1. **Implementing Depth-First-Search (DFS) Algorithm.**
   DFS is an Uninformed search strategy, also known as Blind search strategies. This algorithm is used rostering through a graph or tree data structure. It starts at the top node of a tree and goes as deep as it can dive within a given branch, then via backpropagation it finds an unexplored routes, and then explores them until the entire given data structure (whether tree or graph) has been explored.
   a. **Strategy**.
      DFS uses a Last-In-First-Out (LIFO) approach to keep track of vertices. Depth-first search explores edges that come out of the most recently discovered vertex. Only edges going to unexplored vertices are explored. When all of ss's edges have been explored, the search backtracks until it reaches an unexplored neighbour and the procedure continues till all of the vertices that are reachable from the original source vertex are discovered. Ergo if there are any unvisited vertices, depth-first search selects one of them as a new source and repeats the search. This algorithm is careful not to repeat vertices, so each vertex is explored once.
      DFS expands the targeted node to the deepest level of the data structure therefore sometimes get stuck if the depth is infinite. Although require a low memory requirement.
   b. **Complexity**.
      The time complexity of the DFS algorithm is represented in the form of O(V+E), where V is the number of nodes and E is the number of edges, and the space complexity of the algorithm is O(V).

2. **Implementing Breadth First- Search (BFS) Algorithm.**
   While searching for a tree's data structure over a node that satisfies a given property. 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 horizontally. Usually a queue, as an extra memory is needed to keep track of the child nodes that were encountered but not yet explored.
   1. **Strategy.**
      Algorithm visits the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue. If no adjacent vertex is found, remove the first vertex from the queue. Repeat Rule 1 and Rule 2 until the queue is empty.
   2. **Complexity.**
      Breadth-first search has a running time of **O (V + E) O (V + E) O(V+E)** since every vertex and every edge will be checked once.

3. **Implementing Uniform Cost Search (UCS) Algorithm.**
   This traverse search is like a Dijikstra's algorithm. Here, instead of inserting all vertices into a priority queue, we insert only source, then one by one insert when needed. In every step, we check if the item is already in priority queue (using visited array). If yes, we perform decrease key, else we insert it.
   This variant of Dijkstra is useful for infinite graphs and those graphs which are greatly large to represent in the memory.

1. **Strategy**.
   Insert Root node into the queue repeat till queue is not empty. Remove the next element with the highest priority from the queue. If the node is a destination node, then print the cost and the path and exit else insert all the children of removed elements into the queue with their cumulative cost as their priorities.
   Here root node is the starting node for the path, and a priority queue is being maintained to maintain the path with the least cost to be chosen for the next traversal. In case 2 paths have the same cost of traversal, nodes are considered alphabetically. Completeness is guaranteed provided the cost of every step exceeds some small positive constant.
2. **Complexity.**
   Time complexity? Exponential O($b^{\lfloor C/\epsilon \rfloor}$).
   Space complexity? Exponential O($b^{\lfloor C/\epsilon \rfloor}$).

4. **Comparison Between Depth First Search, Breadth First Search and Uniform Cost Search Traverse Strategies**

   1. **Depth First Search**

Source	Destination	Explorations/Nodes Explored	Path Taken	Cost
Canada Water	Stratford	247	['Canada Water', 'Canary Wharf', 'North Greenwich', 'Canning Town', 'West Ham', 'Stratford']	15
New Cross Gate	Stepney Green	27	['New Cross Gate', 'Surrey Quays', 'Canada Water', 'Canary Wharf', 'North Greenwich', 'Canning Town', 'West Ham', 'Bromley-by-Bow', 'Bow Road', 'Mile End', 'Stepney Green']	25
Ealing Broadway	South Kensington	46	['Ealing Broadway', 'Ealing Common', 'North Ealing', 'Park Royal', 'Alperton', 'Sudbury Town', 'Sudbury Hill', 'South Harrow', 'Rayners Lane', 'West Harrow', 'Harrow-on-the-Hill', 'Northwick Park', 'Preston Road', 'Wembley Park', 'Finchley Road', 'Baker Street', 'Bond Street',	58

			'Green Park', 'Hyde Park Corner', 'Knightsbridge', 'South Kensington']	
Baker Street	Wembley Park	10	['Baker Street', 'Finchley Road', 'Wembley Park']	13

- Advantages of DFS:
  - 1. The memory requirement is Linear with respect to nodes.
  - 2. Less time and space complexity when compared to BFS.
  - 3. The solution can be found out without much more search.
- Disadvantages:
  - 1. Not Guaranteed solution.
  - 2. Time complexity is more, as cut-off depth is smaller.
  - 3. Determination of depth until the search has proceeded.

2. <u>Breadth First Search</u>

Source	Destination	Explorations/ Nodes Explored	Path Taken	Cost
Canada Water	Stratford	37	['Canada Water', 'Canary Wharf', 'North Greenwich', 'Canning Town', 'West Ham', 'Stratford']	
New Cross Gate	Stepney Green	25	['New Cross Gate', 'Surrey Quays', 'Canada Water', 'Rotherhithe', 'Wapping', 'Shadwell', 'Whitechapel', 'Stepney Green']	25
Ealing Broadway	South Kensington	46	['Ealing Broadway', 'Ealing Common', 'Acton Town', 'Turnham Green', 'Hammersmith', 'Barons Court', "Earls' Court", 'Gloucester Road', 'South Kensington']	58
Baker Street	Wembley Park	10	['Baker Street', 'Finchley Road', 'Wembley Park']	13

- Advantages
  - 1. The solution will definitely find out by BFS If there is some solution.
  - 2. BFS will never get trapped in a blind alley, which means unwanted nodes.
  - 3. If there is more than one solution then it will find a solution with minimal steps.
- Disadvantages
  - 1. Memory Constraints As it stores all the nodes of the present level to go for the next level.
  - 2. If a solution is far away then it consumes time.

3. **Uniform Cost Search.**

Source	Destination	Explorations/Nodes Explored	Path Taken	Cost
Canada Water	Stratford	14	['Canada Water', 'Rotherhithe', 'Wapping', 'Shadwell', 'Whitechapel', 'Stepney Green', 'Mile End', 'Stratford'])	14
New Cross Gate	Stepney Green	18	['New Cross Gate', 'Surrey Quays', 'Canada Water', 'Rotherhithe', 'Wapping', 'Shadwell', 'Whitechapel', 'Stepney Green']	14
Ealing Broadway	South Kensington	52	['Ealing Broadway', 'Ealing Common', 'Acton Town', 'Turnham Green', 'Hammersmith', 'Barons Court', "Earls' Court", 'Gloucester Road', 'South Kensington']	20
Baker Street	Wembley Park	81	['Baker Street', 'Finchley Road', 'Wembley Park']	13

- Advantages:
  - It helps to find the path with the lowest cumulative cost inside a weighted graph having a different cost associated with each of its edge from the root node to the destination node.
  - It is considered to be an optimal solution since, at each state, the least path is considered to be followed.
- Disadvantages:
  - The open list is required to be kept sorted as priorities in priority queue needs to be maintained.
  - Requires large storage space (exponentially).
  - As it considers every possible path going from the root node to the destination node there is possibility of algorithm to get stuck in infinite loop.

4. **Comparison**
   - **Any one Method that is consistently best**?
     - Out of all the three given methods UCS performed the best as it gives the least of the values for Cost Function, which depends upon the Average Time Taken Value taken from the Source node to the destination node.

5. **Extending Cost Function**

Cost function computes the cost of the path found by each algorithm (DFS, BFS and UCS), hence evaluating the performance/s of the algorithms via exploring all the possible paths and selecting the shortest one out of them. Therefore, it guarantees the

shortest path possible between two nodes and the optimal solution. Here, "Order of Procession" (**considering number of tube lines changes as the new cost function)** plays the valuable part that is how algorithm proceeds ahead in path instead of having a one unique solution (path).

- **New Cost Function (Number of Tube Lines commuter Changing).** This cost function consider varied Tube Lines (eg , Victoria, Piccadilly, etc) and it detects how many Station lines are changed by the commuter or she/he travelled over the same line to reach the destination.
- New cost function improves the cost significantly when checked for different commutable combinations (refer below drawn table).

Source	Destination	Old Cost (Time Taken)	New Cost (Change of Tube Lines)
Canada Water	Stratford	14	2
New Cross Gate	Stepney Green	14	1
Ealing Broadway	South Kensington	20	1
Baker Street	Wembley Park	13	0

Table: New Cost Function (Changing Tube lines)

6. **Heuristic Search**

Here the new heuristic function is added to the old one which considers one of the Tube Lines (e.g., Bakerloo, Central, Circle, District etc) taken by the commuters to reach the final destination.

Choosing one of the available Tube Line improves the algorithm search by focussing and directing the search in a particular direction, therefore reduces the number of stations a commuter has to board and de-board. Such a search divides the data into the different Zones and also consider the shortest possible distance between the source and destination (using Manhattan distance).

If the commuter is travelling through the Main zones (in higher number), means covering many main zones, a less priority is assigned to that path therefore becomes a Low priority.

We will considering here the path that taking commuter through a lesser number of main zone in path.