Motion Planning (RBE550)
Programming Assignment 2: Dijkstra and A-star Algorithm Implementation

Assignment By:
Swapneel Dhananjay Wagholikar (WPI ID: 257598983)

1. Dijkstra Algorithm:

Explanation:

In Dijkstra Algorithm, priority queue data structure is used which is similar to queue data structure, but we can shuffle the elements in the queue every time before popping any element out of it. In this case, this shuffle is done by taking into consideration g(x) i.e. minimum current cost to come. In the exploration of the grid, the algorithm starts with a start_node which is given as an input. Then it starts exploring its neighbors and so on. In doing so, it continues to append nodes into the queue, shuffling it as per g(x) and pops it out by revised order.

Step 1: the start node is visited.
Step 2: the neighbors are explored in the order specified and pushed into the queue (level 1).
Step 3: g(x) function is updated by adding 1 to it (specific to this case).
Step 4: popping out the node after shuffling.
Step 5: then explore its neighbors by pushing into the queue (level 2).
Step 6:The nodes with lower cost values will be popped and visited when the subsequent iteration begins.

Dijkstra therefore moves through the tree in the direction with the minimum cost to come g(x) to do so.

Pseudocode:

```
1. Initialize start node, goal node, an open queue and a "visited" matrix with 0 values
2. Set visited value of start node as 1 and push it in the open queue
3. Iterate while length of queue is greater than 0
    a. Sort Q based on cost f(x) where f(x) = g(x)
    b. Pop the first object (u) in the queue (increment step count for each visited node)
    c. Check if object is the goal node (break condition)
    d. Explore the neighbors (for loop)
        I. visit the node if and only if not in visited closed list
        ii. create a node (v) and specify the parent of v as u
        Iii. increment the cost g(x) for the node by the weight (1 in this case)
        Iv. updating g(x) value by taking min from updated and original value
        iv. push the v node into the queue
4. Use the generate_path function to return the path from start node to the goal node
```

2. A* Algorithm:

Explanation:

In A* Algorithm, working are very similar to Dijkstra. But while deciding the priority of the queue, we take into consideration h(x) i.e. heuristics along with g(x). As a result, each node has two values: h(x), the heuristic value that is determined by computing the Manhattan Distance between the present node and the goal node, and g(x), the cost to go to that node from the start node.

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

Pseudocode:

```
1. Initialize start node, goal node, a open queue and a “visited” matrix with 0 values
2. Set visited value of start node as 1 and push it in the open queue
3. Iterate while length of queue is greater than 0
        a. Sort Q based on cost f(x) where $f(x) = g(x)+h(x)$
        b. Pop the first object (u) in the queue (increment step count for each visited node)
        c. Check if object is the goal node (break condition)
        d. Explore the neighbors (for loop)
                i. visit the node if and only if not in visited closed list
                ii. create a node (v) and specify the parent of v as u
                iii. increment the cost g(x) for the node by the weight (1 in this case)
                iv. updating g(x) value by taking min from updated and original value
                v. adding heuristic h(x) to g(x) to find final cost function
                vi. push the v node into the queue
3. Use the generate_path function to return the path from start node to the goal node
```

Differences between Dijkstra and A*:

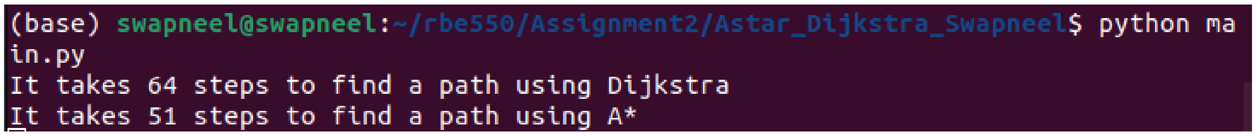
To determine the shortest route from the start node to the current node [$f(x) = g(x)$], the Dijkstra algorithm calculates the cost to come g(x) for each node while A* operates by figuring out the total cost f(x), which is the sum of the cost to come g(x) and the heuristic h(x) of getting to the goal node (Manhattan distance).

Similarities between Dijkstra and A*:

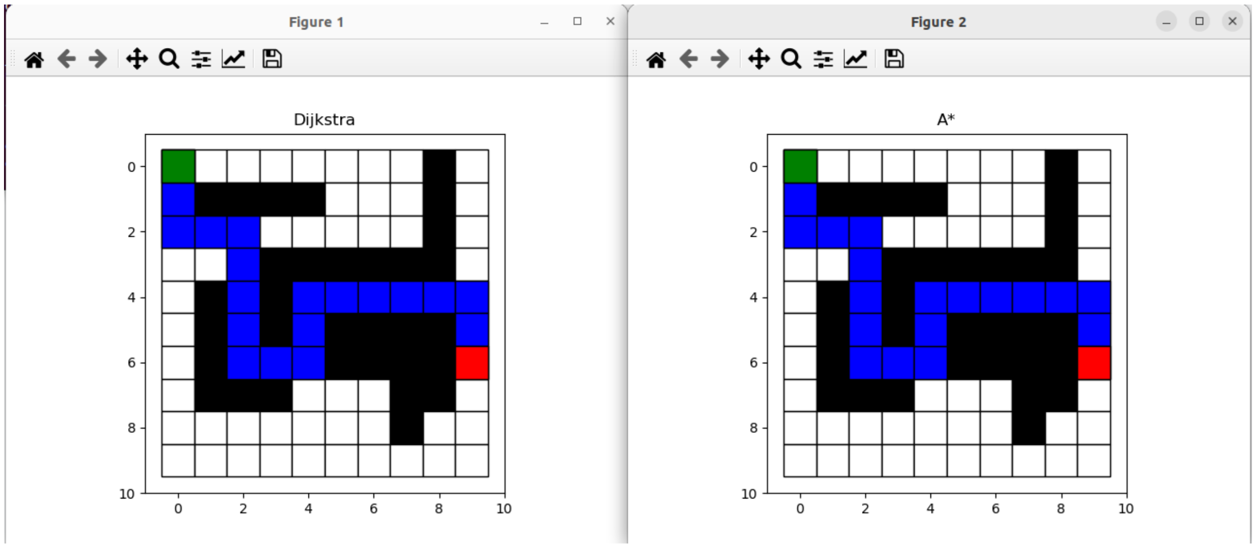
The priority queue (Open Q) is sorted using the cost (f(x)) and is the same for Dijkstra and A*. In other words, the nodes that the algorithm visits are selected from a priority queue according to their cost value f(x).

Test Examples, Test Results and Explanation:

1. **For given test case (map.csv) in the assignment:**

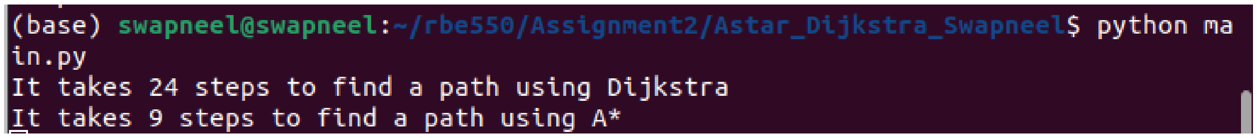


```
(base) swapneel@swapneel:~/rbe550/Assignment2/Astar_Dijkstra_Swapneel$ python ma
in.py
It takes 64 steps to find a path using Dijkstra
It takes 51 steps to find a path using A*
```

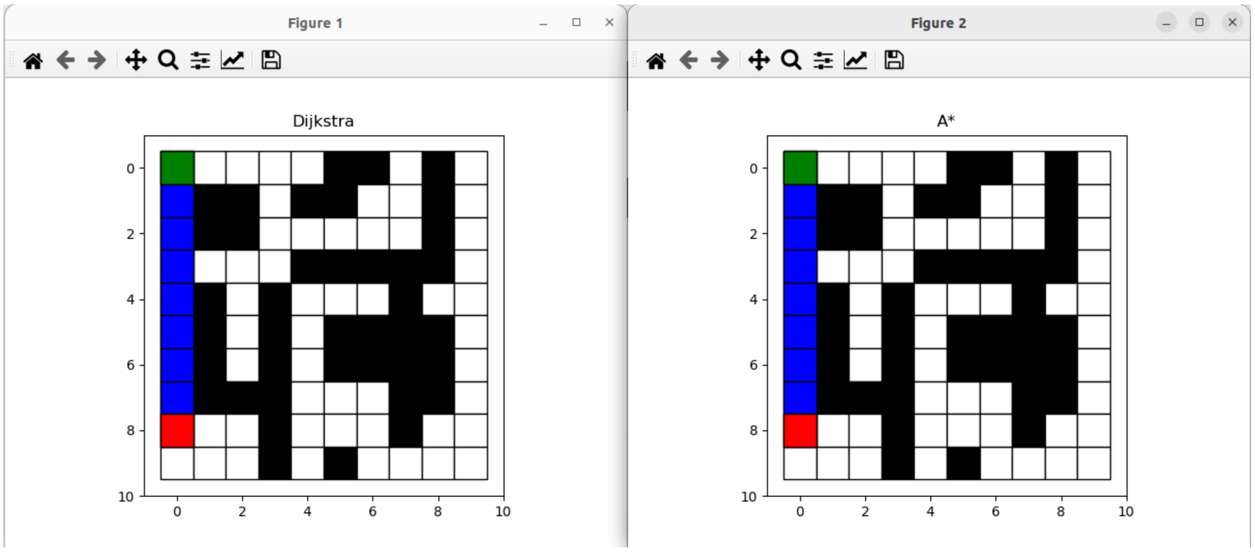


Dijkstra and A* produce results that are similar, but A* requires fewer steps since its heuristic leads the faster graph search process to the goal node.

2. **User Test Case 1:**

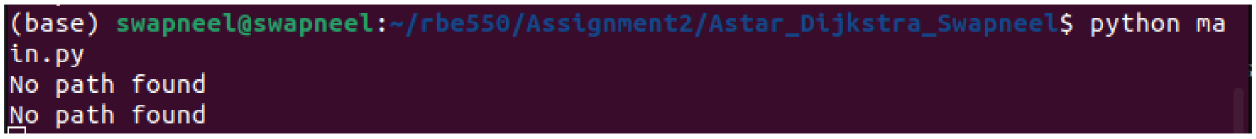


```
(base) swapneel@swapneel:~/rbe550/Assignment2/Astar_Dijkstra_Swapneel$ python ma
in.py
It takes 24 steps to find a path using Dijkstra
It takes 9 steps to find a path using A*
```

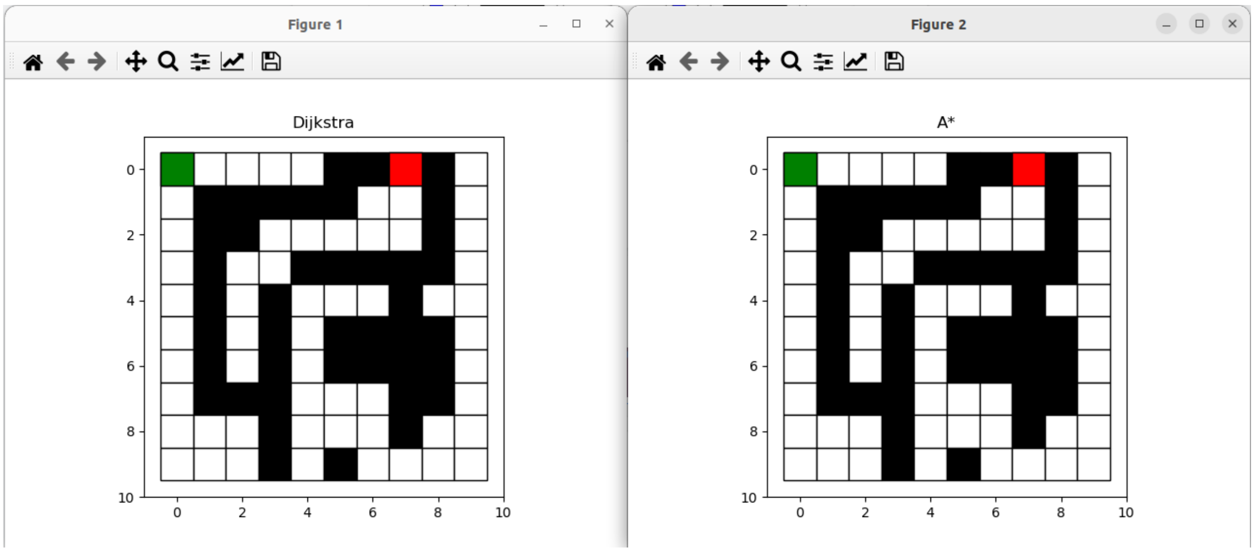


In this instance, the heuristic term's effect is very apparent. It demonstrates how the heuristic directs the A* exploration in the direction of the destination node and, as a result, requires the fewest steps to get there compared to Dijkstra.

3. **User Test Case 2:**



```
(base) swapneel@swapneel:~/rbe550/Assignment2/Astar_Dijkstra_Swapneel$ python ma
in.py
No path found
No path found
```



In this test case, both the algorithms fail to find the path as the path is completely blocked by obstacles.

References:

1. https://levelup.gitconnected.com/dijkstra-algorithm-in-python-8f0e75e3f16e
2. https://www.youtube.com/watch?v=OrJ004Wid4o&t=1889s
3. https://www.geeksforgeeks.org/python-program-for-dijkstras-shortest-path-algorithm-greedy-algo-7/