RBE 550 – Motion Planning

Basic Algorithm Implementation
Assignment -1

By:
Ajith Kumar Jayamoorthy

Contents

Algorithm implementation

Class Node:

1. **row** : Stores the row value of the grid as a node variable
2. **column** : Stores the column value of the grid as a node variable
3. **is_obs** : Stores if the grid is an obstacle or not
4. **g** : Stores the cost for the distance-to-come to current node from start
5. **h** : Stores the cost to-reach the goal from current node (heuristics)
6. **cost** : Stores the total cost of the node (g + h)
7. **parent** : Stores the information of the parent node

As mentioned the grid is explored in the following order: **right, down, left** and **up**

For all the four algorithms, namely, BFS, DFS, Dijkstra and A*, three different types of nodes exist at any given time. They are as follows:

1. **Explored** : The node has been explored and all its neighbours have also been analysed.
2. **Unexplored** : The node is yet to be explored and its information is not available.
3. **Queued_node** : The nodes are currently under exploration and its neighbours yet to be analysed.

Two important data structures are used in the implementation of the following algorithms:

1. **Queue** : This data structure uses the concept of **First-In-First-Out (FIFO)**, like a queue in the real world the element that comes first is removed first from the queue.
2. **Stack** : This data structure uses the concept of **Last-In-First-Out (LIFO)**. The element added last the stack is the first one to be removed.

The following are the data structures used in the four algorithms to store the Queued_nodes:

1. **BFS** uses "**queue"** data structure to store the Queued_nodes
2. **DFS** uses **"stack"** data structure to store the Queued_nodes
3. **Dijkstra** uses **"Priority-queue"** data structure. The order of the nodes in the Priority-queue are sorted in the ascending order of the **cost-to-come** to the current node from the starting node.
4. **A*** also uses **"Priority-queue"** data structure. The order of the nodes in the Priority-queue are sorted in the ascending order of the **Total cost (= cost-to-come + cost-to-reach-goal** (heuristics)**).**

Breadth-First Search:

Breadth first search algorithm initiates from the start node. It explores the neighbours of all the nodes present at a particular distance from the start node consecutively, before going to the next depth distance from the start node. This process goes on until the goal node is found and the loop algorithm then gets terminated. This algorithm is efficient to use when the goal is near to the start node i.e. the distance of the goal from the start is less.

Algorithm:

1. BFS starts by initialising the queue with the start node.
2. The first node of the queue is then added to the explored list.
3. The algorithms enters the loop.
4. The first node of the queue is removed and assigned to a current node
5. If the current node if checked if it is the goal node, if yes, the loop get terminated, else the code continues
6. The algorithm check if the neighbouring nodes are obstacle and within the bound of the graph/grid or have been already explored.
7. If the neighbour node is not an obstacle and not yet explored, then it is added to the queue and added to the explored list.
8. The parameters of the neighbour node is initialised and the parent is set as current node.
9. The loop continues until the queue is empty
10. The path is traced back from the goal node to the start node and then printed out.

Pseudo-code:

```
Function init_graph(grid, start)
        for all the cells in the grid
                create a new node
                initialise the node values

Function search_explored(current_node, explored)
        Flag = False
        For node in explored
                If node found is explored
                        Flag = true
                        Break
        return Flag

Function bfs(grid, goal, start)
        Initialise start and goal node
        Checking if goal is a valid point within the map
        Graph = init_graph(grid, start)   # The grid is converted into a graph of nodes
        Explore = list(start)
        Queue  = list(start)
        While length(Queue) > 0
                Counting steps
                Current_node = queue(0) # Assigning the first element to be explored
                Queue.pop(0)  # Removing the first element from the queue
                If node == goal
                        Break
                For neighbours in adjacent(Graph.Current_node)
                        If neighbour is not obstacle and search_explored(Explore) == False
                                Update neighbour node info
                                Explore.add(neighbour)
                                Queue.add(neighbour)

        Path = traceback()
        return path, steps
```

Depth-First Search:

Depth first search algorithm initiates from the start node. It explores the all the nodes present in a particular branch before going to the next branch from the start node. This process goes on until the goal node is found and the loop algorithm then gets terminated. This algorithm is efficient to use when the goal is in the branched of graph that is explored first.

Algorithm:

1. DFS starts by initialising the stack with the start node.
2. The algorithms enters the loop.
3. The top node of the stack is then added to the current node and then is removed.
4. If the current node if checked if it is the goal node, if yes, the loop get terminated, else the code continues
5. The current node is then added to the explored list.
6. The algorithm check if the neighbouring nodes are obstacle and within the bound of the graph/grid or have been already explored.
7. If the neighbour node is not an obstacle and not yet explored, then it is added to the queue.
8. The parameters of the neighbour node is initialised and the parent is set as current node.
9. The loop continues until the stack is empty
10. The path is traced back from the goal node to the start node and then printed out.

Pseudo-code:

```
Function init_graph(grid, start)
        for all the cells in the grid
                create a new node
                initialise the node values

Function search_explored(current_node, explored)
        Flag = False
        For node in explored
                If node found is explored
                        Flag = true
                        Break
        return Flag

Function dfs(grid, goal, start)
        Initialise start and goal node
        Checking if goal is a valid point within the map
        Graph = init_graph(grid, start)   # The grid is converted into a graph of nodes
        Explore = list()
        Stack  = list(start)

        While length(Stack) > 0
                Counting steps
                Current_node = stack.pop        # Assigning the first element to be explored
                                                # and removing the first element from the queue
                If node == goal
                        Break
                Explore.add(neighbour)
                For neighbours in adjacent(Graph.Current_node)
                        If neighbour is not obstacle and search_explored(Explore) == False
                                Update neighbour node info
                                Queue.add(neighbour)

        Path = traceback()
        return path, steps
```

Dijkstra Search:

Dijkstra search algorithm is used to find the shortest path between two nodes in a graph with the minimum cost to travel. It also initiates from the start node. It explores the neighbours of all the nodes present at a particular distance from the start node consecutively, before going to the next depth distance from the start node. However, as it search the neighbours, it updates the parent of the explored node with the node that gives the minimum cost-to-come to current node. This process goes on until the goal node is found and the loop algorithm then gets terminated. This algorithm is efficient to use than the BFS algorithm as this helps to find the optimal and minimum cost path to reach goal from the start position.

Algorithm:

1. Dijkstra starts by initialising the queue with the start node.
2. The algorithms enters the loop.
3. Th queue is sorted with respect to the **cost-to-come (g)**
4. The first node of the queue is removed and assigned to a current node
5. If the current node if checked if it is the goal node, if yes, the loop get terminated, else the code continues
6. The algorithm check if the neighbouring nodes are obstacle and within the bound of the graph/grid
7. The node is considered if the condition above is satisfied
8. The cost of neighbour node is checked if less than the cost to travel from current node.
9. If the cost is less than the node's g then the parameters of the neighbour node is initialised and the parent is set as current node.
10. Then the node is appended to the Queue
11. The loop continues until the queue is empty
12. The path is traced back from the goal node to the start node and then printed out.

Pseudo-code:

```
Function init_graph(grid, start)
        for all the cells in the grid
                create a new node
                initialise the node values


Function Dijkstra(grid, goal, start)
        Initialise start and goal node
        Checking if goal is a valid point within the map
        Graph = init_graph(grid, start)   # The grid is converted into a graph of nodes
        Queue  = list(start)

        While length(Queue) > 0
                Queue = Sorted(Queue)
                Current_node  = Queue.pop(0) # Saving the first element to be explored and removing node
                Counting steps
                If node == goal
                        Break
                For neighbours in adjacent(Graph.Current_node)
                        If neighbour is not obstacle and within the bounds of the graph:
                                If (current_node.g + cost-to-move) < neighbour.g
                                        Update neighbour node info
                                        Queue.add(neighbour)

        Path = traceback()
        return path, steps
```

A* Search:

A* is an informed search algorithm algorithm. It updates the nodes based on the cost-to-come to a node. That is it searches the neighbours, it updates the parent of the explored node with the node that gives the minimum cost-to-come to reach that node. However, the Queue is sort with respect to the total cost to reach the node and the nodes are explored based on this cost. The total cost is the cost-to-come to that node plus the heuristic cost (I.e. in our case the manhattan distance of the node from the goal node). This process goes on until the goal node is found and the loop algorithm then gets terminated. This algorithm finds the sub-optimal path to reach the goal, however it is faster than the Dijkstra algorithm by few steps.

Algorithm:

1. Dijkstra starts by initialising the queue with the start node.
2. The algorithms enters the loop.
3. Th queue is sorted with respect to the **cost-to-come (g)**
4. The first node of the queue is removed and assigned to a current node
5. If the current node if checked if it is the goal node, if yes, the loop get terminated, else the code continues
6. The algorithm check if the neighbouring nodes are obstacle and within the bound of the graph/grid
7. The node is considered if the condition above is satisfied
8. The cost of neighbour node is checked if less than the cost to travel from current node.
9. If the cost is less than the node's g then the parameters of the neighbour node is initialised and the parent is set as current node.
10. Then the node is appended to the Queue
11. The loop continues until the queue is empty
12. The path is traced back from the goal node to the start node and then printed out.

Pseudo-code:

```
Function init_graph(grid, start)
        for all the cells in the grid
                create a new node
                initialise the node values

Function Astar(grid, goal, start)
        Initialise start and goal node
        Checking if goal is a valid point within the map
        Graph = init_graph(grid, start)   # The grid is converted into a graph of nodes
        Queue  = list(start)

        While length(Queue) > 0
                Queue = Sorted(Queue)
                Current_node  = Queue.pop(0) # Saving the first element to be explored and removing node
                Counting steps
                If node == goal
                        Break
                For neighbours in adjacent(Graph.Current_node)
                        If neighbour is not obstacle and within the bounds of the graph:
                                If (current_node.g + cost-to-move) < neighbour.g
                                        Update neighbour node info
                                        Queue.add(neighbour)

        Path = traceback()
        return path, steps
```

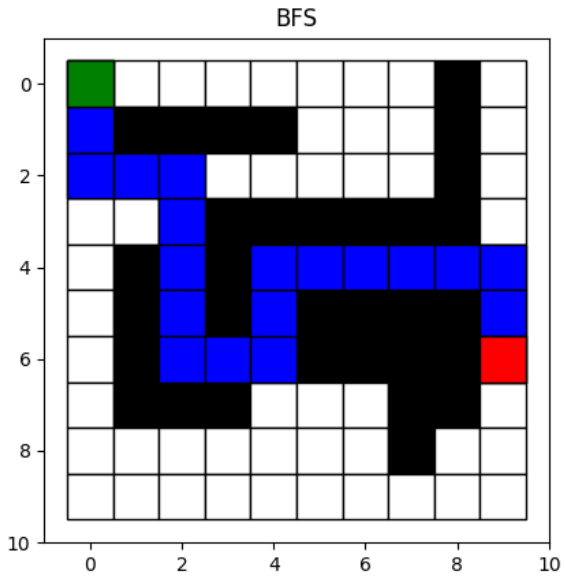
Results:

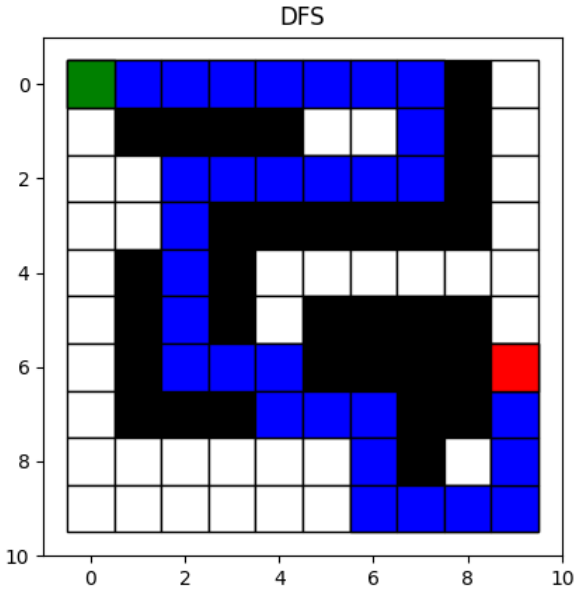
Test Case 1: When a valid goal is given to the search algorithm

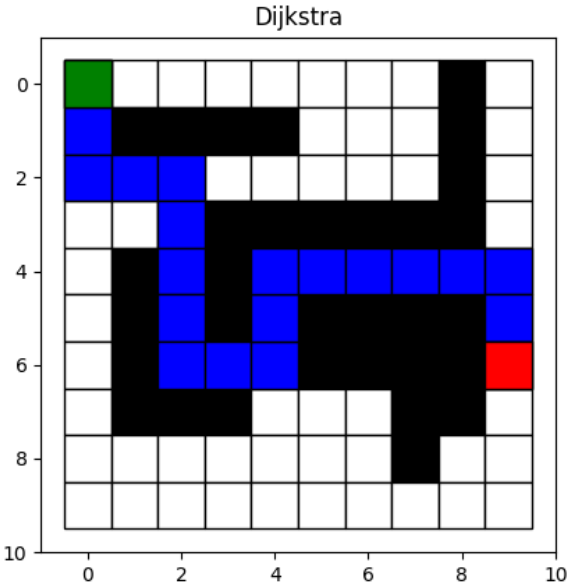
After completing the code in the search.py file, the main.py file was run from the command prompt and the following results were obtained:

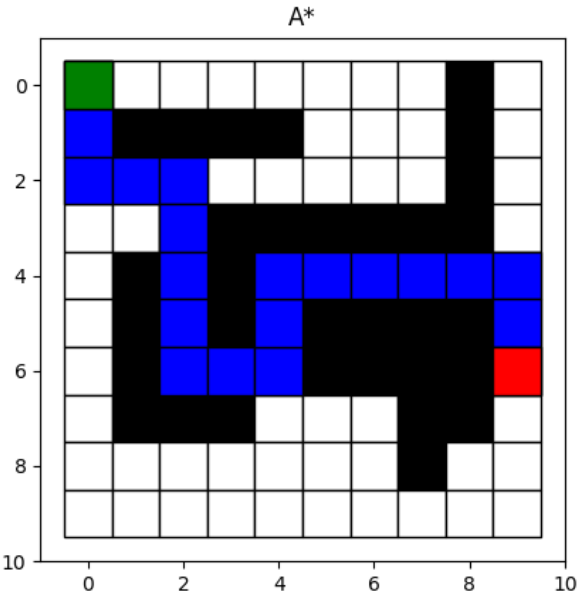
```
C:\Windows\System32\cmd.exe - python  main.py
Microsoft Windows [Version 10.0.19044.1466]
(c) Microsoft Corporation. All rights reserved.

D:\Sem_2 (Spring_2022)\RBE550 - Motion Planning\Lec 3 - Planning for point robots (2D) & HW1\Basic Search Algorithms>pyt
hon search.py

D:\Sem_2 (Spring_2022)\RBE550 - Motion Planning\Lec 3 - Planning for point robots (2D) & HW1\Basic Search Algorithms>pyt
hon main.py
It takes 64 steps to find a path using BFS
It takes 33 steps to find a path using DFS
It takes 64 steps to find a path using Dijkstra
It takes 52 steps to find a path using A*
```

The following paths were obtained for the testing map file:

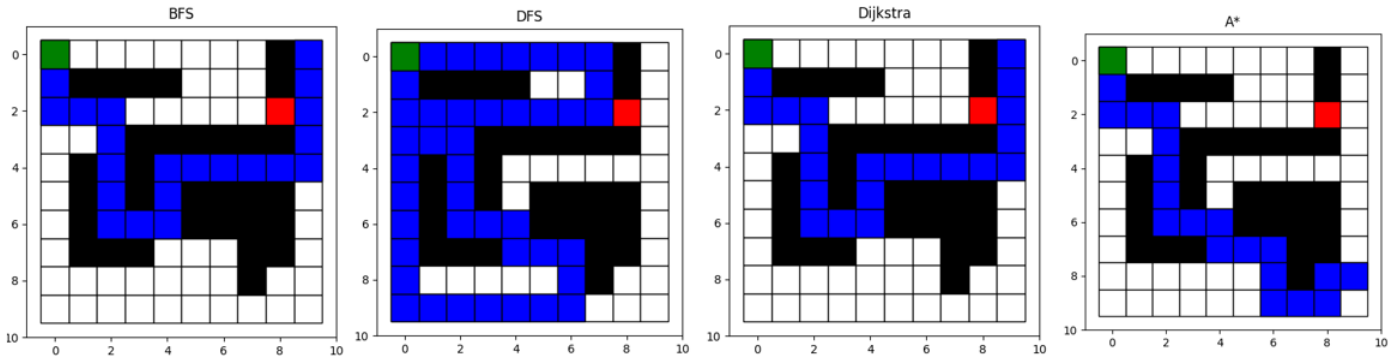








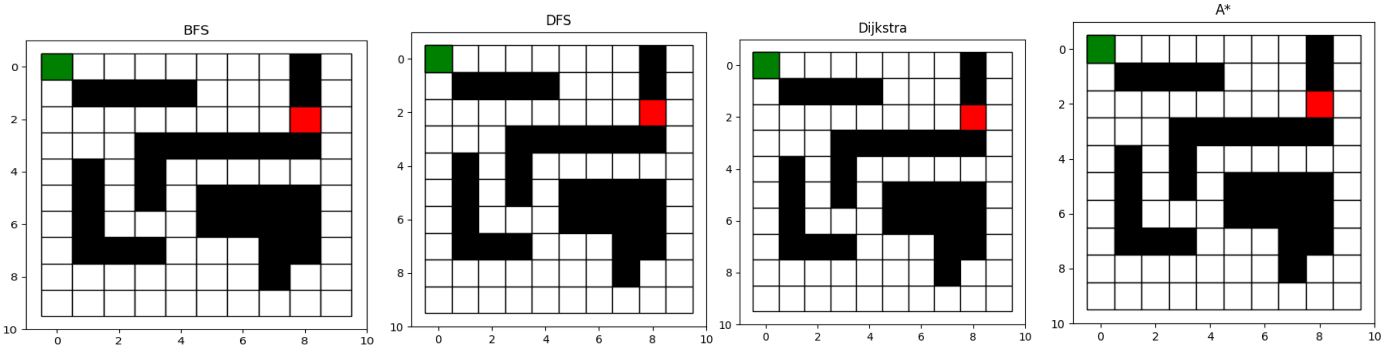
Test Case 2: When a goal is an obstacle



The result of the path was displayed as below:

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19044.1466]
(c) Microsoft Corporation. All rights reserved.

D:\Sem_2 (Spring_2022)\RBE550 - Motion Planning\Lec 3 - Planning for poi
hon main.py
No path found
No path found
No path found
No path found
```

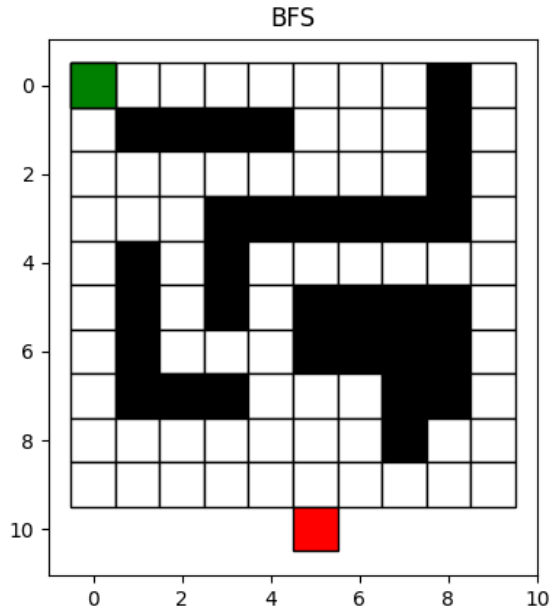
After condition to check if the goal is obstacle was updated the following plots and output were obtained:

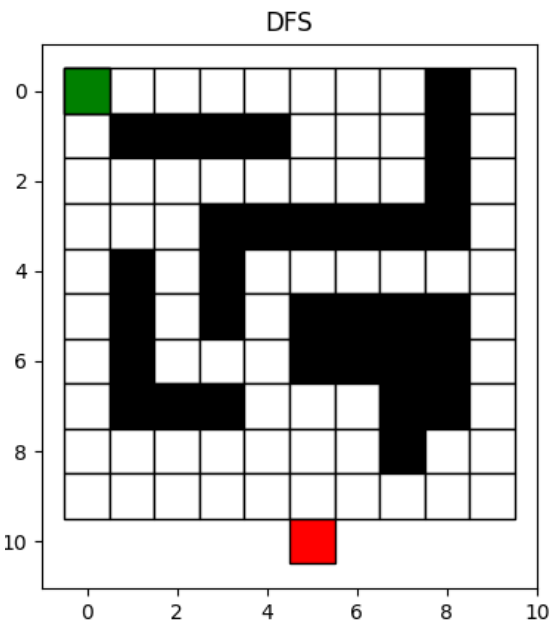


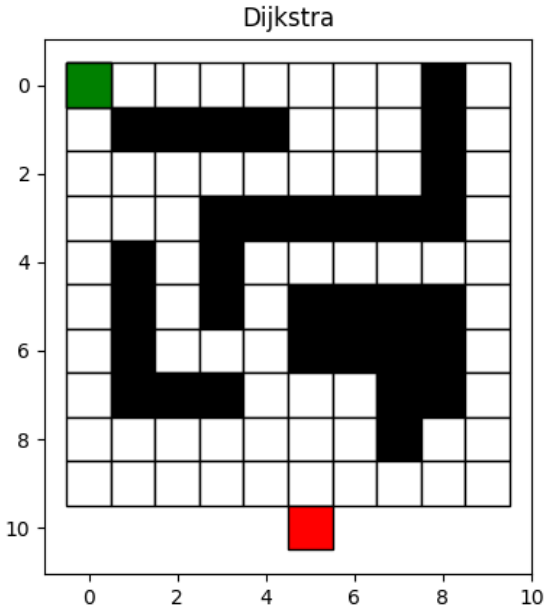
The result of the path was displayed as below:

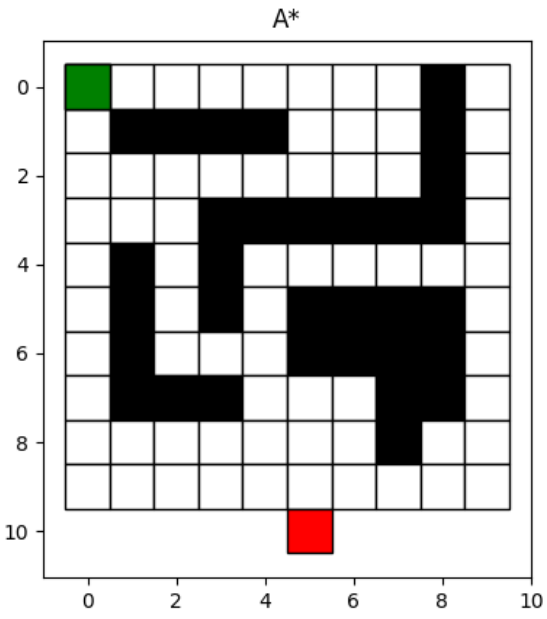
```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19044.1466]
(c) Microsoft Corporation. All rights reserved.

D:\Sem_2 (Spring_2022)\RBE550 - Motion Planning\Lec 3 - Planni
hon main.py
No path found
No path found
No path found
No path found
```

Test Case 3: When a goal is outside the grid

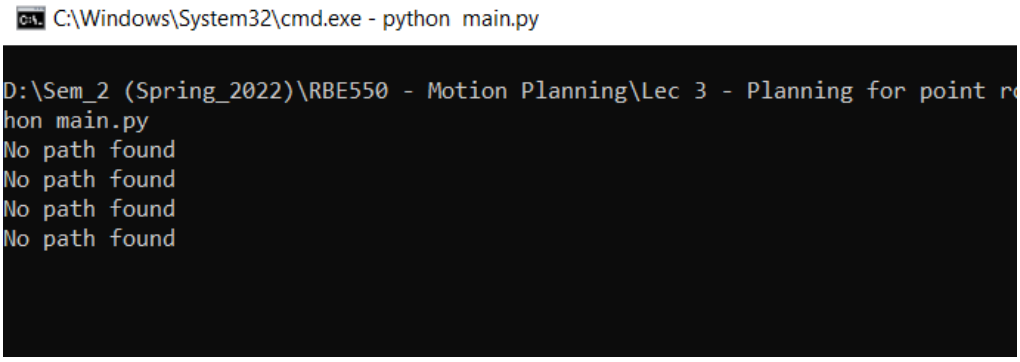








The output from the terminal is as follows:



Observations:

Test Case 1:

1. We have valid goal position that is within the map and not an obstacle.
2. All the four algorithms were able to reach the goal position and with varying number of steps as follows:
    a. Breath-First Search - 64 Steps
    b. Depth-First Search - 33 Steps
    c. Dijkstra Search - 64 Steps
    d. A* Search - 52 Steps
3. It can be observed that the BFS, Dijkstra and A* algorithm follow the same path from start to goal. This is because BFS is the base framework for both Dijkstra and A* algorithm
4. From above it can be observed that both BFS and Dijkstra take the same number of steps. This is usually not the case as Dijkstra algorithm takes a smaller number of steps as well as more optimum path based on the **cost-to-come** from the start position. In our case the cause to move has been considered as **1** for all the nodes and thus the Dijkstra algorithm becomes similar to the BFS algorithm.
5. DFS algorithm has a different path because it explores branch-wise unlike the other algorithms. Moreover, DFS has a smaller number of steps in this case. The number of steps purely depends on the position of the goal. Had the goal been somewhere in the bottom left corner, then it might have taken much longer step to reach goal.
6. The same above also applies to other algorithms. If the goal is changed, then the algorithms might reach it in different number of steps as seen above.
7. Though A* has same base as BFS and Dijkstra, it takes lesser number of steps to reach the goal because of the heuristics cost, i.e., the cost calculated from the goal to the nodes. It helps the A* algorithm to navigate faster, however the result might be a sub-optimal solution. The corner case where A* can perform poorly compared to other algorithms is when it is stuck in between obstacles. The algorithm might take longer number of steps to come out of it.

Test Case 2:

The goal is assigned to an obstacle, so there is no valid path to be evaluated between the start and the goal nodes. Thus, the as no path is found, the algorithm prints **“No path found”**

Test Case 3:

The goal is assigned values outside the grid map. Therefore, no valid path can be found in this scenario as well. Thus, the algorithm print **“No path found”.**

Conclusion:

A **complete algorithm** is an algorithm is one which can output if a path exists or not for a given map, start and goal position, with confidence. The above four algorithms being able to satisfy all the above three test cases ensure, that the algorithm implementation has made it a complete algorithm.

References:

1. BFS Wiki : https://en.wikipedia.org/wiki/Breadth-first_search
2. DFS Wiki : https://en.wikipedia.org/wiki/Depth-first_search
3. Dijkstra Wiki : https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
4. A* Wiki : https://en.wikipedia.org/wiki/A*_search_algorithm
5. RBE550 Motion planning slides
6. Udemy Robotics Course