}

Sumeet Shanbhag

Motion Planning Assignment 1

Pseudocode of recursive Breadth-First search algorithm

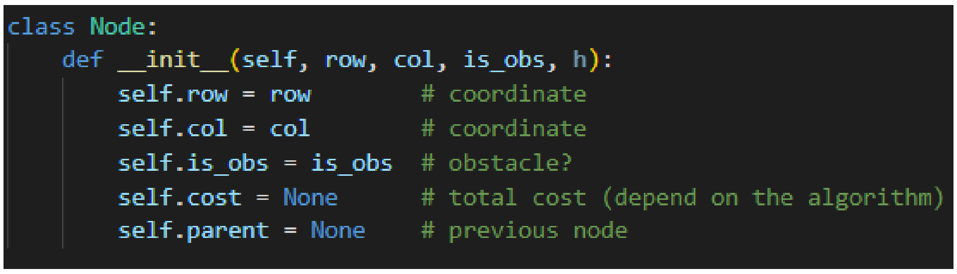
```
Breadth_First_Search( G, A ) // G ie the graph and A is the source node
        Let q be the queue
        q.enqueue( A ) // Inserting source node A to the queue
        Mark A node as visited.
        While ( q is not empty )
        B = q.dequeue( ) // Removing that vertex from the queue, which will be visited by its
neighbour
Processing all the neighbors of B
For all neighbors of C of B
          If C is not visited, q. enqueue( C ) //Stores C in q to visit its neighbour
Mark C a visited
```

Pseudocode of recursive Depth-First search algorithm

```
Depth_First_Search(matrix[ ][ ] ,source_node, visited, value)
{
 If ( sourcce_node == value)
 return true // we found the value
 visited[source_node] = True
 for node in matrix[source_node]:
   If visited [ node ] == false
   Depth_first_search ( matrix, node, visited)
   end if
end for
return false //If it gets to this point, it means that all nodes have been explored.
               //And we haven't located the value yet.
}
```

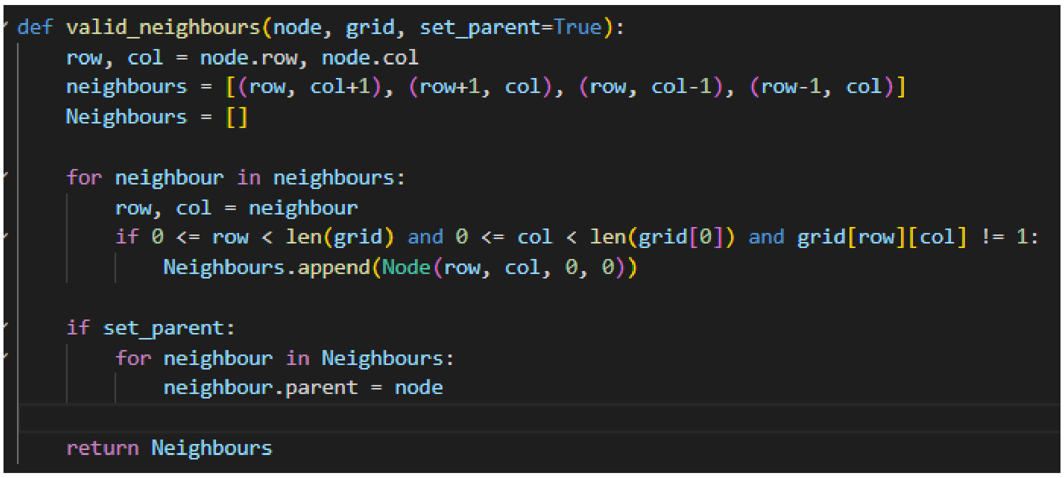
Code Explanation :

The **Node** class represents each node in the grid, storing its coordinates (row, col), whether it is an obstacle (is_obs), the total cost (cost), and its parent node (parent) which is the previous node visited.



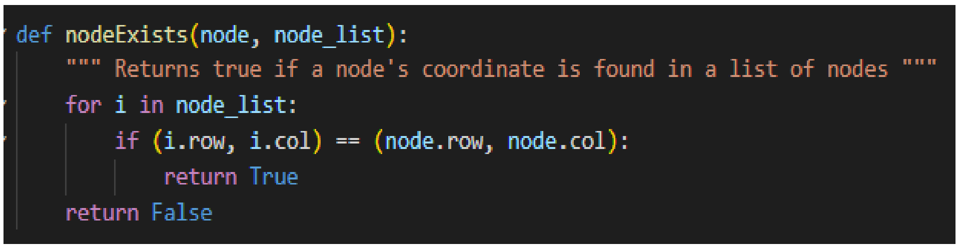
```
class Node:
    def __init__(self, row, col, is_obs, h):
        self.row = row          # coordinate
        self.col = col          # coordinate
        self.is_obs = is_obs    # obstacle?
        self.cost = None        # total cost (depend on the algorithm)
        self.parent = None      # previous node
```

valid_neighbours function takes a node, a grid and a boolean value (set_parent). It returns a list of neighbor nodes (neighbours) that can be visited from the current node. The function checks whether a node is in the grid and whether it is an obstacle, then creates a new node with the same row and column values as the neighbor with the cost and parent set to None. It also sets the parent of each neighbor to the current node if set_parent is True.



```
def valid_neighbours(node, grid, set_parent=True):
    row, col = node.row, node.col
    neighbours = [(row, col+1), (row+1, col), (row, col-1), (row-1, col)]
    Neighbours = []

    for neighbour in neighbours:
        row, col = neighbour
        if 0 <= row < len(grid) and 0 <= col < len(grid[0]) and grid[row][col] != 1:
            Neighbours.append(Node(row, col, 0, 0))

    if set_parent:
        for neighbour in Neighbours:
            neighbour.parent = node

    return Neighbours
```

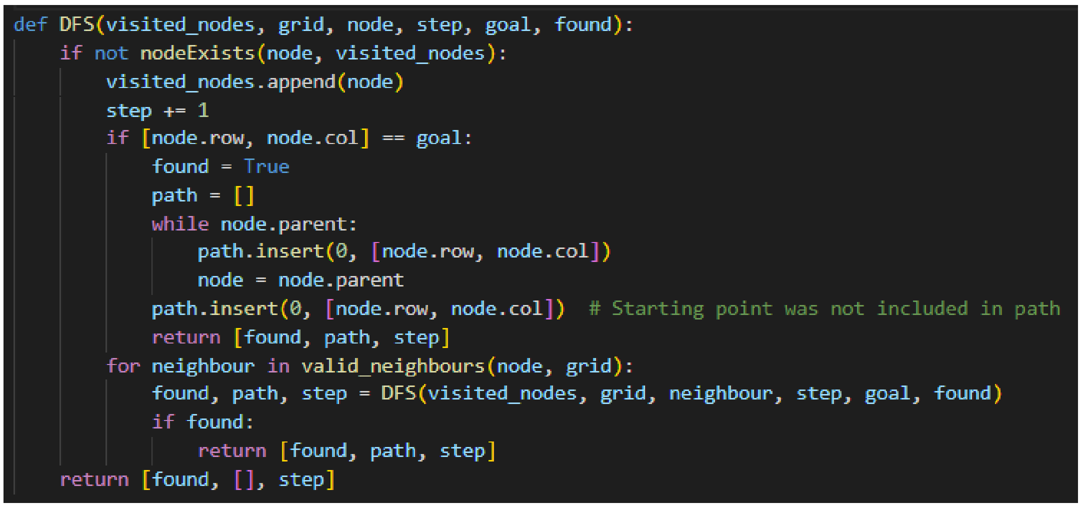
nodeExists function checks if a node is in the list of nodes. It returns True if the node's coordinates match any node in the list.



```
def nodeExists(node, node_list):
    """ Returns true if a node's coordinate is found in a list of nodes """
    for i in node_list:
        if (i.row, i.col) == (node.row, node.col):
            return True
    return False
```

DFS function performs the depth-first search algorithm. It takes the list of visited nodes, a grid, the current node, the number of steps (step), the goal node, and a boolean value (found) as input. It returns a list of three values [found, path, step], where found is True if a path from start to goal is found, path is a list of coordinates from start to goal, and step is the number of nodes visited before finding the path.

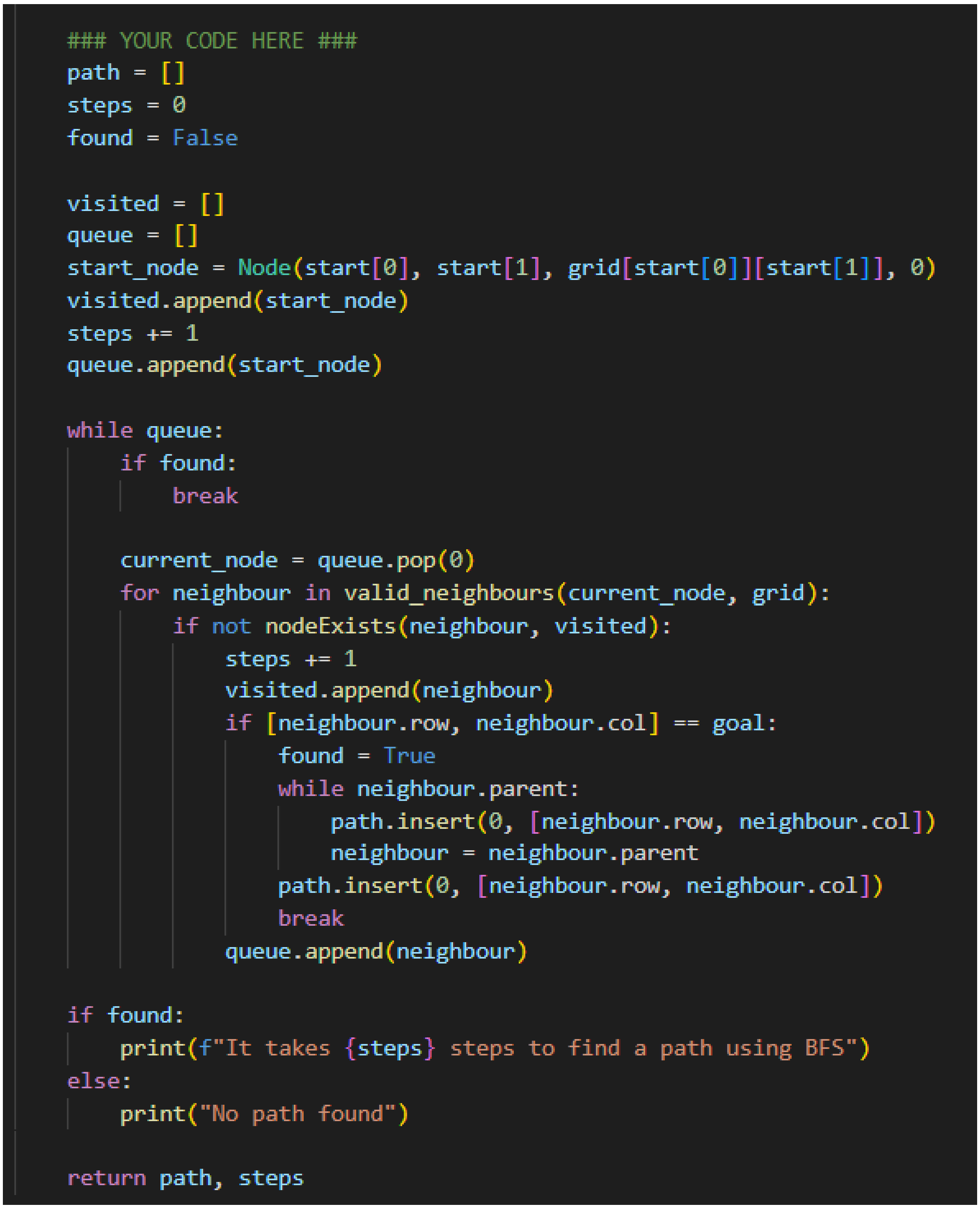
The function checks if the current node is in the visited nodes list. If not, it adds it to the visited nodes list and increments the step. If the current node is the goal node, it sets found to True and returns the path to the goal node by traversing the parent nodes from goal to start. Otherwise, it loops through the neighbors of the current node, calls the DFS function recursively for each neighbor, and updates the found, path, and step variables if the path to the goal is found.



```
def DFS(visited_nodes, grid, node, step, goal, found):
    if not nodeExists(node, visited_nodes):
        visited_nodes.append(node)
        step += 1
        if [node.row, node.col] == goal:
            found = True
            path = []
            while node.parent:
                path.insert(0, [node.row, node.col])
                node = node.parent
            path.insert(0, [node.row, node.col])  # Starting point was not included in path
            return [found, path, step]
        for neighbour in valid_neighbours(node, grid):
            found, path, step = DFS(visited_nodes, grid, neighbour, step, goal, found)
            if found:
                return [found, path, step]
    return [found, [], step]
```

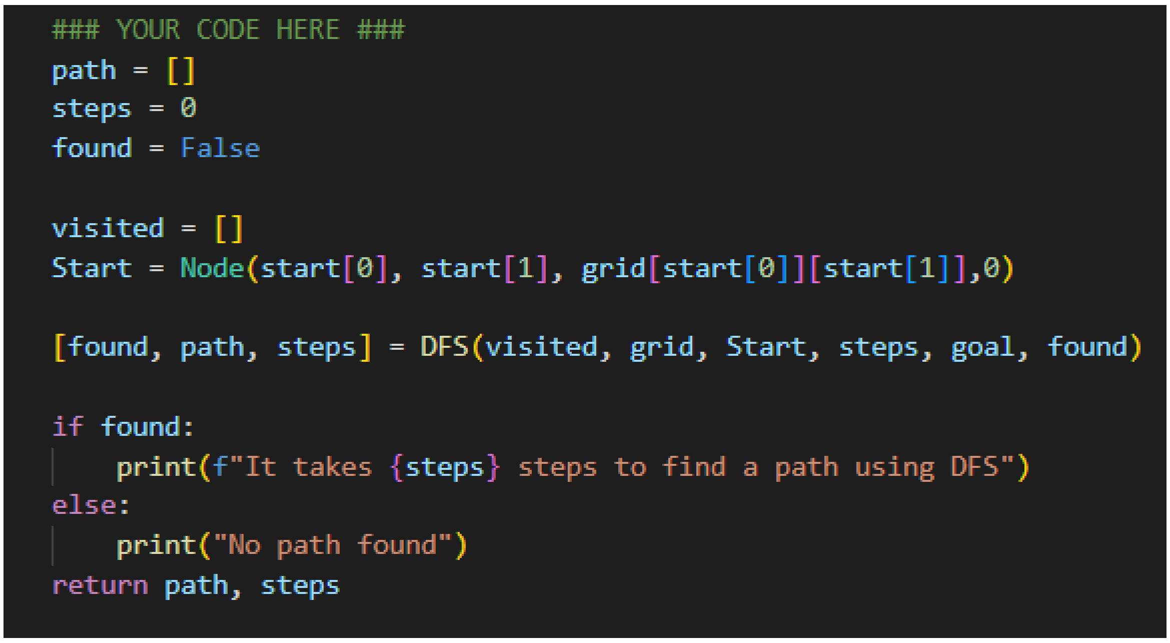
bfs function performs the breadth-first search algorithm. It takes a grid, the start node, and the goal node as input. It returns a list of two values [path, steps], where path is a list of coordinates from start to goal, and steps is the number of nodes visited before finding the path.

The function initializes the path and steps variables to empty lists and zero, respectively. It also initializes a found variable to False, a visited list containing the start node, and a queue containing the start node. The function then enters a while loop, which runs as long as the queue is not empty and the path to the goal is not found. The function dequeues the first node in the queue and loops through its neighbors. For each unvisited neighbor, the function adds it to the visited list, increments the steps variable, and checks if it is the goal node. If it is, the function sets found to True, constructs the path by traversing the parent nodes from the goal to the start node, and breaks out of the loop. Otherwise, it adds the neighbor to the queue. If the goal node is found, the function returns the path and the steps variables.



```python
    ### YOUR CODE HERE ###
    path = []
    steps = 0
    found = False

    visited = []
    queue = []
    start_node = Node(start[0], start[1], grid[start[0]][start[1]], 0)
    visited.append(start_node)
    steps += 1
    queue.append(start_node)

    while queue:
        if found:
            break

        current_node = queue.pop(0)
        for neighbour in valid_neighbours(current_node, grid):
            if not nodeExists(neighbour, visited):
                steps += 1
                visited.append(neighbour)
                if [neighbour.row, neighbour.col] == goal:
                    found = True
                    while neighbour.parent:
                        path.insert(0, [neighbour.row, neighbour.col])
                        neighbour = neighbour.parent
                    path.insert(0, [neighbour.row, neighbour.col])
                    break
                queue.append(neighbour)

    if found:
        print(f"It takes {steps} steps to find a path using BFS")
    else:
        print("No path found")

    return path, steps
```

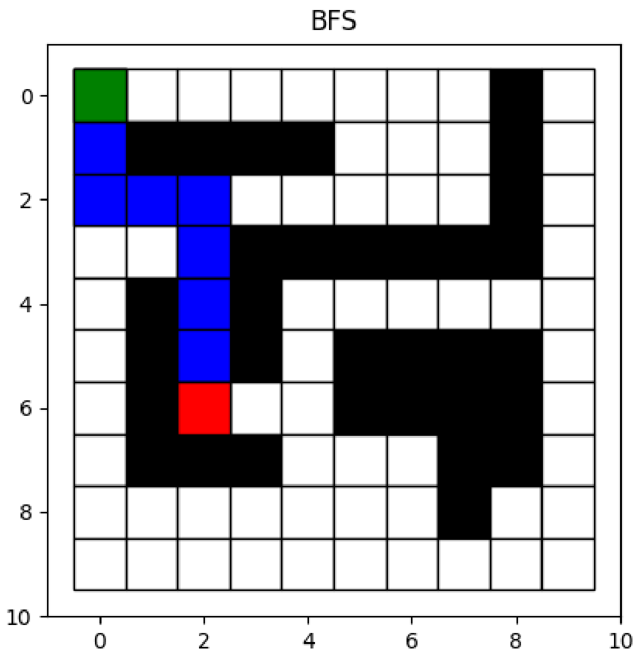
dfs function performs the depth-first search algorithm. It takes a grid, the start node, and the goal node as input. It returns a list of two values [path, steps], where path is a list of coordinates from start to goal, and steps is the number of nodes visited before finding the path.

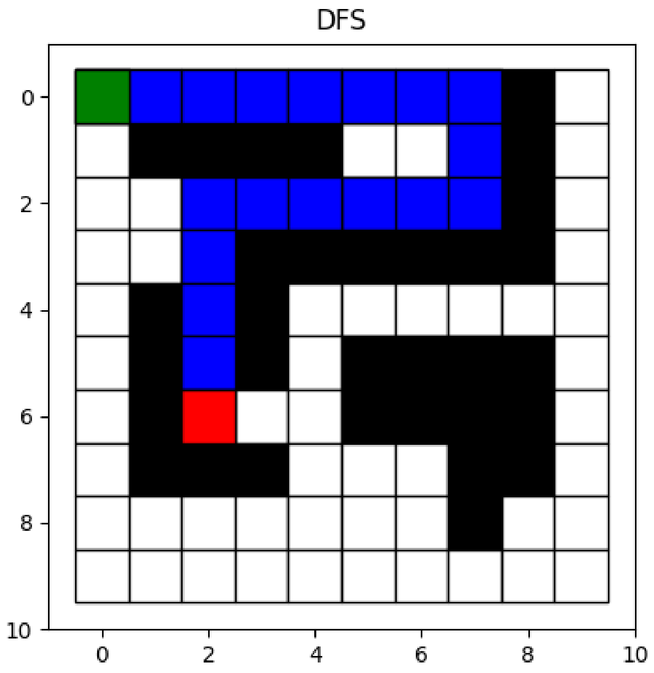


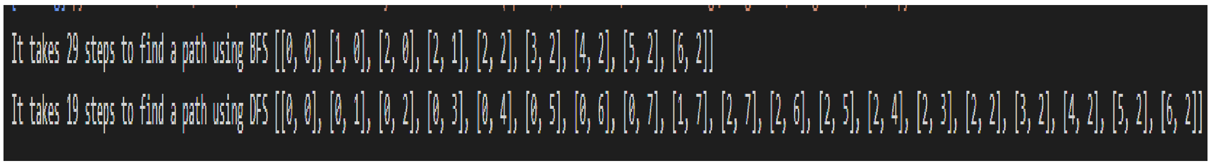
```python
### YOUR CODE HERE ###
path = []
steps = 0
found = False

visited = []
Start = Node(start[0], start[1], grid[start[0]][start[1]],0)

[found, path, steps] = DFS(visited, grid, Start, steps, goal, found)

if found:
    print(f"It takes {steps} steps to find a path using DFS")
else:
    print("No path found")
return path, steps
```

What is BFS and DFS and their differences?

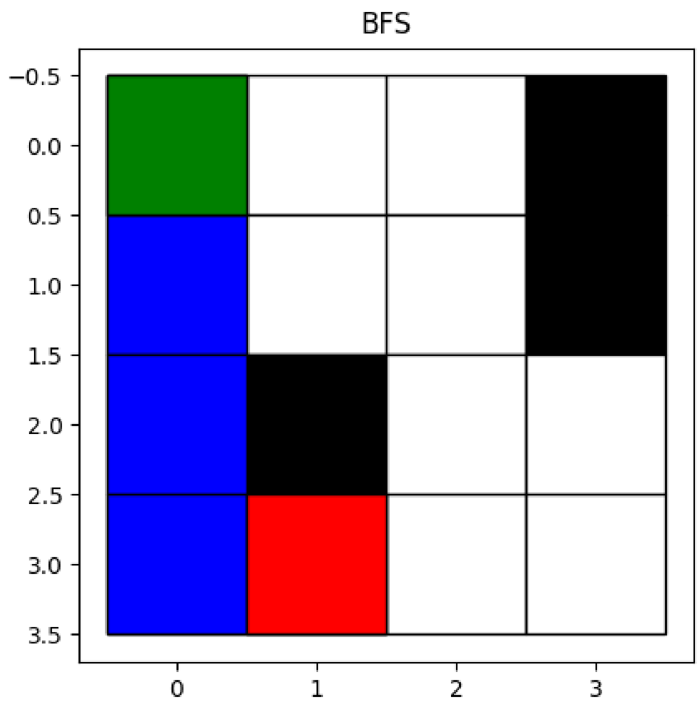
Breadth First Search	**Depth First Search**
BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
Here, the nodes that are on the same level or quite simply the sibling node is visited first.	Here, the child nodes are visited before the sibling nodes.
BFS requires more memory.	DFS requires less memory.
BFS is optimal for finding the shortest path.	DFS does not necessarily give the shortest path.

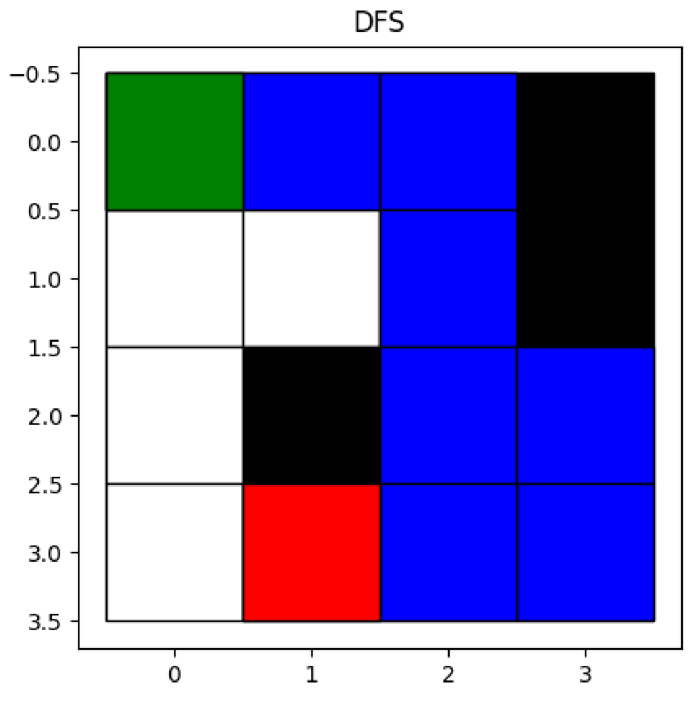


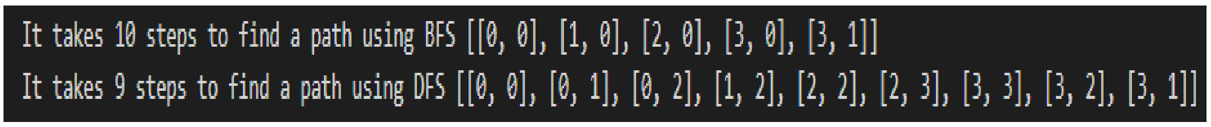




```
It takes 29 steps to find a path using BFS [[0, 0], [1, 0], [2, 0], [2, 1], [2, 2], [3, 2], [4, 2], [5, 2], [6, 2]]
It takes 19 steps to find a path using DFS [[0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [0, 5], [0, 6], [0, 7], [1, 7], [2, 7], [2, 6], [2, 5], [2, 4], [2, 3], [2, 2], [3, 2], [4, 2], [5, 2], [6, 2]]
```







```
It takes 10 steps to find a path using BFS [[0, 0], [1, 0], [2, 0], [3, 0], [3, 1]]
It takes 9 steps to find a path using DFS [[0, 0], [0, 1], [0, 2], [1, 2], [2, 2], [2, 3], [3, 3], [3, 2], [3, 1]]
```

References : -

- https://www.simplilearn.com/tutorials/data-structure-tutorial/bfs-algorithm
- https://www.simplilearn.com/tutorials/data-structure-tutorial/dfs-algorithm
- https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/