Thorpe Park Navigator

Analysis

Background

Thorpe Park is one of the most visited amusement parks in the UK, attended by children and adults alike, ranking 3rd for the most visited theme park in 2019, and 2nd in 2020. The park is located between towns Chertsey and Staines-upon-Thames in Surrey, and operates approximately 30 attractions. Thorpe Park was initially opened to the public in 1979 by Lord Louis Mountbatten, labelled as a 'leisure based visitor attraction'. Since then, numerous rides have been added to the 'roster', and now, Thorpe Park is home to major attractions such as Tidal Wave, Colossus, Nemesis Inferno, Stealth, Saw: The Ride, and the Swarm.

Identification of problem

My project is a Thorpe Park Navigator, which will determine the shortest path to a ride requested by the user. The program is aimed towards people who want to waste no time wandering around the park, which would therefore allow them to go on more rides within the park's open times. Currently, visitors do not have a way to determine the shortest path through the park, and instead, are most likely to go on rides that are the closest to them as they enter the park, common ones being "Rush" and "Vortex". This tendency is especially prevalent in people who are visiting Thorpe Park for the first time, where they do not have a specific preference for certain rides.

Identification of prospective users

- Teens and young adults
- People who are very busy and do not want to waste time trying to find rides to go on
- People who want to make full use of their ticket and go on as many rides as possible
- People going to Thorpe Park for the first time

Prospective User Interview

In order to gauge a better understanding of the problem, I interviewed a random student about their prior experiences of Thorpe Park, and how it could be enhanced.

How many times have you been to Thorpe Park?

3 times now.

Are you able to go on every ride when you visit Thorpe Park?

No, usually I am able to go on around 5-8 rides.

Why?

Unfortunately, a lot of my time is unnecessarily wasted trying to locate the rides I'd like to go on which also have the shortest queues. For example, last time I was at Thorpe Park I wanted to go on Stealth but when I got there I wasn't willing to wait an hour in line, so I had to find a new ride to go on.

Do you feel like you are able to get the most out of your ticket?

Sadly, no. Too much time is wasted wandering around and I can never go on all the best rides in one visit.

How do you navigate around Thorpe Park?

Similarly to most people I have no specific way of navigating the park. Normally I would go on the rides that I come across while wandering around. Although, after visiting 3 times, I now have certain rides that I prefer and therefore I prioritise going on those rides.

Is this time efficient?

No, because many others have the same thinking and consequently the queue situation worsens.

When people go to Thorpe Park for the first time, how many rides would they usually be able to go on?

When people visit for the first time, they will find it very difficult to navigate the park in a way that will allow them to go on anymore than 5 rides.

Why is this the case?

Because people visiting for the first time would most likely be unaware of what rides they would like to go on and are unfamiliar with the park, which would likely result in them wasting a lot of time trying to find rides.

What features would benefit you the most when it comes to a Thorpe Park Navigator?

One feature that I would like to see is the ability to enter a preference list, which will create a path for me to visit my preferred rides as fast as possible. Another ideal feature would be that the program asks me for the maximum amount of time I would be willing to wait in a queue, and if the queue time exceeds how long I would be willing to wait, then the ride is excluded from the shortest path.

In conclusion, this interview highlighted to me the issues that arise as a result of people being unaware of how to navigate the park in a time efficient manner, such as people being unable to make the most out of their ticket and struggling to go on more than 5 rides in one visit. Moreover, the interviewee provided suggestions for features that could be implemented.

Questionnaire

To identify which rides are to be included in the application, I sent out a questionnaire to 50 people in my school. The result of this would allow me to identify any rides that are unpopular and not required to be graphed and included in the program.

Ride Name	**Ride Category**	**Best attraction (Number of votes)**	**Worst attraction (Number of votes)**
The Swarm	Rollercoaster	5	0
SAW - The Ride	Rollercoaster	7	0
Stealth	Rollercoaster	11	0
Nemesis Inferno	Rollercoaster	2	0
Colossus	Rollercoaster	3	0
Walking Dead: The Ride	Rollercoaster	2	1
Flying Fish	Rollercoaster	0	0
Samurai	Thrilling flat ride	1	0
Quantum	Thrilling flat ride	0	0
Zodiac	Thrilling flat ride	0	1
Vortex	Thrilling flat ride	2	0
Detonator: Bombs Away	Thrilling flat ride	1	0
Rush	Thrilling flat ride	0	0
Rumba Rapids	Water ride	6	1
Storm Surge	Water ride	0	1
Tidal Wave	Water ride	4	0
Depth Charge	Water ride	1	0
Mr Monkey's Banana Ride	Family flat ride	0	16
Storm in a Teacup	Family flat ride	1	1
Dodgems	Family flat ride	0	0
Ghost Train	Dark Ride	3	1

Black Mirror Labyrinth	Maze Attraction	0	7
Angry Birds Experience	4D Cinema	0	12
Amity Beach	Beach	0	9

Rows highlighted in red show attractions to be excluded from the program

Identification of user needs / Objectives

1. **Account creation and management**
   - 1.1. Users should be able to create an account by providing a unique username and password.
   - 1.2. A table within the database should exist to store user account information, including the username, password, and a unique user identifier.

2. **User input and preferences**
   - 2.1. Users should be able to input their current location within the theme park by specifying the identifier of the ride or attraction they are currently at.
   - 2.2. Users should be able to specify the ride or attraction they want to go to by inputting the identifier of the ride or attraction.
   - 2.3. Users should be able to input their top 3 preferred rides or attractions in order of priority.
   - 2.4. Program should ensure that all 3 preferences are unique.
   - 2.5. A table within the database should exist to store user preferences, including the unique user identifier, the ride or attraction identifier, and the priority order for each preference.

3. **Shortest path calculation and display**
   - 3.1. The program should calculate the shortest path between the user's current location and their destination ride or attraction.
   - 3.2. The path must include all 3 of the user's preferred rides or attractions in the order of priority specified by the user.
   - 3.3. The path should be displayed to the user with the names of the rides or attractions, rather than just the ride identifiers.

4. Ride information

4.1. A table within the database should exist to store information about the rides or attractions, including the ride names and identifiers.

4.2. A table within the database should exist to store information about the edges between rides or attractions, including the ride identifier for the start and end points of each edge and the weight of the edge.

5. Journey time calculation and queue time input

5.1. The program should output the total time it will take for the user to complete the journey between their current location and their destination, taking into account the time it will take to travel between each ride or attraction on the path.

5.2. Users should be able to input the current queue times for the rides or attractions they are visiting.

5.3. The program should add the queue times to the total journey time to determine the overall cost of the journey for the user.

5.4. A table within the database should exist to store the current queue times for each ride or attraction.

5.5. The program should ask the user for their maximum waiting time in a queue.

5.6. If the waiting time for a ride or attraction exceeds the user's maximum waiting time, the program should exclude that ride or attraction from the shortest path and recalculate the path to find an alternative route.

Possible solutions to the problem

One possible solution to the problem is to use depth-first search. Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node, or any arbitrary node, and visits each adjacent node that hasn't been visited already. This process continues as a loop until there is no unvisited adjacent node. Then, the algorithm must backtrack and check for other unvisited nodes and visit them. Once all nodes have been visited, the nodes can be printed in the path taken. However, a drawback of depth-first search is that the path found may not be the most optimal, as more nodes may be visited than what is required for optimisation.

Another possible solution is to use breadth-first search. Breadth-first search, similarly to depth-first search, is an algorithm for searching a tree data structure. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory in the form of a queue is needed to keep track of the child nodes that were encountered but not yet explored. A drawback of this search is that more memory is consumed, seeing as all connected vertices must be stored in memory.

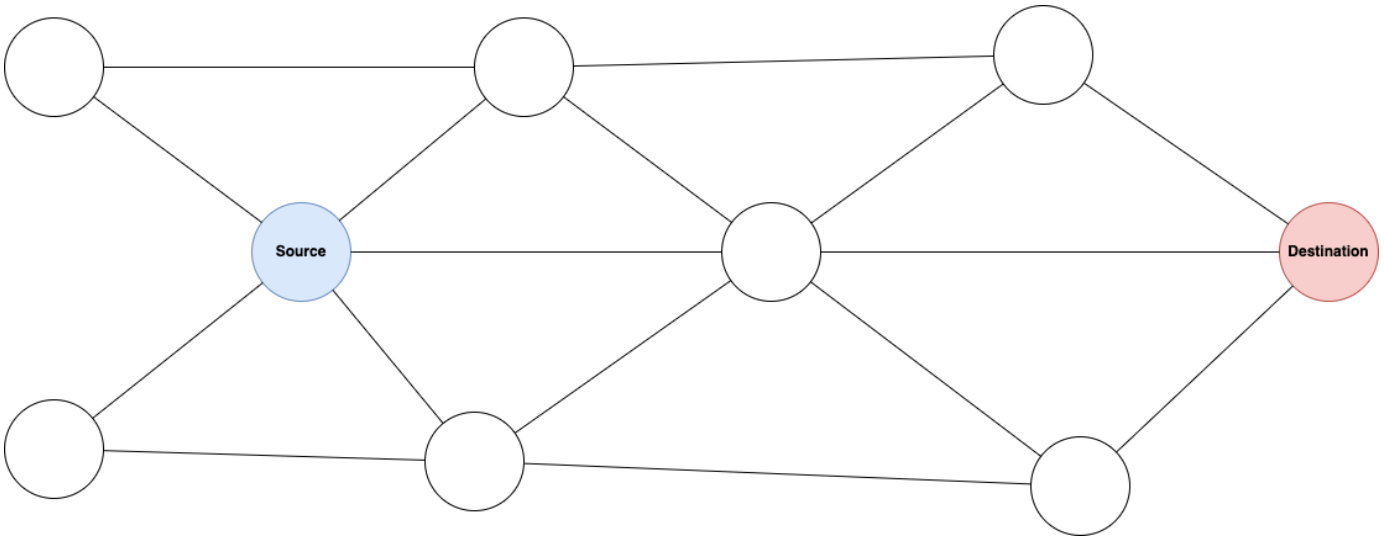
My Chosen Solution: Dijkstra algorithm

Research into Dijkstra

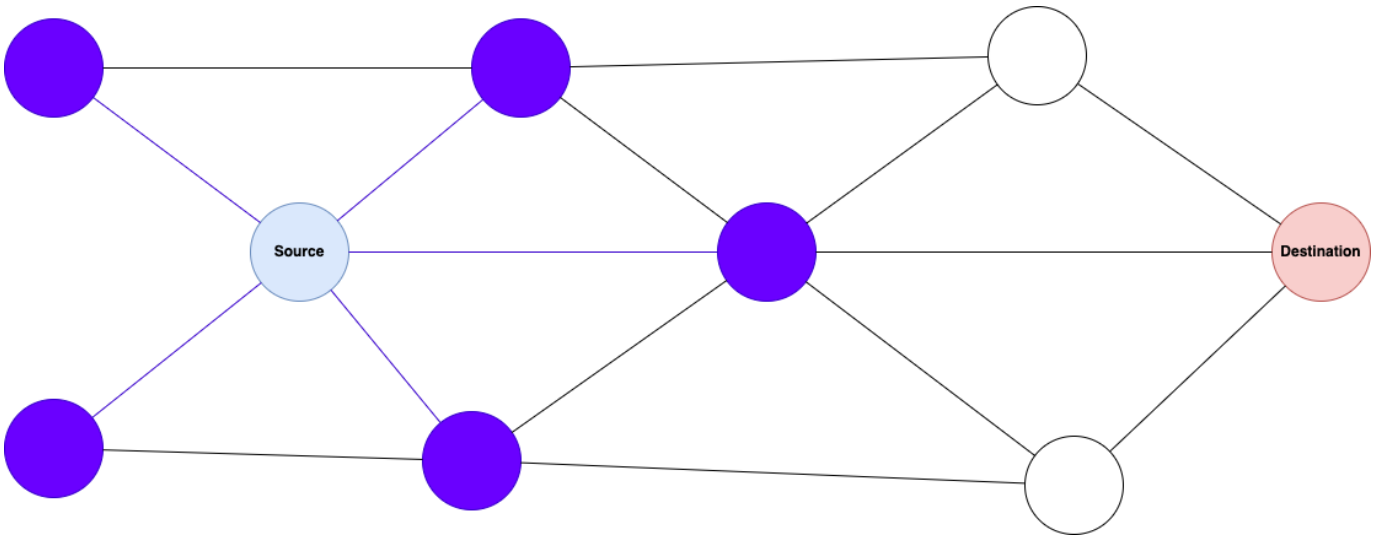
Created by Edsger W. Dijkstra in 1956, Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph. The algorithm is written in the context of weighted graphs, with the graph containing nodes and edges, which are the connections between nodes and have weights associated with them. Therefore, the shortest path is identified by calculating the path that has the lowest cost between a source node and destination node, as the weights of the edges that belong to the path are added together to find the total cost. The algorithm works by maintaining a priority queue of nodes, ordered by their distance from the starting node. It repeatedly extracts the node with the minimum distance from the priority queue and updates the distances of its neighbours based on the newly discovered shorter path. The algorithm terminates when the destination node is extracted from the priority queue.

Dijkstra Step-by-step example

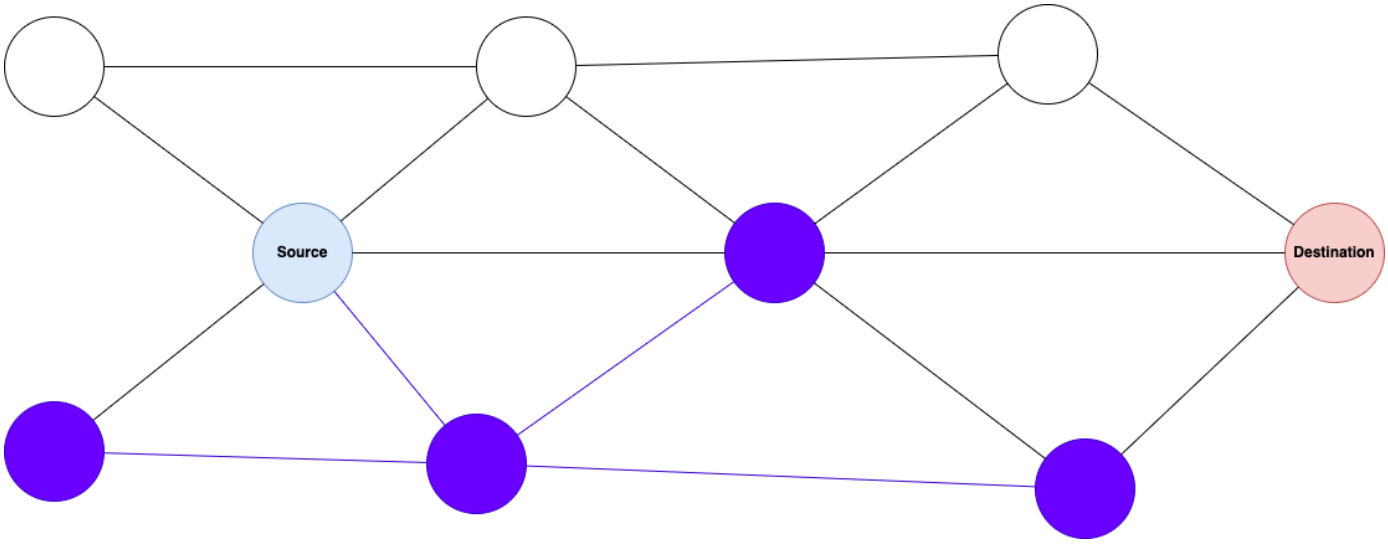
Step 1: Select Source and Destination nodes



Step 2: From the source node, visit paths to all adjacent nodes and make note of the distances of each node from the source. All unvisited nodes will have a distance of infinity. Mark the source node as fully explored.



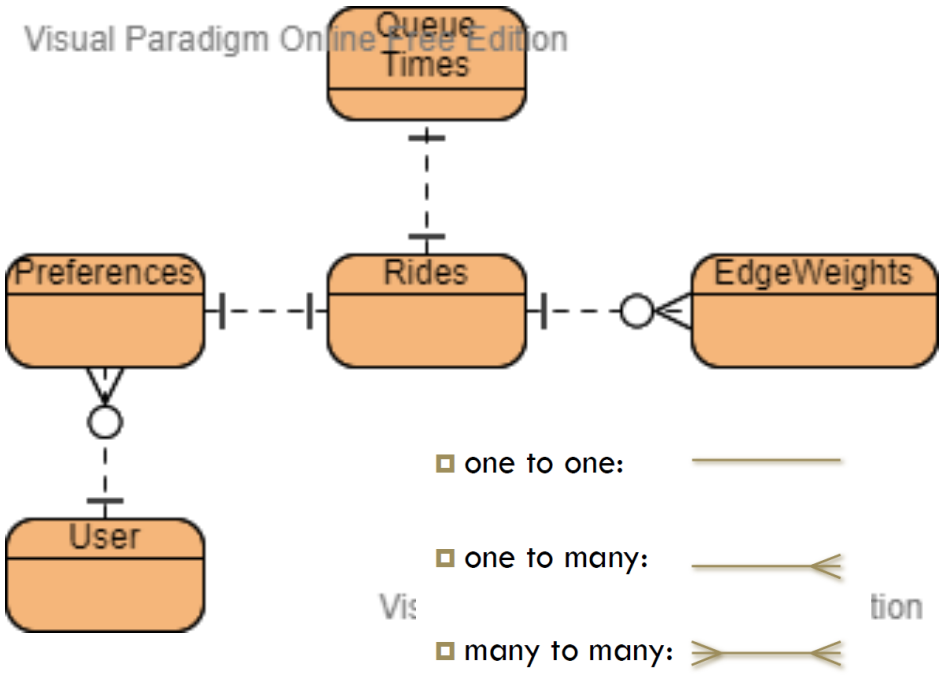
Step 3: Choose the node that is closest to the source node and visit the paths to all adjacent nodes, making note of the total distance of each node from the source. The total distance can be found by adding together the distance between the source node and node B and the distance between node B and an adjacent node.



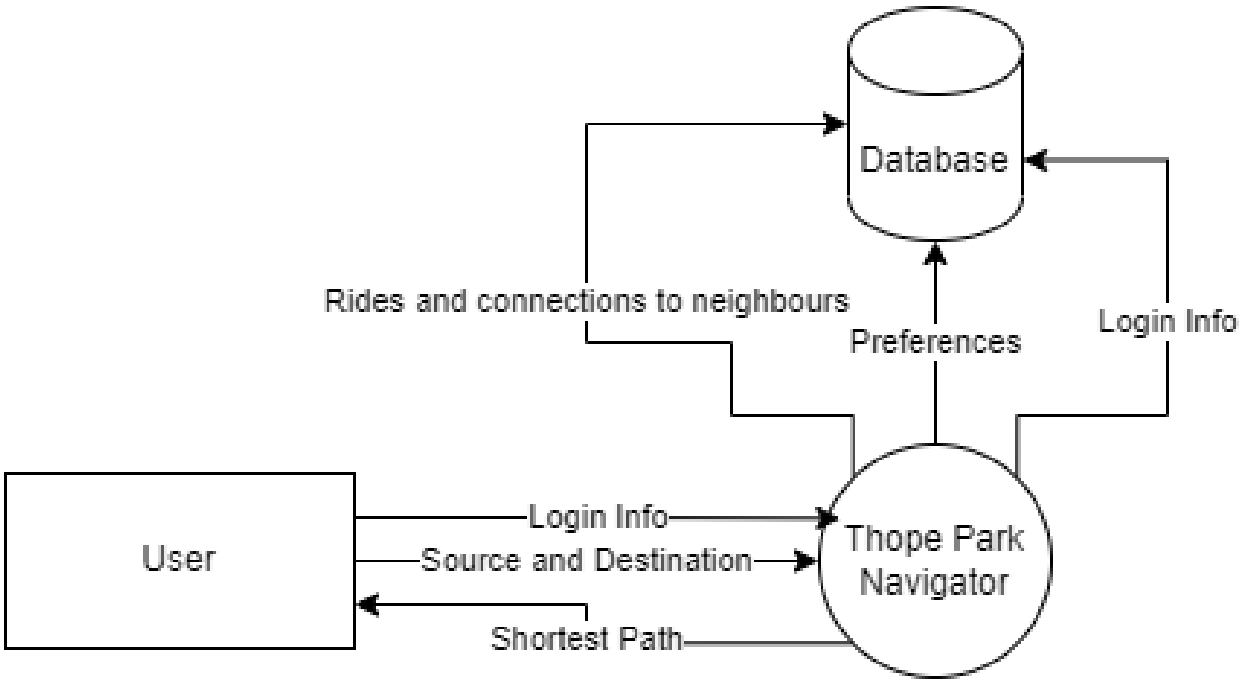
Step 4: Repeat until all paths have been explored, and the destination node is extracted to return the shortest path.

Initial Modelling

Entity-Relationship Diagram

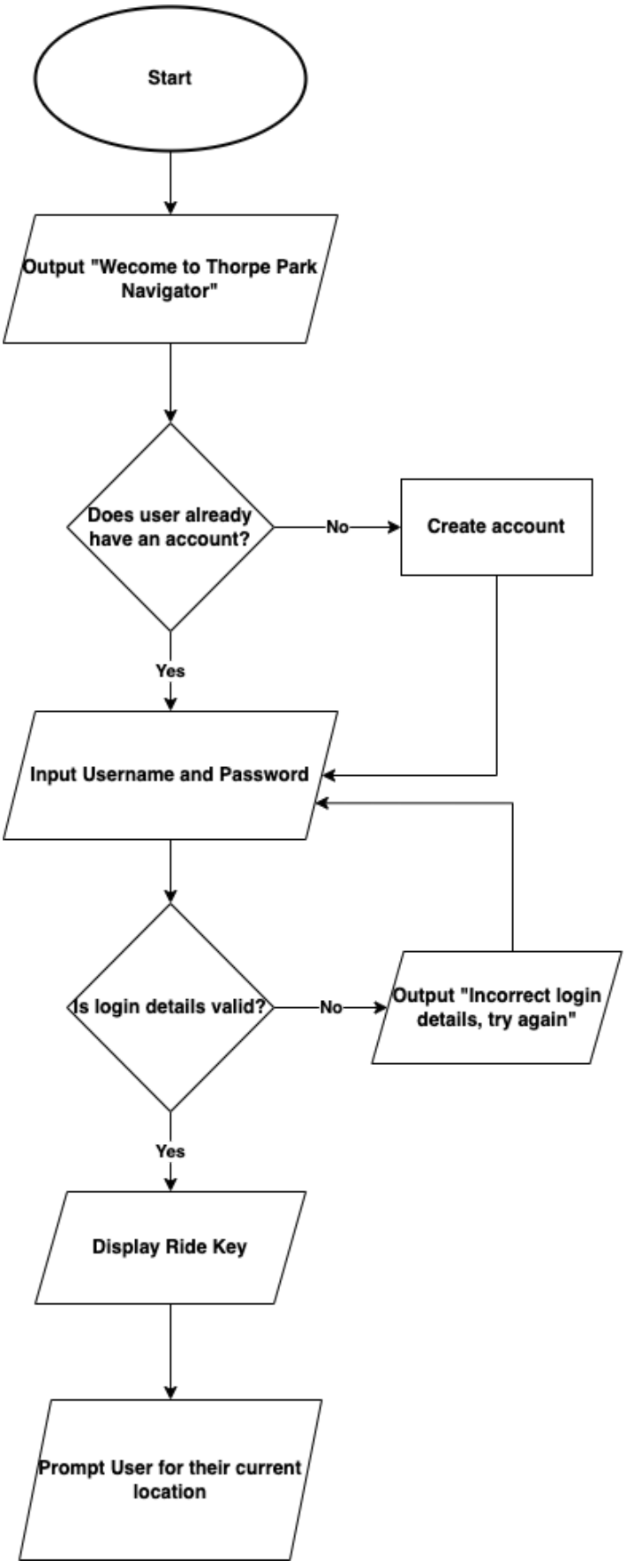


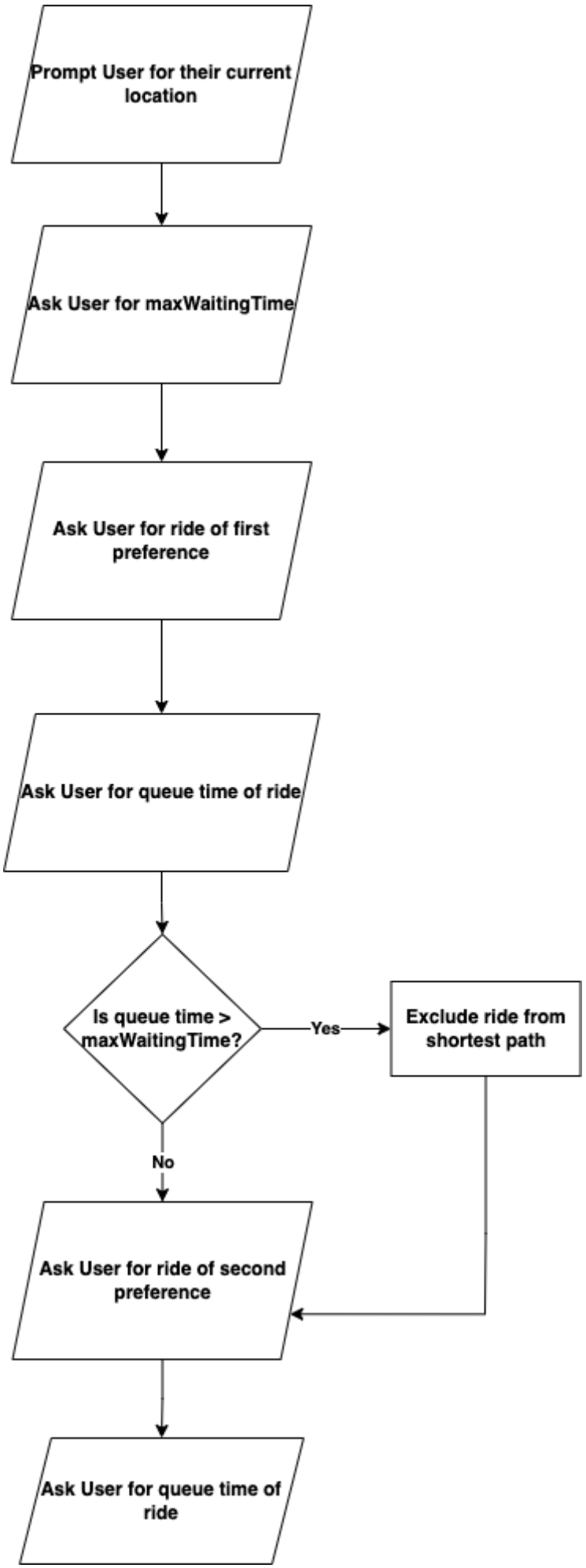
Data Flow Diagram

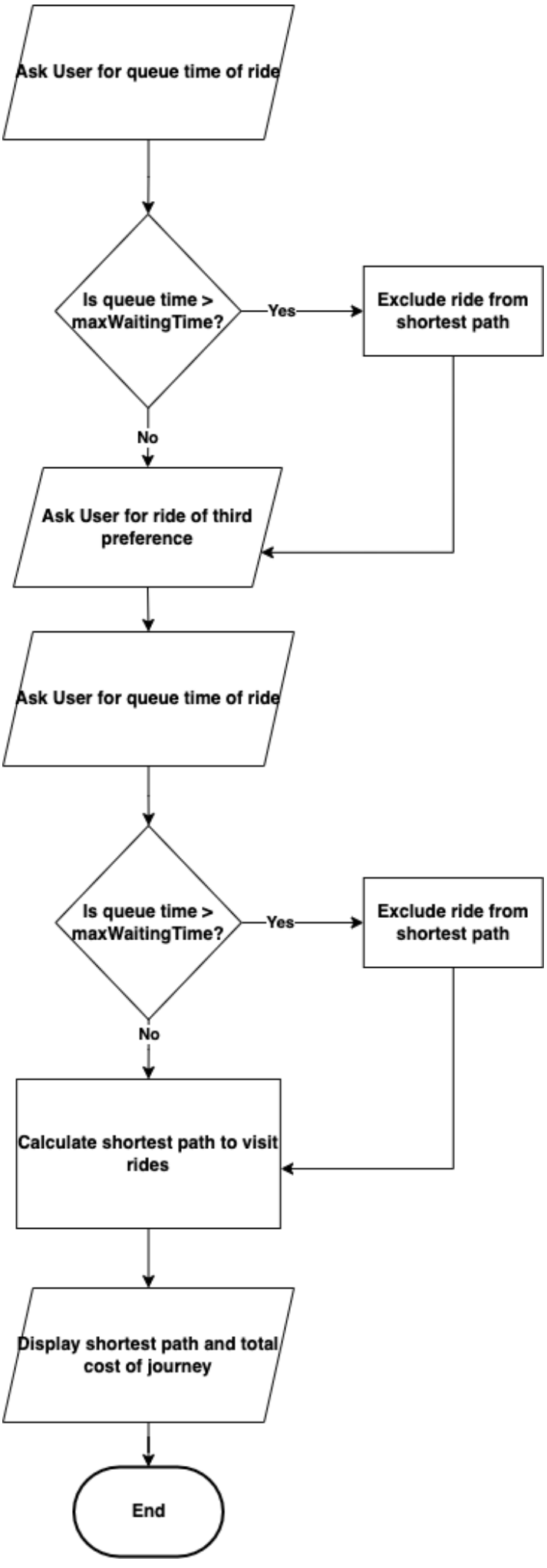


Design

System Flowchart

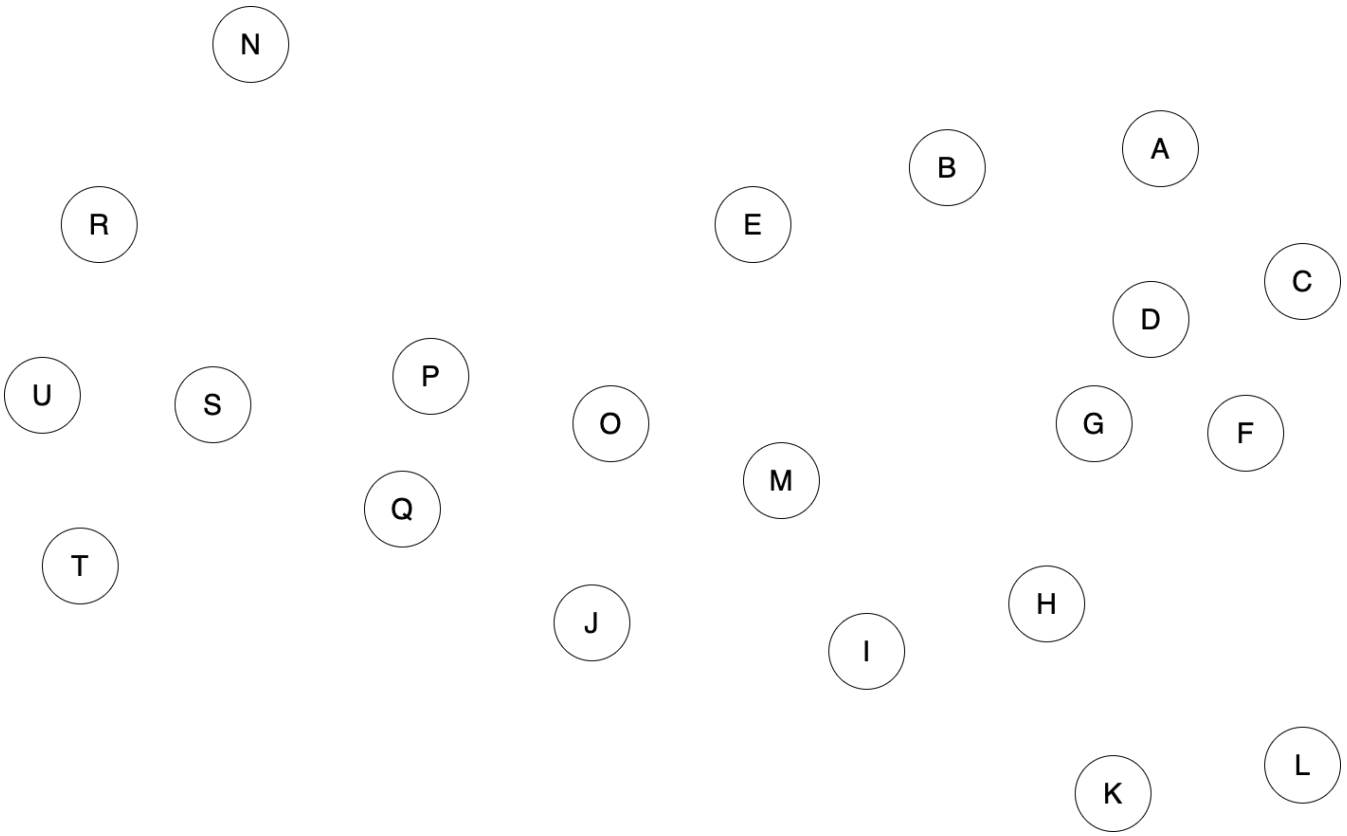



Prompt User for their current location
Ask User for maxWaitingTime
Ask User for ride of first preference
Ask User for queue time of ride
Is queue time > maxWaitingTime?
Yes
Exclude ride from shortest path
No
Ask User for ride of second preference
Ask User for queue time of ride


Ask User for queue time of ride
Is queue time > maxWaitingTime?
Yes
Exclude ride from shortest path
No
Ask User for ride of third preference
Ask User for queue time of ride
Is queue time > maxWaitingTime?
Yes
Exclude ride from shortest path
No
Calculate shortest path to visit rides
Display shortest path and total cost of journey
End

Graphing Thorpe Park

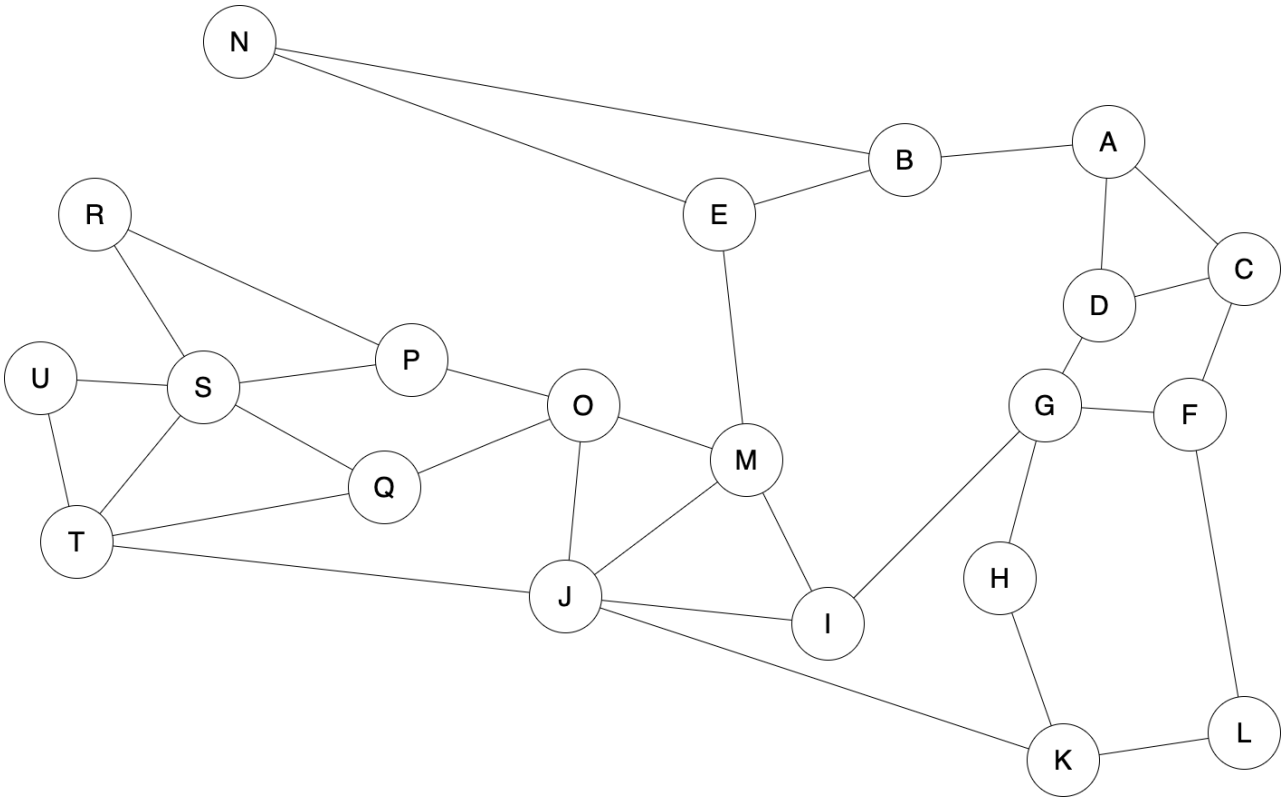
Using the above map, I mapped out the 20 rides - which were suggested to be included with the questionnaire carried out - into a graph, and produced the following result:



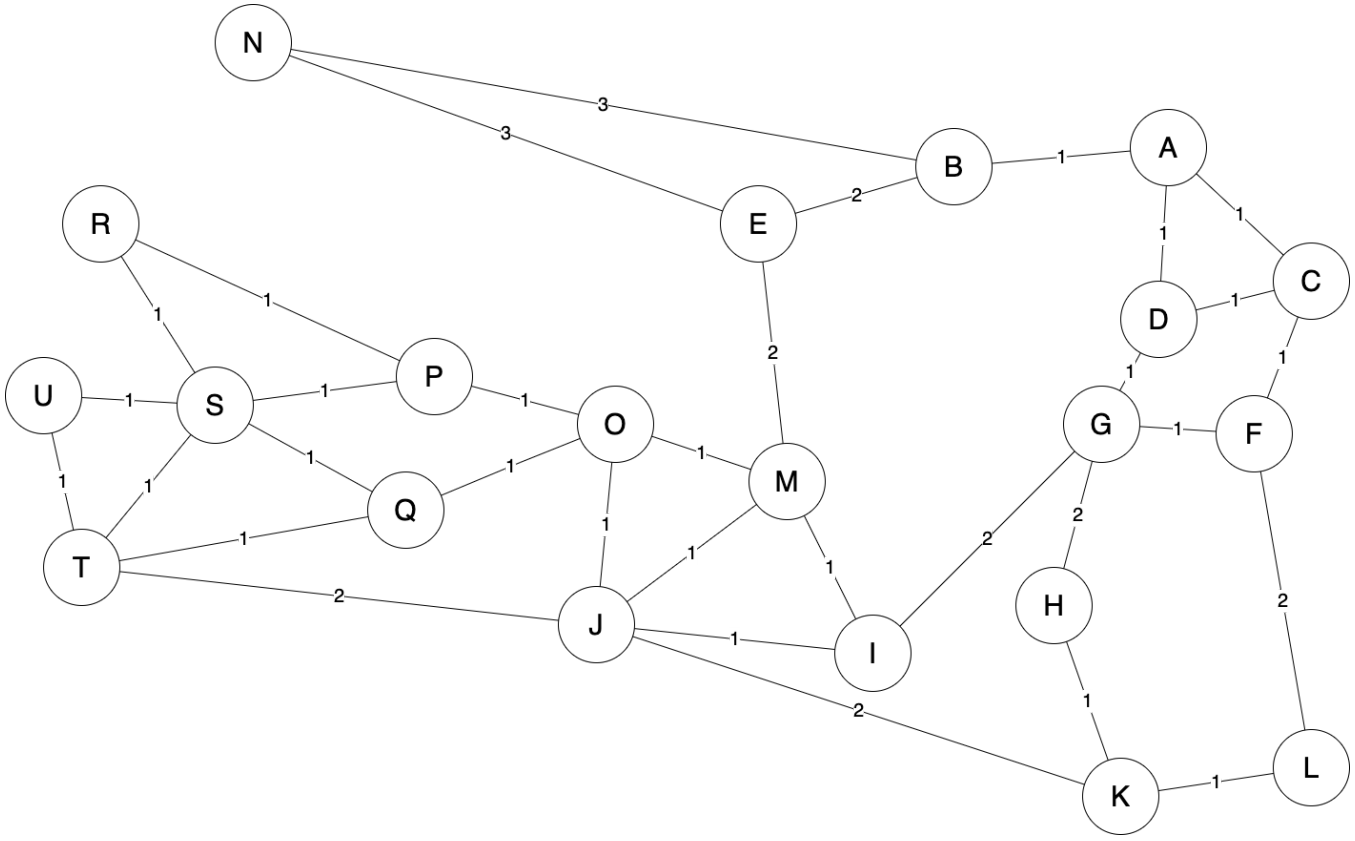
Node A represents the Entrance - "The Dome"

The next step was to create edges between the nodes. The edges on the graph are to mirror the paths that exist between rides in Thorpe Park. To aid me with this, I utilised OpenStreetMap to accurately establish the edges.

Footways represented with red dotted lines


N
B
A
R
E
C
D
P
U
S
O
G
F
M
Q
T
H
J
I
K
L

The next step was to associate the edges with weights, with the value of the weight being determined on the basis of the time taken in minutes to travel along the edge. For this, I used Google Maps, which enabled me to use point to point navigation to determine the time taken to walk through the paths.



Dijkstra's algorithm

Pseudocode

```
Procedure dijkstra(Graph, StartingNode)
Begin
  for each node N in Graph
    path[N] <- infinite
    previous[N] <- NULL
    IF N != StartingNode, add N to PriorityQueue
  path [StartingNode] <- 0

  WHILE PriorityQueue IS NOT EMPTY
    X <- Extract lowest from PriorityQueue
    for each unvisited neighbouring node of current node X
      tempDistance <- path [X] + edgeweight(X, N)
      IF tempDistance < path [N]
        path [N] <- tempDistance
        previous[N] <- X
  return path[], previous[]
End
```

Explanation of pseudocode:

- Initialise a priority queue to store the nodes, ordered by their distance from the starting node. Initialise the distance of the starting node to 0, and the distance of all other nodes to infinity.
- While the priority queue is not empty:
    - Extract the node with the minimum distance from the priority queue.
    - For each neighbour of the node:
        - Calculate the distance from the starting node to the neighbour through the current node.
        - If the calculated distance is less than the current distance of the neighbour, update the distance of the neighbour and add it to the priority queue.
- The distance of the destination node represents the shortest path from the starting node to the destination node.

Data structures used to implement Dijkstra's algorithm

Linked list

A linked list is a linear data structure that consists of a sequence of nodes, where each node stores a reference to an object and a reference to the next node in the sequence. The first node in the sequence is called the head, and the last node is called the tail. The use of a linked list allows for elements to be inserted and removed in constant time.

Adjacency list

An adjacency list is a way of representing a graph as a list of lists, where each node in the graph is represented by a list of its neighbours. Adjacency lists are a compact way to represent sparse graphs, which are graphs where most pairs of nodes are not connected by an edge. Adjacency lists are also useful for storing graphs that are not well suited for indexing, such as graphs with string labels on the nodes. In an adjacency list, the nodes can be represented by any object with a unique identity, such as a string, and the list can be indexed by the node object itself, rather than by its label.

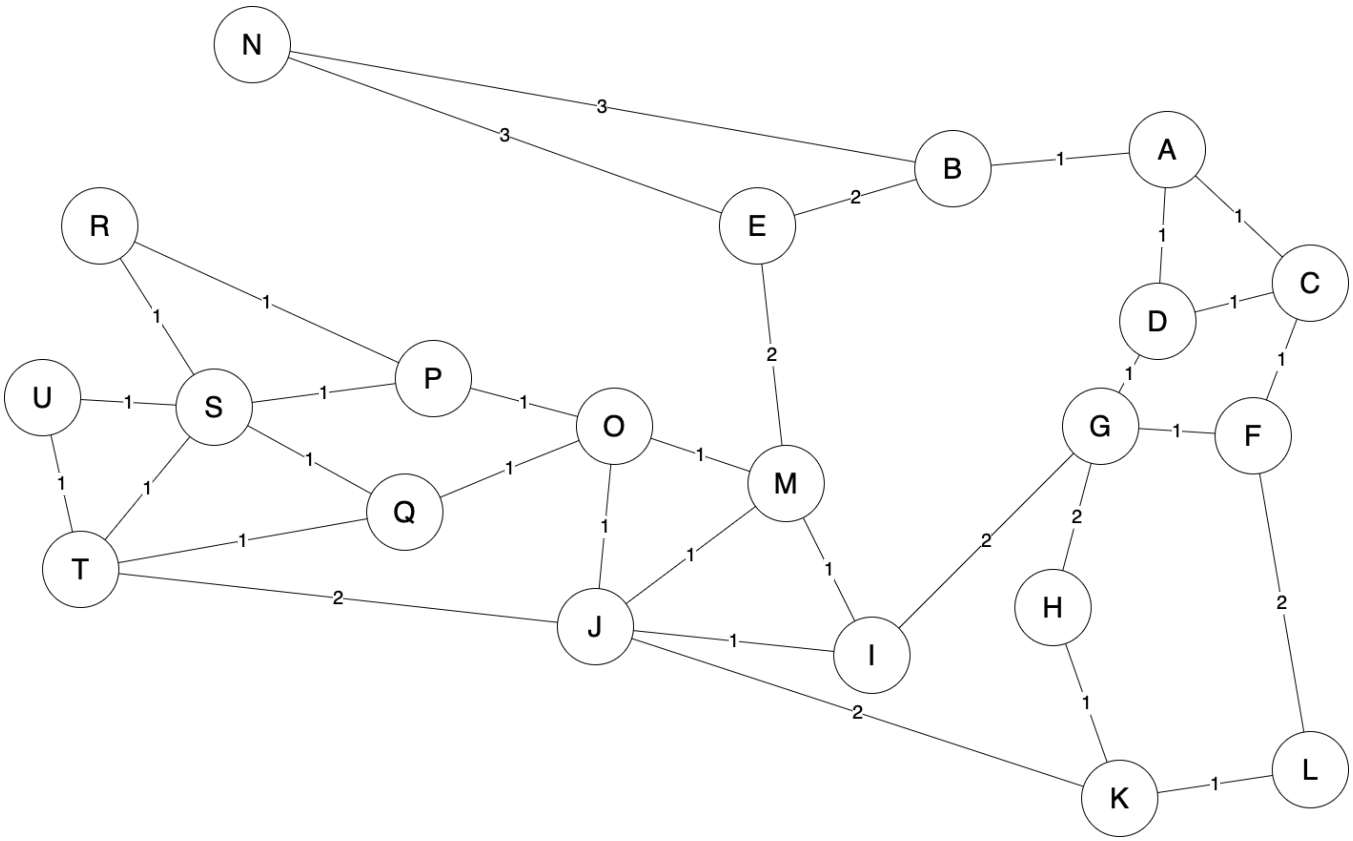
Adjacency linked list

An adjacency linked list is a variant of an adjacency list that uses linked lists to store the neighbours of each node in a graph. This can be more memory efficient than using a standard list, because a linked list only stores the references to the neighbours, rather than a complete copy of the list. Adjacency linked lists, as with normal adjacency lists, are useful for storing graphs that are not well suited for indexing, such as graphs with string labels on the nodes. They are also useful for storing graphs that have a large number of edges, because they only store the references to the neighbours, rather than a complete copy of the list.

Priority Queue

A priority queue is a data structure that stores a collection of elements and allows them to be retrieved in a particular order, based on their priority. In a priority queue, elements are added with a priority, and the element with the highest priority is always retrieved first. The priority can be any type of value that is comparable, such as a number or a string. This makes a priority queue useful for implementing graph search algorithms, such as Dijkstra's algorithm.

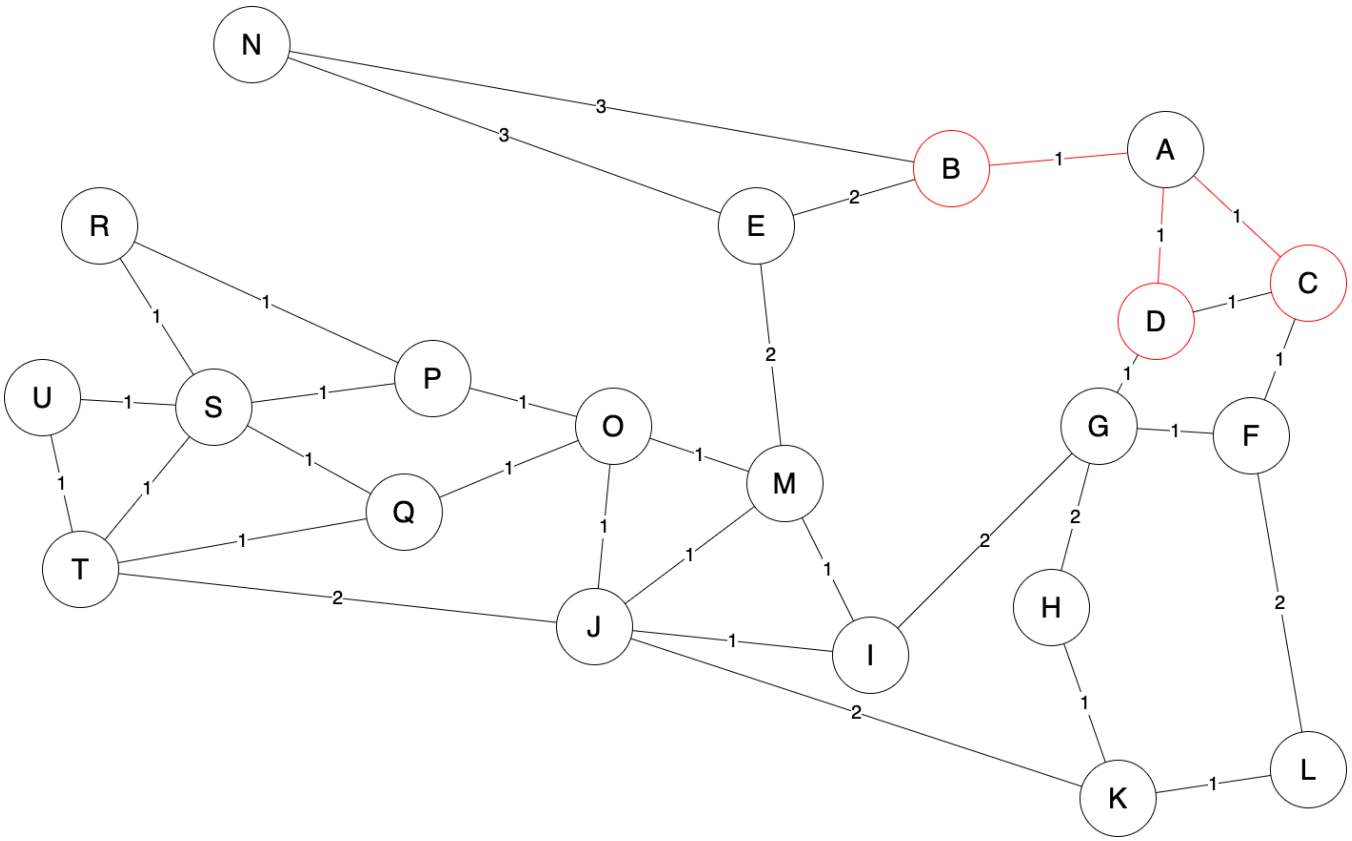
Dijkstra Dry Run - Using Dijkstra's algorithm to search through Thorpe Park graph



Here is the graph of Thorpe Park with nodes A to U. The numbers associated with the edges represent how many minutes it would take to travel along the edge.

The task is to get from node A to node U in the shortest amount of time. The first step is to create a table containing all the nodes:

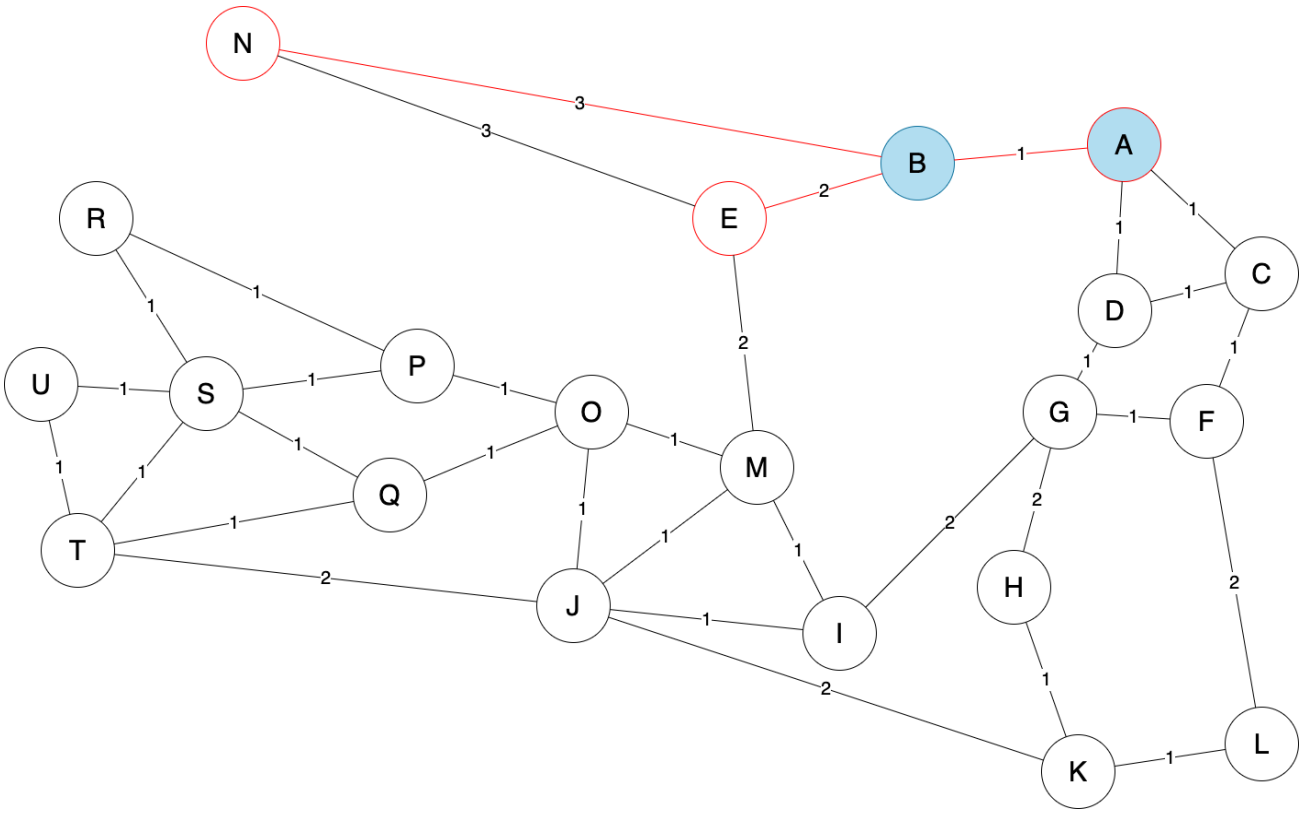
Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A																					



The connections to other nodes from A are to be filled into the table, therefore nodes B, C and D are filled in with weight 1. The subscripted A means that the weight is relative to A. Where A is not connected to a node, the infinity symbol (∞) is entered. This shows that there is no current way to reach the nodes, suggesting that it takes an infinite amount of time to reach them. A has now been fully investigated and can be shaded.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A

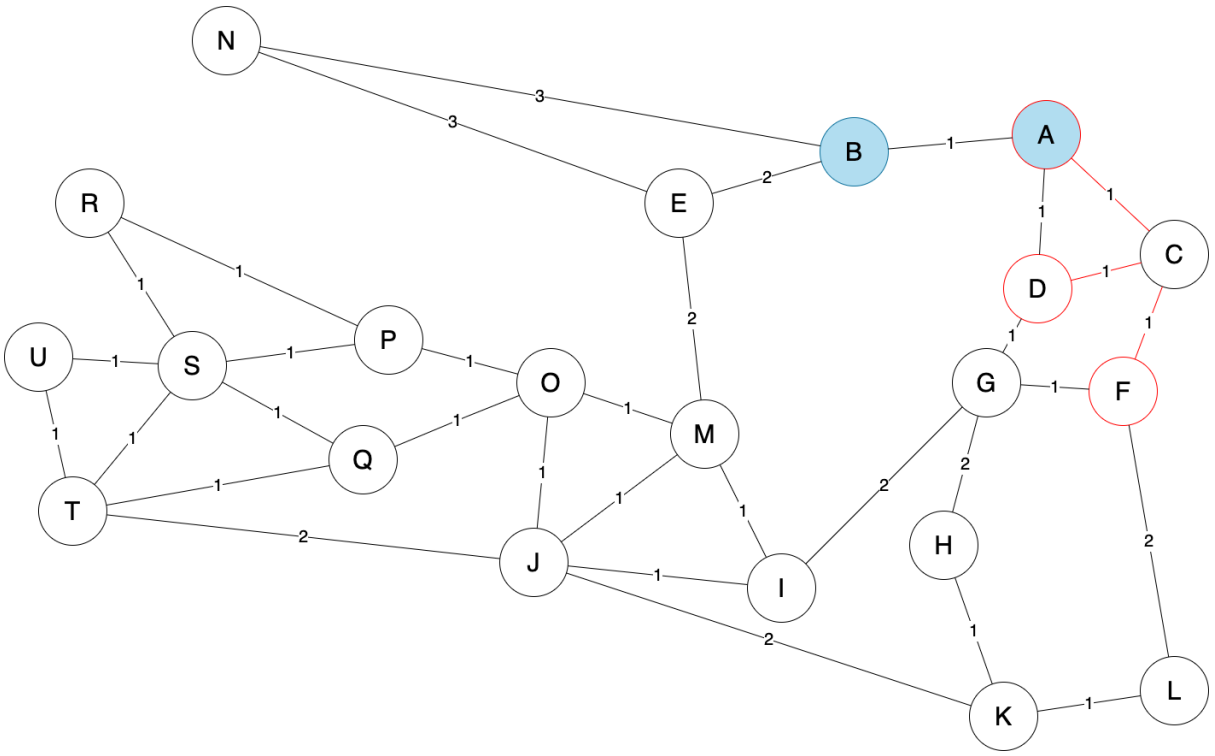
The next step is to select the node with the smallest weight from the table. However, in this case, nodes B, C and D have the same weight of 1. As a result, there is no direct node to prioritise, so any node can be visited. In this example, B will be visited.



Node B has connections to E and N, therefore the time taken to travel to other nodes is still infinite. According to the graph, it takes 2 minutes to travel from B to E, and because the table is relative to A, subsequently taking 1 + 2 minutes to get to E from A. 3 is less than infinity and takes priority - a shorter path to E has been discovered. In the same way, it takes 1 + 3 minutes to travel to N from A, and 4 is less than infinity and takes priority - discovering a shorter path to N. The subscript B shows that node B is the previous node visited. B has now been fully investigated and can be shaded.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A

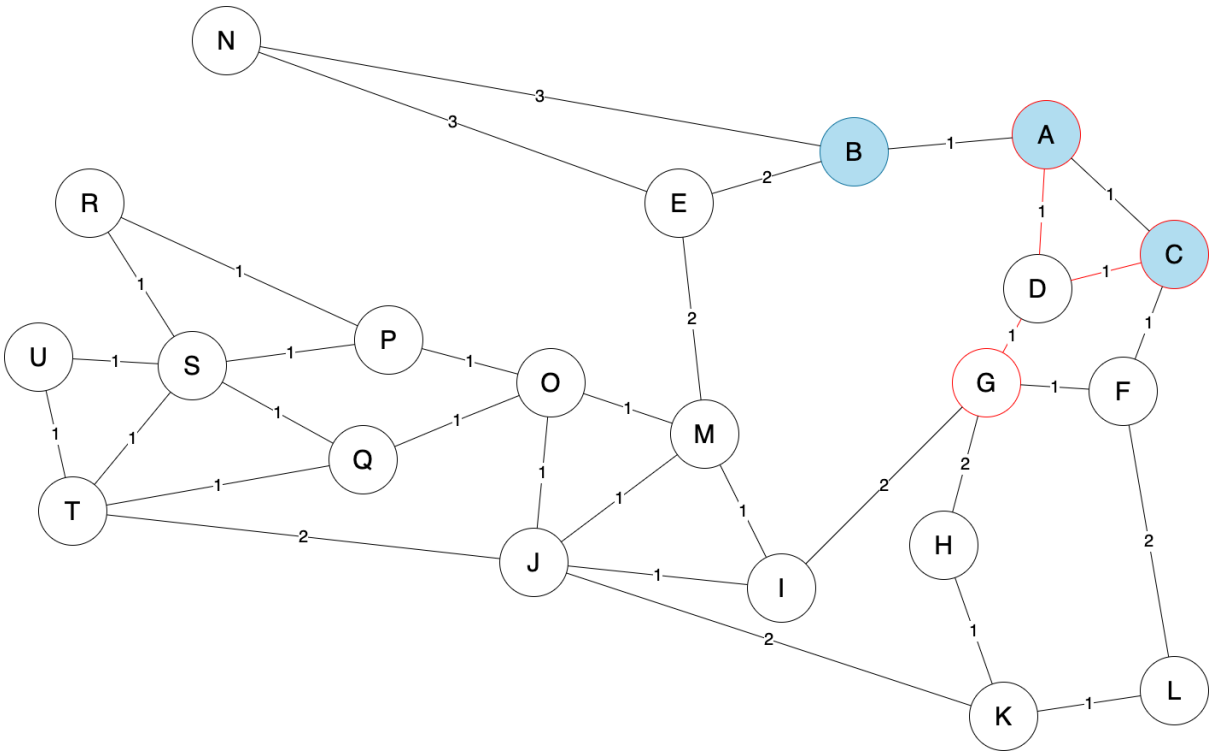
Once again, C and D are equal with lowest weight in the table, resulting in no order of priority being set, so C can be visited next.



C has connections to D and F. It takes 1 minute to travel from C to D, and therefore 1 + 1 minutes from A to D via C. However, a connection from A to D already exists, and costs less, which means that the existing weight can remain unchanged. It takes 1 minute to travel from C to F, which is 1 + 1 minutes from A to F. 2 is less than infinity so a shorter path to F has been discovered. The subscript shows that C is the previous node visited. Node C has been investigated fully and can be shaded.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A

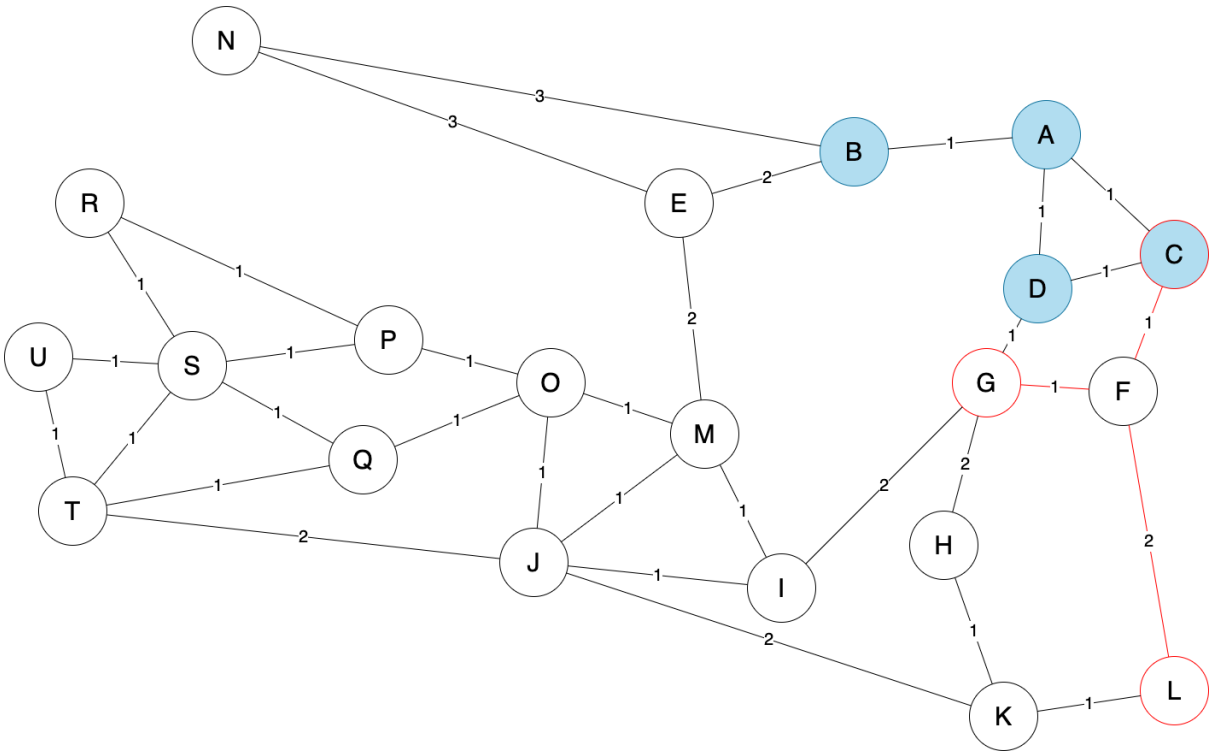
Column D contains 1, which is the lowest weight in the table, which means D is to be visited next.



D has connections to C and G. A connection from A to C already exists with a shorter time taken to travel, rather than going via D, as 1 is less than 2, 1 takes priority and remains unchanged. It takes 1 minute to go to G from D, which is 1 + 1 minutes from A to G, 2 is less than infinity and takes priority - a shorter path to G has been discovered. The subscript D shows that D is the previous node visited. D has been fully explored and can be shaded.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A

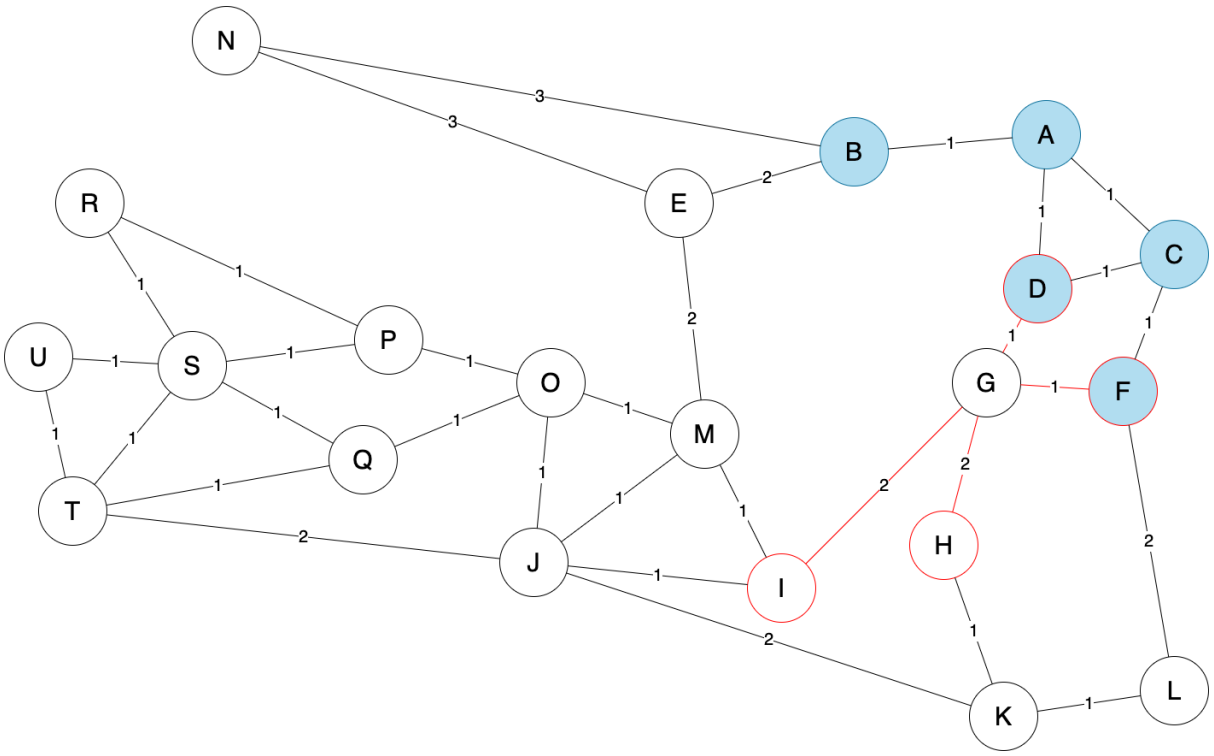
Both columns F and G contain 2, so neither take priority. As a result, F can be visited next.



F has paths to G and L. It takes 1 minute to travel to G from F, which means 3 minutes from A, which is greater than the existing weight, therefore it doesn't take priority and replace the existing connection. It takes 2 minutes to travel from F to L, which means 4 minutes from A. 4 is less than infinity and takes priority - a shorter path to L has been found.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A

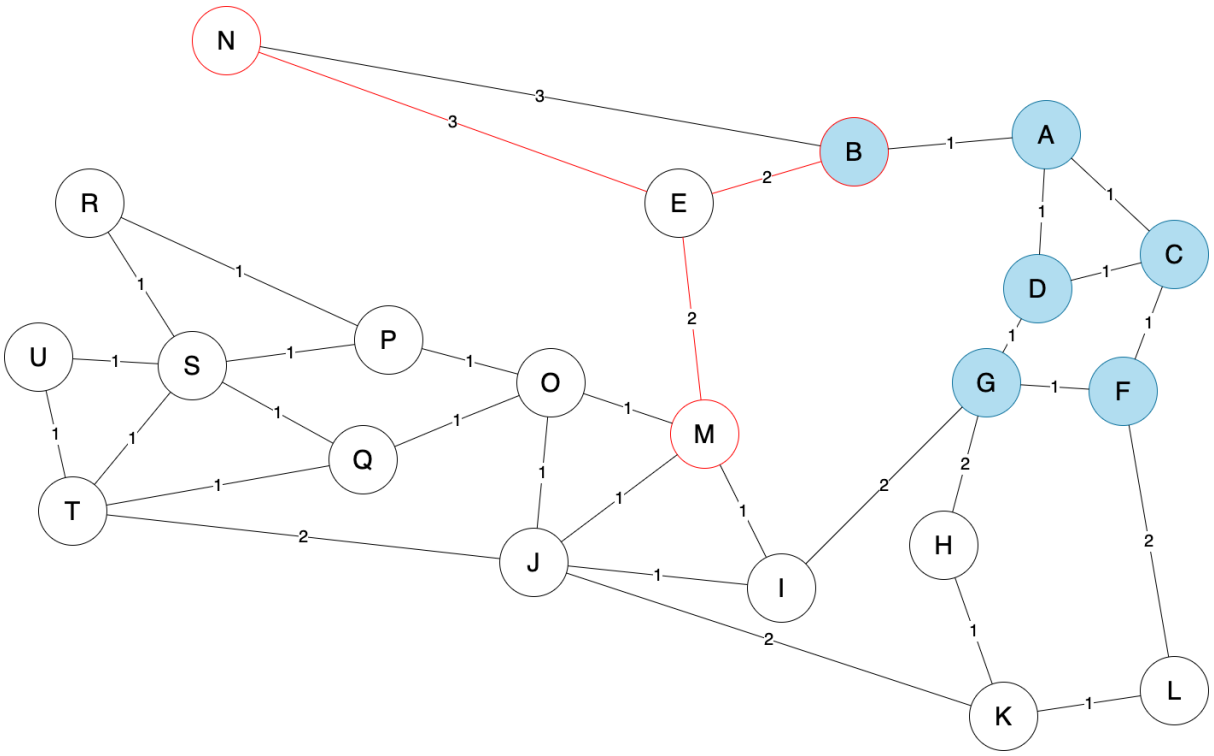
Column G contains the lowest unvisited weight in the table, and is visited next.



G has paths to F, H and I. It takes 1 minute to travel from G to F, which is 3 minutes from A. This is greater than the existing weight of 2 in the table, and as a result does not take priority. It takes 2 minutes to go to H from G, which is 4 minutes from A. 4 is less than infinity and takes priority, resulting in a shorter path to H being discovered. It takes 2 minutes to go to I from G, which is 4 minutes from A. 4 is less than infinity and takes priority, resulting in a shorter path to I being discovered.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A

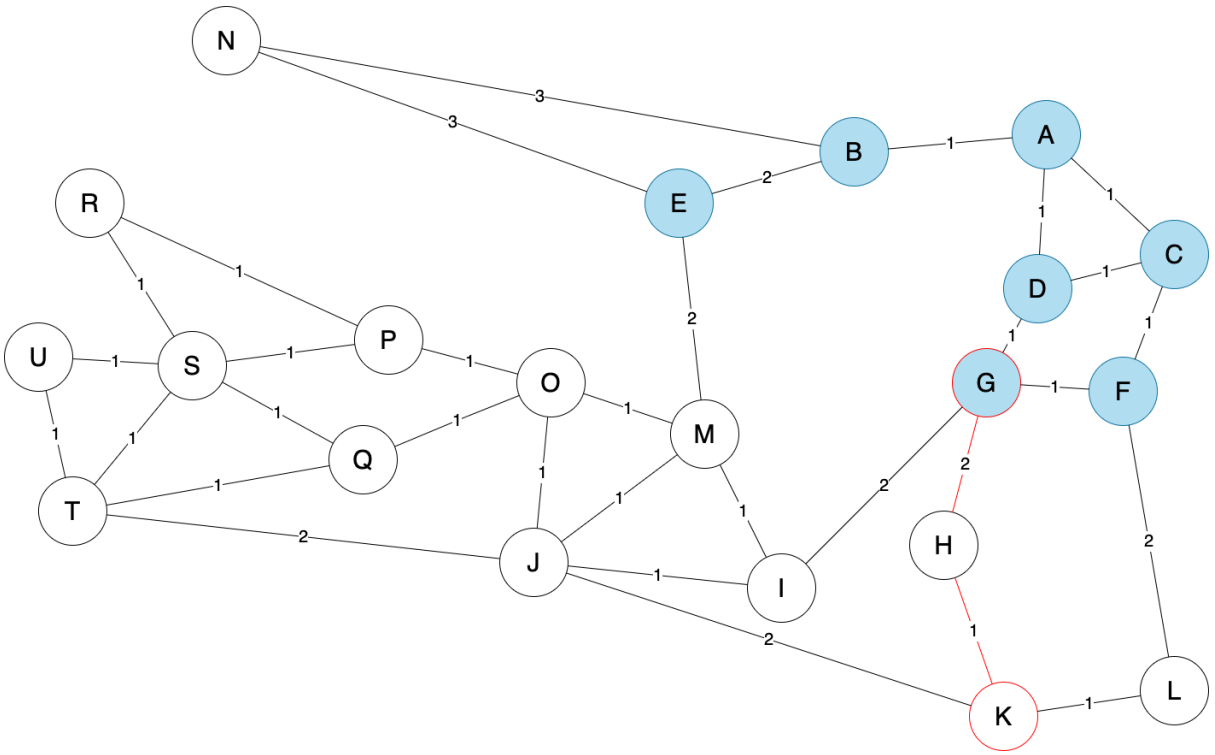
Column E contains the smallest weight with 3, so it is visited next.



Node E has connections to N and M. It takes 3 minutes to travel to N from E, which means 6 minutes from A. 6 is greater than the existing amount of time taken to get to N in the table, which is 4, and therefore does not take priority. It takes 2 minutes to travel from E to M, which is 5 minutes from A. 5 is less than infinity, and takes priority, with a shorter path to M being found as a result.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
E	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A

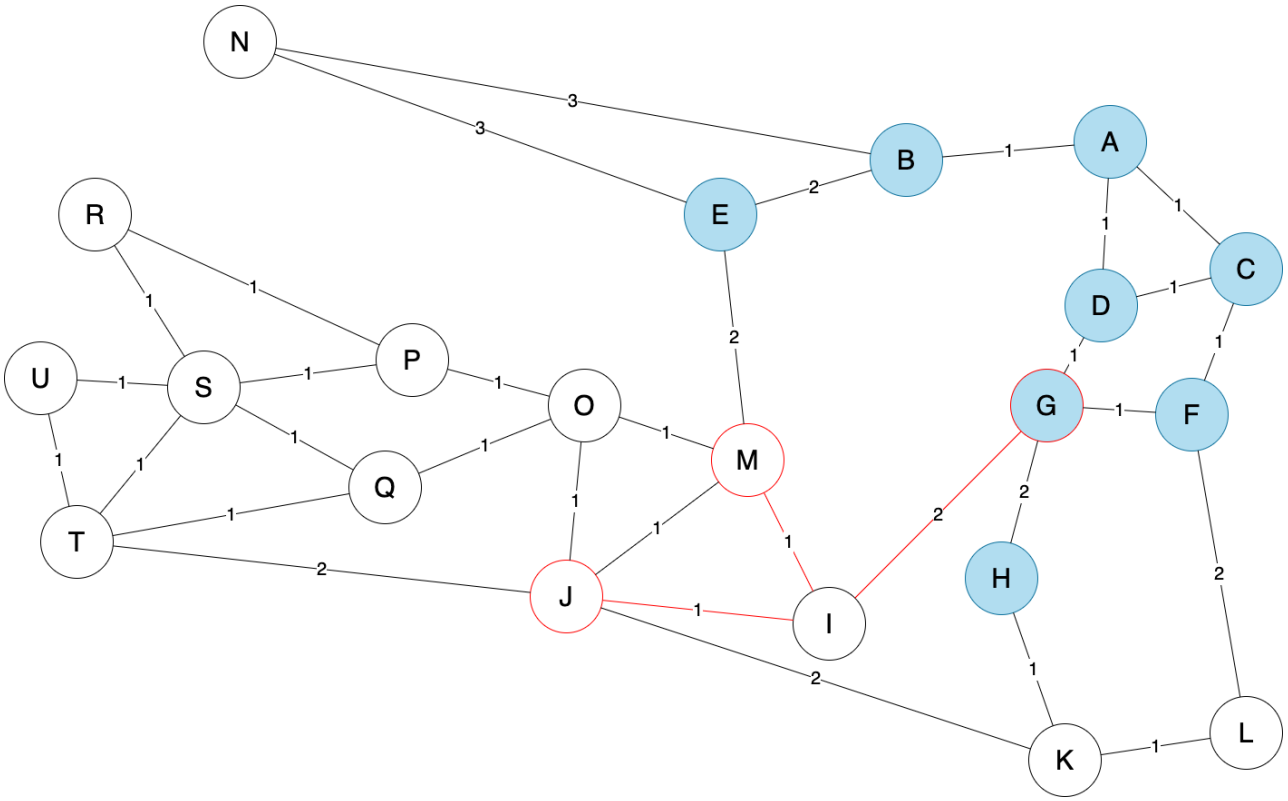
Columns H, I, L and N all contain 4. There is no specific node to be visited, so node H can be visited.



H has a connection to K. It takes 1 minute to travel to K from H, which means 5 minutes from A. 5 is less than infinity and takes priority, which results in a shorter path to K being found.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
E	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
H	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A

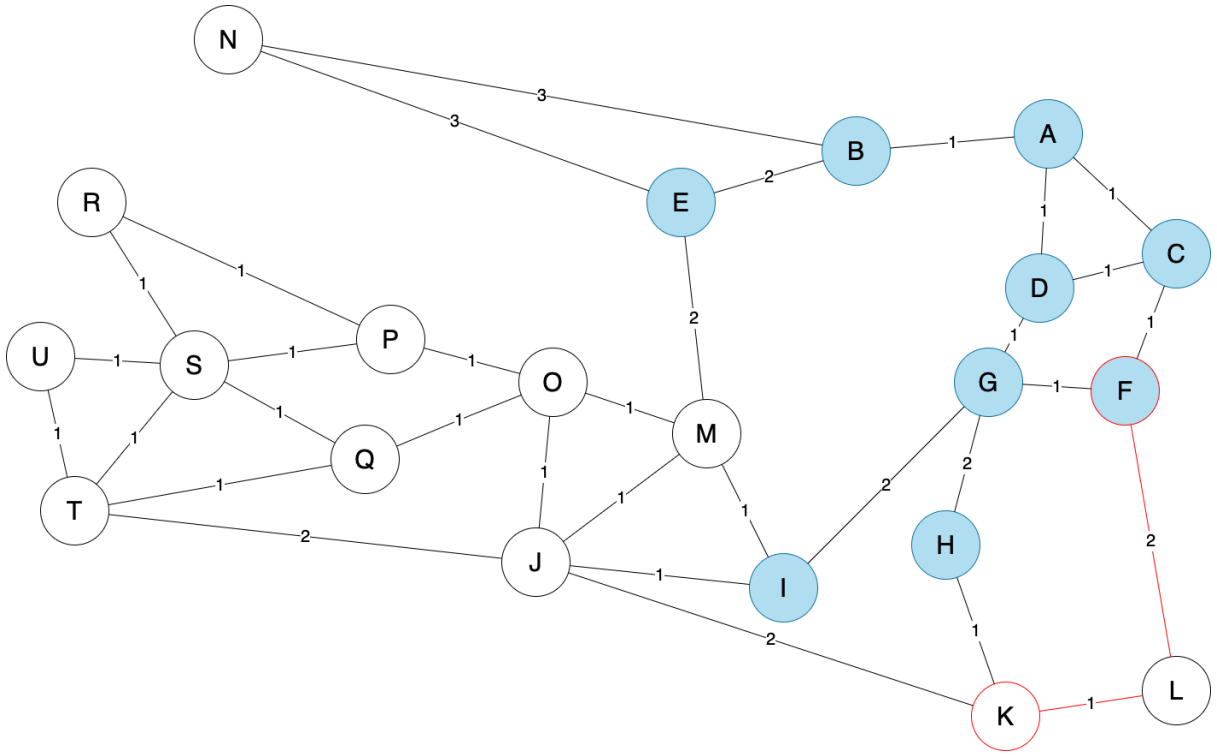
Columns I, L and N all contain the lowest weights in the table, being equal with 4. This means that they have equal priority, so node I can be visited next.



Node I has connections to M and J. It takes 1 minute to travel along the path from I to M, which is 5 minutes from A. This is the same as the weight in the table, so change is not required. It takes 1 minute to travel to J from I, which is 5 minutes from A. 5 is less than infinity, so a shorter path to J has been discovered.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
E	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
H	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
I	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A

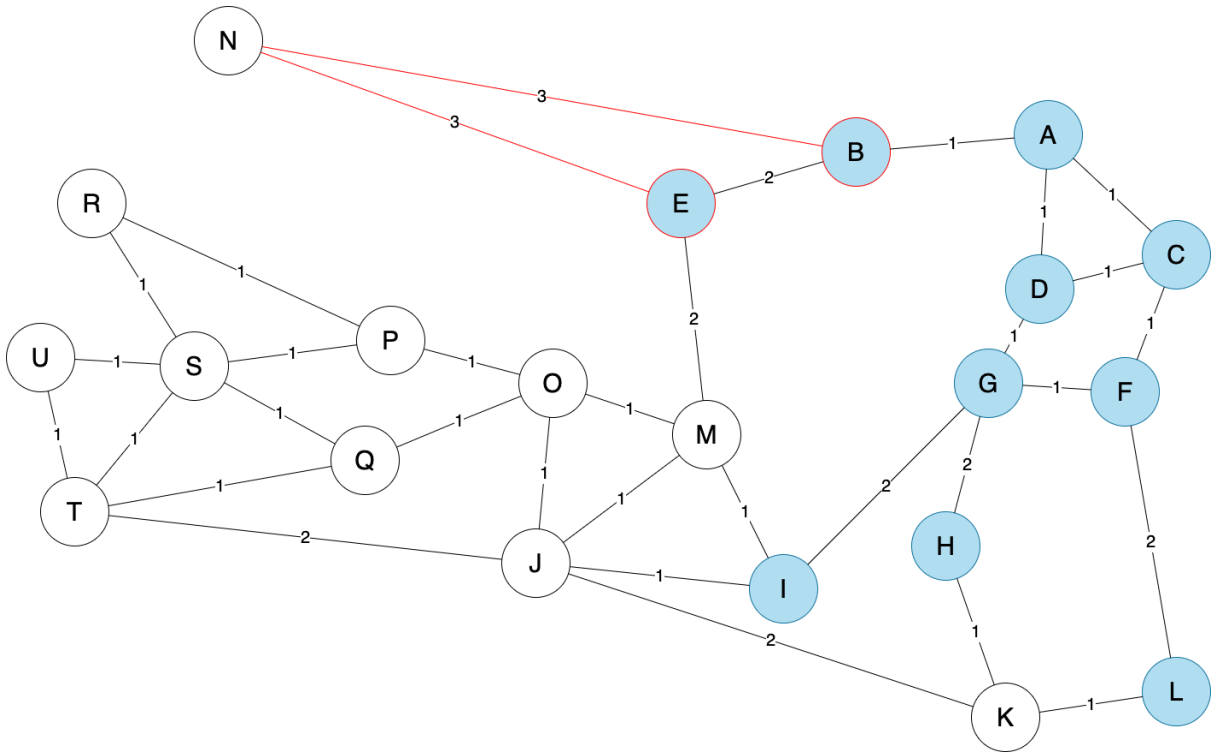
Columns L and N are equal lowest, so L can be visited next.



L has connections to F and K, the path between F and L has already been explored. It takes 1 minute to travel along the edge from L to K, which is 5 minutes from A. This is the same as the weight in the table, so change is not required.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
E	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
H	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
I	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
L	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A

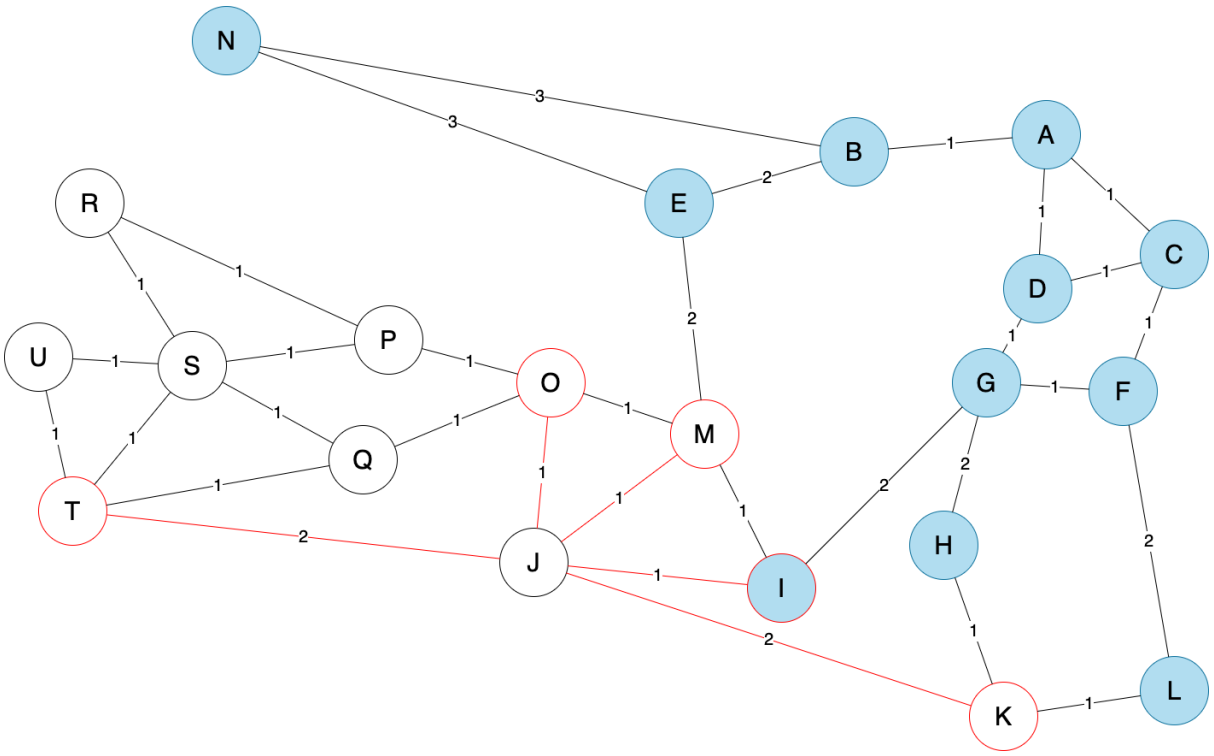
Column N contains the lowest value with 4, so it is visited next.



N has connections to E and B, which are both paths that have already been explored. Therefore, no further changes to the table are required.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
E	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
H	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
I	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
L	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
N	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A

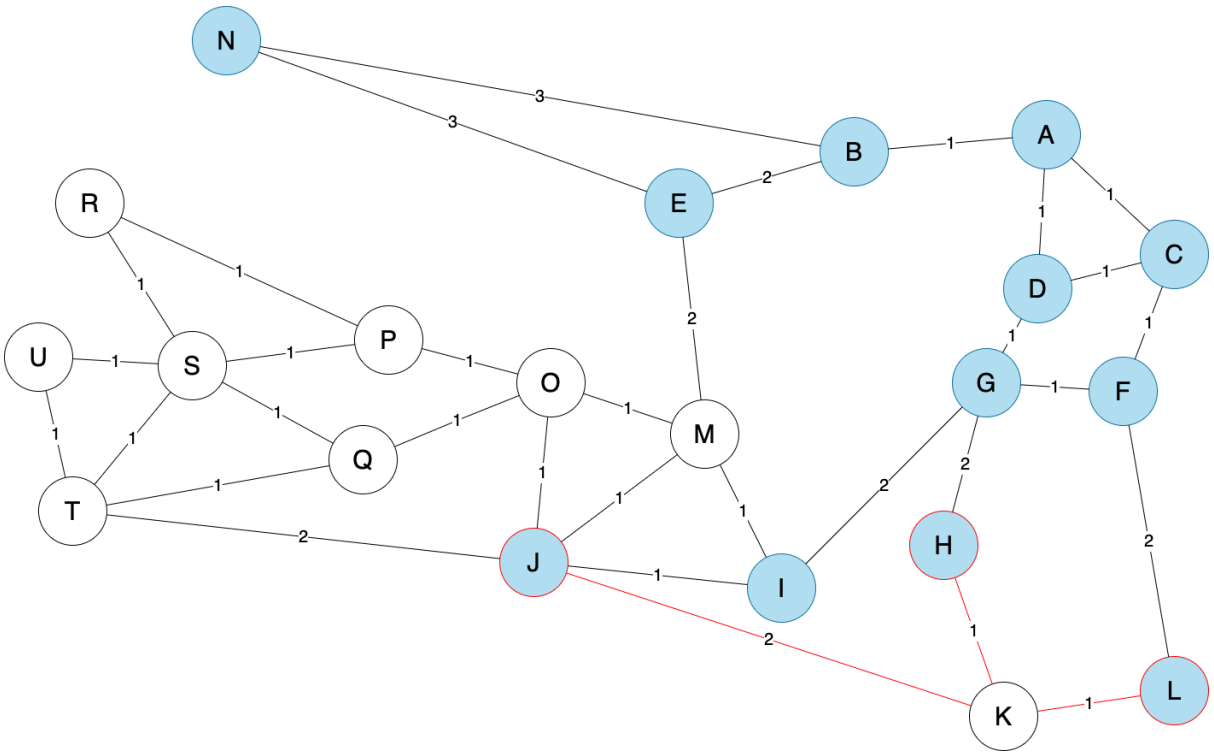
Columns J, K and M are equal with the lowest weights in the table, so node J can be visited next.



J has edges connecting to K, M, O and T. It takes 2 minutes to get to K from J, which is 7 minutes from A. 7 is greater than 5, so this path does not take priority. It takes 1 minute to get to M from J, which is 6 minutes from A. 6 is greater than 5, so this path does not take priority. It takes 1 minute to get to O from J, which is 6 minutes from A. 6 is less than infinity, so this path takes priority, resulting in a shorter path to O being discovered. It takes 2 minutes to get to T from J, which is 7 minutes from A. 7 is less than infinity, so this path takes priority, resulting in a shorter path to T being discovered.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
E	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
H	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
I	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
L	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
N	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
J	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A

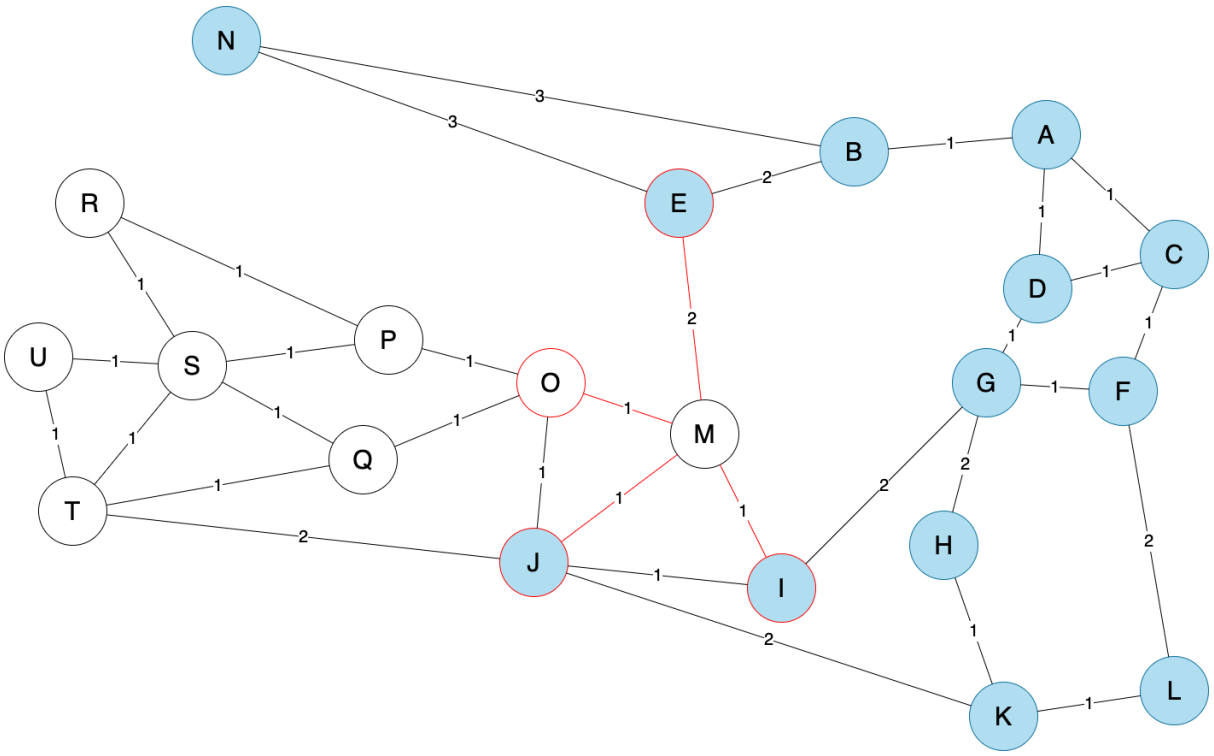
Columns K and M are equal with the lowest weights in the table, so node K can be visited next.



K has connections to J and L. It takes 2 minutes to get to J from K, which is 7 minutes from A. 7 is greater than 5, so it does not take priority and no changes are to be made. It takes 1 minute to get from K to L, which is 6 minutes from A. 6 is greater than 5, so it does not take priority and no changes are to be made.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
E	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
H	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
I	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
L	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
N	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
J	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
K	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A

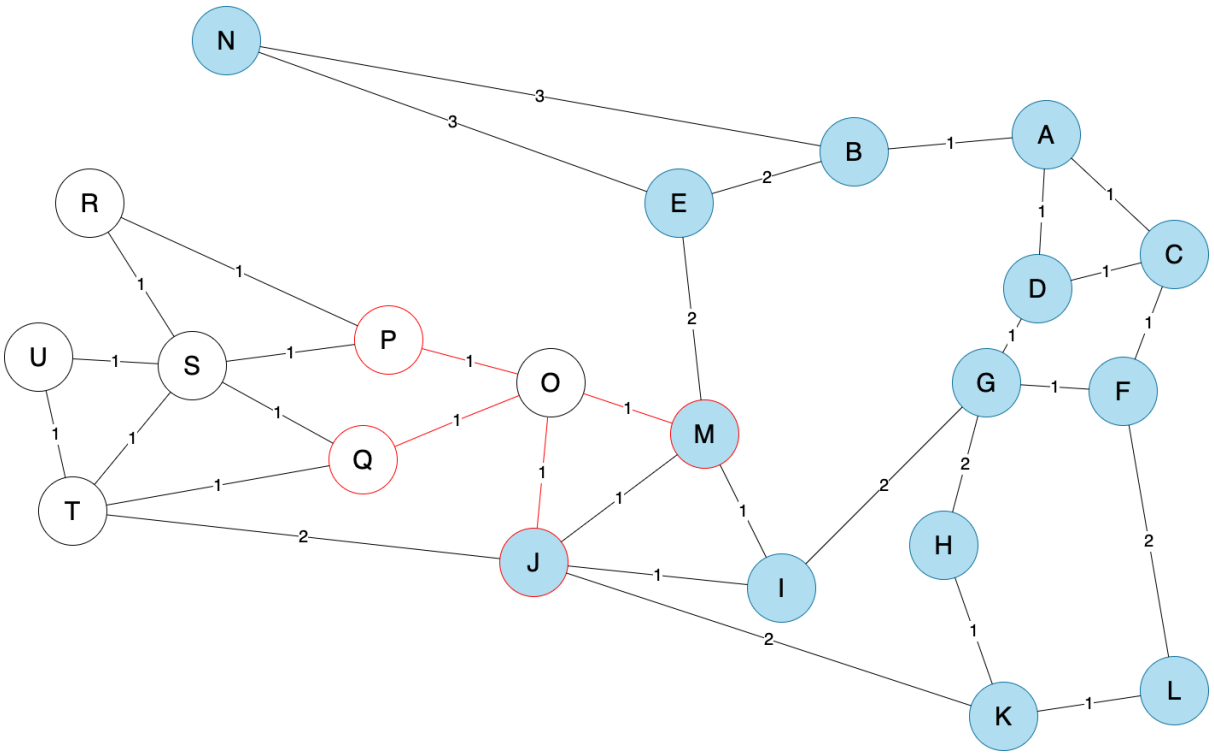
Column M has the lowest value in the table with 5, so it is visited next.



M has connections to I, J and O. It takes 1 minute to travel along the path from M to I, which is 6 minutes from A. 6 is greater than the existing weight of 4 to I, so no changes are made. It takes 1 minute to travel along the path from M to J, which is 6 minutes from A. 6 is greater than the existing weight of 5 to J, so no changes are made. It takes 1 minute to travel along the path from M to O, which is 6 minutes from A. 6 is the same as the existing weight at I, so no changes are made to the table.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
E	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
H	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
I	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
L	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
N	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
J	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
K	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
M	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A

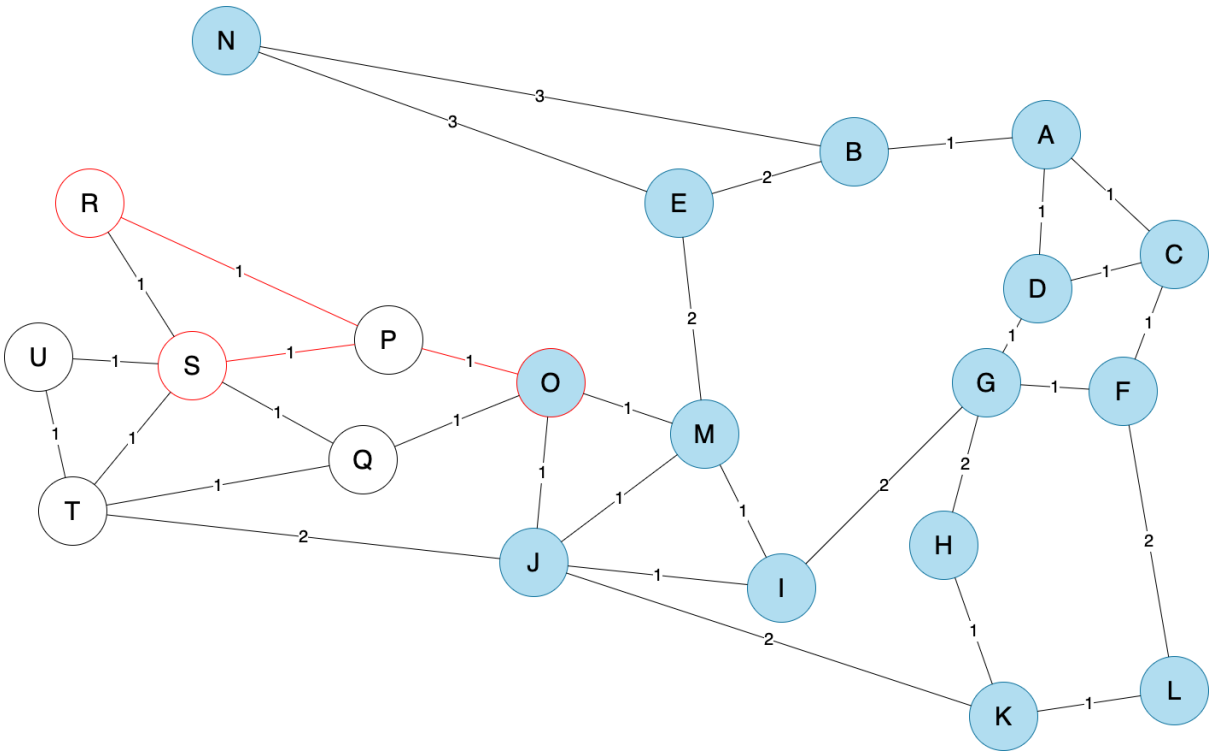
Column O has the lowest weight in the table, therefore node O is visited next.



O has connections to M, P and Q. It takes 1 minute to travel along the path from O to M, which is 7 minutes from A. 7 is greater than the existing weight of 5 to node M, so no changes are made. It takes 1 minute to travel along the path from O to P, which is 7 minutes from A. 7 is less than infinity, which means that a shorter path to P has been discovered. It takes 1 minute to travel along the path from O to Q, which is 7 minutes from A. 7 is less than infinity, which means that a shorter path to Q has been discovered.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
E	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
H	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
I	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
L	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
N	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
J	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
K	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
M	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
O	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	∞_A	∞_A	7_J	∞_A

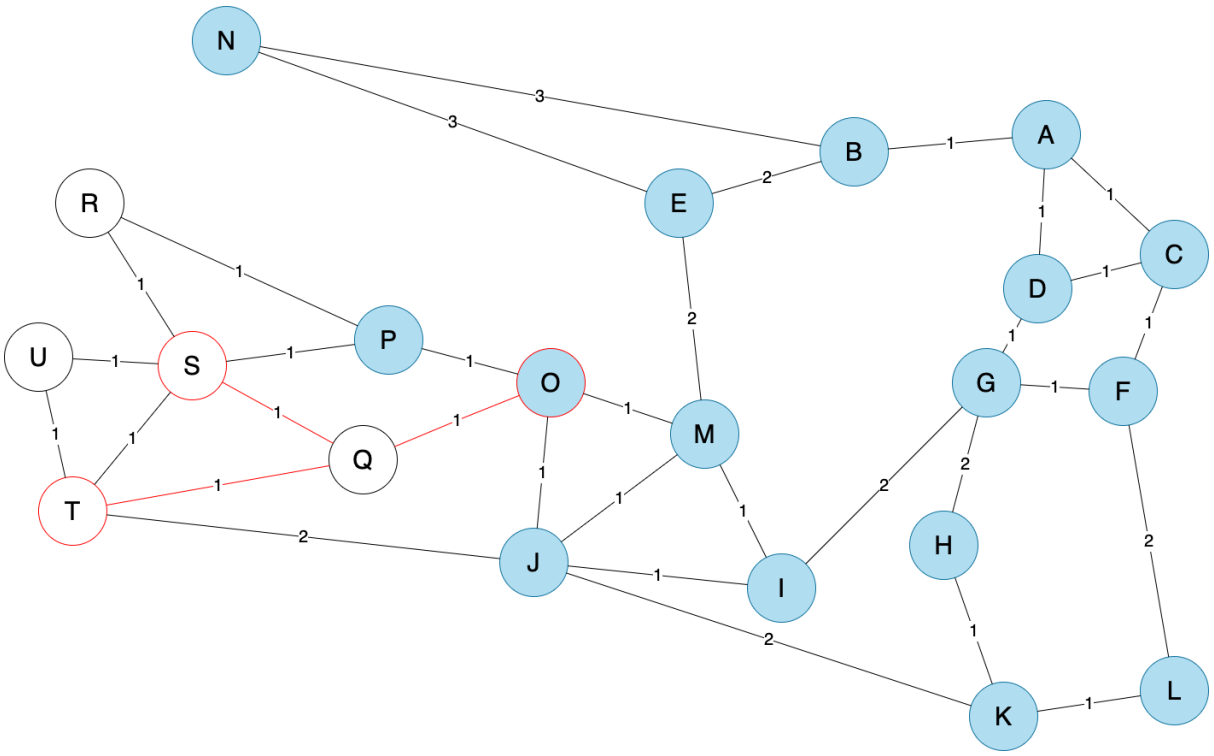
Columns P, Q and T are level as the lowest weights in the table. As there is no obvious node to visit next, P can be visited.



P has connections to R and S. It takes 1 minute to travel the path from P to R, which is 8 minutes from A. 8 is less than infinity, therefore a shorter path to R has been discovered. It takes 1 minute to travel the path from P to S, which is 8 minutes from A. 8 is less than infinity, therefore a shorter path to S has been discovered.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
E	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
H	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
I	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
L	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
N	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
J	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
K	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
M	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
O	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	∞_A	∞_A	7_J	∞_A
P	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	∞_A

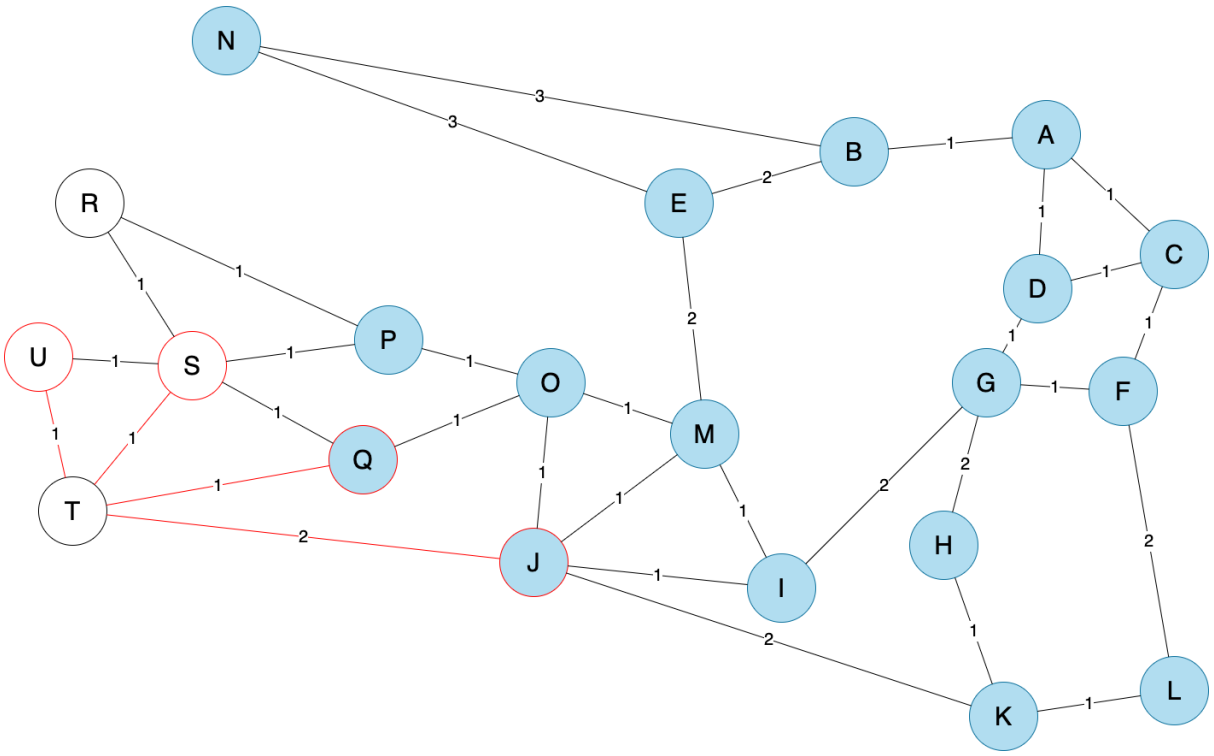
Columns Q and T both have the lowest values in the table, so Q can be visited next.



Node Q has paths to S and T. It takes 1 minute to travel the path from Q to S, which is 8 minutes from A. 8 is the same as the weight that is already in the table for S, so no change is required. It takes 1 minute to travel the path from Q to T, which is 8 minutes from A. 8 is greater than 7, so a shorter path to T is not discovered and the weight remains at 7.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
E	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
H	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
I	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
L	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
N	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
J	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
K	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
M	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
O	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	∞_A	∞_A	7_J	∞_A
P	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	∞_A
Q	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	∞_A

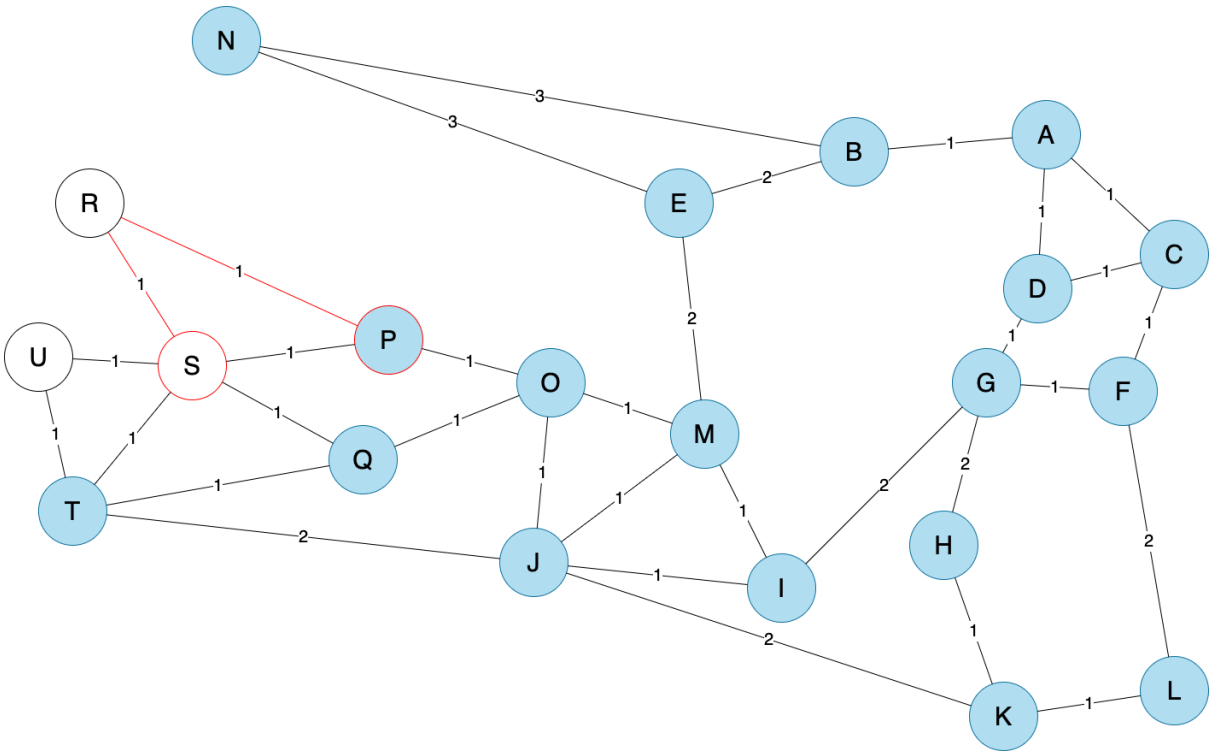
Column T contains the lowest weight with 7, so node T is visited next.



T has connections to Q, S and U. It takes 1 minute to travel to Q from T, which is 8 minutes from A. 8 is greater than 7, therefore a shorter path to Q has not been discovered. It takes 1 minute to travel from T to S, which is 8 minutes from A. 8 is equal to the value at S, so no change is required. It takes 1 minute to travel from T to U, which is 8 minutes from. 8 is less than infinity - a shorter path to U has been discovered.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
E	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
H	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
I	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
L	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
N	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
J	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
K	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
M	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
O	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	∞_A	∞_A	7_J	∞_A
P	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	∞_A
Q	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	∞_A
T	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	8_T

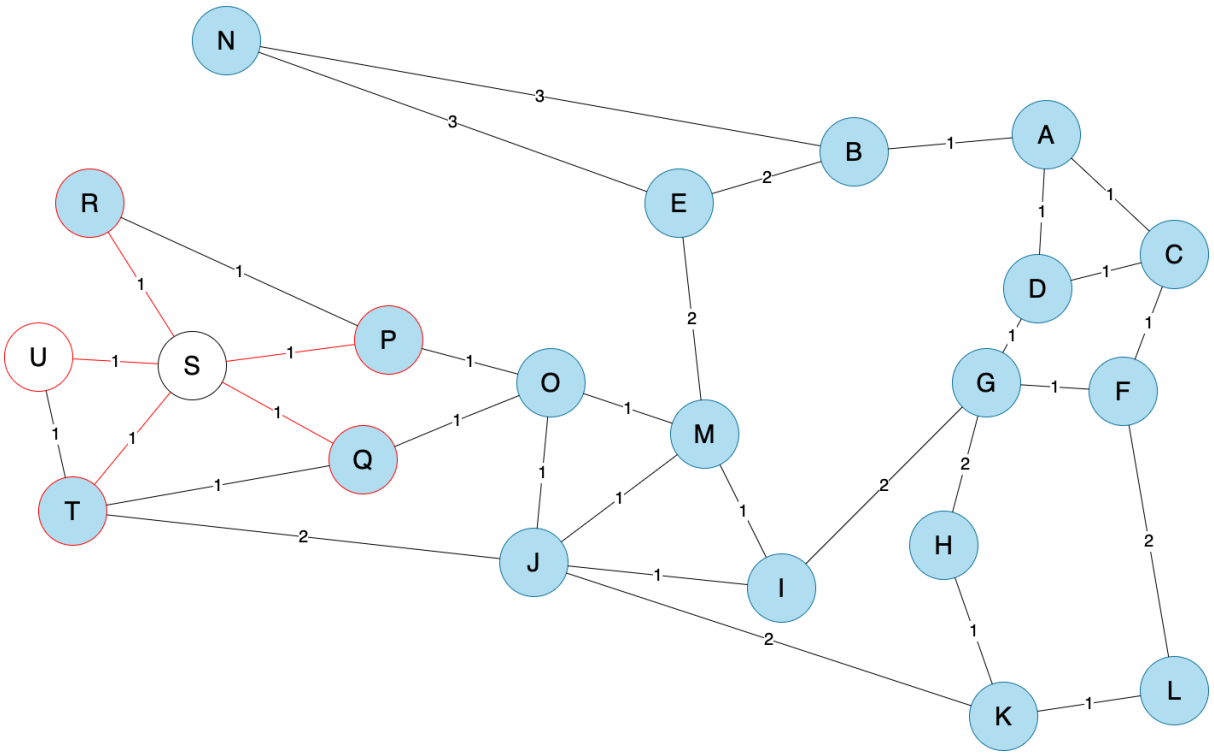
Columns R, S and U have equal values of 8, so R can be visited next.



Node R has a connection to S. It takes 1 minute to go to S from R, which is 9 minutes from A. 9 is greater than the current value of S, this means that a shorter path to S has not been discovered, and the value is to remain at 8.

Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
E	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
H	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
I	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
L	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
N	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
J	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
K	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
M	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
O	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	∞_A	∞_A	7_J	∞_A
P	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	∞_A
Q	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	∞_A
T	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	8_T
R	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	8_T

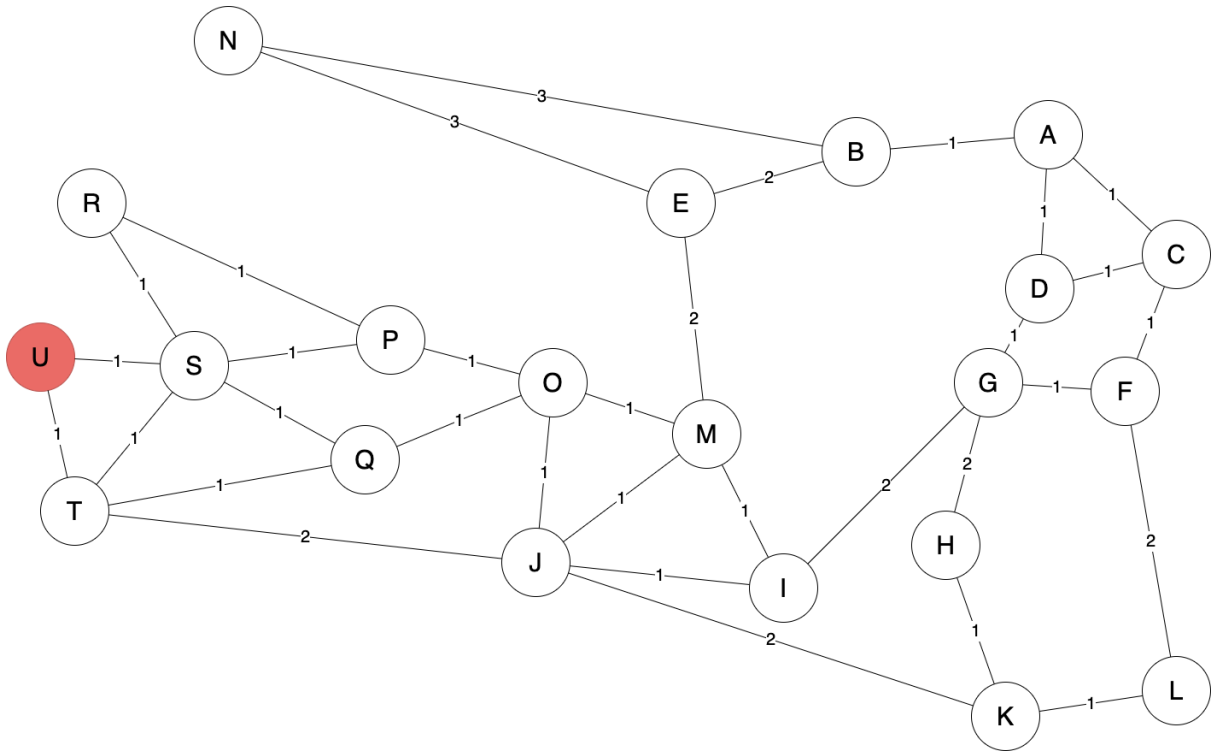
Columns S and U both have values of 8, so S can be visited next.



S has connections to Q, R, T and U. It takes 1 minute to go to Q from S, which is 9 minutes from A. 9 is greater than the current value of Q, this means that a shorter path to Q has not been discovered, and the value is to remain at 7. It takes 1 minute to go to R from S, which is 9 minutes from A. 9 is greater than the current value of R, this means that a shorter path to R has not been discovered, and the value is to remain at 8. It takes 1 minute to go to T from S, which is 9 minutes from A. 9 is greater than the current value of T, this means that a shorter path to T has not been discovered, and the value is to remain at 7. It takes 1 minute to go to U from S, which is 9 minutes from A. 9 is greater than the current value of U, this means that a shorter path to U has not been discovered, and the value is to remain at 8.

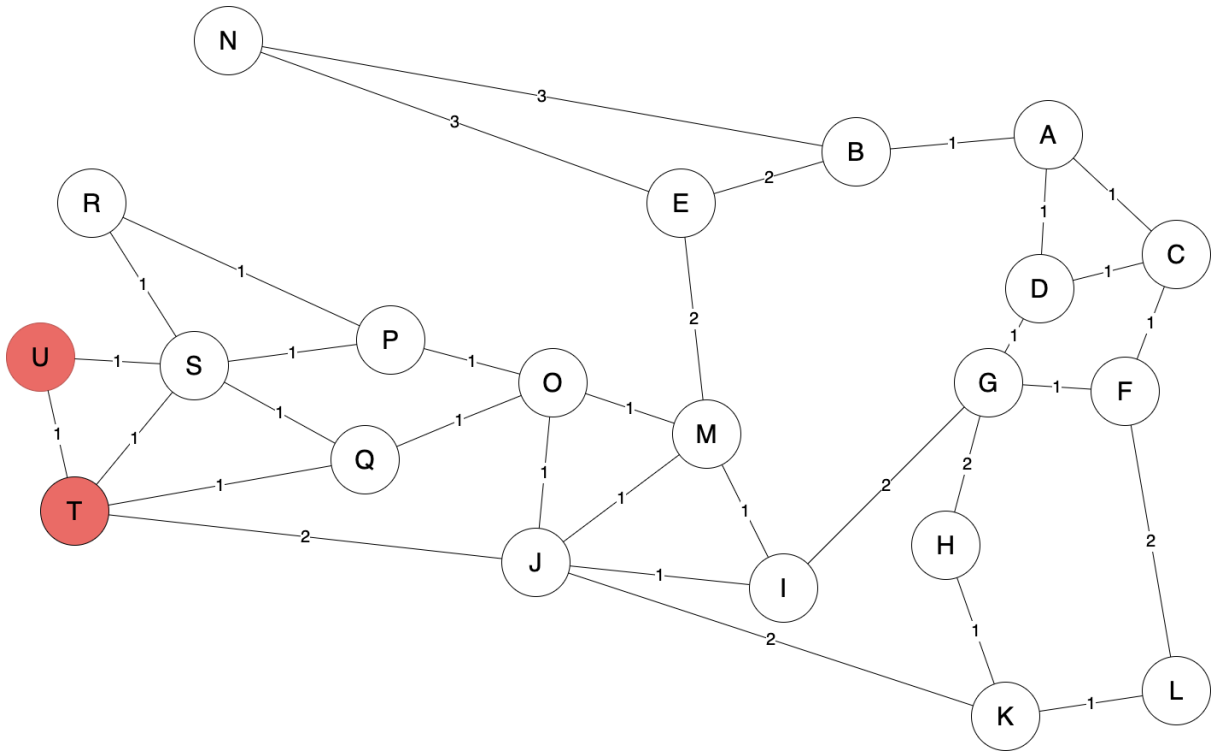
Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	0_A	1_A	1_A	1_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
B	0_A	1_A	1_A	1_A	3_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
C	0_A	1_A	1_A	1_A	3_B	2_C	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
D	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
F	0_A	1_A	1_A	1_A	3_B	2_C	2_D	∞_A	∞_A	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
G	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	∞_A	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
E	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	∞_A	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
H	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	∞_A	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
I	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
L	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
N	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
J	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
K	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
M	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	∞_A	∞_A	∞_A	∞_A	7_J	∞_A
O	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	∞_A	∞_A	7_J	∞_A
P	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	∞_A
Q	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	∞_A
T	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	8_T
R	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	8_T
S	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	8_T

The table is now complete, as U is the last node, all edges must have been explored now. Using the bottom row of the table, the shortest path from node A to node U can be determined by working backwards. Furthermore, it can be noted that the total cost of the path will be 8 minutes.



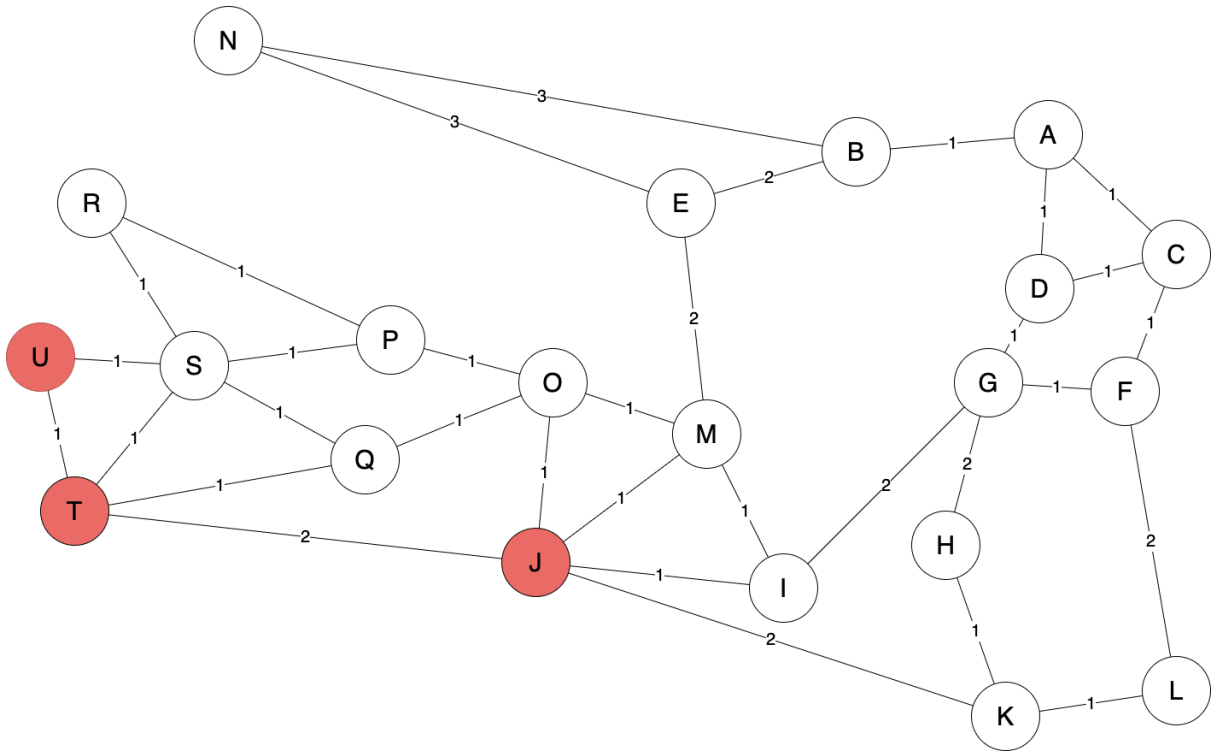
Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
S	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	8_T

The subscripted letter next to 8 is T, therefore the node visited before U is T.



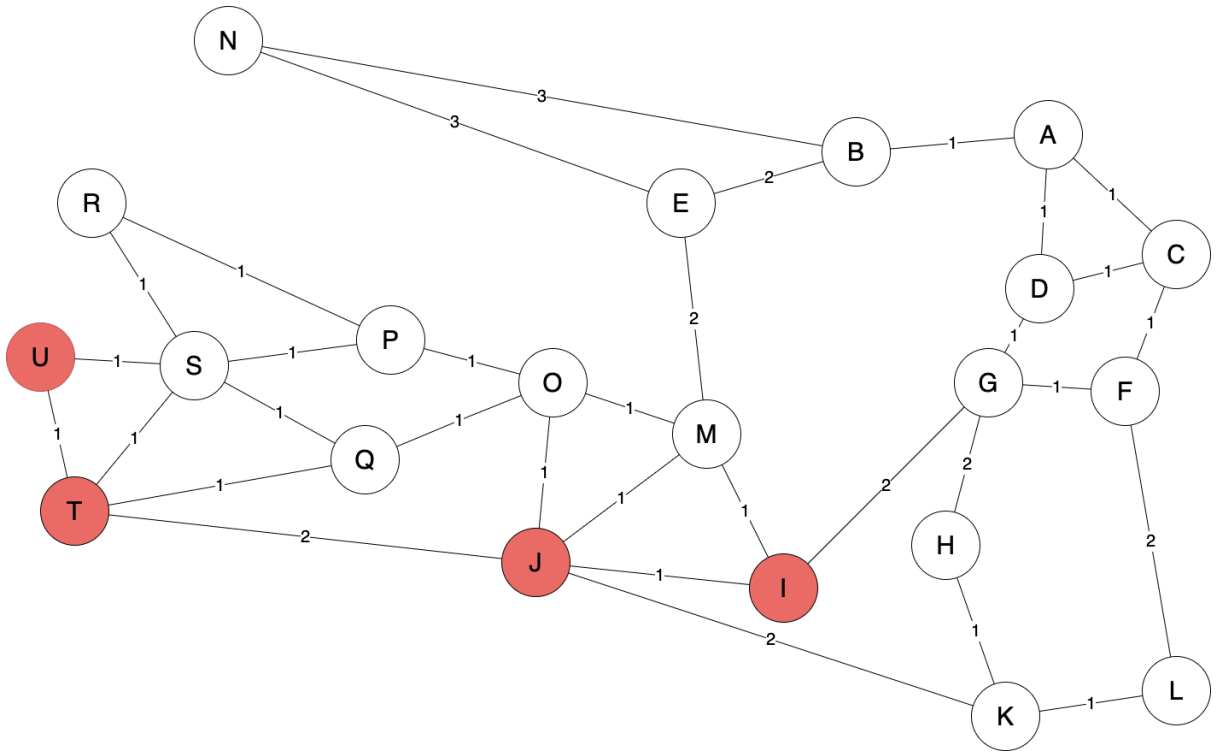
Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
S	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	8_T

The subscripted letter next to 7 is J, therefore the node visited before T is J.



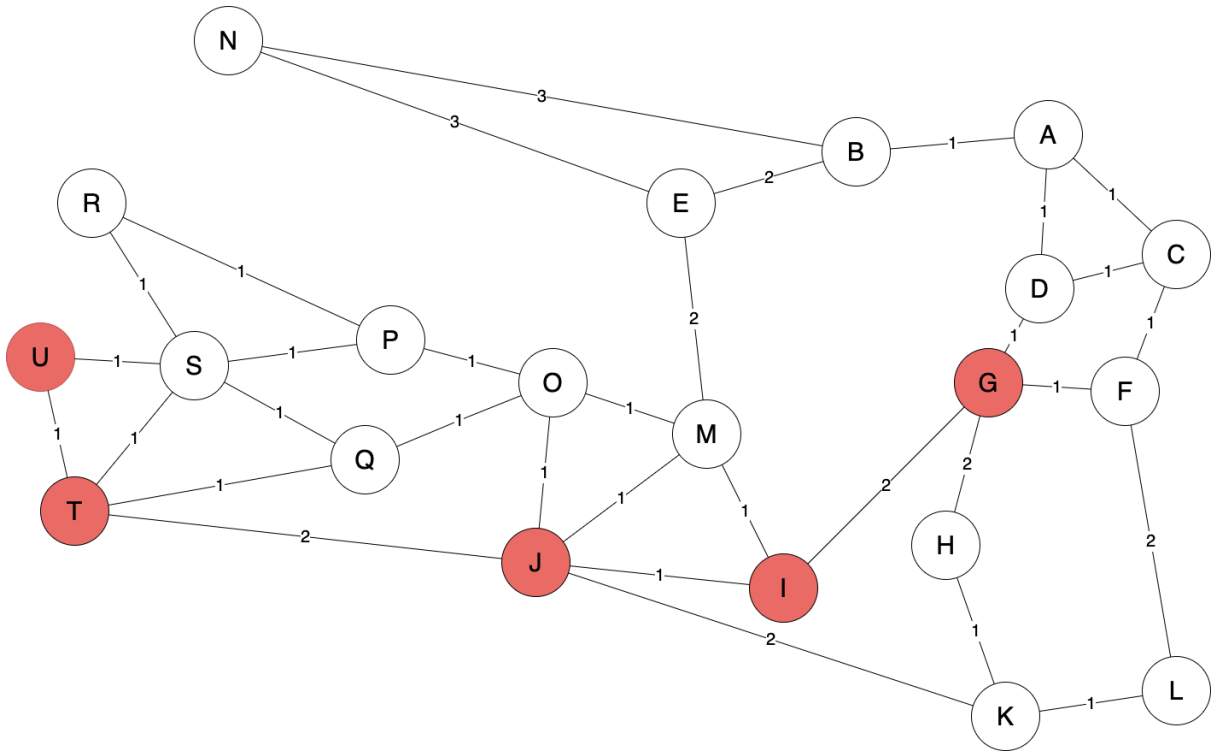
Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
S	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	8_T

The subscripted letter next to 5 is I, therefore the node visited before J is I.



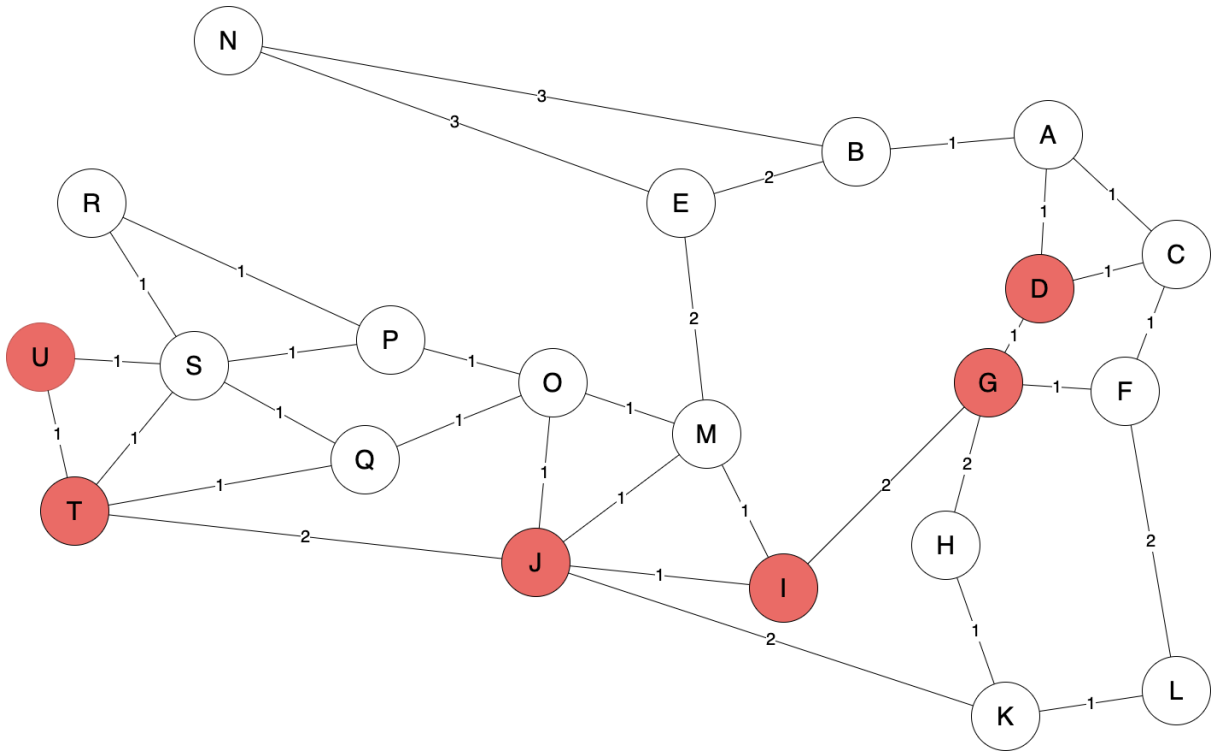
Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
S	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	8_T

The subscripted letter next to 4 is G, therefore the node visited before I is G.



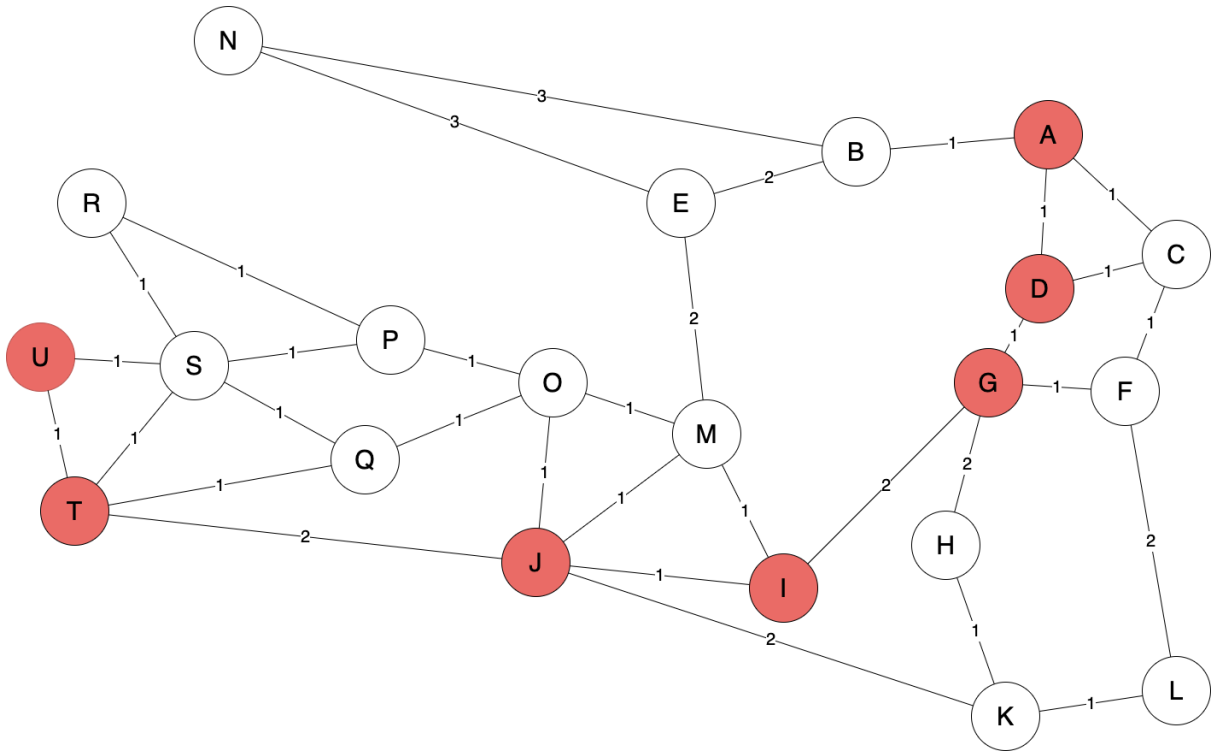
Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
S	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	8_T

The subscripted letter next to 2 is D, therefore the node visited before G is D.



Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
S	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	8_T

The subscripted letter next to 1 is A, therefore the node visited before D is A.



Node	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
S	0_A	1_A	1_A	1_A	3_B	2_C	2_D	4_G	4_G	5_I	5_H	4_F	5_E	4_B	6_J	7_O	7_O	8_P	8_P	7_J	8_T

The path is complete, the shortest path to go from A to U is: A D G I J T U, and will take 8 minutes.

Database Design

Creating a normalised database

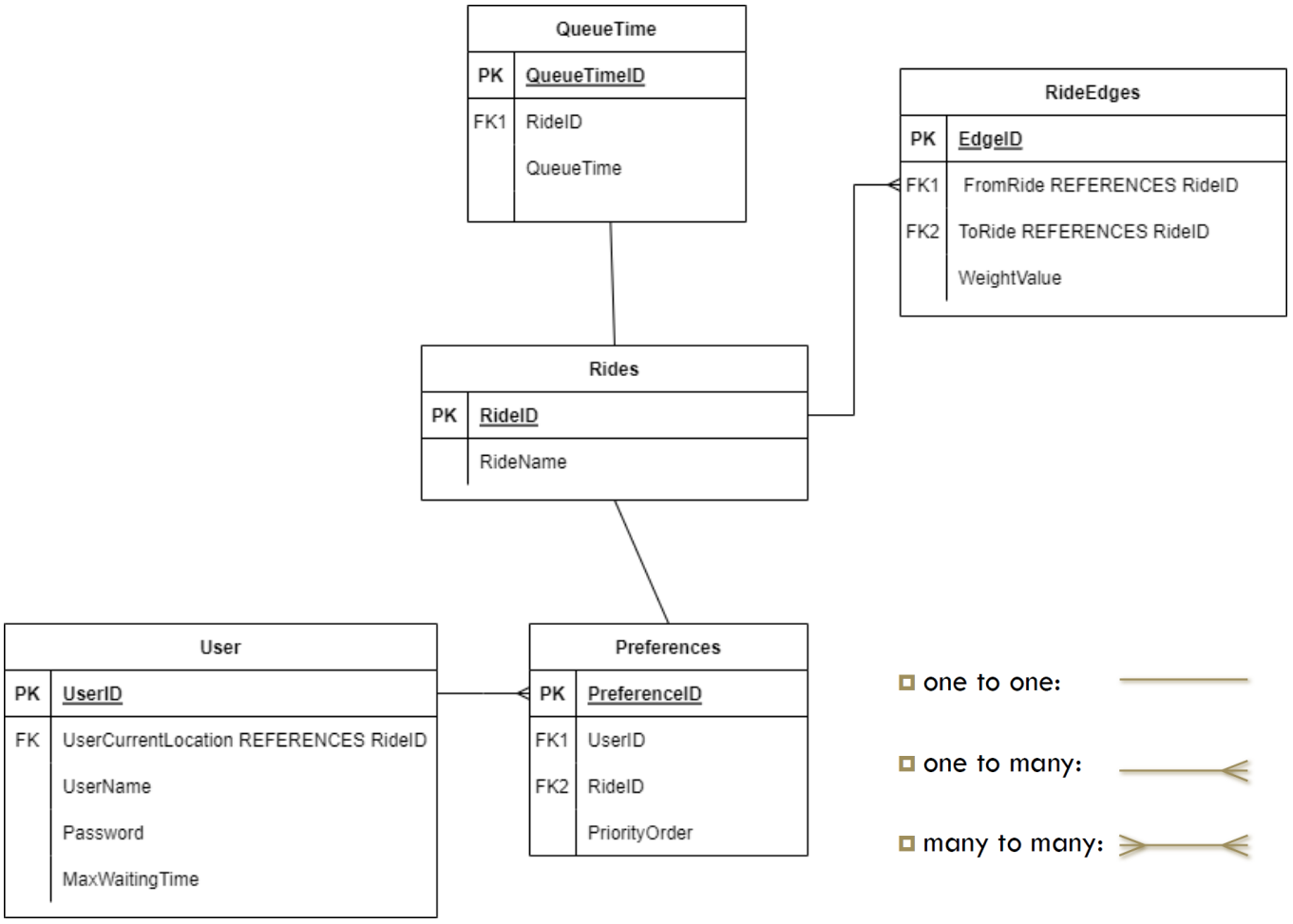
Database normalisation is the process of organising a database in a way that reduces redundancy and dependency. Normalisation typically involves dividing a database into tables and defining entity relationships between the tables. This enables data to be isolated so that additions, deletions, and modifications of a field can be made in just one table and then spread to the rest of the database through the relationships.

There are three normal forms used to normalise a database:

1. First Normal Form
    - Data must be atomic: each attribute containing a single value, not multiple
    - No repeating groups of attributes
2. Second Normal Form
    - No partial key dependencies: all non-key attributes in a table must be fully dependent on the primary key
3. Third Normal Form
    - No non-key dependencies: no non-key attribute is dependent on another non-key attribute

The database that I will create must be normalised with third normal form. In order to achieve third normal form, a relation must also follow second normal form and first normal form. This means that the relation must meet the requirements of both second normal form and first normal form in addition to the requirements of third normal form.

Entity-Relationship Diagram



Rides (RideID, RideName)

Field	Type	Description
RideID	String	Unique identifier for each ride
RideName	String	Name of the ride

RideEdges(EdgeID, FromRide, ToRide, WeightValue)

Field	Type	Description
EdgeID	Integer	Unique identifier for each edge
FromRide	String	Identifier for the first ride in the edge (refers to RideID in Rides table)
ToRide	String	Identifier for the second ride in the edge (refers to RideID in Rides table)
WeightValue	Integer	Weight of the edge between the two rides

Users (UserID, UserName, Password, UserCurrentLocation, MaxWaitingTime)

Field	Type	Description
UserID	Integer	Unique identifier for each user
UserName	String	Username of the user
Password	String	Password for the user
UserCurrentLocation	String	Identifier of the ride where the user is currently located (refers to RideID in Rides table)
MaxWaitingTime	Integer	Users maximum waiting time, which is common for all rides

Preferences (PreferenceID, UserID, RideID, PriorityOrder)

Field	Type	Description
PreferenceID	Integer	Unique identifier for each preference
UserID	Integer	Identifier for the user who has the preference (refers to UserID in Users table)
RideID	String	Identifier for the ride that the preference applies to (refers to RideID in Rides table)
PriorityOrder	Integer	Order of priority for the preference

QueueTimes (QueueTimeID, RideID, QueueTime)

Field	Type	Description
QueueTimeID	Integer	Unique identifier for each queue time
RideID	String	Identifier for the ride (refers to RideID in Rides table)
QueueTime	Integer	Current waiting time for the ride

The Rides table and the RideEdges table are connected by a one-to-many relationship, since a ride can have multiple edges, but each edge is associated with a single ride. The primary key of the Rides table (RideID) is used as a foreign key in the RideEdges table (FromRide and ToRide), which establishes the one-to-many relationship between the two tables.

The Users table and the Preferences table are connected by a one-to-many relationship, since a user can have multiple preferences, but each preference is associated with a single user. The primary key of the Users table (UserID) is used as a foreign key in the Preferences table (UserID), which establishes the one-to-many relationship between the two tables.

The Rides table and the QueueTime table are connected by a one-to-one relationship, since a ride can have only one queue time, and each queue time is associated with a single ride. The primary key of the Rides table (RideID) is used as a foreign key in the QueueTime table (RideID), which establishes the one-to-one relationship between the two tables.

DDL

```
Create table Rides
(
RideID VARCHAR(255) PRIMARY KEY NOT NULL,
RideName VARCHAR(255)
)
```

Table name: Rides WITHOUT ROWID

	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate
1	RideID	VARCHAR (255)	🔑				🔒	
2	RideName	VARCHAR (255)						

RideID	RideName
A	Entrance
B	Depth Charge
C	Vortex
D	Quantum
E	Flying fish
F	Zodiac
G	Rush
H	Colossus
I	The Walking Dead: The Ride
J	Ghost Train
K	Samurai
L	Saw - The Ride
M	Storm Surge
N	The Swarm
O	Tidal Wave
P	Dodgems
Q	Detonator
R	Stealth
S	Storm in a Tea Cup
T	Nemesis Inferno
U	Rumba Rapids

```
Create table RideEdges
(
EdgeID INTEGER PRIMARY KEY NOT NULL,
FromRide VARCHAR(255) REFERENCES Rides(RideID),
ToRide VARCHAR(255) REFERENCES Rides(RideID),
WeightValue INTEGER
)
```

Table name: RideEdges WITHOUT ROWID

	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate
1	EdgeID	INTEGER						
2	FromRide	VARCHAR (255)						
3	ToRide	VARCHAR (255)						
4	WeightValue	INTEGER						

	A	B	C	D
1	EdgeId	FromRide	ToRide	WeightValu
2	1	A	C	1
3	2	C	A	1
4	3	A	D	1
5	4	D	A	1
6	5	C	D	1
7	6	D	C	1
8	7	B	E	2
9	8	E	B	2
10	9	C	F	1
11	10	F	C	1
12	11	G	D	1
13	12	D	G	1
14	13	G	H	2
15	14	H	G	2
16	15	F	G	1
17	16	G	F	1
18	17	G	I	2
19	18	I	G	2
20	19	T	J	2
21	20	J	T	2
22	21	J	K	2
23	22	K	J	2
24	23	H	K	1
25	24	K	H	1
26	25	K	L	1
27	26	L	K	1
28	27	F	L	2
29	28	L	F	2
30	29	I	M	1
31	30	M	I	1
32	31	E	M	2
33	32	M	E	2
34	33	E	N	3
35	34	N	E	3
36	35	B	N	3
37	36	N	B	3
38	37	J	O	1
39	38	O	J	1
40	39	J	M	1
41	40	M	J	1
42	41	M	O	1

38	O	J	1
39	J	M	1
40	M	J	1
41	M	O	1
42	O	M	1
43	T	Q	1
44	Q	T	1
45	T	S	1
46	S	T	1
47	T	U	1
48	U	T	1
49	U	S	1
50	S	U	1
51	S	R	1
52	R	S	1
53	S	P	1
54	P	S	1
55	S	Q	1
56	Q	S	1
57	R	P	1
58	P	R	1
59	O	P	1
60	P	O	1
61	O	Q	1
62	Q	O	1
63	A	B	1
64	B	A	1
65	I	J	1
66	J	I	1

Data is read from the csv file into the database

Create table User
(
UserID INTEGER PRIMARY KEY NOT NULL,
Username VARCHAR(255),
Password VARCHAR(255)
UserCurrentLocation VARCHAR(255) REFERENCES Rides(RideID),
MaxWaitingTime INTEGER
)

Table name: User ☐ WITHOUT ROWID

	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate
1	UserID	INTEGER	🔑				✓	
2	Username	VARCHAR (255)						
3	Password	VARCHAR (255)						
4	UserCurrentLocation	VARCHAR (255)		🔗				
5	MaxWaitingTime	INTEGER						

Create table Preferences
(
PreferenceID INTEGER PRIMARY KEY NOT NULL,
UserID INTEGER REFERENCES User(UserID),
RideID VARCHAR(255) REFERENCES Rides(RideID),
PriorityOrder INTEGER,
)

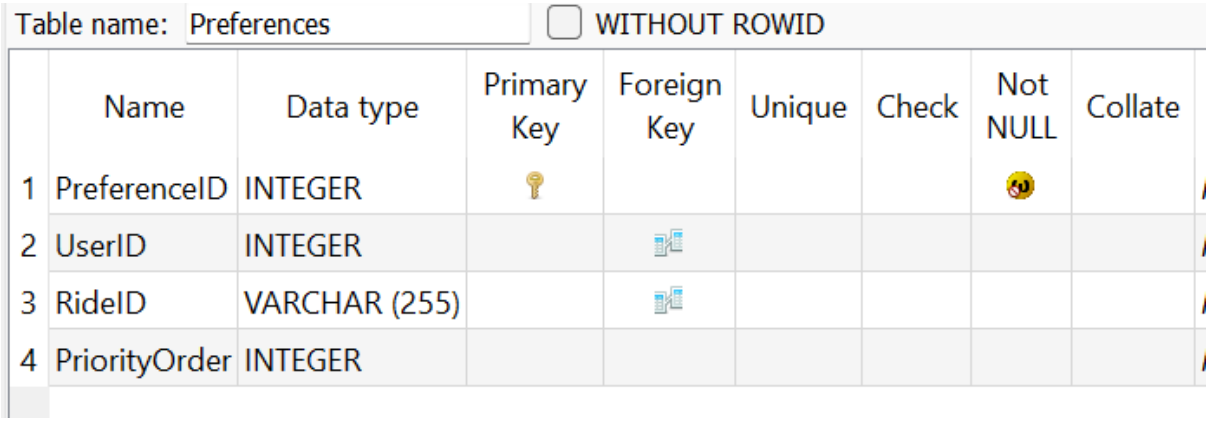


Table name: Preferences WITHOUT ROWID

	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate
1	PreferenceID	INTEGER						
2	UserID	INTEGER						
3	RideID	VARCHAR (255)						
4	PriorityOrder	INTEGER						

Create table QueueTimes
(
QueueTimeID INTEGER PRIMARY KEY NOT NULL,
RideID VARCHAR(255) REFERENCES Rides(RideID),
QueueTime INTEGER
)

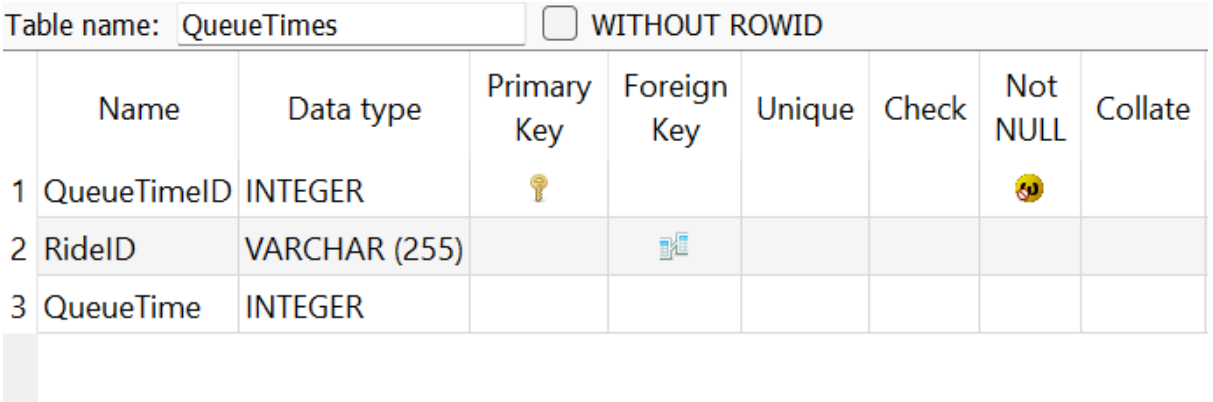


Table name: QueueTimes WITHOUT ROWID

	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate
1	QueueTimeID	INTEGER						
2	RideID	VARCHAR (255)						
3	QueueTime	INTEGER						

User Interface

For the Thorpe Park Navigator, I have opted for a command line user interface as opposed to a graphical user interface. A command line interface for a Thorpe Park Navigator would be suitable for the prospective users identified in the Analysis section of development for the following reasons:

- **Teens and young adults:** Teens and young adults are likely to be more comfortable with using a command line interface as they are more familiar with technology and may prefer a more straightforward interface.

- **People who are very busy and do not want to waste time trying to find rides to go on:** A command line interface allows users to quickly input their desired information and receive the shortest path without the need for navigating through menus or clicking through multiple screens. This can save time for busy users who do not want to waste time using a more complex interface.

- **People who want to make full use of their ticket and go on as many rides as possible:** A command line interface allows users to quickly input their desired information and receive the shortest path, which can help them make the most out of their ticket by maximising the number of rides they are able to go on.

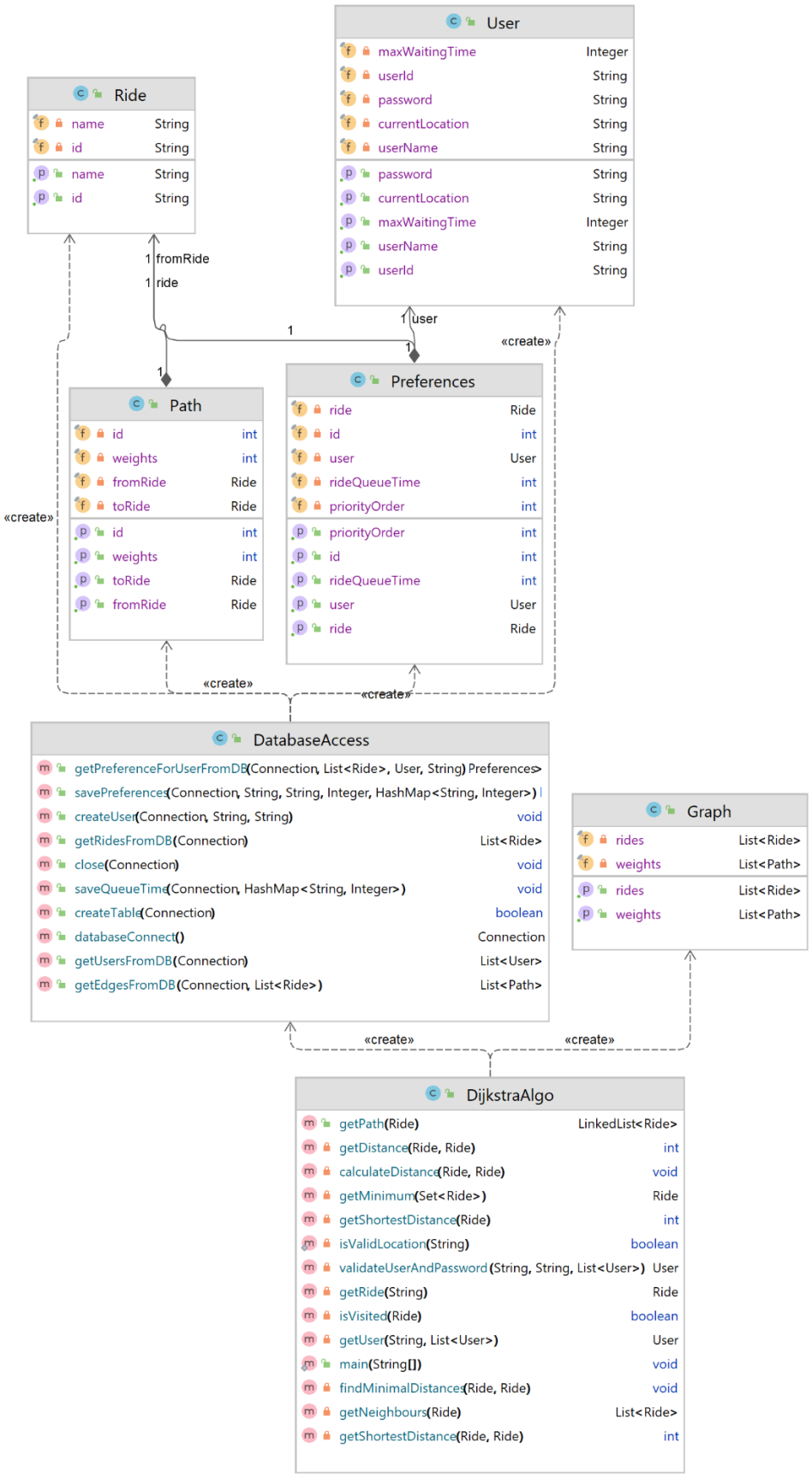
- **People going to Thorpe Park for the first time:** A command line interface can be easier to use for people visiting Thorpe Park for the first time as it allows them to quickly input their desired information and receive the shortest path without the need to navigate through unfamiliar menus or screens. This can help them make the most out of their visit and avoid wasting time trying to find rides.

Overall, a command line interface for a Thorpe Park Navigator would be suitable for the identified prospective users as it allows for quick and efficient input and output of information, making it easier for users to navigate the park and make the most out of their visit.

Example

```
 Welcome to Thorpe Park Navigator

Login / Account Registration

Enter your exisitng / desired username:

Enter your existing / desired password:


  Ride Name                      Ride Key
Entrance                           A
Depth Charge                       B
Vortex                             C
Quantum                            D
Flying fish                        E
Zodiac                             F
Rush                               G
Colossus                           H
The Walking Dead: The Ride         I
Ghost Train                        J
Samurai                            K
SAW - The Ride                     L
Storm Surge                        M
THE SWARM                          N
Tidal Wave                         O
Dodgems                            P
Detonator                          Q
Stealth                            R
Storm in a Tea Cup                 S
Nemesis Inferno                    T
Rumba Rapids                       U

Using the key above, enter where in the park you currently are:
```

Class Diagram



Subroutines used

Subroutines Used	Description
Class DatabaseAccess	
getPreferenceForUserFromDB	Queries the Preferences table and QueueTimes table, returns back a list of preferences objects
savePreferences	Saves the preferences entered by the user into the Preferences table
createUser	Creates a record in the User table
getRidesFromDB	Queries the Rides table, returning back a list of rides
close	Closes database connection
saveQueueTime	Saves the queue times entered by the user into the Queue Times table
createTable	Creates all the tables that are necessary for the program, inserts the list of rides into the Rides table, insert list of edges and weights from a csv file into the RideEdges table
databaseConnect	Creates database connection
getUsersFromDB	Queries the User table, returning back a list of users
getEdgesFromDB	Queries the RideEdges table, returning back a list of edges and their weights
Class DijkstraAlgo	
calculateDistance	Calculates distance from source ride to destination ride
findMinimalDistances	Finds the shortest path to get to destination ride from a given source ride
getDistance	Returns the cost of the shortest path
isVisited	Checks whether the destination node (ride) has been visited
getMinimum	Checks for the nearest unvisited ride
getNeighbours	Returns list of unvisited neighbouring rides
getShortestDistance(destination)	Returns weight of a ride
getPath	Calculates path to destination, checking for existing paths
getShortestDistance(source, destination)	Returns cost of journey from source to destination

validateUserAndPassword	Checks if username and password match
getUser	Returns back the user object
getRide	Returns back the ride object
isValidLocation	Checks if user input for a location is valid (single letter)
Class Graph	
Graph	Creates graph using rides and weights
getWeights	Returns a list of weights
getRides	Returns a list of rides
Class Path	Create a path object, containing source ride, destination ride, and the weight between the two
Path	
getId	
getDistance	
getToRide	
getFromRide	
Class Preferences	Creates a preference object, containing user details, a ride, its priority as a preferences and its queue time
Preferences	
getId	
getSkipAfterMins	
getRide	
getUser	
getRideWaitTime	
getRideOrder	
Class Ride	Creates a ride object, containing rideID and ride name
Ride	
getId	
getName	

Class User	Creates a user object, containing UserId, username, password, user's current location, and user's maximum wait time
User	
getCurrentLocation	
getMaxWaitingTime	
getUserName	
getPassword	
getUserId	
Class LinkedList	
add	Adds destination nodes to end of the list
get	Returns the node at a particular index
items	Returns an array list of all the nodes

Justification of programming language (Java)

Java is a widely used programming language with a large developer community and a wealth of libraries and frameworks that make it an attractive choice for many different types of projects. It is platform independent, which means that you can write code on one platform and then run it on another without needing to make any changes. Additionally, it is a powerful and efficient language that can be used to build a wide range of applications.

Technical

```
public class DatabaseAccess {

    public Connection databaseConnect() {
        Connection conn = null;
        try {
            Class.forName("org.sqlite.JDBC");//Specifying the SQLite Java
driver
            conn =
DriverManager.getConnection("jdbc:sqlite:C://Users//adima//Downloads//TP13
012023//ThorpeParkNavigator//ThorpeParkNavigatorDatabase.db");//Specify
the database, since relative in the main project folder
            conn.setAutoCommit(false);// Controls when data is written
            System.out.println("\nOpened database successfully\n");
        } catch (SQLException e) {
            System.out.println(e.getClass().getName() + ": " +
e.getMessage());
            System.exit(0);
        } catch (ClassNotFoundException ex) {

Logger.getLogger(DatabaseAccess.class.getName()).log(Level.SEVERE, null,
ex);
        }
        return conn;
    }

    public void close(Connection conn) {
        try {
            conn.close();
        } catch (SQLException ex) {
            Logger.getLogger(Connection.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }

    public void createUser(Connection conn, String userName, String
password) {
        String insertsql = "INSERT INTO User (UserID, Username, Password,"
                + " UserCurrentLocation , MaxWaitingTime ) "
                + "VALUES (?,? ,? , ?, ?);";
        try {
            PreparedStatement ps = conn.prepareStatement(insertsql);
            ps.setString(2, userName);
            ps.setString(3, password);
            ps.executeUpdate();
```

```
            conn.commit();
        } catch (SQLException ex) {

Logger.getLogger(DijkstraAlgo.class.getName()).log(Level.SEVERE, null,
ex);
        }

    }

    public void saveQueueTime(Connection conn, HashMap<String, Integer>
qTimesTable) {

        String insertsql = "REPLACE INTO QueueTimes "
                + "(RideID, QueueTime) VALUES (?,?) ";

        for (String rideId : qTimesTable.keySet()) {
            try {
                int qTimeId = 0;
                Statement stmt = conn.createStatement();
                ResultSet rs = stmt.executeQuery("Select * from QueueTimes
where RideID ='" + rideId + "'");
                if (rs.next()) { //Ride exits, so update it
                    qTimeId = rs.getInt("QueueTimeID");
                    insertsql = "REPLACE INTO QueueTimes "
                            + "(QueueTimeID, RideID, QueueTime) VALUES
(?,?,?) ";
                    PreparedStatement ps =
conn.prepareStatement(insertsql);
                    ps.setInt(1, qTimeId);
                    ps.setString(2, rideId);
                    ps.setInt(3, qTimesTable.get(rideId));
                    ps.executeUpdate();
                    ps.close();
                } else {
                    insertsql = "REPLACE INTO QueueTimes "
                            + "(RideID, QueueTime) VALUES (?,?) ";
                    PreparedStatement ps =
conn.prepareStatement(insertsql);
                    ps.setString(1, rideId);
                    ps.setInt(2, qTimesTable.get(rideId));
                    ps.executeUpdate();
                    ps.close();
                }
                conn.commit();
            } catch (SQLException ex) {

Logger.getLogger(DijkstraAlgo.class.getName()).log(Level.SEVERE, null,
```

```
ex);
            }
        }
    }

    public void savePreferences(Connection conn, String userId, String
currentLocationOfUser,
            Integer maxWaitingTime, HashMap<String, Integer> preferences)
{

        String updateSQL = "UPDATE User set UserCurrentLocation = ? ,
MaxWaitingTime = ?"
                + " where UserID = ?";
        try {
            PreparedStatement ps = conn.prepareStatement(updateSQL);
            ps.setString(1, currentLocationOfUser);
            ps.setInt(2, maxWaitingTime);
            ps.setString(3, userId);
            ps.executeUpdate();
            conn.commit();
        } catch (SQLException ex) {

Logger.getLogger(DatabaseAccess.class.getName()).log(Level.SEVERE, null,
ex);
        }

        for (String rideId : preferences.keySet()) {

            try {
                String insertsql = "INSERT OR IGNORE INTO Preferences "
                        + "( PreferenceID, UserID , RideID, PriorityOrder
) VALUES ( ?,?,?,? ) ";
                PreparedStatement ps = conn.prepareStatement(insertsql);
                ps.setString(2, userId);
                ps.setString(3, rideId);
                ps.setInt(4, preferences.get(rideId));
                ps.executeUpdate();
                conn.commit();

            } catch (SQLException ex) {

Logger.getLogger(DatabaseAccess.class.getName()).log(Level.SEVERE, null,
ex);
            }
        }

    }
```

```
    public boolean createTable(Connection conn) {
        Statement stmt = null;
        boolean returnValue = false;
        try {
            stmt = conn.createStatement();
            String dropRides = "DROP TABLE if exists Rides";
            stmt.executeUpdate(dropRides);
            String dropRideEdges = "DROP TABLE if exists RideEdges";
            stmt.executeUpdate(dropRideEdges);
//            String dropUser = "DROP TABLE if exists User";
//            stmt.executeUpdate(dropUser);
            String dropPreferences = "DROP TABLE if exists Preferences";
            stmt.executeUpdate(dropPreferences);
//            String dropQueueTimes = "DROP TABLE if exists QueueTimes";
//            stmt.executeUpdate(dropQueueTimes);
            conn.commit();

            String sql1 = "CREATE TABLE Rides"
                    + " (RideID VARCHAR(255) PRIMARY KEY     NOT NULL,"
                    + " RideName VARCHAR(255))";

            stmt.executeUpdate(sql1);
            conn.commit();

            Map<String, String> ridesTable = new HashMap<String,
String>();

            ridesTable.put("A", "Entrance");
            ridesTable.put("B", "Depth Charge");
            ridesTable.put("C", "Vortex");
            ridesTable.put("D", "Quantum");
            ridesTable.put("E", "Flying fish");
            ridesTable.put("F", "Zodiac");
            ridesTable.put("G", "Rush");
            ridesTable.put("H", "Colossus");
            ridesTable.put("I", "The Walking Dead: The Ride");
            ridesTable.put("J", "Ghost Train");
            ridesTable.put("K", "Samurai");
            ridesTable.put("L", "Saw - The Ride");
            ridesTable.put("M", "Storm Surge");
            ridesTable.put("N", "The Swarm");
            ridesTable.put("O", "Tidal Wave");
            ridesTable.put("P", "Dodgems");
            ridesTable.put("Q", "Detonator");
            ridesTable.put("R", "Stealth");
            ridesTable.put("S", "Storm in a Tea Cup");
```

```
            ridesTable.put("T", "Nemesis Inferno");
            ridesTable.put("U", "Rumba Rapids");

            for (String rideCode : ridesTable.keySet()) {
                String rideDescription = (String)
ridesTable.get(rideCode);

                String insertsql = "INSERT INTO Rides (RideID,RideName) "
                        + "VALUES (?,? );";

                PreparedStatement ps = conn.prepareStatement(insertsql);
                ps.setString(1, rideCode);
                ps.setString(2, rideDescription);
                ps.executeUpdate();
            }

            conn.commit();
            //System.out.println("Rides table created.");

            String sql2 = "CREATE TABLE IF NOT EXISTS RideEdges"
                    + "(EdgeID INTEGER PRIMARY KEY NOT NULL,"
                    + "FromRide VARCHAR(255) REFERENCES Rides(RideID),"
                    + "ToRide VARCHAR(255) REFERENCES Rides(RideID),"
                    + "WeightValue INTEGER NOT NULL)";

            String sql3 = "CREATE TABLE IF NOT EXISTS User"
                    + "(UserID INTEGER PRIMARY KEY NOT NULL,"
                    + "Username VARCHAR(255),"
                    + "Password VARCHAR(255),"
                    + "UserCurrentLocation VARCHAR(255) REFERENCES
Rides(RideID),"
                    + "MaxWaitingTime INTEGER)";

            String sql4 = "CREATE TABLE IF NOT EXISTS Preferences"
                    + "(PreferenceID INTEGER PRIMARY KEY NOT NULL,"
                    + "UserID INTEGER REFERENCES User(UserID),"
                    + "RideID VARCHAR(255) REFERENCES Rides(RideID),"
                    + "PriorityOrder INTEGER)";

            String sql5 = "CREATE TABLE IF NOT EXISTS QueueTimes"
                    + "(QueueTimeID INTEGER PRIMARY KEY NOT NULL,"
                    + "RideID VARCHAR(255) REFERENCES Rides(RideID),"
                    + "QueueTime INTEGER)";

            stmt.executeUpdate(sql2);
            //System.out.println("RideEdges table created.");
```

```
            BufferedReader input;
            try {
                input = new BufferedReader(new FileReader(new
File("C:\\Users\\adima\\Downloads\\TP13012023\\ThorpeParkNavigator\\\\Dist
anceBetweenRides.csv")));

                String line = null;
                String[] st = null;

                try {
                    while ((line = input.readLine()) != null) {
                        st = line.replace("\"", "").split(",");
                        if (!st[0].equals("EdgeId")) {
                            String insertsql = "INSERT INTO RideEdges
(EdgeId,FromRide,ToRide,WeightValue) "
                                    + "VALUES (?,?,?,? );";

                            PreparedStatement ps =
conn.prepareStatement(insertsql);
                            ps.setString(1, st[0]);
                            ps.setString(2, st[1]);
                            ps.setString(3, st[2]);
                            int distance = Integer.valueOf(st[3]);
                            ps.setInt(4, distance);
                            ps.executeUpdate();
                            conn.commit();
                        }
                    }
                } catch (IOException ex) {

Logger.getLogger(DatabaseAccess.class.getName()).log(Level.SEVERE, null,
ex);
                }

            } catch (FileNotFoundException ex) {

Logger.getLogger(DatabaseAccess.class.getName()).log(Level.SEVERE, null,
ex);
            }

            stmt.executeUpdate(sql3);
            conn.commit();
            stmt.executeUpdate(sql4);
            conn.commit();
            stmt.executeUpdate(sql5);
            conn.commit();
            stmt.close();
```

```
            returnValue = true;
        } catch (SQLException e) {
            System.out.println(e.getClass().getName() + ": " +
e.getMessage());
            System.exit(0);
        }
        return returnValue;
    }

    public List< Ride> getRidesFromDB(Connection conn) {
        Statement stmt = null;
        ResultSet rs = null;
        List<Ride> nodeList = new ArrayList< Ride>();
        try {
            stmt = conn.createStatement();
            rs = stmt.executeQuery("select * from Rides");
            while (rs.next()) {
                String rideId = rs.getString("RideID");
                String description = rs.getString("RideName");
                Ride ride = new Ride(rideId, description);
                nodeList.add(ride);
            }
            stmt.close();
            rs.close();
        } catch (SQLException e) {
            System.out.println(e.getClass().getName() + ": " +
e.getMessage());
        }
        return nodeList;
    }

    public List< Path> getEdgesFromDB(Connection conn, List<Ride> rides) {
        Statement stmt = null;
        ResultSet rs = null;
        List<Path> edgeList = new ArrayList< Path>();
        try {
            stmt = conn.createStatement();
            rs = stmt.executeQuery("select * from RideEdges");
            while (rs.next()) {
                int id = rs.getInt("EdgeId");
                String fromRideId = rs.getString("FromRide");
                String toRideId = rs.getString("ToRide");
                int distance = rs.getInt("WeightValue");

                Ride sourceRide = rides.stream()
                        .filter(ride -> fromRideId.equals(ride.getId()))
                        .findAny()
```

```
                        .orElse(null);

                Ride destRide = rides.stream()
                        .filter(ride -> toRideId.equals(ride.getId()))
                        .findAny()
                        .orElse(null);

                Path edge = new Path(id, sourceRide, destRide, distance);
                edgeList.add(edge);
            }
            stmt.close();
            rs.close();
        } catch (SQLException e) {
            System.out.println(e.getClass().getName() + ": " +
e.getMessage());
        }
        return edgeList;
    }

    public List< User> getUsersFromDB(Connection conn) {
        Statement stmt = null;
        ResultSet rs = null;
        List<User> userList = new ArrayList< User>();
        try {
            stmt = conn.createStatement();
            rs = stmt.executeQuery("select * from User");
            while (rs.next()) {
                String userId = rs.getString("UserId");
                String userName = rs.getString("UserName");
                String password = rs.getString("Password");
                String currentLocation =
rs.getString("UserCurrentLocation");
                Integer maxWaitingTime = rs.getInt("MaxWaitingTime");
                User user = new User(userId, userName, password,
currentLocation,
                        maxWaitingTime);
                userList.add(user);
            }
            stmt.close();
            rs.close();
        } catch (SQLException e) {
            System.out.println(e.getClass().getName() + ": " +
e.getMessage());
        }
        return userList;
    }
```

```
    public List< Preferences> getPreferenceForUserFromDB(Connection conn,
List<Ride> rides, User user, String userId) {
        Statement stmt = null;
        ResultSet rs = null;
        List<Preferences> preferencesList = new ArrayList< Preferences>();

        try {
            stmt = conn.createStatement();
            rs = stmt.executeQuery("select * from Preferences where
UserID='" + userId + "' order by PriorityOrder asc");
            while (rs.next()) {

                int id = rs.getInt("PreferenceID");
                String rideId = rs.getString("RideID");
                int priorityOrder = rs.getInt("PriorityOrder");

                //Get wait time for the ride
                Statement rideWaitstmt = conn.createStatement();
                int rideQueueTime = 0;
                ResultSet rideWaitrs = rideWaitstmt.executeQuery("select *
from QueueTimes where RideID='" + rideId + "'");
                if (rideWaitrs.next()) {
                    rideQueueTime = rideWaitrs.getInt("QueueTime");
                }

                Ride toRide = rides.stream()
                        .filter(ride -> rideId.equals(ride.getId()))
                        .findAny()
                        .orElse(null);

                Preferences preferences = new Preferences(id, user,
toRide, priorityOrder, rideQueueTime);
                preferencesList.add(preferences);
            }
            stmt.close();
            rs.close();
        } catch (SQLException e) {
            System.out.println(e.getClass().getName() + ": " +
e.getMessage());
        }
        return preferencesList;
    }

}

public class DijkstraAlgo {
```

```
    private final List<Ride> rides;
    private final List<Path> paths;
    private Set<Ride> visitedNodes;
    private Set<Ride> unvisitedNodes;
    private Map<Ride, Ride> predecessors;
    private Map<Ride, Integer> distance;
    private Map<Ride, Integer> eachStep;

    public DijkstraAlgo(Graph graph) {
        // create a copy of the array so that we can operate on this array
        this.rides = new ArrayList<Ride>(graph.getRides());
        this.paths = new ArrayList<Path>(graph.getWeights());
    }

    public static void main(String arg[]) {

        boolean exit = false;
        while (!exit) {

            Scanner scanner = new Scanner(System.in);

            List<Ride> tnodes;
            List<Path> tedges;
            List<User> users;
            List<Preferences> preferences;

            tnodes = new ArrayList<Ride>();
            tedges = new ArrayList<Path>();
            users = new ArrayList<User>();
            preferences = new ArrayList<Preferences>();

            DatabaseAccess dbAccess = new DatabaseAccess();
            Connection conn = dbAccess.databaseConnect();
            dbAccess.createTable(conn);
            if (conn == null) {
                System.out.println(" DB connection is null. Cannot
proceed.");
            }

            tnodes = dbAccess.getRidesFromDB(conn);
            tedges = dbAccess.getEdgesFromDB(conn, tnodes);
            users = dbAccess.getUsersFromDB(conn);

            Graph graph = new Graph(tnodes, tedges);

            DijkstraAlgo dijkstra = new DijkstraAlgo(graph);
```

```
            String userName = "", password = "";
            boolean valid = false;

            User user = null;
            String userId = null;

            System.out.println("\n\n                Welcome to Thorpe Park
Navigator");
            System.out.println("\nLogin / Account Registration\n");

            while (true) {

                while (true) {
                    System.out.println("Enter your existing / desired
username that is between 5 and 12 characters long:");
                    userName = scanner.nextLine();
                    if (!userName.isEmpty() && userName.length() >= 5 &&
userName.length() <= 12) {
                        valid = true;
                        break;
                    }
                    System.out.println("The username entered is invalid.
Username cannot be empty, and must be between 5 and 12 characters long.");
                }
                while (true) {
                    System.out.println("Enter your existing / desired
password:");
                    password = scanner.nextLine();
                    if (!password.isEmpty()) {
                        user = dijkstra.validateUserAndPassword(userName,
password, users);
                        break;
                    }
                    System.out.println("Password cannot be empty.");
                }

                if (user != null) {
                    userId = user.getUserId();
                    break;
                } else if (dijkstra.getUser(userName, users) == null &&
valid) {
                    System.out.print("\nThis username and password
combination does not exist. Would you like to create a new account?\nEnter
'Y' to proceed\nEnter any key to go back to the start\n");
                    String createUser = scanner.nextLine();
                    createUser = createUser.toUpperCase();
                    if (createUser.equals("Y")) {
```

```
                        dbAccess.createUser(conn, userName, password);
                        valid = false;
                        users = dbAccess.getUsersFromDB(conn);
                        user = dijkstra.validateUserAndPassword(userName,
password, users);
                        userId = user.getUserId();
                        System.out.println("Username: '" + userName + "'
and Password: '" + password + "' are valid. Account creation
successful!");
                        break;
                    }
                } else {
                    System.out.println("Username and password do not
match.\n");
                }

            }

            System.out.println("\n  Ride Name                          Ride
Key"); // Map Key
            System.out.println("Entrance                             A");
            System.out.println("Depth Charge                         B");
            System.out.println("Vortex                               C");
            System.out.println("Quantum                              D");
            System.out.println("Flying fish                          E");
            System.out.println("Zodiac                               F");
            System.out.println("Rush                                 G");
            System.out.println("Colossus                             H");
            System.out.println("The Walking Dead: The Ride           I");
            System.out.println("Ghost Train                          J");
            System.out.println("Samurai                              K");
            System.out.println("SAW - The Ride                       L");
            System.out.println("Storm Surge                          M");
            System.out.println("THE SWARM                            N");
            System.out.println("Tidal Wave                           O");
            System.out.println("Dodgems                              P");
            System.out.println("Detonator                            Q");
            System.out.println("Stealth                              R");
            System.out.println("Storm in a Tea Cup                   S");
            System.out.println("Nemesis Inferno                      T");
            System.out.println("Rumba Rapids
U\n");

            HashMap<String, Integer> preferencesTable = new
HashMap<String, Integer>();

            HashMap<String, Integer> queueTimesTable = new HashMap<String,
```

```
Integer>();

            valid = false;
            String pref1;
            String pref2;
            String pref3;
            String userCurrentLocation = "";
            Integer maxWaitingTime = null;
            Integer qTime1 = 0;
            Integer qTime2 = 0;
            Integer qTime3 = 0;

            while (!valid || preferencesTable.size() < 3) {

                Ride ride = null;
                while (ride == null) {
                    System.out.print("Using the key above, enter where in
the park you currently are:\n");
                    userCurrentLocation = scanner.nextLine();
                    if (isValidLocation(userCurrentLocation) == true) {
                        userCurrentLocation =
userCurrentLocation.toUpperCase();

                        ride = dijkstra.getRide(userCurrentLocation);
                        if (ride == null) {
                            System.out.println("The ride key you have
entered does not exist.");
                        }
                    }
                }

                ride = null;

                while (ride == null) {
                    boolean formatException = true;
                    System.out.println("Using the key above, enter which
ride/attraction you would like to visit first:");
                    pref1 = scanner.nextLine();
                    if (isValidLocation(pref1) == true) {
                        pref1 = pref1.toUpperCase();
                        ride = dijkstra.getRide(pref1);
                        if (pref1.equals(userCurrentLocation)) {
                            System.out.println("The ride you have chosen
to visit first is the same as your current location.");
                            ride = null;
                            formatException = false;
                        } else if (ride == null) {
```

```
                            System.out.println("The ride key you have
entered does not exist.");
                            formatException = false;
                        } else {
                            preferencesTable.put(pref1, 1);
                            formatException = true;
                        }
                    } else {
                        formatException = false;
                    }

                    while (formatException) {
                        Scanner in = new Scanner(System.in);
                        System.out.println("Enter the current queue time
for " + dijkstra.getRide(pref1).getName() + " (" + pref1 + ")");
                        try {
                            qTime1 = in.nextInt();
                            formatException = false;
                            queueTimesTable.put(pref1, qTime1);
                        } catch (InputMismatchException ime) {
                            System.out.println("Invalid input. Please
enter the queue time using numbers");
                            formatException = true;
                        }
                    }

                }

                ride = null;
                while (ride == null) {
                    boolean formatException = true;
                    System.out.println("Using the key above, enter which
ride/attraction you would like to visit second:");
                    pref2 = scanner.nextLine();
                    if (isValidLocation(pref2) == true) {
                        pref2 = pref2.toUpperCase();
                        ride = dijkstra.getRide(pref2);

                        if (preferencesTable.get(pref2) != null) {
                            System.out.println("The ride you have chosen
to visit second is the same as the ride you would like to visit first.");
                            ride = null;
                            formatException = false;
                        } else if (ride == null) {
                            System.out.println("The ride key you have
entered does not exist.");
                            formatException = false;
```

```
                        } else {
                            preferencesTable.put(pref2, 2);
                            formatException = true;
                        }
                    } else {
                        formatException = false;
                    }

                    while (formatException) {
                        Scanner in = new Scanner(System.in);
                        System.out.println("Enter the current queue time
for " + dijkstra.getRide(pref2).getName() + " (" + pref2 + ")");
                        try {
                            qTime2 = in.nextInt();
                            formatException = false;
                            queueTimesTable.put(pref2, qTime2);
                        } catch (InputMismatchException ime) {
                            System.out.println("Invalid input. Please
enter the queue time using numbers");
                            formatException = true;
                        }
                    }
                }

                ride = null;
                while (ride == null) {
                    boolean formatException = true;
                    System.out.println("Using the key above, enter which
ride/attraction you would like to visit third:");
                    pref3 = scanner.nextLine();
                    if (isValidLocation(pref3) == true) {
                        pref3 = pref3.toUpperCase();
                        ride = dijkstra.getRide(pref3);
                        if (ride == null) {
                            System.out.println("The ride key you have
entered does not exist.");
                        }
                        if (preferencesTable.get(pref3) != null) {
                            System.out.println("Invalid input. The ride
you have chosen to visit third has already been selected as another one of
your choices.");
                            ride = null;
                            formatException = false;
                        } else if (ride == null) {
                            System.out.println("The ride key you have
entered does not exist.");
                            formatException = false;
```

```
                        } else {
                            preferencesTable.put(pref3, 3);
                            formatException = true;
                        }

                    } else {
                        formatException = false;
                    }

                    while (formatException) {
                        Scanner in = new Scanner(System.in);
                        System.out.println("Enter the current queue time
for " + dijkstra.getRide(pref3).getName() + " (" + pref3 + ")");
                        try {
                            qTime3 = in.nextInt();
                            formatException = false;
                            queueTimesTable.put(pref3, qTime3);
                        } catch (InputMismatchException ime) {
                            System.out.println("Invalid input. Please
enter the queue time using numbers");
                            formatException = true;
                        }
                    }
                }

                while (maxWaitingTime == null) {
                    scanner = new Scanner(System.in);
                    System.out.println("What is the maximum amount of
time, in minutes, you will be willing to wait in a queue? Enter a
number:");
                    try {
                        maxWaitingTime = scanner.nextInt();
                    } catch (InputMismatchException ime) {
                        System.out.println("Invalid input. Please enter
your maximum waiting time in numbers");
                        maxWaitingTime = null;

                    }

                }
                valid = true;

            }

            dbAccess.savePreferences(conn, userId, userCurrentLocation,
maxWaitingTime,
                    preferencesTable); //Save to DB
```

```
            dbAccess.saveQueueTime(conn, queueTimesTable); //Save to DB

            users = dbAccess.getUsersFromDB(conn);

            user = dijkstra.validateUserAndPassword(userName, password,
users);

            preferences = dbAccess.getPreferenceForUserFromDB(conn,
tnodes, user, userId); //Read back from DB

            if (conn != null) {
                dbAccess.close(conn);
                System.out.println("\nDatabase closed.\n");
            }

            userCurrentLocation = user.getCurrentLocation();
            Ride fromRide = dijkstra.getRide(userCurrentLocation);

            int totalCost = 0;
            int totalQueueTime = 0;
            for (Preferences preference : preferences) {
                Ride toRide = preference.getRide();
                System.out.println("");

                System.out.println("Cost of journey from '"
                        + fromRide.getName() + "' (" + fromRide.getId() +
") To '" + toRide.getName()
                        + "' (" + toRide.getId() + ") is "
                        + dijkstra.getShortestDistance(fromRide, toRide) +
" minute(s)");

                if (preference.getRideQueueTime() >
user.getMaxWaitingTime()) {
                    System.out.println("Queue time for "
                            + toRide.getName()
                            + "(" + toRide.getId()
                            + ") is "
                            + preference.getRideQueueTime() + " minute(s).
This is greater than your maximum waiting time of "
                            + user.getMaxWaitingTime() + " minute(s),
therefore, "
                            + toRide.getName()
                            + "(" + toRide.getId()
                            + ") is skipped");

                } else {
```

```
                    totalCost = totalCost +
dijkstra.getShortestDistance(fromRide, toRide);
                    LinkedList<Ride> myPath = dijkstra.getPath(toRide);
                    String pref = preference.getRide().getId();
                    if (queueTimesTable.get(pref) != null) {
                        totalQueueTime = totalQueueTime +
queueTimesTable.get(pref);
                    }

                    boolean startPosition = true;

                    for (Ride step : myPath.items()) {
                        if (startPosition) {
                            System.out.println("Starting from '" +
fromRide.getName() + "' (" + fromRide.getId() + ")");
                            startPosition = false;
                        } else {
                            System.out.println("Go To -> '" +
step.getName() + "' (" + step.getId() + ")");
                        }
                        fromRide = toRide;
                    }

                }
            }

            totalCost = totalCost + totalQueueTime;

            if (totalCost == 0) {
                System.out.println("\nAll rides have been skipped, cost of
journey is 0\n");
            } else {
                System.out.println("\nTotal cost of journey is " +
totalCost + " minutes\n");
            }

            System.out.println("Would you like to go back to the
start?\nEnter 'Y' to proceed\nEnter any other key to terminate the
program");

            Scanner scan = new Scanner(System.in);

            String backToStart = scan.nextLine();

            backToStart = backToStart.toUpperCase();
```

```
            if (backToStart.equals("Y")) {
                exit = false;

                System.out.println("");
            } else {

                exit = true;

            }
        }
        System.out.println("\nThank you for using Thorpe Park
Navigator!");

    }

    private void calculateDistance(Ride source, Ride destination) {
        visitedNodes = new HashSet<Ride>();
        unvisitedNodes = new HashSet<Ride>();
        distance = new HashMap<Ride, Integer>();
        eachStep = new HashMap<Ride, Integer>();
        predecessors = new HashMap<Ride, Ride>();
        distance.put(source, 0);
        unvisitedNodes.add(source);
        while (unvisitedNodes.size() > 0) {
            Ride ride = getMinimum(unvisitedNodes);
            visitedNodes.add(ride);
            unvisitedNodes.remove(ride);
            findMinimalDistances(ride, destination);
        }

    }

    private void findMinimalDistances(Ride ride, Ride destination) {
        List<Ride> adjacentNodes = getNeighbours(ride);
        for (Ride target : adjacentNodes) {
            int distanceToTarget = getShortestDistance(target);
            int distanceToRide = getShortestDistance(ride);
            int distanceFromRideToTarget = getDistance(ride, target);

            if (distanceToTarget > distanceToRide +
distanceFromRideToTarget) {
                distance.put(target, distanceToRide +
distanceFromRideToTarget);
                eachStep.put(target, distanceFromRideToTarget);
                predecessors.put(target, ride);
                unvisitedNodes.add(target);
            }
```

```
        }
    }

    private int getDistance(Ride source, Ride destination) {
        int dist = 0;
        for (Path path : paths) {
            if (path.getFromRide().equals(source) &&
path.getToRide().equals(destination)) {
                dist = path.getWeights();
                break;
            }
        }
        return dist;
    }

    private List<Ride> getNeighbours(Ride ride) {
        List<Ride> neighbours = new ArrayList<Ride>();
        for (Path path : paths) {
            if (path.getFromRide().equals(ride)
                    && !isVisited(path.getToRide())) {
                neighbours.add(path.getToRide());
            }
        }
        return neighbours;
    }

    private Ride getMinimum(Set<Ride> nodes) {
        Ride minimum = null;
        for (Ride node : nodes) {
            if (minimum == null) {
                minimum = node;
            } else {
                if (getShortestDistance(node) <
getShortestDistance(minimum)) {
                    minimum = node;
                }
            }
        }
        return minimum;
    }

    private boolean isVisited(Ride node) {
        return visitedNodes.contains(node);
    }

    private int getShortestDistance(Ride source, Ride destination) {
        calculateDistance(source, destination);
```

```
        Integer weight = distance.get(destination);
        if (weight == null) {
            return Integer.MAX_VALUE;
        } else {
            return weight;
        }
    }

    private int getShortestDistance(Ride destination) {
        Integer weight = distance.get(destination);
        if (weight == null) {
            return Integer.MAX_VALUE;
        } else {
            return weight;
        }
    }

    public LinkedList<Ride> getPath(Ride target) {  //This method returns
the path from the source to the selected target and NULL if no path exists
        LinkedList<Ride> path = null;
        path = new LinkedList<Ride>();
        Ride step = target;
        // check if a path exists
        if (predecessors.get(step) == null) {
            return null;
        }
        path.add(step);
        while (predecessors.get(step) != null) {
            step = predecessors.get(step);
            path.add(step);
        }
        return path;
    }

    private Ride getRide(String rideId) {
        Ride tride = this.rides.stream()
                .filter(ride -> rideId.equalsIgnoreCase(ride.getId()))
                .findAny()
                .orElse(null);
        return tride;
    }

    private User getUser(String userName, List<User> users) {
        User tuser = users.stream()
                .filter(user ->
userName.equalsIgnoreCase(user.getUserName()))
                .findAny()
```

```
                .orElse(null);
        return tuser;
    }

    private User validateUserAndPassword(String userName, String password,
List<User> users) {
        User tuser = users.stream()
                .filter(user ->
userName.equalsIgnoreCase(user.getUserName()))
                .filter(user -> password.equals(user.getPassword()))
                .findAny()
                .orElse(null);
        return tuser;
    }

    private static boolean isValidLocation(String location) {
        if (location.length() != 1 ||
!Character.isLetter(location.charAt(0))) {
            System.out.println("Invalid input. Please enter a single
letter.");
            return false;
        } else {
            return true;
        }
    }
}

public class Graph {

    private final List<Ride> rides;
    private final List<Path> weights;

    public Graph(List<Ride> rides, List<Path> weights) {
        this.rides = rides;
        this.weights = weights;
    }

    public List<Ride> getRides() {
        return rides;
    }

    public List<Path> getWeights() {
        return weights;
    }

}
```

```
public class Path {

    private final int id;
    private final Ride fromRide;
    private final Ride toRide;
    private final int weights;

    public Path(int id, Ride fromRide, Ride toRide, int weights) {
        this.id = id;
        this.fromRide = fromRide;
        this.toRide = toRide;
        this.weights = weights;
    }

    public int getId() {
        return id;
    }

    public Ride getToRide() {
        return toRide;
    }

    public Ride getFromRide() {
        return fromRide;
    }

    public int getWeights() {
        return weights;
    }

}

public class Preferences {

    private final int id;
    private final User user;
    private final Ride ride;
    private final int priorityOrder;
    private final int rideQueueTime;

    public Preferences(int id, User user, Ride ride, int priorityOrder,
int rideQueueTime) {
        this.id = id;
        this.user = user;
        this.ride = ride;
        this.priorityOrder = priorityOrder;
        this.rideQueueTime = rideQueueTime;
```

```
    }

    public int getId() {
        return id;
    }

    public User getUser() {
        return user;
    }

    public Ride getRide() {
        return ride;
    }

    public int getPriorityOrder() {
        return priorityOrder;
    }

    public int getRideQueueTime() {
        return rideQueueTime;
    }

}

public class Ride {

    final private String id;
    final private String name;

    public Ride(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}

public class User {

    final private String userId;
    final private String userName;
```

```
    final private String password;
    final private String currentLocation;
    final private Integer maxWaitingTime;

    public User(String userId, String userName, String password,
            String currentLocation, Integer maxWaitingTime) {
        this.userId = userId;
        this.userName = userName;
        this.password = password;
        this.currentLocation = currentLocation;
        this.maxWaitingTime = maxWaitingTime;
    }

    public String getUserId() {
        return userId;
    }

    public String getUserName() {
        return userName;
    }

    public String getPassword() {
        return password;
    }

    public String getCurrentLocation() {
        return currentLocation;
    }

    public Integer getMaxWaitingTime() {
        return maxWaitingTime;
    }

}

public class LinkedList<E> {

    static class Node<E> {

        E value;
        Node<E> next;

        Node(E value) {
            this.value = value;
        }
    }
```

```
    Node<E> head = new Node<E>(null);
    Node<E> tail = head;
    int size;

    void add(E value) {
        tail = tail.next = new Node<E>(value);
        size++;
    }

    E get(int index) {
        if (index < 0 || size <= index) {
            System.out.println("Out of bounds return null");
            return null;
        }
        Node<E> node = head.next;
        while (index > 0) {
            node = node.next;
            index--;
        }
        return node.value;
    }

    public ArrayList<E> items() {
        ArrayList<E> arrayList = new ArrayList();
        int counter = 0;
        while (counter < size) {
            arrayList.add(this.get(counter));
            counter += 1;
        }
        Collections.reverse(arrayList);
        return arrayList;
    }
}
```

Testing

Input and Output Testing

In order to test my program's success in terms of providing a solution to the original problem outlined in the Analysis section, as well as how robustly coded it is, I have conducted numerous tests which test the functionalities of the program, shown in the table below. The tests have been conducted with normal data, erroneous data, and boundary data.

Test Number	Video Time-stamp	Description	Input	Expected Output	Success or Fail
1	0:13	Creating account: Testing username input (Boundary)	Username = "user1"	"Enter your existing / desired password:"	Success
2	0:35	Creating account: Testing username input (Boundary)	Username = "username1234"	"Enter your existing / desired password:"	Success
3	0:50	Creating account: Testing username input (Erroneous)	Username = " "	"The username entered is invalid. Username cannot be empty, and must be between 5 and 12 characters long."	Success
4	0:57	Creating account: Testing username input (Boundary)	Username = "user"	"The username entered is invalid. Username cannot be empty, and must be between 5 and 12 characters long."	Success
5	1:06	Creating account: Testing username input (Boundary)	Username = "username12345"	"The username entered is invalid. Username cannot be empty, and must be between 5 and 12 characters long."	Success
6	1:17	Creating account: Testing password input (Erroneous)	Username = "username" Password = " "	"Password cannot be empty."	Success
7	1:27	Creating account: Testing password input (Normal)	Password = "password"	This username and password combination does not exist. Would you like to create a new account? Enter 'Y' to proceed Enter any key to go back to the start"	Success

8	1:37	Denying account creation	Choice = “@”	“Enter your existing / desired username that is between 5 and 12 characters long:"	Success
9	1:42	Confirming account creation	Username = “username” Password = “password” Choice = “Y”	“Account creation successful!"	Success
10	2:04	Checking database insertion for User table		Login details stored in database correctly (1, username, password, ,)	Success
11	2:14	Logging in	Username = “username” Password = “pass”	“Username and password do not match.”	Success
12	2:31	Logging in	Username = “username” Password = “password”	“Using the key above, enter where in the park you currently are:”	Success
13	2:39	Testing user location input (Erroneous)	CurrentLocation = “Aa”	“Invalid input. Please enter a single letter.”	Success
14	2:45	Testing user location input (Erroneous)	CurrentLocation = “1”	“Invalid input. Please enter a single letter.”	Success
15	2:52	Testing user location input (Normal)	CurrentLocation = “A”	“Using the key above, enter which ride/attraction you would like to visit first:”	Success
16	2:59	Testing preference input	Preference1 = “a”	“The ride you have chosen to visit first is the same as your current location.”	Success
17	3:06	Testing preference input (Erroneous)	Preference1 = “2”	“Invalid input. Please enter a single letter.”	Success
18	3:11	Testing preference input (Erroneous)	Preference1 = “v”	"The ride key you have entered does not exist."	Success
19	3:17	Testing preference input (Normal)	Preference1 = “u”	“Enter the current queue time for Rumba Rapids (U)”	Success
20	3:24	Testing queue time input (Erroneous)	QueueTime1 = “aaa”	“Invalid input. Please enter the queue time using numbers”	Success

21	3:30	Testing queue time input (Normal)	QueueTime1 = “7”	“Using the key above, enter which ride/attraction you would like to visit second:”	Success
22	3:36	Testing preference input (ensuring unique preferences)	Preference2 = “U”	“The ride you have chosen to visit second is the same as the ride you would like to visit first.”	Success
23	3:51	Testing preference input (Normal)	Preference2 = “b”	“Enter the current queue time for Depth Charge (B)”	Success
24	3:58	Testing queue time input (Erroneous)	QueueTime2 = “aaa”	“Invalid input. Please enter the queue time using numbers”	Success
25	4:06	Testing queue time input (Normal)	QueueTime2 = “9”	“Using the key above, enter which ride/attraction you would like to visit third:”	Success
26	4:14	Testing preference input (ensuring unique preferences)	Preference3 = “B”	“Invalid input. The ride you have chosen to visit third has already been selected as another one of your choices.”	Success
27	4:23	Testing preference input (ensuring unique preferences)	Preference3 = “U”	“Invalid input. The ride you have chosen to visit third has already been selected as another one of your choices.”	Success
28	4:29	Testing preference input (Normal)	Preference3 = “g”	“Enter the current queue time for Rush (G)”	Success
29	4:35	Testing queue time input (Erroneous)	QueueTime3 = “aaa”	“Invalid input. Please enter the queue time using numbers”	Success
30	4:41	Testing queue time input (Normal)	QueueTime3 = “8”	“What is the maximum amount of time, in minutes, you will be willing to wait in a queue? Enter a number:”	Success
31	4:48	Testing maximum wait time input (Erroneous)	MaximumWaitTime = “aaa”	“Invalid input. Please enter your maximum waiting time in numbers”	Success

32	4:58	Testing maximum wait time input (Boundary)	MaximumWaitTime = “9”	Cost of journey from 'Entrance' (A) To 'Rumba Rapids' (U) is 8 minute(s) Starting from 'Entrance' (A) Go To -> 'Quantum' (D) Go To -> 'Rush' (G) Go To -> 'The Walking Dead: The Ride' (I) Go To -> 'Ghost Train' (J) Go To -> 'Nemesis Inferno' (T) Go To -> 'Rumba Rapids' (U) Cost of journey from 'Rumba Rapids' (U) To 'Depth Charge' (B) is 8 minute(s) Starting from 'Rumba Rapids' (U) Go To -> 'Nemesis Inferno' (T) Go To -> 'Ghost Train' (J) Go To -> 'Storm Surge' (M) Go To -> 'Flying fish' (E) Go To -> 'Depth Charge' (B) Cost of journey from 'Depth Charge' (B) To 'Rush' (G) is 3 minute(s) Starting from 'Depth Charge' (B) Go To -> 'Entrance' (A) Go To -> 'Quantum' (D) Go To -> 'Rush' (G) Total cost of journey is 43 minutes Would you like to go back to the start? Enter 'Y' to proceed Enter any other key to terminate the program	Success *(Refer back to Design section for Dijkstra dry run which shows path from A to U)*

33	5:58	Testing maximum wait time input (Boundary)	Choice = “n” Username = “username” Password = “password” CurrentLocation = “A” Preference1 = “u” QueueTime1 = “7” Preference2 = “b” QueueTime2 = “9” Preference3 = “g” QueueTime3 = “8” MaximumWaitTime = “8”	Cost of journey from 'Entrance' (A) To 'Rumba Rapids' (U) is 8 minute(s) Starting from 'Entrance' (A) Go To -> 'Quantum' (D) Go To -> 'Rush' (G) Go To -> 'The Walking Dead: The Ride' (I) Go To -> 'Ghost Train' (J) Go To -> 'Nemesis Inferno' (T) Go To -> 'Rumba Rapids' (U) Cost of journey from 'Rumba Rapids' (U) To 'Depth Charge' (B) is 8 minute(s) Queue time for Depth Charge(B) is 9 minute(s). This is greater than your maximum waiting time of 8 minute(s), therefore, Depth Charge(B) is skipped Cost of journey from 'Rumba Rapids' (U) To 'Rush' (G) is 6 minute(s) Starting from 'Rumba Rapids' (U) Go To -> 'Nemesis Inferno' (T) Go To -> 'Ghost Train' (J) Go To -> 'The Walking Dead: The Ride' (I) Go To -> 'Rush' (G) Total cost of journey is 29 minutes Would you like to go back to the start? Enter 'Y' to proceed Enter any other key to terminate the program	Success
34	6:40	Program termination	Choice = “n”	“Thank you for using Thorpe Park Navigator!”	Success

35	6:53	Testing maximum wait time input (Boundary)	Username = “username” Password = “password” CurrentLocation = “A” Preference1 = “u” QueueTime1 = “7” Preference2 = “b” QueueTime2 = “9” Preference3 = “g” QueueTime3 = “8” MaximumWaitTime = “7”	Cost of journey from 'Entrance' (A) To 'Rumba Rapids' (U) is 8 minute(s) Starting from 'Entrance' (A) Go To -> 'Quantum' (D) Go To -> 'Rush' (G) Go To -> 'The Walking Dead: The Ride' (I) Go To -> 'Ghost Train' (J) Go To -> 'Nemesis Inferno' (T) Go To -> 'Rumba Rapids' (U) Cost of journey from 'Rumba Rapids' (U) To 'Depth Charge' (B) is 8 minute(s) Queue time for Depth Charge(B) is 9 minute(s). This is greater than your maximum waiting time of 7 minute(s), therefore, Depth Charge(B) is skipped Cost of journey from 'Rumba Rapids' (U) To 'Rush' (G) is 6 minute(s) Queue time for Rush(G) is 8 minute(s). This is greater than your maximum waiting time of 7 minute(s), therefore, Rush(G) is skipped Total cost of journey is 15 minutes Would you like to go back to the start? Enter 'Y' to proceed Enter any other key to terminate the program	Success

36	7:26	Testing maximum wait time input (Boundary)	Choice = “s” Username = “username” Password = “password” CurrentLocation = “A” Preference1 = “u” QueueTime1 = “7” Preference2 = “b” QueueTime2 = “9” Preference3 = “g” QueueTime3 = “8” MaximumWaitTime = “6”	Cost of journey from 'Entrance' (A) To 'Rumba Rapids' (U) is 8 minute(s) Queue time for Rumba Rapids(U) is 7 minute(s). This is greater than your maximum waiting time of 6 minute(s), therefore, Rumba Rapids(U) is skipped Cost of journey from 'Entrance' (A) To 'Depth Charge' (B) is 1 minute(s) Queue time for Depth Charge(B) is 9 minute(s). This is greater than your maximum waiting time of 6 minute(s), therefore, Depth Charge(B) is skipped Cost of journey from 'Entrance' (A) To 'Rush' (G) is 2 minute(s) Queue time for Rush(G) is 8 minute(s). This is greater than your maximum waiting time of 6 minute(s), therefore, Rush(G) is skipped All rides have been skipped, cost of journey is 0 Would you like to go back to the start? Enter 'Y' to proceed Enter any other key to terminate the program	Success
37	7:53	Checking database insertion for User table		UserCurrentLocation and MaxWaitingTime stored in database correctly (1, username, password, A, 6)	Success

38	8:00	Checking database insertion for Preferences table		UserID, RideID, and PriorityOrder stored in database correctly (1, 1, B, 2) (2, 1, U, 1) (3, 1, G, 3)	Success
39	8:13	Checking database insertion for QueueTimes table		RideID and QueueTime stored in database correctly. (1, B, 9) (2, U, 7) (3, G, 8)	Success
40	8:23	Going back to start	Choice = 'y'	"Enter your existing / desired username that is between 5 and 12 characters long:"	Success
41	8:33	Checking database update for User table	Username = "username" Password = "password" CurrentLocation = "D" Preference1 = "g" QueueTime1 = "7" Preference2 = "h" QueueTime2 = "9" Preference3 = "b" QueueTime3 = "8" MaximumWaitTime = "10"	UserCurrentLocation and MaxWaitingTime updated in database correctly (1, username, password, D, 10)	Success
42	9:14	Checking database update for Preferences table		RideID, and PriorityOrder updated correctly (1, 1, B, 3) (2, 1, G, 1) (3, 1, H, 2)	Success
43	9:23	Checking database update for QueueTimes table		Queue Time updated correctly, and new record for H (1, B, 8) (2, U, 7) (3, G, 7) (4, H, 9)	Success

Evidence of Test Outcome

In video format

URL: https://youtu.be/85NsdJVhlRE

Evaluation

Achievement of Objectives

Objective	Achieved?	Comments
Account creation and management		Users are able to create unique accounts with unique usernames and passwords, and a table within the database exists to store this information.
- Users can create an account	☑	
- User account information is stored in a table	☑	
User input and preferences		Users are able to input their current location, destination, and top 3 preferred rides, and a table within the database exists to store this information.
- Users can input their current location	☑	
- Users can input their destination	☑	
- Users can input their top 3 preferred rides or attractions in order of priority	☑	
- Program ensures all 3 preferences are unique	☑	
- User preferences are stored in a table	☑	
Shortest path calculation and display		The program calculates the shortest path between a user's current location and destination, including their top 3 preferred rides in order of priority. The path is displayed to the user with the names of the rides.
- The program calculates the shortest path between the user's current location and destination	☑	
- The path includes all 3 of the user's preferred rides or attractions in the specified priority order	☑	
- The path is displayed to the user with ride names	☑	

Ride information		Tables within the database exist to store information about the rides or attractions and the edges between them.
- A table stores information about the rides or attractions	☑	
- A table stores information about the connections between rides or attractions	☑	
Journey time calculation and queue time input		The program calculates the total time it will take for a user to complete their journey, taking into account travel time and queue times. Users can input current queue times and the program will add this to the total journey time. The program also asks for the user's maximum waiting time in a queue and recalculates the path if a ride or attraction exceeds this time.
- The program outputs the total journey time	☑	
- Users can input the current queue times for the rides or attractions they are visiting	☑	
- The program adds the queue times to the total journey time to determine the overall cost of the journey	☑	
- A table stores the current queue times for each ride or attraction	☑	
- The program asks the user for their maximum waiting time in a queue	☑	
- If a ride or attraction's waiting time exceeds the user's maximum waiting time, it is excluded from the path and the program recalculates the path	☑	

User Interview

As a way to evaluate the success of the program, I re-interviewed the same student who was previously interviewed during the Analysis stage of development. The idea of this was to understand whether my program helped solve the original problem, as well as receiving feedback on where improvements could be made.

It's been a few months since we last spoke and I'm curious to hear about your experience with the Thorpe Park Navigator. Can you tell me what you did with the program?

I registered an account with a username and password, and then input my current location and preferred attractions, in order of priority.

How did the program help you plan your visit to Thorpe Park?

The program calculated the shortest path between my current location and destination, taking into account my preferred attractions. It displayed the shortest path in terms of ride names, which made it easier for me to understand which rides I would be visiting.

Did the program consider queue times when calculating the shortest path?

Yes, the program asked me to input the queue times for the rides I was visiting, and then added the queue times to the total cost of the path to determine the total cost of the journey. It also asked me for my maximum waiting time in a queue, and skipped any rides that had a queue time that exceeded my maximum waiting time.

How many times have you used the program since we last spoke?

I've used the program a few times now, and it's really helped me make the most of my visits to Thorpe Park.

How so?

Well, the ability to enter my preference list and set a maximum waiting time in the queue has made it much easier for me to navigate the park and go on the rides that I want to go on. I no longer waste time wandering around or standing in long queues for rides that I'm not interested in.

How many rides are you able to go on now with the help of the Thorpe Park Navigator?

With the program's help, I've been able to go on around 7-9 rides per visit, which is a big improvement compared to before.

Do you feel like you're able to get the most out of your ticket now?

Definitely, the program has really helped me make the most of my time at the park and go on as many rides as possible.

Overall, how would you rate the program?

I would give the program a high rating. It was easy to use and helped me plan my visit to Thorpe Park effectively.

Did you encounter any issues while using the program?

Not really, the program was very straightforward and easy to use.

Do you have any suggestions for how the program could be improved?

One thing that I think would be useful is if the program had a feature that let users see the queue times for the rides in real-time. That way, they could make more informed decisions about which rides to visit and in what order.

Conclusion and identification of areas for improvement

In conclusion, the interview highlighted to me that overall, my solution to the original problem has been a success. It has eliminated the common issues that arise when visiting Thorpe Park, as well as achieving the objectives that were set out in the Analysis stage of development. As mentioned by the student in the interview, one way in which the program could be improved is by implementing a feature that lets users see the queue times for the rides in real-time. Currently, queue times are entered by the user, which could be considered as a flawed system because the user must be aware of the queue time for the ride by physically going to the ride to see its estimated queue time, so it can be entered into the navigator via the prompt. As opposed to this, the program could be made more seamless by utilising the 'Real Time API'. The Real Time API offers live waiting times and current ride statuses for Thorpe Park, with the data being updated every 5 minutes. Therefore, the navigator can be enhanced, as queue times can be derived from the API instead of the user, and allows for the queue times to be updated more frequently.