RBE 550 -Motion Planning
HW1-Basic Search algorithms

Krishna Sathwik Durgaraju

Code and algorithm explanation:

Common code for all algorithms:

Class Node
A node class is defined with two attributes
1.Map coordinate tuples (x, y)
2.Node object to store parent node data
Start node's parent node will be none.

Path Tracking Function:
Output of this function provides shortest path i.e., list of nodes from goal node to start node. This is achieved by backtracking parent node starting from goal node to start node.

Neighbour node's function:

This function provides list of valid neighbour nodes.

Neighbour nodes order: right, down, left, up

Conditions to check valid neighbour nodes:

1. Neighbour's coordinates should be within map.
2. Neighbour's coordinates should not be obstacle (1)
3. Neighbour's coordinates should not be in Visited list

Flag to check which algorithm calls this function to implement below condition:

BFS, Dijkstra, A* = Same List as it uses priority queue

DFS = Reverse of this List as it uses priority stack (LIFO)

Breadth First Search (BFS):

BFS explores adjacent nodes first in queue order and stops exploring until it reaches goal node if exists. BFS is uninformed search algorithm since it doesn't consider distance between nodes.

Code explanation:

Map coordinates are converted tuples and node class. Queue list is created using deque python library. A while loop is used to continuous exploration of adjacent nodes for corresponding frontier node. Open list begins from start node and valid adjacent nodes are added to the queue subsequently. Finally, when goal node is reached break the loop and trace the path form goal node to start node using backtracking of parent node.

Pseudo code:

Function BFS (Grid, start, goal):

1.Initialize Open list (priority queue) with start node.

2.Initialize Visited list with start node.

3.While Open list not empty:

1. Pop node from left most in Queue (FIFO).
2. If the current node is Goal node, then track path and exit loop
3. Add unexplored adjacent nodes to the Open list.
4. Add current node to the visited node list.
5. Count number of steps (Total Number of Visited nodes)

4.if path is found output path list else raise exception no path found.

Depth First Search (DFS):

DFS explores each adjacent node to the depth in **stack** order and stops exploring until it reaches goal node if exists. DFS is also uninformed search algorithm since it doesn't consider distance between nodes.

Only difference with BFS code is it uses priority stack for open list.

Code explanation:

Like BFS except open list type (LIFO). Map coordinates are converted tuples and node class. Stack list is created using deque python library. A while loop is used to continuous exploration of adjacent nodes for corresponding frontier node. Open list begins from start node and valid adjacent nodes are added to the

stack subsequently. Finally, when goal node is reached break the loop and trace the path form goal node to start node using backtracking of parent node.

Pseudo code:

Function DFS (Grid, start, goal):

1.Initialize Open list (priority Stack) with start node.

2.Initialize Visited list with start node.

3.While Open list not empty:

1. Pop node from left most in Stack (LIFO).
2. If the current node is Goal node, then track path and exit loop
3. Add unexplored adjacent nodes to the Open list.
4. Add current node to the visited node list.
5. Count number of steps (Total Number of Visited nodes)

4.if path is found output path list else raise exception no path found.

Dijkstra:

Dijkstra algorithm finds the shortest path by exploring adjacent nodes with heuristic function. It is like BFS algorithm except it uses Manhattan distance function as heuristic to decide which adjacent node to explore first.

Code explanation:

Code is like BFS except heuristic function.

Manhattan distance = | current node x -goal x | + | current node y -goal y |

Adjacent nodes are sorted based on distance form goal to each adjacent node.

Lowest distance (cost) is considered as high priority node.

Pseudo code:

Function Dijkstra (Grid, start, goal):

1.Initialize Open list (priority queue) with start node.

2.Initialize Visited list with start node.

3.While Open list not empty:

1. Pop node from left most in Queue (FIFO).
2. If the current node is Goal node, then track path and exit loop
3. Calculate Manhattan distance for each adjacent node to goal node
4. Sort adjacent list based on distance value.

5. Add sorted adjacent nodes to the Open list.
6. Add current node to the visited node list.
7. Count number of steps (Total Number of Visited nodes)

4.If path is found output path list else raise exception no path found.

A*:

A* algorithm finds the shortest path by exploring adjacent nodes with additional heuristic function. It is like Dijkstra algorithm except it uses heuristic function as distance explored and distance to be travelled to decide which adjacent node to explore first.

Code explanation:

Code is like Dijkstra except heuristic function.

Manhattan distance -1 = | current node x -goal x | + | current node y -goal y |

Manhattan distance -2 = | current node x -start x | + | current node y -start y |

Total distance = D1 + D2

Adjacent nodes are sorted based on Total distance form goal to each adjacent node.Lowest distance (cost) is considered as high priority node.

Pseudo code:

Function A* (Grid, start, goal):

1.Initialize Open list (priority queue) with start node.

2.Initialize Visited list with start node.

3.While Open list not empty:

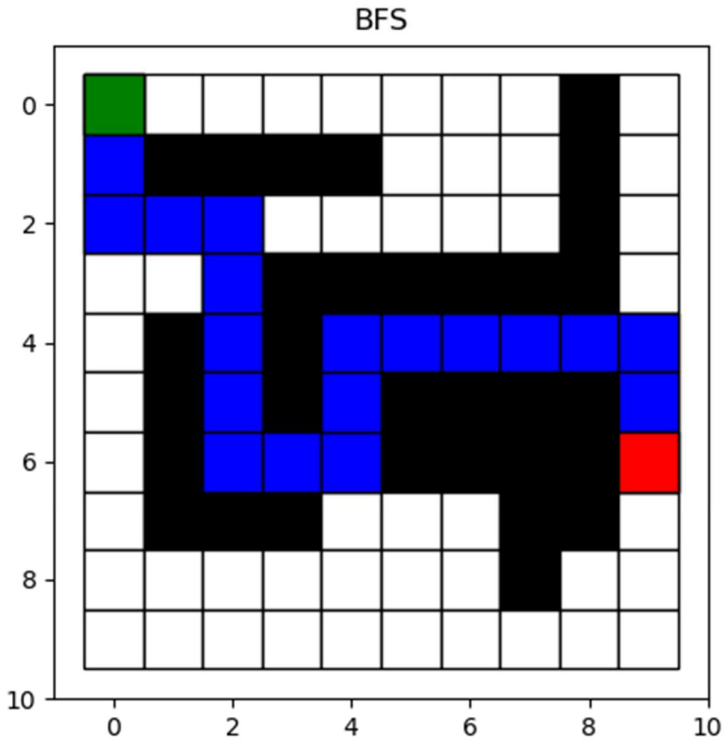
1. Pop node from left most in Queue (FIFO).
2. If the current node is Goal node, then track path and exit loop
3. Calculate Manhattan distance for each adjacent node to goal node
4. Calculate Manhattan distance for each adjacent node to start node
5. Add 2 distances to get total distance.
6. Sort adjacent list based on distance value.
7. Add sorted adjacent nodes to the Open list.
8. Sort open list with distance value
9. Add current node to the visited node list.
10. Count number of steps (Total Number of Visited nodes)

4.If path is found output path list else raise exception no path found.

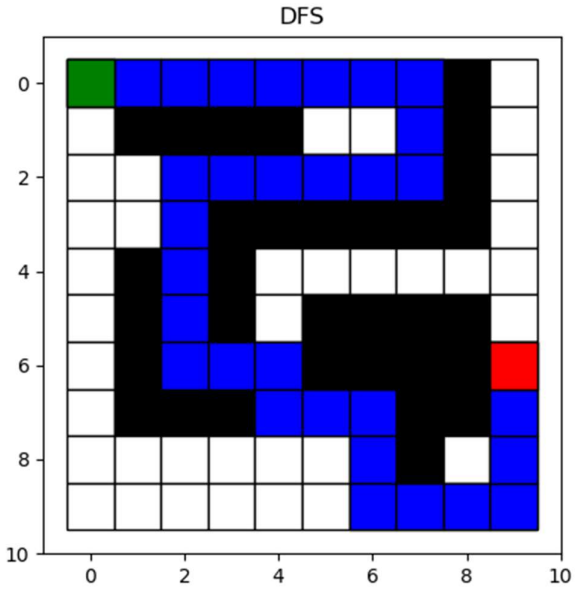
Test example, test result and explanation:

```
Console 1/A
\GoogleDriveFSPipe_krish_shell

In [115]: runfile('C:/Users/krish/Desktop/Motion_Planning/01_Assignments/Basic Search Algorithms/Final/main.py',
wdir='C:/Users/krish/Desktop/Motion_Planning/01_Assignments/Basic Search Algorithms/Final')
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*

In [116]:
```

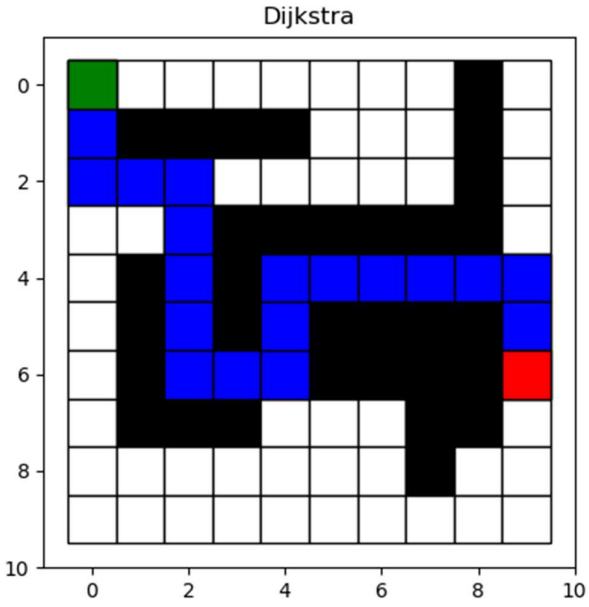
BFS:



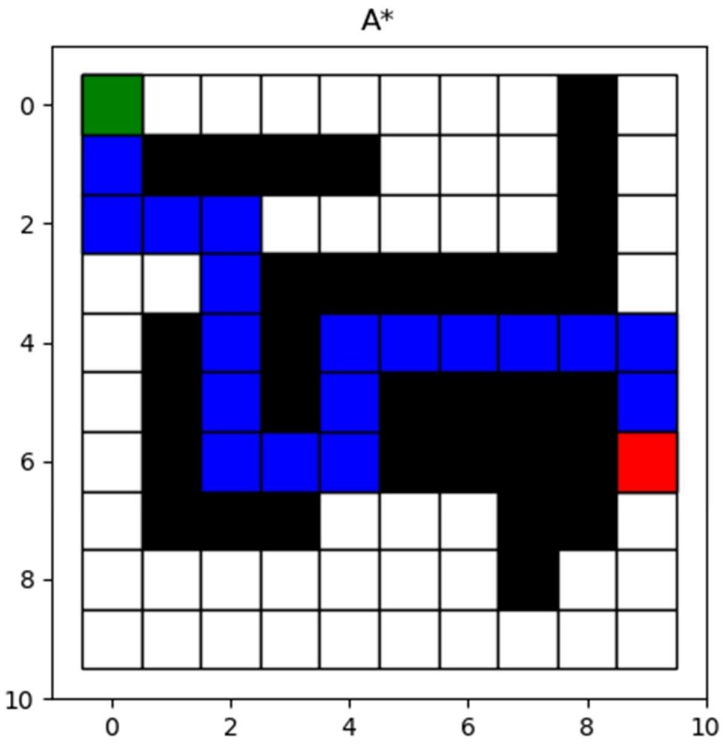
DFS



Dijkstra



A*

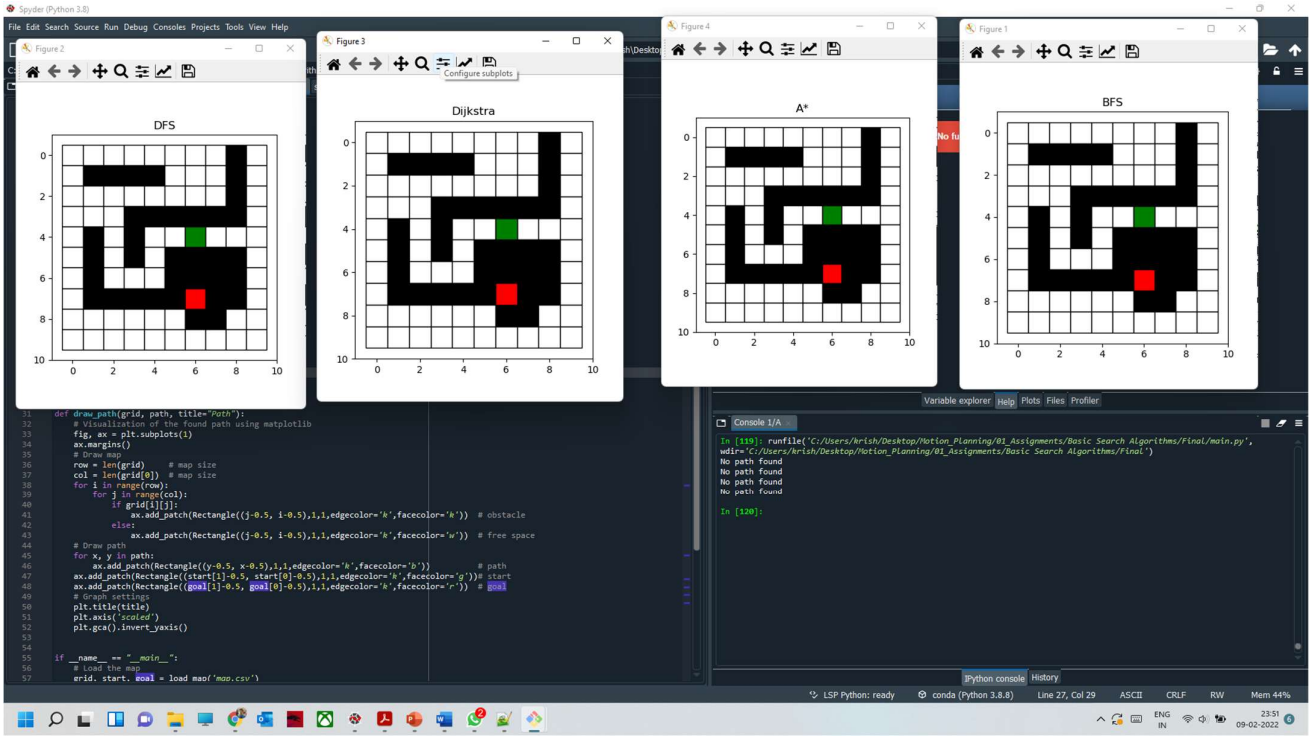


Explanation:

- Shortest path provided by BFS, Dijkstra, A* algorithms are same because heuristic value for every edge is same (Edge value =1).
- DFS provide different path because it considers LIFO as open list.
- A* explores less nodes (51) compared to BFS(64) and Dijkstra(64) because of distance heuristic function.
- DFS finds path to the goal with less exploration steps but traced path is not the shortest path to the goal.

Different Map:

Output: No path found for all algorithms



Conclusion:

1. BFS and Dijkstra provides same shortest path if adjacent node costs are equal.
2. No guarantee of shortest path by using DFS
3. A* algorithm provides shortest path with less exploring nodes. So, it is most efficient algorithm if heuristic is admissible.
4. BFS always provides shortest path but more exploration of nodes.

References:

Implemented code by understanding concepts from below links

1. https://www.youtube.com/watch?v=pcKY4hjDrxk
2. https://www.geeksforgeeks.org/shortest-path-in-a-binary-maze/