Motion Planning - Basic Search Algorithm Implementation

Name: Aniket Patil Email: apatil2@wpi.edu

Breadth First Search (BFS) Algorithm:

Explanation:

The BFS algorithm works on the principle of queues which is a first in first out (FIFO) type data structure. As the algorithm explores the grid, it traverses through the objects by pushing them into a queue and popping the object which was first pushed to the queue.

Example: In the first step, the start node is visited, and the neighbors are explored in the order right, bottom, left, top and pushed into the queue (level 1). We then explore the right node by pushing its neighbors into the queue. However, these neighbors (level 2) are now at the end of the queue hence the other neighbors of the start node (level 1) are explored first. Therefore, this algorithm is referred to as the Breadth First Algorithm as it explores Level 1 first before moving down to Level 2 of the graph.

Difference:

1. BFS works by exploring the breadth of the tree first, and then goes deeper unlike the DFS approach (siblings visited before visiting children of each node)
2. It works on the First In Last Out type data structure (queue)
3. BFS will give the shortest path

Similarity:

Both BFS and DFS are exploratory methods which do not take into consideration the heuristics to the goal node or the "cost to come" from the start node to the current node

Pseudocode:

1. Initialize **start node**, **goal node**, a **queue** and a "**visited**" matrix with False values
2. Set visited value of **start node** as True and push it in the **queue**
3. Iterate while length of **queue** is greater than 0
    a. Pop the first object (**u**) in the **queue** (increment step count for each visited node)
    b. Check if object is the **goal node** (break condition)
    c. Explore the neighbors (for loop)
        i. If you have already visited, then continue to next neighbor
        ii. If not visited, create a node (**v**) and specify the parent of v as u
        iii. Push the **v** node into the **queue** and set **visited** = True
4. Use the **get_path** function to return the path to the **goal node**

Depth First Search (DFS) Algorithm:

Explanation:

The DFS algorithm works on the principle of recursion or using stacks which is a first in last out type data structure. As the algorithm explores the grid, it enters a recursive function and explores the grid inside the branch till it reaches the end of the branch.

Example: In the first step, the start node is visited, and the neighbors are explored in the order right, bottom, left, top and the algorithm enters the node. We then explore the neighbors of the current node. When it finds suitable neighbors, it enters the same function again in a recursive manner. Therefore, it

goes on entering the function till it reaches a dead end or the goal node, in which case it encounters a return statement and exits the function.

Difference:

1. DFS works by exploring the depth of the tree first, and then explores other branches (children visited before visiting siblings of each node)
2. It works on the First In Last Out type data structure (stack) or recursion functions
3. Based on the graph structure, DFS may traverse through a lot of edges before reaching the goal node

Similarity:

Both BFS and DFS are exploratory methods which do not take into consideration the heuristics to the goal node or the "cost to come" from the start node to the current node

Pseudocode:

1. Initialize **start node**, **goal node**, a **queue** and a "**visited**" matrix with False values
2. Recursion function with input as the start node
    a. Set visited as True for input node
    b. Check if input node is goal node, if yes, set found = 1 and return
    c. Explore neighbors (for loop)
        i. If visited, continue to other neighbors
        ii. Recurse into the function using the neighbor node as input
        iii. If found = 1, return
    d. Return if no new neighbors found (reached end of the branch)
3. Use the **get_path** function to return the path to the **goal node**

Dijkstra Algorithm:

Explanation:

Dijkstra works by calculating the cost of traversing to each node from the start node. It does not take into consideration the heuristics of reaching the goal node from the current node.

Example: Let us start from the start node, and traverse through the graph. The algorithm will first check all the neighbors in the order: right, bottom, left, top and assign the cost of traversing to that node from the start node. In our case, since the step is weighted 1, the cost g(x) for all neighbors of the start node is 1. When the next node pops from the queue (also known as Open List) based on cost value, its neighbors (if not visited) will be assigned value 2, that is the cost of traversing to that node and added to the priority queue. When the next iteration starts, the nodes with lower cost value will be popped and visited. Hence, Dijkstra traverses through the tree by moving along the direction which has the lowest "cost to come g(x)".

Difference:

It works by calculating the "cost to come g(x)" for each node to figure out the shortest path to reach the current node from the start node [f(x) = g(x)]

Similarity:

Sorting of the priority queue (Open List) is the same for both Dijkstra and A* and based on the cost (f(x)). That is, the nodes that are visited by the algorithm pop from a priority queue based on their cost value f(x).

Pseudocode:

1. Initialize start node and end node
2. Create a priority queue (Q) and push start_node to it
3. While length of Q > 0
    a. Sort Q based on cost f(x) where f(x) = g(x)
    b. Pop a node u with lowest cost from Q
    c. If visited(u) is True, continue to next iteration, else, set visited(u) = True (Closed List check)
    d. If u is goal_node, set parameters(cost, parent, visited) of goal_node and break
    e. Explore neighbors of u (for loop)
        i. If neighbor is visited (in closed list), continue to next neighbor
        ii. Increment the cost g(x) for the node by the weight (1 in this case)
        iii. Calculate f(x) = g(x)
        iv. Add the neighbor to Q
4. Retrace path using get_path function

A* Algorithm:

Explanation:

A* works in a similar way to Dijkstra, however it also takes into consideration the cost of reaching the goal from the current node, also known as the heuristics h(x). Therefore, each node has two values g(x) that is the cost to come to that node from the start node and h(x) which is the heuristic value which is calculated by finding the Manhattan Distance of the current node from the goal node. Manhattan Distance is calculated using:
h(x) = abs(curr_node.row - goal_node.row) + abs(curr_node.col - goal_node.col)
Where curr_node is the current node and goal_node is the goal node

Therefore, the priority queue (also known as Open List) pops out a node with the lowest cost value f(x) = g(x) + h(x) and visits that node.

Difference:

A* works by calculating the total cost f(x) which is the total of the “cost to come g(x)” and the heuristic h(x) of reaching the goal node (Manhattan distance)

Similarity:

Sorting of the priority queue (Open List) is the same for both Dijkstra and A* and based on the cost (f(x)). That is, the nodes that are visited by the algorithm pop from a priority queue based on their cost value f(x).

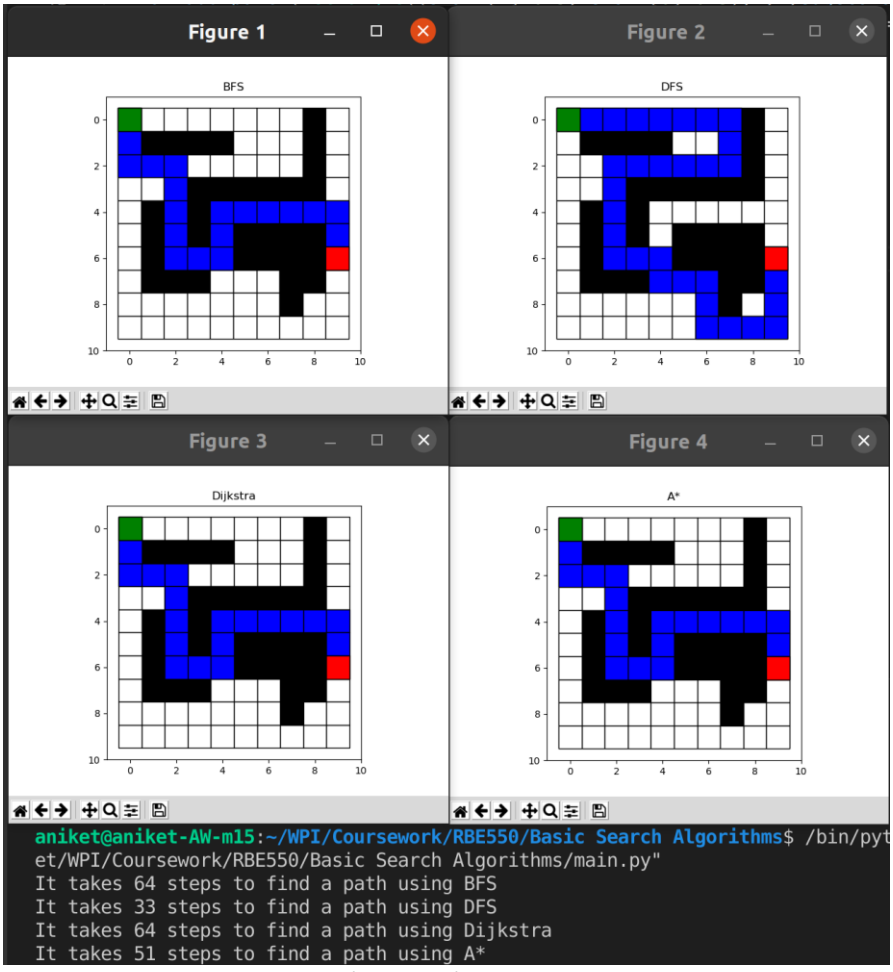
Pseudocode:

1. Initialize start node and end node
2. Create a priority queue (Q) and push start_node to it
3. While length of Q > 0
    a. Sort Q based on cost f(x) where f(x) = g(x) + h(x)

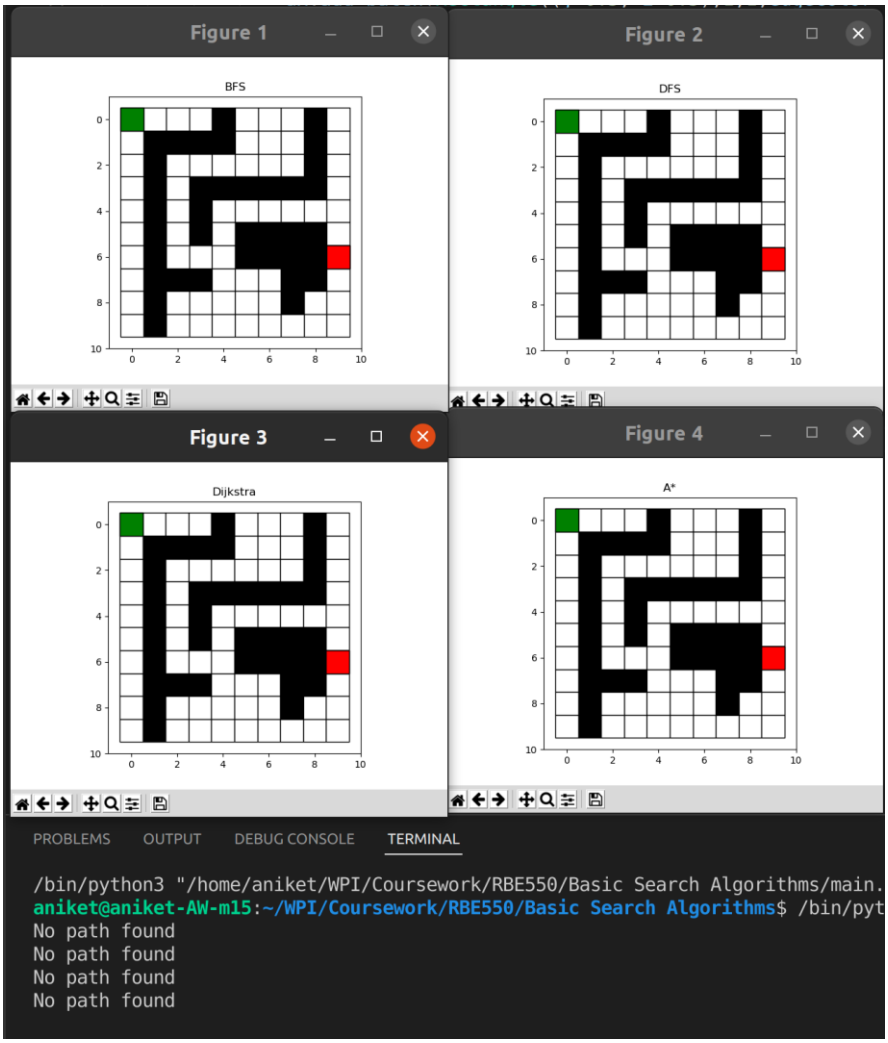
b. Pop a node u with lowest cost from Q
c. If visited(u) is True, continue to next iteration, else, set visited(u) = True (Closed List check)
d. If u is goal_node, set parameters(cost, parent, visited) of goal_node and break
e. Explore neighbors of u (for loop)
    i. If neighbor is visited (in closed list), continue to next neighbor
    ii. Increment the cost g(x) for the node by the weight (1 in this case)
    iii. Calculate heuristic h(x) and total cost f(x) = g(x) + h(x)
    iv. Add the neighbor to Q

4. Retrace path using get_path function

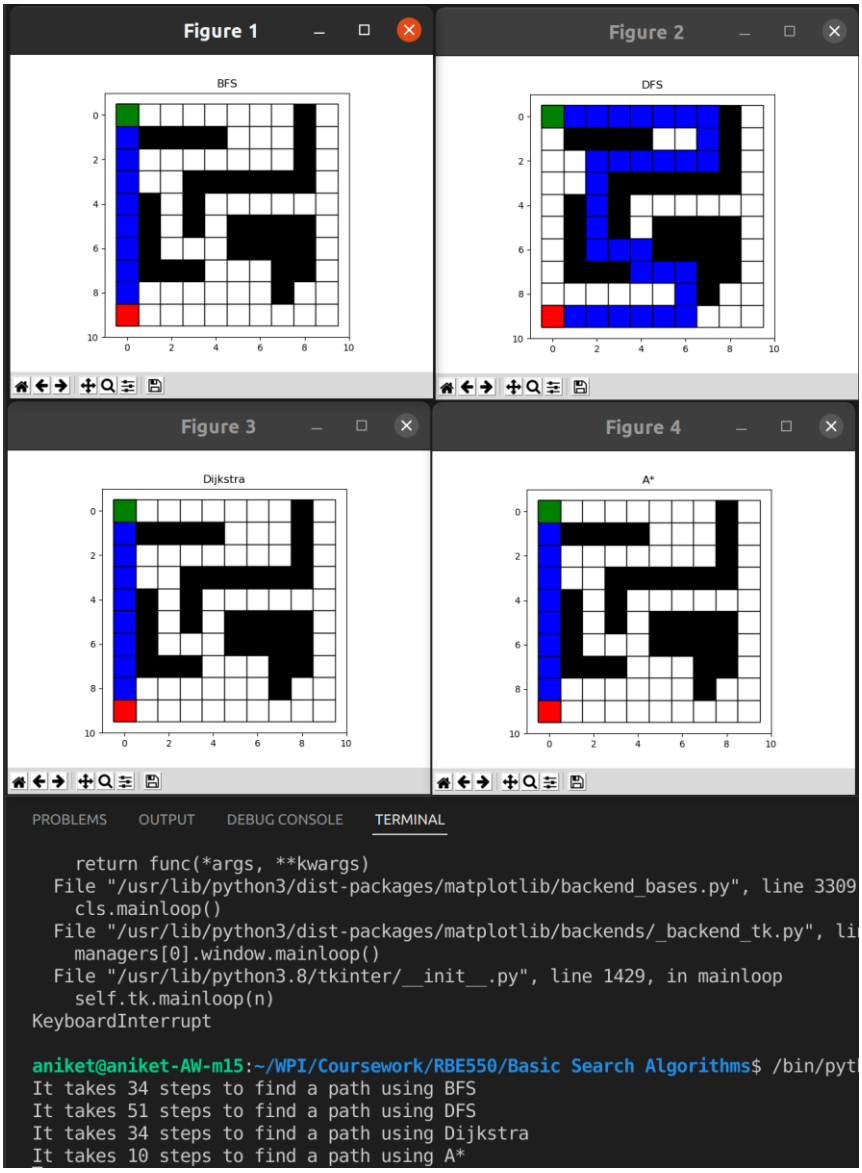
Test Examples and Explanations:



This is the test case given in the assignment (map.csv). We see that the DFS goes on exploring the right node as the algorithm works on depth exploration. Dijkstra and A* give similar outputs, but A* uses a smaller number of steps as the heuristic guides the graph search process towards the goal node.

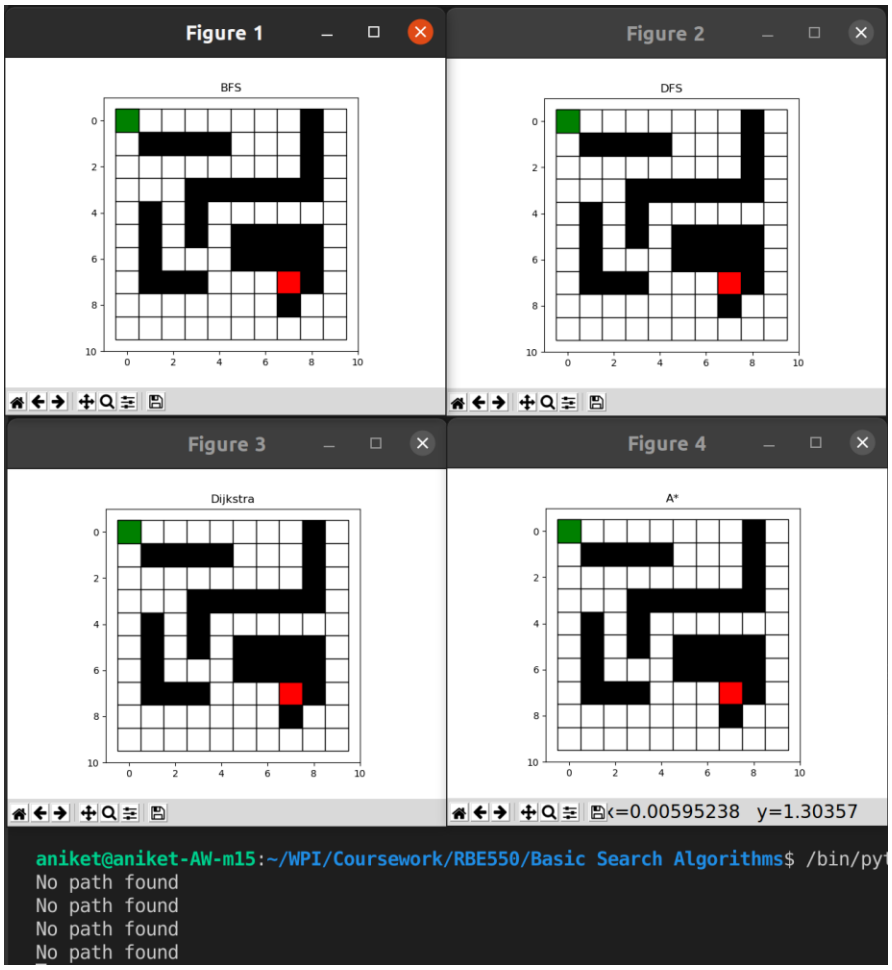


In this test example, all algorithms found no path to the goal. The reason for that is that no matter what direction you traverse in, an obstacle blocks the path to the goal.



In this test example, DFS clearly takes the longest route as the algorithm explores the right node first every time. This makes it explore the paths to the right entirely before moving on to other branches. However, since that path also leads to the goal node, other branches are not explored.

Coming to Dijkstra and A*, we can clearly see the impact of the heuristic term in this example. It shows how the heuristic guides the A* exploration towards the goal node and hence takes the least number of steps to reach the goal node.



In this case, the goal node does have a clear path leading to it. However, the goal node itself is placed in an obstacle. I have added this case as an exception too, which does not classify paths leading to/from obstacles as viable paths.

References:

https://www.geeksforgeeks.org/a-search-algorithm/

Playlist for Algorithm explanations:
https://www.youtube.com/playlist?list=PLDV1Zeh2NRsDGO4--qE8yH72HFL1Km93P