Motion Planning - Basic Algorithm Implementation

-Sailesh Rajagopalan

Breadth First Search

BFS algorithm starts with the vertex first lowest level and it discovers other vertices that it can approach from the initial vertex and keeps them marked, Similarly from the next level it marks the vertices after identifying them and making them as visited noted, The process is continuous and runs till there are not new vertices that are present and the goal is reached

We initialise a que and start node will be explored and appended to the que and then the queue will be queued and checked for it to be a goal or not then the algorithm checks the neighbouring nodes and if they haven't been previous visited they will be added to the queue and the loop will continue until there is nothing more left.

<u>Pseudocode:</u>

```
Function BFS(grid,start,goal)
        Define start node
        Initialise queue, previous nodes

        while True
                Check if goal or not
                Backtrack and reverse
        For coordinate for all valid nodes
                Check if the queue is empty
                previous_nodes.add(node.coordinate) add all nodes to avoid repeat
                Print steps taken
                Or no path found
Return function
```

Depth First Search

The DFS algorithm is when the traversal in a tree or graph starts with the root node and it does exploration of every node as far as possible by marking them visited and then moving to the adjacent unmarked node and continuing the same process until there are no unmarked node present, the important aspect is backtracking and checking for no unmarked nodes for traversal is present.

A stack is initialised and the start node is appended to the stack and the loop starts. The stack will be popped to check if it is goal or not and if it's a goal, the loop will break or the loop continues checking the neighbouring loops and if they're not previously visited the stack is added and the loop continues till there are no explorations to do.

<u>Pseudocode:</u>

```
Function DFS(grid, start,goal):
    Define start node
    Add start node to previous node

While condition:
    Check if goal or not
    Backtracking and reverse

    Get all valid neighbour nodes
    Check queue priority

Print steps if found
Return function
```

Dijkstra Algorithm

The dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, fixing the first as a source node and finding the shortest paths from source node to all other existent nodes and traversing.

A graph is initialised and the total cost and cost to come for the nodes are and the queue is initialised entering the loop the priority queue us dequeued and checked if it is a goal or not if it is the loop will break or the cost to come concept is updated and the total cost is calculated and added to the priority queue, the loop will continue until the priority queue is empty

<u>PseudoCode:</u>

```
Function (grid, starter, goal):
    Define start node
    Priority queue initialization
    Adding starting node to set

    While condition:
        Check if its goal or not
        Backtrack and append
    Get all valid neighbours
    Sort based on neighbour nides
    Check for visited or not
    Update priority queue
    Print steps
Return function
```

A*

It is a graph traversal search algorithm, it updates nodes based on cost to come, which is similar to dijkstra. If the cost to come is less than the cost of the discovered node the cost of the discovered node is updated. The algorithm starts exploring node with least total cost is the sum of cost to come and cost to reach goal, A* finds sub optimal solutions to pathfinding problems.

Graph is initialised and the total cost and cost to comes, along with the heuristic for each nodes, the priority queue is initialised and the loop staters where queue is dequeued and checked for node if its goal or not, if it is breaks out of loop or it checked for less cost to come and update. The total code is calculated for every node and added to the priority queue and the loop continues till its empty.

Pseudocode:

```
Function(grid, start, goal):
    Define starting node
    Initialise previous and start node appended
    While loop:
        If the node is goal or not
        Backtrack

    Initialise node list to sorted based on heuristics
    Get all valid neighbours

    Sort nodes based on heuristic
    Check previous and duplicate

    If statement:
        The queue is empty
        Break

    Print steps
Return function
```

Differences:

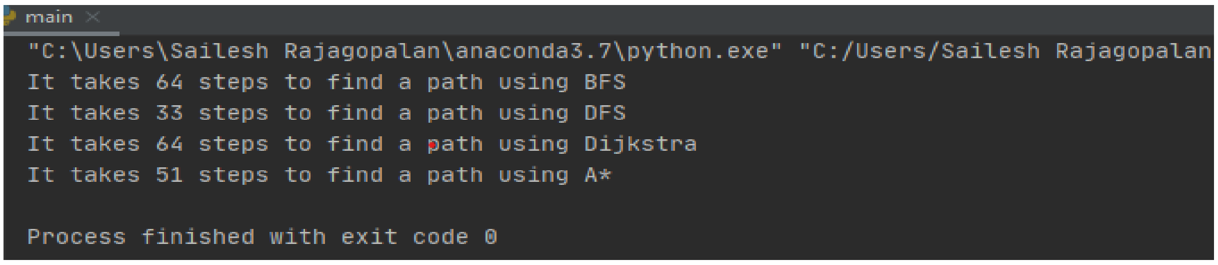
The four algorithms here BFS, DFS, Dijkstra and A* are all forward seach algorithms and always have unvisited nodes or states, Dead (states which have been visited every possible next state been visited) and alive (stated that have been visited and possibly some adjacent have not been visited).

BFS and DFS are very similar other than

- BFS checks the node as explored before enqueueing the node, while DFS checks the node as explored only after dequeuing
- BFS searches depth wise, hence it uses queue, whereas DFS searches branch wise, hence it uses stack

BFS, Dijkstra and A*:
Difference - Dijkstra and A* use priority queue but BFS does not, Djikstra and A* update the cost of exploring nodes if they are less than their previous value.

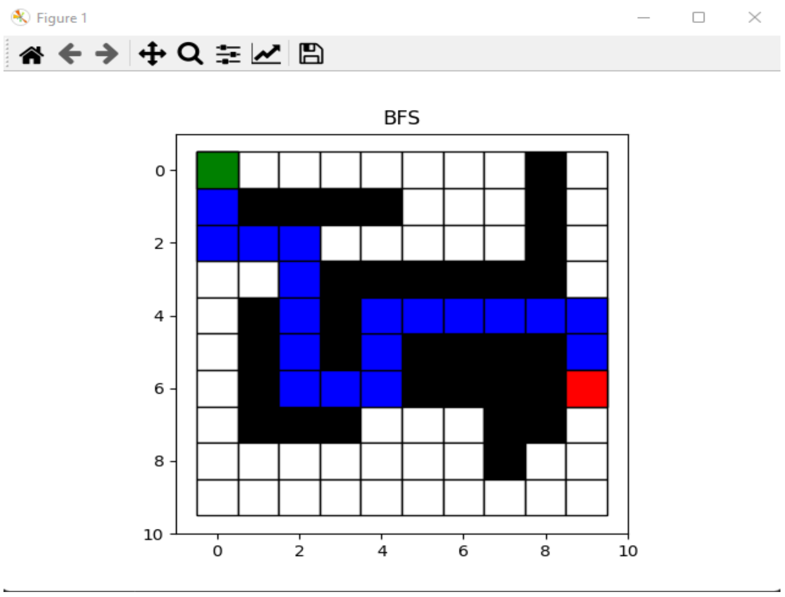
Results:



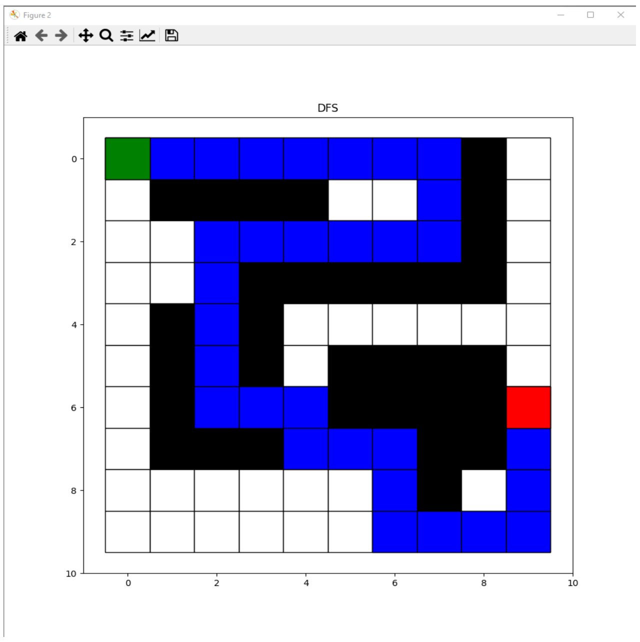
```
"C:\Users\Sailesh Rajagopalan\anaconda3.7\python.exe" "C:/Users/Sailesh Rajagopalan
It takes 64 steps to find a path using BFS
It takes 33 steps to find a path using DFS
It takes 64 steps to find a path using Dijkstra
It takes 51 steps to find a path using A*

Process finished with exit code 0
```

BFS



DFS



Djikstra

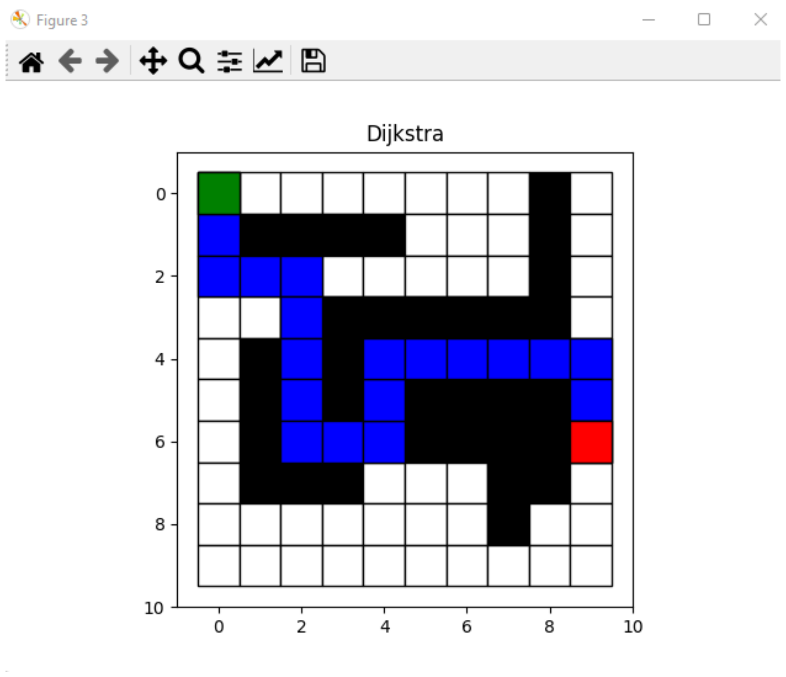


Figure 3
Dijkstra
0
2
4
6
8
10
0
2
4
6
8
10

A*

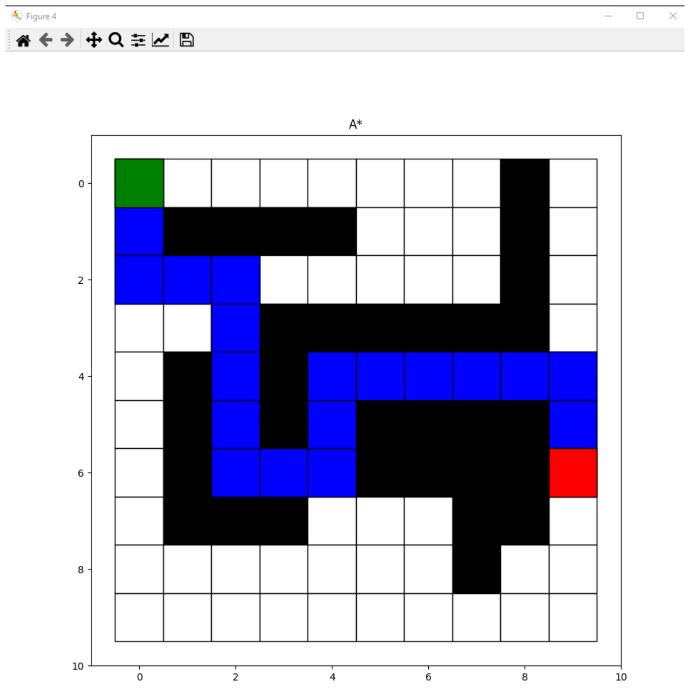


Figure 4
A*
0
2
4
6
8
10
0
2
4
6
8
10