**ALGORITHMS EXPLANATION AND THEIR RESULTS**

**Breadth First Search:**

The BFS algorithm will first visit all the neighbors of node v (it's child nodes, on level one), in the order that is given in the adjacency list. Next, it takes the child nodes of those neighbors (level two) into consideration, and so on. Also, we keep track of the nodes that have been visite, so that we don’t fall into infinite loop of revisiting same nodes.

The BFS Algorithm steps are as follows:

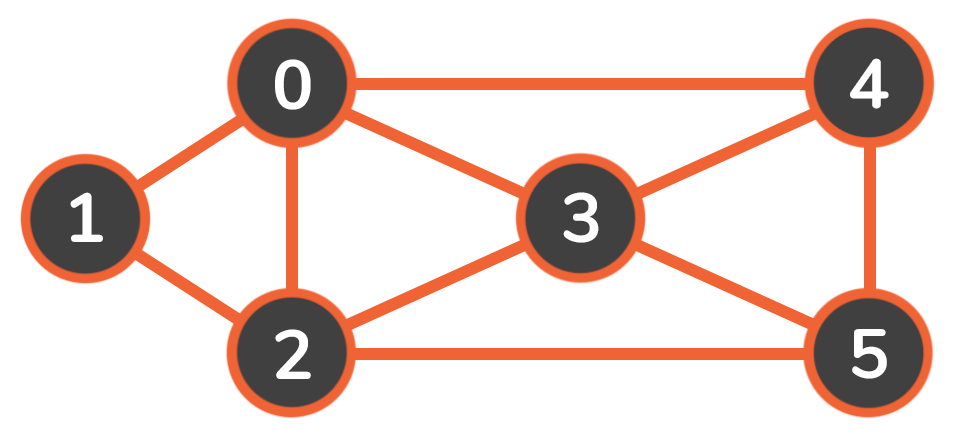
1. Add the root/start node to the Queue.
2. For every node, set that they don't have a defined parent node.
3. Until the Queue is empty:

- Extract the node from the beginning of the Queue.

- Perform output processing.

- For every neighbor of the current node that doesn't have a defined parent (is not visited), add it to the Queue, and set the current node as their parent.

Example



As we start the traversal from the start node 0, it is put into the visited set and into the queue as well. While we still have nodes in the queue, we extract the first one, print it, and check all of its neighbors.

When going through the neighbors, we check if each of them is visited, and if not we add them to the queue and mark them as visited:

Steps Queue Visited

Add start node 0 [0] {0}

Visit 0, add 1 & 2 to Queue [1, 2] {0}

Visit 1, add 4 to Queue [2, 4] {0, 2}

Visit 2, add 3 to Queue [4, 3] {0, 1, 2}

Visit 4, no unvisited neighbours [3] {0, 1, 1, 4}

Visit 3, no unvisited neighbours [ ] {0, 1, 2, 4, 3}

--------------------------------------------------------------------------------------------------------------------------

**Depth First Search:**

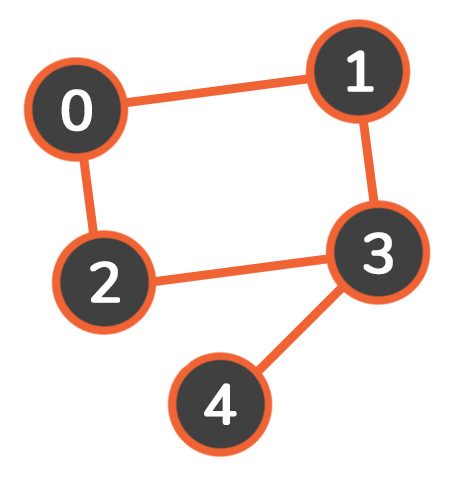
Depth-First Search (DFS) is an algorithm used to traverse or locate a target node in a graph or tree data structure. It priorities depth and searches along one branch, as far as it can go - until the end of that branch. Once there, it backtracks to the first possible divergence from that branch, and searches until the end of that branch, repeating the process.

The DFS algorithm steps are as follows:

1. Mark the current node as visited.
2. Traverse the neighboring nodes that aren't visited and recursively call the DFS function for that node.

The algorithm stops either when the target node is found, or the whole graph has been traversed (all nodes are visited).

Example



Add node 0 to the traversal path and mark it as visited. Check if node 0 is equal to target node 3, and since it's not, continue and traverse its neighbors (1 and 2).

If neighbor 1 is not visited, then the algorithm calls the function recursively for that node.

Recursive call for node 1: Add node 1 to the traversal path and mark it as visited. Continue and traverse its neighbors (0 and 3). Since neighbor 0 is visited, move on to the next one. Then, 3 is not visited so call the function recursively for this node.

Current Node Path Visited

0 [0] {0}

1 [0, 1] {0, 1}

3 [0, 1, 3] {0, 1, 3}

The algorithm stops and our program prints out the resulting traversal path from node 0 to node 3:

[0, 1, 3]

----------------------------------------------------------------------------------------------------

**Dijkstra's Algorithm:**

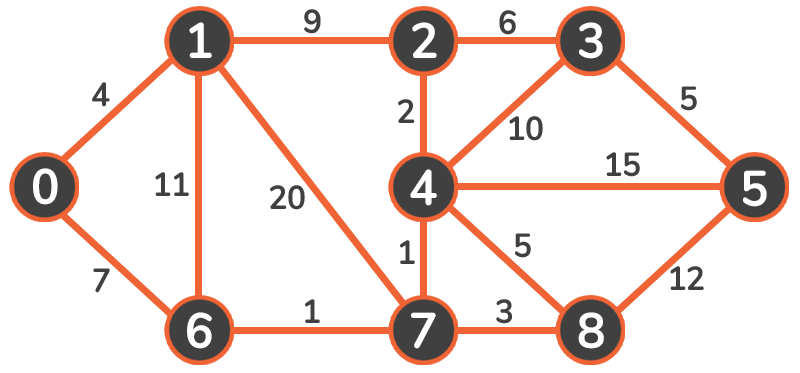
In the beginning, we'll want to create a set of visited vertices, to keep track of all of the vertices that have been assigned their correct shortest path. We will also need to set costs of all vertices in the graph (ie. lengths of the current shortest path that leads to it).

All of the costs will be set to 'infinity' at the beginning, to make sure that every other cost we may compare it to would be smaller than the starting one. The only exception is the cost of the first, starting vertex - this vertex will have a cost of 0, because it has no path to itself.

Then, we repeat two main steps until the graph is traversed (as long as there are vertices without the shortest path assigned to them):

1. We pick a vertex with the shortest current cost, visit it, and add it to the visited vertices set.
2. We update the costs of all its adjacent vertices that are not visited yet. For every edge between n and m where m is unvisited - if the cheapestPath(s, n) + cheapestPath(n,m) < cheapestPath(s,m), update the cheapest path between s and m to equal cheapestPath(s,n) + cheapestPath(n,m).

Example



vertex cost to get to it from vertex 0

0 0

1 4

2 11

3 17

4 9

5 24

6 7

7 8

8 11

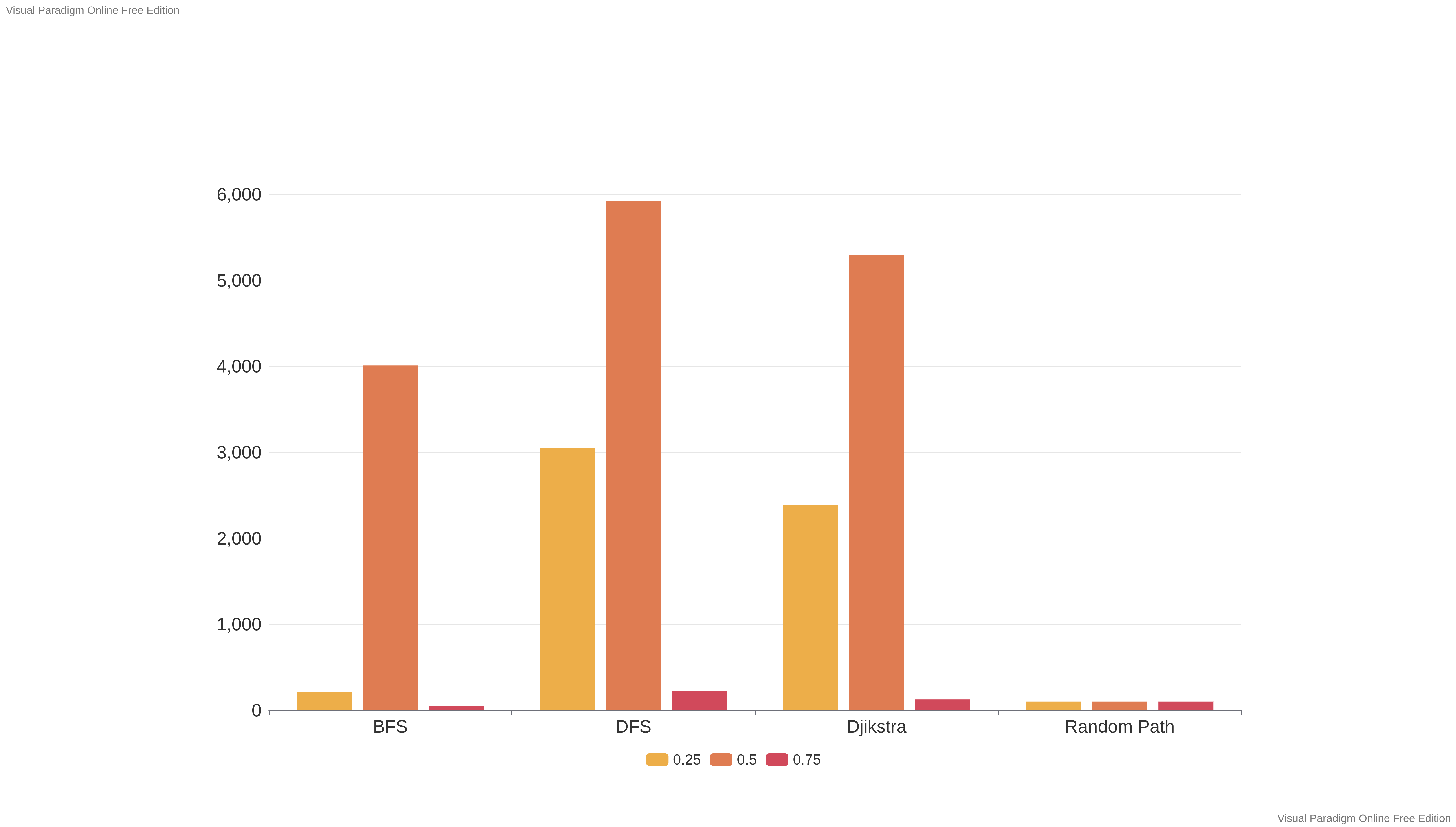
--------------------------------------------------------------------------------------------------------

Results obtained:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 25% occupancy |  | Total iterations consumed | Total path cost |  |
| bfs |  | 214 | 14 |  |
| dfs |  | 3050 | 159 |  |
| Djilkstra |  | 2381 | 95 |  |
| random |  | 100 | NA |  |
| 50% occupancy |  |  |  |  |
| bfs |  | 4008 | 81 |  |
| dfs |  | 5917 | 607 |  |
| Djilkstra |  | 5294 | 340 |  |
| random |  | 100 | NA |  |
| 75% occupancy |  |  |  | Here, goals unreachable for all |
| bfs |  | 47 | NA |  |
| dfs |  | 223 | NA |  |
| Djilkstra |  | 98 | NA |  |
| random |  | 100 | NA |  |

From the results obtained, it can be concluded that the search algorithms consumed more nodes to traverse obviously with more obstacles in occupancy grid.

And as far as algorithms are concerned, The performance graph of the algorithms is as below:



As can be seen that **BFS** performed the best in comparison to all. As per expectations, **Djikstra** gave slightly off results, and my guess is this was because of random path values initialized. This could make the algorithm go off the course if the values are random weights are initialized instead.

**Random path planner:**

And in random path planner, as I hoped randomly selected adjacent node of the current node, there were lower chances of convergence and thus I terminated it after 100 iterations since it explored all the nodes in the surroundings randomly with high redundancy in nodes explored. If we don’t allow the random path planner to visit the node again, then it will get stuck since it didn’t work like a tree node.

Initially I took inspiration from BFS itself, where I selected next adjacent node whose Euclidean distance with the goal node is shortest, but this would still make the bot stuck since there was no parameter to come out of the local minima when stuck in a corner or in-between the crevices of the obstacle. Thus this approach was discarded and then I selected any one of the adjacent nodes of the robots choice as per it got from random.choice() function.

To conclude, Djikstra would perform still better than all others since it could prioritize the paths to take over others, unlike BFS and DFS, where the branches or child nodes selected are anyways randomly, since weights to all the edges or nodes are equal.