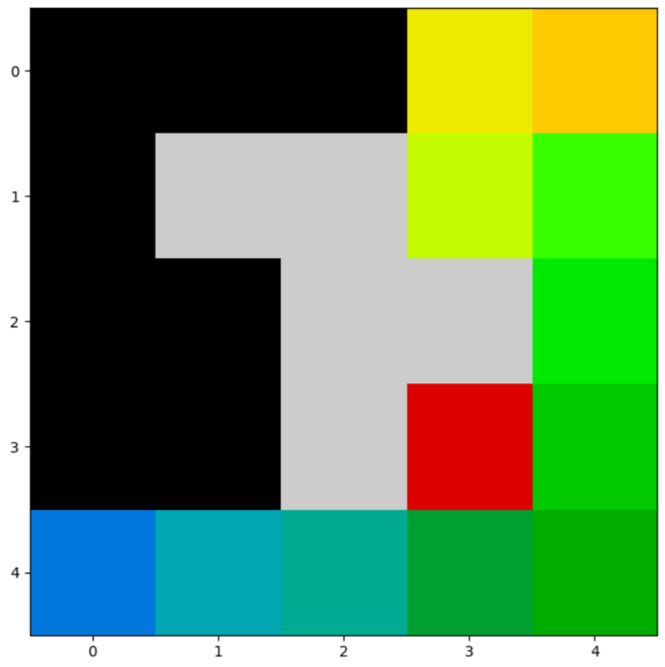
HW01

Manju Shivdasan

1. Note: The algorithm code is highly influenced from the pseudocode (few minor changes are made to make the pseudocode work as an dfs algorithm)
   1.1 Run DFS on map0.txt. Report the path and set of states visited. Include an image of the path taken in your answer. Is the path found the shortest path? Explain why the algorithm did or did not find the shortest path.
   i. Done
   ii. Path: [(4, 0), (4, 1), (4, 2), (4, 3), (4, 4), (3, 4), (2, 4), (1, 4), (1, 3), (0, 3), (0, 4)]
   Visited: [(4, 0), (4, 1), (4, 2), (4, 3), (4, 4), (3, 4), (3, 3), (2, 4), (1, 4), (1, 3), (0, 3), (0, 4)]



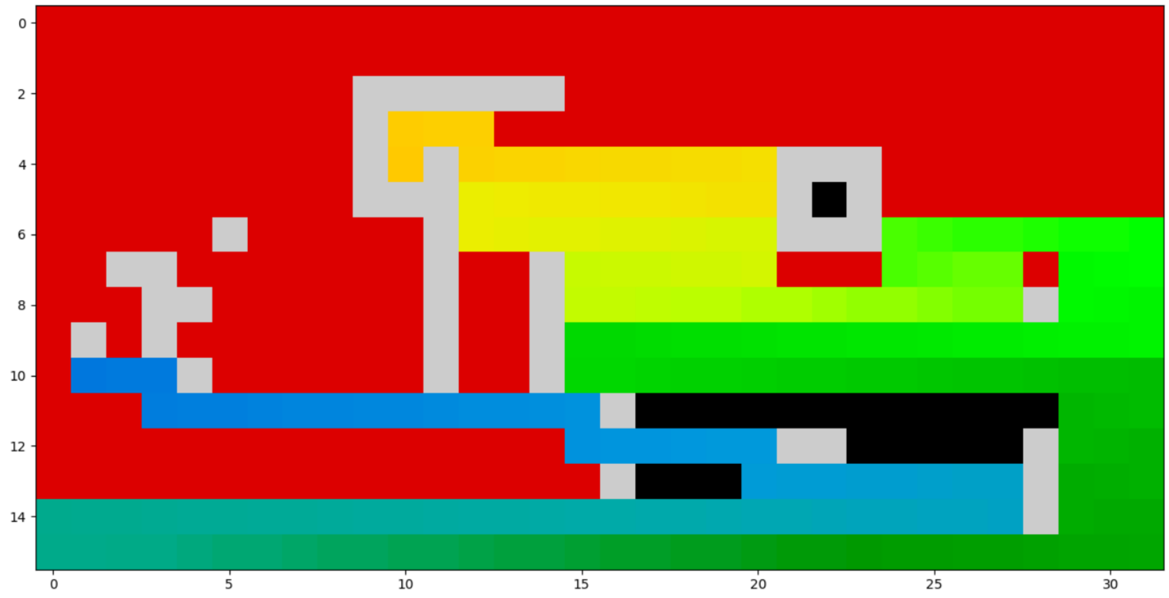
   iii. No the path found is not the shortest path.
   iv. In DFS search, the children nodes are explored in a specific order that is predefined in the algorithm (by actions) i.e. it proceeds in an arbitrary order. Also, DFS utilizes the STACK (LIFO) principle, which means that even though the goal state is present in the STACK it won't be visited until it is the top most element in the STACK.

1.2 Perform the same test and analysis for map1.txt.

i. Done

ii. Path: [(10, 1), (11, 1), (12, 1), (13, 1), (14, 1), (15, 1), (15, 0), (14, 0), (13, 0), (12, 0), (11, 0), (10, 0), (9, 0), (8, 0), (7, 0), (6, 0), (5, 0), (4, 0), (3, 0), (2, 0), (1, 0), (0, 0), (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (8, 2), (9, 2), (10, 2), (11, 2), (12, 2), (13, 2), (14, 2), (15, 2), (15, 3), (14, 3), (13, 3), (12, 3), (11, 3), (11, 4), (12, 4), (13, 4), (14, 4), (15, 4), (15, 5), (14, 5), (13, 5), (12, 5), (11, 5), (10, 5), (9, 5), (8, 5), (7, 5), (7, 4), (6, 4), (5, 4), (4, 4), (3, 4), (2, 4), (1, 4), (0, 4), (0, 5), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (5, 6), (4, 6), (3, 6), (2, 6), (1, 6), (0, 6), (0, 7), (1, 7), (2, 7), (3, 7), (4, 7), (5, 7), (6, 7), (7, 7), (8, 7), (9, 7), (10, 7), (11, 7), (12, 7), (13, 7), (14, 7), (15, 7), (15, 8), (14, 8), (13, 8), (12, 8), (11, 8), (10, 8), (9, 8), (8, 8), (7, 8), (6, 8), (5, 8), (4, 8), (3, 8), (2, 8), (1, 8), (0, 8), (0, 9), (1, 9), (1, 10), (0, 10), (0, 11), (1, 11), (1, 12), (0, 12), (0, 13), (1, 13), (1, 14), (0, 14), (0, 15), (1, 15), (2, 15), (3, 15), (4, 15), (5, 15), (6, 15), (7, 15), (8, 15), (9, 15), (10, 15), (11, 15), (12, 15), (13, 15), (14, 15), (15, 15), (15, 14), (14, 14), (13, 14), (12, 14), (11, 14), (11, 13), (10, 13), (9, 13), (8, 13), (7, 13), (6, 13), (5, 13), (4, 13), (3, 13), (3, 12), (3, 11), (3, 10), (4, 10)]

Visited: [(10, 1), (11, 1), (12, 1), (13, 1), (14, 1), (15, 1), (15, 0), (14, 0), (13, 0), (12, 0), (11, 0), (10, 0), (9, 0), (8, 0), (7, 0), (6, 0), (5, 0), (4, 0), (3, 0), (2, 0), (1, 0), (0, 0), (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (8, 2), (9, 2), (10, 2), (11, 2), (12, 2), (13, 2), (14, 2), (15, 2), (15, 3), (14, 3), (13, 3), (12, 3), (11, 3), (10, 3), (11, 4), (12, 4), (13, 4), (14, 4), (15, 4), (15, 5), (14, 5), (13, 5), (12, 5), (11, 5), (10, 5), (9, 5), (8, 5), (7, 5), (7, 4), (6, 4), (5, 4), (4, 4), (3, 4), (2, 4), (1, 4), (0, 4), (0, 3), (1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (6, 3), (6, 2), (5, 2), (4, 2), (3, 2), (2, 2), (1, 2), (0, 2), (0, 5), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (5, 6), (4, 6), (3, 6), (2, 6), (1, 6), (0, 6), (0, 7), (1, 7), (2, 7), (3, 7), (4, 7), (5, 7), (6, 7), (7, 7), (8, 7), (9, 7), (10, 7), (11, 7), (12, 7), (13, 7), (14, 7), (15, 7), (15, 6), (14, 6), (13, 6), (12, 6), (11, 6), (10, 6), (9, 6), (8, 6), (7, 6), (6, 6), (15, 8), (14, 8), (13, 8), (12, 8), (11, 8), (10, 8), (9, 8), (8, 8), (7, 8), (6, 8), (5, 8), (4, 8), (3, 8), (2, 8), (1, 8), (0, 8), (0, 9), (1, 9), (1, 10), (0, 10), (0, 11), (1, 11), (1, 12), (0, 12), (0, 13), (1, 13), (1, 14), (0, 14), (0, 15), (1, 15), (2, 15), (3, 15), (4, 15), (5, 15), (6, 15), (7, 15), (8, 15), (9, 15), (10, 15), (11, 15), (12, 15), (13, 15), (14, 15), (15, 15), (15, 14), (14, 14), (13, 14), (12, 14), (11, 14), (11, 13), (10, 13), (9, 13), (8, 13), (7, 13), (6, 13), (5, 13), (4, 13), (3, 13), (3, 12), (4, 12), (5, 12), (6, 12), (7, 12), (8, 12), (9, 12), (10, 12), (11, 12), (12, 12), (13, 12), (14, 12), (15, 12), (15, 11), (14, 11), (13, 11), (12, 11), (11, 11), (11, 10), (10, 10), (9, 10), (8, 10), (7, 10), (6, 10), (6, 9), (7, 9), (8, 9), (9, 9), (10, 9), (11, 9), (12, 9), (13, 9), (14, 9), (15, 9), (15, 10), (14, 10), (13, 10), (12, 10), (15, 13), (14, 13), (13, 13), (12, 13), (3, 11), (3, 10), (4, 10)]



iii. No, the path found is not the shortest path.

iv. In DFS search, the children nodes are explored in a specific order that is predefined in the algorithm (by actions) i.e. it proceeds in an arbitrary order. Also, DFS utilizes the STACK (LIFO) principle, which means that even though the goal state is present in the STACK it won't be visited until it is the top most element in the STACK.

1.3 Perform the same test and analysis for map2.txt.

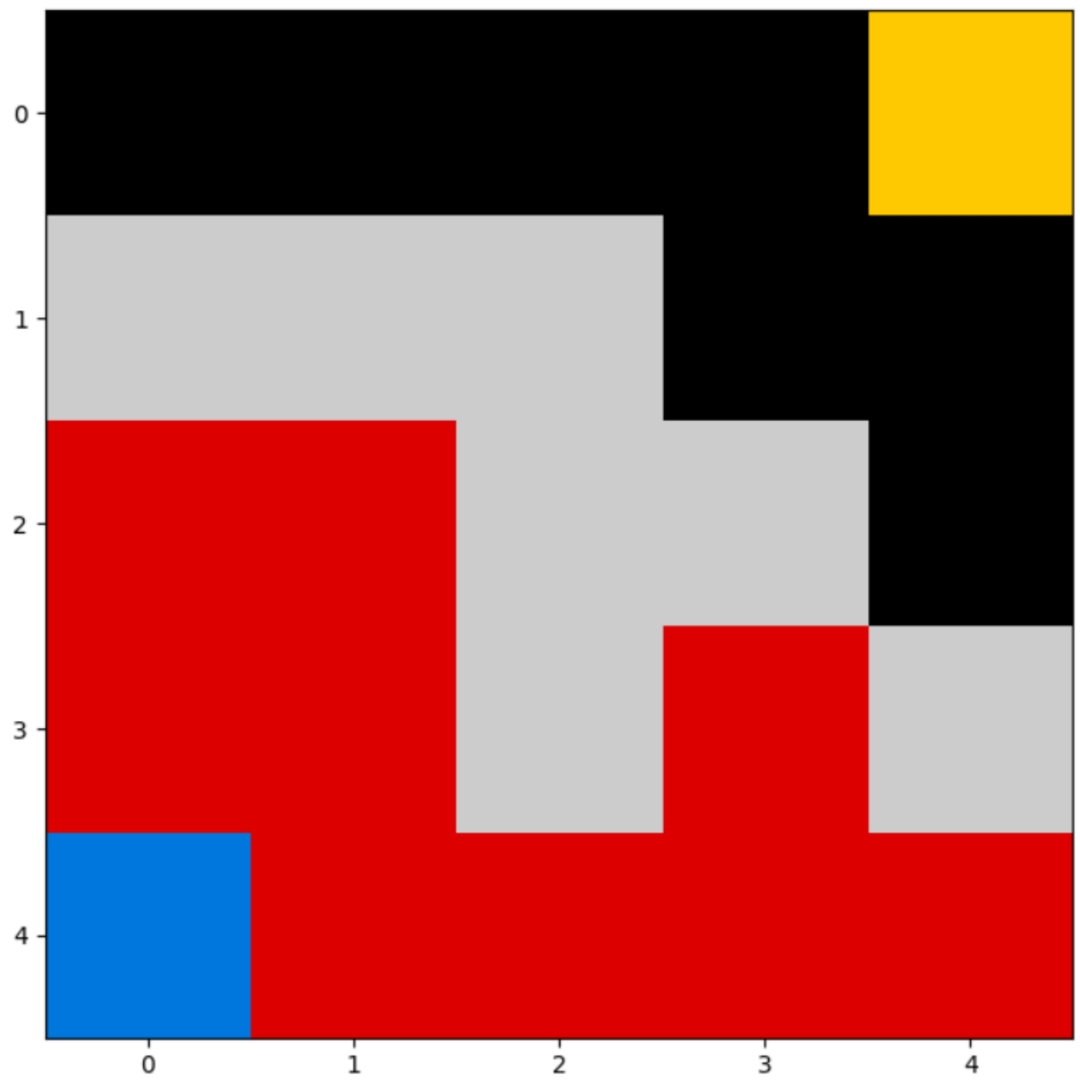
i. Done

ii. Path taken by the algorithm:

The initial path the algorithm takes is r u l u r r after which it realizes there is not path from Initial position to Goal state, thus the algorithm throws an statement stating "No path Found".

Path: None

Visited: [(4, 0), (3, 0), (2, 0), (2, 1), (3, 1), (4, 1), (4, 2), (4, 3), (3, 3), (4, 4)]



iii. There is no path found.

iv. Since there is not path available to goal, therefore all the nodes before the obstacles are visited.

1.4 The DFS algorithm explained in the psuedocode above adds search nodes for all actions from a given node at once. What issues may arise from doing this? How could this be alleviated?

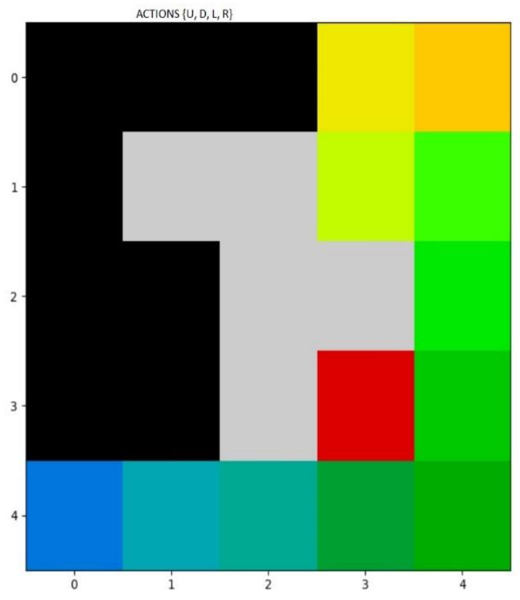
Ans:

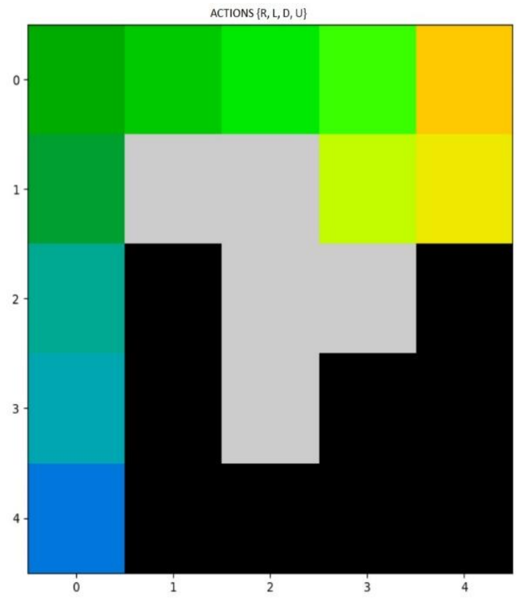
When the map grid is small, adding search nodes for all actions on a given node is fast. But, when the map is large, the number of actions associated with the search nodes increase exponentially. this would affect the time complexity and memory utilization.

This problem can be alleviated by visiting the first legal node. By doing this, we would be adding only one search nodes, making it faster and utilizing less memory.

1.5 Reverse the order of the actions explored by DFS and run it again on map0.txt and map1.txt. Does anything change in your results? What? Why is this the case?

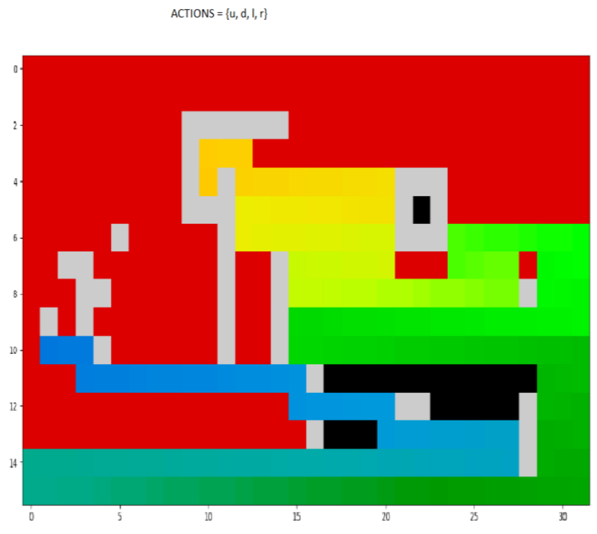
The result is totally dependent on the action sequence performed on the node. If the actions were to be reversed the path traversed on the map grid will also change. This is because dfs utilizes the STACK principle of LIFO. So when the order of the action is changed, a different node value is popped out.

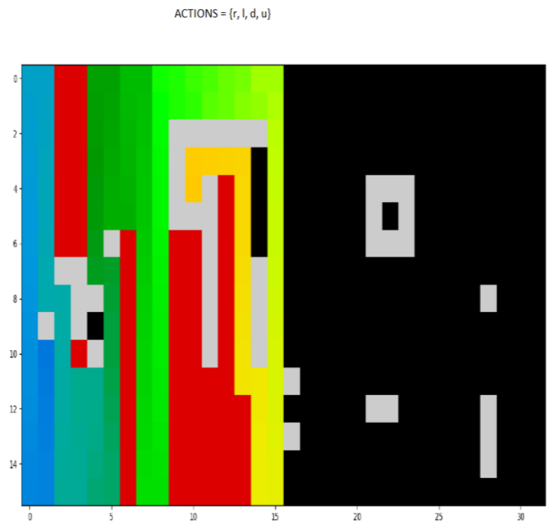




Old Path: r r r r u u u l u r

New Path: u u u u r r r d r u





Old Path: r r d r r r r r r r r r r r r d r r r r r d r r r r r r r d l l l l l l l l l l l l l l l l l l l l l l l l l l l d r r r r r r r r r r r r r r r r r r r r r r r r r r r r r r r u l l u r r u l l u r r u l l l l l l l l l l l l l l l l u r r r r r r r r r r r r r r r r u l l u r r u l l l l l l l d r r r d l l l l l l l l l l l l u r r r r r u l l l l l l l l u r r r r r r r r u l l l l l l r

New path: d d d d d l u u u u u u u u u u u u u u u r d d d d d d d r d d d d d d d r u u u u r d d d d r u u u u u u u u l u u u u u u u r d d d d d r u u u u u r d d d d d d d d d d d d d d d r u u u u u u u u u u u u u u u r d r u r d r u r d r u r d d d d d d d d d d d d d d d d l u u u u l u u u u u u u u l l l u

1.6 Extend DFS to iterative deepening DFS. You need to add an argument to your DFS function that it only explores to a maximum depth m during execution. Additionally, you need to modify your visited set to take into account the depth at which nodes are visited, since they may be revisited at a shallower path which leads to the goal in fewer than m steps. Run your iterative deepening implementation on map0.txt and map1.txt. Did your algorithm find the shortest path? Explain why it did or did not.

Ans:

Map0:

Path: [(4, 0), (4, 1), (4, 2), (4, 3), (4, 4), (3, 4), (2, 4), (1, 4), (0, 4)]

Visited: [(4, 0), (4, 1), (4, 2), (4, 3), (4, 4), (3, 4), (3, 3), (2, 4), (1, 4), (1, 3), (0, 4)]

Map1:

Path: [(10, 1), (10, 2), (10, 3), (11, 3), (11, 4), (11, 5), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), (11, 11), (11, 12), (11, 13), (10, 13), (10, 12), (9, 12), (9, 13), (8, 13), (8, 12), (7, 12), (7, 13), (6, 13), (6, 14), (5, 14), (5, 13), (5, 12), (4, 12), (3, 12), (3, 11), (3, 10), (4, 10)]

Visited: [(10, 1), (10, 2), (10, 3), (11, 3), (11, 4), (11, 5), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), (11, 11), (11, 12), (11, 13), (11, 14), (11, 15), (12, 15), (12, 16), (12, 17), (12, 18), (12, 19), (12, 20), (13, 20), (13, 21), (13, 22), (13, 23), (13, 24), (13, 25), (13, 26), (13, 27), (14, 27), (14, 26), (15, 27), (12, 27), (12, 26), (11, 27), (14, 25), (14, 24), (14, 23), (14, 22), (15, 23), (15, 24), (15, 25), (12, 25), (12, 24), (12, 23), (11, 23), (11, 24), (11, 25), (10, 24), (14, 21), (14, 20), (14, 19), (14, 18), (14, 17), (14, 16), (14, 15), (14, 14), (15, 15), (13, 15), (15, 16), (15, 17), (13, 17), (13, 18), (13, 19), (15, 18), (15, 19), (15, 20), (15, 21), (11, 20), (11, 21), (11, 22), (10, 22), (10, 23), (9, 23), (9, 24), (9, 25), (9, 26), (9, 27), (10, 26), (8, 26), (10, 25), (8, 25), (8, 24), (7, 25), (9, 22), (9, 21), (9, 20), (9, 19), (10, 20), (8, 20), (10, 21), (8, 21), (8, 22), (7, 21), (8, 23), (7, 23), (7, 24), (6, 24), (7, 22), (11, 19), (11, 18), (11, 17), (10, 17), (10, 18), (10, 19), (9, 18), (9, 17), (9, 16), (9, 15), (10, 16), (8, 16), (8, 17), (8, 18), (7, 17), (12, 14), (12, 13), (12, 12), (12, 11), (12, 10), (12, 9), (12, 8), (12, 7), (12, 6), (12, 5), (12, 4), (12, 3), (12, 2), (12, 1), (12, 0), (13, 1), (11, 1), (13, 2), (13, 3), (14, 2), (11, 2), (13, 4), (13, 5), (13, 6), (13, 7), (14, 6), (14, 5), (14, 4), (15, 5), (13, 8), (13, 9), (13, 10), (13, 11), (13, 12), (13, 13), (13, 14), (14, 13), (14, 12), (15, 13), (14, 11), (14, 10), (14, 9), (14, 8), (15, 9), (15, 10), (15, 11), (10, 15), (10, 13), (10, 12), (9, 12), (9, 13), (8, 13), (8, 12), (7, 12), (7, 13), (6, 13), (6, 14), (6, 15), (6, 16), (6, 17), (6, 18), (6, 19), (6, 20), (7, 20), (7, 19), (5, 20), (5, 19), (4, 20), (7, 18), (5, 18), (5, 17), (5, 16), (5, 15), (4, 16), (4, 17), (4, 18), (3, 17), (7, 16), (7, 15), (8, 15), (5, 14), (5, 13), (5, 12), (6, 12), (4, 12), (4, 13), (4, 14), (4, 15), (3, 15), (3, 14), (3, 13), (3, 12), (3, 11), (3, 10), (4, 10)]

Yes, my algorithm found the shortest path.

In a depth-first search, you begin at some node in the graph and continuously explore deeper and deeper into the graph while you can find new nodes that you haven't yet reached (or until you find the solution). Any time the DFS runs out of moves, it backtracks to the latest point where it could make a different choice, then explores out from there. This can be a serious problem if your graph is extremely large and there's only one solution, since you might end up exploring the entire graph along one DFS path only to find the solution after looking at each node. Worse, if the graph is infinite (perhaps your graph consists of all the numbers, for example), the search might not terminate. Moreover, once you find the node you're looking for, you might not have the optimal path to it (you could have looped all over the graph looking for the solution even though it was right next to the start node!)

One potential fix to this problem would be to limit the depth of any one path taken by the DFS. For example, we might do a DFS search, but stop the search if we ever take a path of length greater than 5. This ensures that we never explore any node that's of distance greater than five from the start node, meaning that we never explore out infinitely or (unless the graph is extremely dense) we don't search the entire graph. However, this does mean that we might not find the node we're looking for, since we don't necessarily explore the entire graph.

The idea behind iterative deepening is to use this second approach but to keep increasing the depth at each level. In other words, we might try exploring using all paths of length one, then all paths of length two, then length three, etc. until we end up finding the node in question. This means that we never end up exploring along infinite dead-end paths, since the length of each path is capped by some length at each step. It also means that we find the shortest possible path to the destination node, since if we didn't find the node at depth m but did find it at depth m + 1, there can't be a path of length m (or we would have taken it), so the path of length m + 1 is indeed optimal.

2. Note: The algorithm code is highly influenced from the pseudocode of DFS (few minor changes are made to make the pseudocode work as an bfs algorithm)

   2.1 Run BFS on map0.txt. Report the path and set of states visited. Is the path found the shortest path? Explain why the algorithm did or did not find the shortest path.

   i. Done
   ii. Path: [(4, 0), (3, 0), (2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (0, 3), (0, 4)]
   Visited: [(4, 0), (3, 0), (4, 1), (2, 0), (3, 1), (4, 2), (1, 0), (2, 1), (4, 3), (0, 0), (3, 3), (4, 4), (0, 1), (3, 4), (0, 2), (2, 4), (0, 3), (1, 4), (1, 3), (0, 4)]
   iii. The algorithm found the shortest path from init to goal. This is because bfs always visits the m node before moving on to the (m+1) node.

2.2 Perform the same test and analysis for map1.txt.

i. Done

ii. **Path:**

Path: [(10, 1), (11, 1), (11, 2), (11, 3), (11, 4), (11, 5), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), (11, 11), (11, 12), (10, 12), (9, 12), (8, 12), (7, 12), (6, 12), (5, 12), (4, 12), (3, 12), (3, 11), (3, 10), (4, 10)]

Visited: [(10, 1), (11, 1), (10, 0), (10, 2), (12, 1), (11, 0), (11, 2), (9, 0), (9, 2), (10, 3), (13, 1), (12, 0), (12, 2), (11, 3), (8, 0), (8, 2), (14, 1), (13, 0), (13, 2), (12, 3), (11, 4), (7, 0), (8, 1), (15, 1), (14, 0), (14, 2), (13, 3), (12, 4), (11, 5), (6, 0), (7, 1), (15, 0), (15, 2), (14, 3), (13, 4), (12, 5), (10, 5), (11, 6), (5, 0), (6, 1), (15, 3), (14, 4), (13, 5), (12, 6), (9, 5), (10, 6), (11, 7), (4, 0), (5, 1), (6, 2), (15, 4), (14, 5), (13, 6), (12, 7), (8, 5), (9, 4), (9, 6), (10, 7), (11, 8), (3, 0), (4, 1), (5, 2), (6, 3), (15, 5), (14, 6), (13, 7), (12, 8), (7, 5), (8, 6), (9, 7), (10, 8), (11, 9), (2, 0), (3, 1), (4, 2), (5, 3), (6, 4), (15, 6), (14, 7), (13, 8), (12, 9), (7, 4), (7, 6), (8, 7), (9, 8), (10, 9), (11, 10), (1, 0), (2, 1), (3, 2), (4, 3), (5, 4), (15, 7), (14, 8), (13, 9), (12, 10), (6, 6), (7, 7), (8, 8), (9, 9), (10, 10), (11, 11), (0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (15, 8), (14, 9), (13, 10), (12, 11), (5, 6), (6, 7), (7, 8), (8, 9), (9, 10), (11, 12), (0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (15, 9), (14, 10), (13, 11), (12, 12), (4, 6), (5, 7), (6, 8), (7, 9), (8, 10), (10, 12), (11, 13), (0, 2), (1, 3), (2, 4), (3, 5), (15, 10), (14, 11), (13, 12), (12, 13), (3, 6), (4, 7), (5, 8), (6, 9), (7, 10), (9, 12), (10, 13), (11, 14), (0, 3), (1, 4), (2, 5), (15, 11), (14, 12), (13, 13), (12, 14), (2, 6), (3, 7), (4, 8), (6, 10), (8, 12), (9, 13), (11, 15), (0, 4), (1, 5), (15, 12), (14, 13), (13, 14), (12, 15), (1, 6), (2, 7), (3, 8), (7, 12), (8, 13), (10, 15), (0, 5), (15, 13), (14, 14), (13, 15), (12, 16), (0, 6), (1, 7), (2, 8), (6, 12), (7, 13), (9, 15), (10, 16), (15, 14), (14, 15), (12, 17), (0, 7), (1, 8), (5, 12), (6, 13), (8, 15), (9, 16), (10, 17), (15, 15), (14, 16), (11, 17), (13, 17), (12, 18), (0, 8), (1, 9), (4, 12), (5, 13), (6, 14), (7, 15), (8, 16), (9, 17), (10, 18), (15, 16), (14, 17), (11, 18), (13, 18), (12, 19), (0, 9), (1, 10), (3, 12), (4, 13), (5, 14), (6, 15), (7, 16), (8, 17), (9, 18), (10, 19), (15, 17), (14, 18), (11, 19), (13, 19), (12, 20), (0, 10), (1, 11), (3, 11), (3, 13), (4, 14), (5, 15), (6, 16), (7, 17), (8, 18), (9, 19), (10, 20), (15, 18), (14, 19), (11, 20), (13, 20), (0, 11), (1, 12), (3, 10), (3, 14), (4, 15), (5, 16), (6, 17), (7, 18), (8, 19), (9, 20), (10, 21), (15, 19), (14, 20), (11, 21), (13, 21), (0, 12), (1, 13), (4, 10)]

iii. The algorithm found the shortest path from init to goal. This is because bfs always visits the m node before moving on to the (m+1) node.

2.3 **Perform the same test and analysis for map2.txt.**

i. Done

Path: None

Visited: [(4, 0), (3, 0), (4, 1), (2, 0), (3, 1), (4, 2), (2, 1), (4, 3), (3, 3), (4, 4)]

The initial path traced by the algorithm is as shown above. Since there is no physical path from the init state to the goal, the algorithm explores all the possible path till it hits the obstacles and then finally prompts us that there is no physical path between the 2 states.

ii. Since no path exists, the algorithm couldn't find a path to the goal.

2.4 Compare the performance of your algorithm to DFS and iterative deepening DFS above. How do the paths found differ? How do the states visited differ? Which was easier to implement and why? Discuss anything else you found interesting or difficult in implementing and evaluating the three algorithms.

Ans:

From 1.6, we know the difference between DFS and IDDFS.

The difference between DFS and BFS is that, DFS explores each branch till the end and then starts off with the next one whereas BFS explores all the nearby m nodes before proceeding to m+1 node. It means that we find the shortest possible path to the destination node, since if we didn't find the node at depth m but did find it at depth m + 1, there can't be a path of length m (or we would have taken it), so the path of length m + 1 is indeed optimal.

The difference between BFS and IDDFS is that in a BFS, you must hold all of the fringe nodes in memory at once. This takes memory O(bm), where b is the branching factor. Compare this to the O(m) memory usage from iterative deepening (to hold the state for each of the m nodes in the current path). Of course, BFS never explores the same path multiple times, while iterative deepening may explore any path several times as it increases the depth limit. However, asymptotically the two have the same runtime. BFS terminates in O(bm) steps after considering all O(bm) nodes at distance m. Iterative deepening uses O(bm) time per level, which sums up to O(bm) overall, but with a higher constant factor.

The path found by DFS is different when compared to IDDFS and BFS, whereas the path found by IDDFS and BFS, is the same, given the depth mentioned in the program is greater than or equal to the depth of the goal. The states visited by DFS is different from that of BFS and IDDFS. DFS expands each branch till the end and then moves on to the next one, whereas BFS and IDDFS expands the nearby node m and then moves on to the next node, m+1. All 3 algorithms were easy to implement ones the DFS algorithm was implemented. Minor changes to DFS algorithm lead us to the BFS and IDDFS algorithm.

Out of the 3 algorithms, IDDFS is the most efficient one, as it utilizes the pros of BFS and DFS algorithm. It helps us to find the shortest path by utilizing the characteristic of BFS of expanding the m depth node before moving on to the m+1 node, thus providing us the optimal solution and all of this with a memory complexity of O(m)

3. Uniform Cost Search (20 pts)

Note: The algorithm code is highly influenced from the pseudocode taught in class (few minor changes are made to make the pseudocode work as an usc algorithm)

3.1 (5 pts) Run uniform cost search on map0.txt. Report the path and set of states visited. Is the path found the lowest cost path? Is the path found the shortest path? Explain why the algorithm did or did not find the lowest cost path.

i. Done

ii. **Path:**

[(4, 0), (3, 0), (2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (0, 3), (0, 4)]

Visited List:

[(4, 0), (3, 0), (4, 1), (4, 2), (3, 1), (2, 0), (1, 0), (4, 3), (2, 1), (3, 3), (0, 0), (4, 4), (3, 4), (0, 1), (0, 2), (2, 4), (0, 3), (1, 4), (0, 4)]

iii. Yes, the algorithm did find the lowest cost and the shortest path. Since the nodes are expanded in increasing order of path cost and all m nodes are visited first before m+1 nodes, the least cost path and the shortest path is found.

3.2 **(5 pts) Perform the same test and analysis for map1.txt.**

i. Done

ii. Path:

[(10, 1), (10, 2), (10, 3), (11, 3), (11, 4), (11, 5), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), (11, 11), (11, 12), (10, 12), (9, 12), (8, 12), (7, 12), (6, 12), (5, 12), (4, 12), (3, 12), (3, 11), (3, 10), (4, 10)]

Visited List:

[(10, 1), (11, 1), (10, 0), (10, 2), (12, 1), (9, 0), (10, 3), (9, 2), (11, 2), (11, 0), (13, 1), (8, 2), (11, 3), (12, 2), (8, 0), (12, 0), (11, 4), (7, 0), (12, 3), (8, 1), (14, 1), (13, 0), (13, 2), (12, 4), (15, 1), (13, 3), (14, 0), (7, 1), (11, 5), (6, 0), (14, 2), (14, 3), (11, 6), (6, 1), (10, 5), (15, 2), (12, 5), (15, 0), (13, 4), (5, 0), (13, 5), (15, 3), (10, 6), (4, 0), (14, 4), (12, 6), (5, 1), (11, 7), (6, 2), (9, 5), (13, 6), (10, 7), (15, 4), (9, 6), (4, 1), (3, 0), (14, 5), (12, 7), (8, 5), (5, 2), (11, 8), (6, 3), (9, 4), (12, 8), (5, 3), (14, 6), (9, 7), (13, 7), (10, 8), (8, 6), (15, 5), (4, 2), (3, 1), (2, 0), (7, 5), (11, 9), (6, 4), (1, 0), (7, 4), (11, 10), (2, 1), (12, 9), (4, 3), (13, 8), (5, 4), (8, 7), (15, 6), (14, 7), (9, 8), (7, 6), (10, 9), (3, 2), (12, 10), (3, 3), (2, 2), (0, 0), (1, 1), (11, 11), (13, 9), (10, 10), (14, 8), (4, 4), (7, 7), (5, 5), (8, 8), (6, 6), (15, 7), (9, 9), (9, 10), (11, 12), (5, 6), (8, 9), (1, 2), (12, 11), (13, 10), (0, 1), (3, 4), (2, 3), (14, 9), (15, 8), (6, 7), (4, 5), (7, 8), (5, 7), (12, 12), (4, 6), (15, 9), (10, 12), (8, 10), (3, 5), (11, 13), (2, 4), (0, 2), (7, 9), (14, 10), (13, 11), (1, 3), (6, 8), (4, 7), (1, 4), (0, 3), (6, 9), (14, 11), (11, 14), (10, 13), (5, 8), (7, 10), (13, 12), (15, 10), (2, 5), (9, 12), (12, 13), (3, 6), (4, 8), (6, 10), (14, 12), (9, 13), (15, 11), (12, 14), (2, 6), (13, 13), (8, 12), (11, 15), (3, 7), (1, 5), (0, 4), (8, 13), (3, 8), (13, 14), (15, 12), (12, 15), (1, 6), (2, 7), (14, 13), (10, 15), (0, 5), (7, 12), (9, 15), (10, 16), (6, 12), (14, 14), (0, 6), (1, 7), (2, 8), (13, 15), (12, 16), (7, 13), (15, 13), (12, 17), (5, 12), (9, 16), (10, 17), (15, 14), (6, 13), (8, 15), (0, 7), (14, 15), (1, 8), (11, 17), (8, 16), (5, 13), (13, 17), (12, 18), (4, 12), (9, 17), (0, 8), (1, 9), (10, 18), (15, 15), (6, 14), (14, 16), (7, 15), (12, 19), (1, 10), (0, 9), (15, 16), (10, 19), (13, 18), (9, 18), (6, 15), (5, 14), (7, 16), (8, 17), (3, 12), (14, 17), (4, 13), (11, 18), (13, 19), (6, 16), (14, 18), (5, 15), (4, 14), (7, 17), (10, 20), (15, 17), (1, 11), (12, 20), (3, 13), (9, 19), (3, 11), (11, 19), (8, 18), (0, 10), (14, 19), (1, 12), (15, 18), (4, 15), (7, 18), (3, 14), (9, 20), (11, 20), (10, 21), (0, 11), (3, 10), (8, 19), (6, 17), (13, 20), (5, 16), (0, 12), (9, 21), (4, 16), (15, 19), (8, 20), (3, 15), (7, 19), (11, 21), (6, 18), (10, 22), (5, 17), (4, 10)]

iii. Yes, the algorithm did find the lowest cost and the shortest path. Since the nodes are expanded in increasing order of path cost and all m nodes are visited first before m+1 nodes, the least cost path and the shortest path is found.

3.3 (10 pts) Now extend the set of actions to allow the robot to move diagonally on the grid. Make diagonal action cost 1.5 as opposed to 1 for lateral and vertical moves. Perform the same test and analysis for map0.txt, map1.txt, and map2.txt as in 3.1.

i. Map0:
   a. Path: [(4, 0), (4, 1), (4, 2), (3, 3), (2, 4), (1, 4), (0, 4)]
   b. Visited: [(4, 0), (3, 0), (4, 1), (3, 1), (4, 2), (2, 0), (2, 1), (1, 0), (4, 3), (3, 3), (0, 0), (4, 4), (3, 4), (0, 1), (2, 4), (0, 2), (1, 4), (1, 3), (0, 3), (0, 4)]

ii. Map 1
   a. Path: [(10, 1), (10, 2), (10, 3), (11, 4), (11, 5), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), (11, 11), (10, 12), (9, 12), (8, 12), (7, 12), (6, 12), (5, 12), (4, 12), (3, 11), (4, 10)]
   b. Visited: [(10, 1), (11, 1), (10, 0), (10, 2), (9, 0), (11, 2), (11, 0), (9, 2), (10, 3), (12, 1), (8, 0), (8, 2), (11, 3), (12, 0), (12, 2), (13, 1), (8, 1), (12, 3), (7, 0), (13, 2), (13, 0), (9, 4), (11, 4), (7, 1), (14, 1), (13, 3), (12, 4), (13, 4), (6, 0), (9, 5), (14, 0), (14, 2), (11, 5), (8, 5), (6, 1), (10, 5), (14, 3), (12, 5), (15, 1), (6, 2), (15, 0), (15, 2), (5, 0), (14, 4), (13, 5), (9, 6), (11, 6), (15, 3), (7, 5), (14, 5), (5, 1), (8, 6), (10, 6), (12, 6), (15, 4), (6, 3), (4, 0), (7, 4), (7, 6), (5, 2), (13, 6), (9, 7), (11, 7), (15, 5), (5, 3), (4, 1), (8, 7), (14, 6), (10, 7), (12, 7), (6, 4), (4, 2), (6, 6), (3, 0), (13, 7), (15, 6), (7, 7), (9, 8), (11, 8), (3, 1), (5, 4), (14, 7), (4, 3), (8, 8), (6, 7), (10, 8), (12, 8), (4, 4), (9, 9), (15, 7), (3, 2), (7, 8), (13, 8), (11, 9), (5, 6), (2, 0), (8, 9), (10, 9), (6, 8), (5, 5), (12, 9), (14, 8), (3, 3), (5, 7), (2, 1), (2, 2), (4, 6), (5, 8), (9, 10), (1, 0), (11, 10), (15, 8), (4, 5), (3, 4), (13, 9), (7, 9), (2, 3), (6, 9), (14, 9), (8, 10), (10, 10), (4, 7), (1, 1), (3, 5), (12, 10), (15, 9), (0, 0), (13, 10), (2, 4), (11, 11), (3, 6), (7, 10), (1, 2), (4, 8), (2, 5), (14, 10), (6, 10), (12, 11), (3, 7), (0, 1), (1, 3), (0, 2), (13, 11), (15, 10), (3, 8), (1, 4), (2, 6), (11, 12), (14, 11), (1, 5), (12, 12), (10, 12), (2, 7), (0, 3), (15, 11), (2, 8), (13, 12), (11, 13), (0, 4), (1, 6), (0, 5), (14, 12), (1, 7), (10, 13), (9, 12), (12, 13), (9, 13), (0, 6), (1, 8), (11, 14), (15, 12), (13, 13), (1, 9), (14, 13), (8, 12), (0, 7), (12, 14), (15, 13), (8, 13), (0, 8), (13, 14), (11, 15), (0, 9), (10, 15), (14, 14), (1, 10), (12, 15), (7, 12), (0, 10), (13, 15), (15, 14), (7, 13), (10, 16), (9, 15), (12, 16), (14, 15), (6, 12), (1, 11), (9, 16), (15, 15), (0, 11), (6, 13), (10, 17), (8, 15), (12, 17), (14, 16), (6, 14), (5, 12), (1, 12), (8, 16), (9, 17), (11, 17), (0, 12), (13, 17), (15, 16), (5, 13), (10, 18), (5, 14), (12, 18), (14, 17), (6, 15), (8, 17), (4, 12), (1, 13), (7, 15), (11, 18), (9, 18), (0, 13), (13, 18), (4, 13), (5, 15), (7, 16), (15, 17), (10, 19), (4, 14), (12, 19), (6, 16), (3, 12), (1, 14), (7, 17), (8, 18), (14, 18), (4, 15), (9, 19), (7, 18), (15, 18), (5, 16), (13, 19), (3, 13), (3, 11), (0, 14), (11, 19), (3, 14), (4, 16), (6, 17), (12, 20), (1, 15), (10, 20), (14, 19), (8, 19), (11, 20), (3, 15), (13, 20), (5, 17), (0, 15), (2, 15), (6, 18), (15, 19), (7, 19), (3, 10), (9, 20), (14, 20), (4, 17), (1, 16), (4, 10)]

iii. Map2:
   a. Path: [(4, 0), (4, 1), (4, 2), (3, 3), (2, 4), (1, 4), (0, 4)]
   b. Visited: [(4, 0), (3, 0), (4, 1), (3, 1), (4, 2), (2, 0), (2, 1), (4, 3), (3, 3), (4, 4), (2, 4), (1, 4), (1, 3), (0, 4)]

4. A* Search (35 pts)

Note: The algorithm code is highly influenced from the pseudocode of A * (few minor changes are made to make the pseudocode work as an a * algorithm)

4.1 (10 pts) Implement a euclidean distance heuristic for evaluating A*. Run your algorithm on maps map0.txt, map1.txt, and map2.txt and give the results. How does the path you get compare to uniform cost search? How do the states explored compare to uniform cost search? Did you notice any other differences in comparing the two?
Ans:
The path taken by both the algorithm is the same, but the nodes visited by A* is less when compared to the UCS. A* explores the nodes closer to the goal whereas UCS explores all the nodes based on the cost factor.

i. Map0:
Path:
[(4, 0), (3, 0), (2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (0, 3), (0, 4)]
Visited:
[(4, 0), (3, 0), (4, 1), (3, 1), (2, 0), (4, 2), (2, 1), (1, 0), (4, 3), (3, 3), (3, 4), (0, 0), (0, 1), (0, 2), (0, 3), (0, 4)]

ii. Map1:
Path:
Path: [(10, 1), (10, 2), (10, 3), (11, 3), (11, 4), (11, 5), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), (11, 11), (11, 12), (10, 12), (9, 12), (8, 12), (7, 12), (6, 12), (5, 12), (4, 12), (3, 12), (3, 11), (3, 10), (4, 10)]
Visited:
Visited: [(10, 1), (10, 2), (10, 3), (9, 2), (8, 2), (11, 1), (11, 2), (10, 0), (11, 3), (9, 0), (11, 4), (11, 5), (8, 0), (10, 5), (8, 1), (12, 1), (11, 6), (9, 5), (11, 0), (10, 6), (12, 2), (8, 5), (9, 6), (7, 0), (7, 1), (11, 7), (12, 3), (8, 6), (10, 7), (7, 5), (9, 7), (12, 4), (7, 6), (8, 7), (6, 0), (6, 1), (7, 7), (6, 2), (11, 8), (6, 3), (10, 8), (6, 4), (9, 8), (12, 5), (6, 6), (8, 8), (6, 7), (7, 8), (13, 1), (12, 0), (9, 4), (6, 8), (12, 6), (13, 2), (5, 0), (5, 1), (5, 2), (11, 9), (5, 3), (10, 9), (5, 4), (9, 9), (5, 5), (8, 9), (5, 6), (7, 9), (5, 7), (6, 9), (5, 8), (13, 3), (12, 7), (7, 4), (13, 4), (4, 0), (4, 1), (4, 2), (11, 10), (10, 10), (4, 3), (9, 10), (4, 4), (4, 5), (8, 10), (7, 10), (4, 6), (6, 10), (4, 7), (4, 8), (12, 8), (13, 5), (13, 0), (14, 1), (14, 2), (13, 6), (3, 0), (3, 1), (12, 9), (3, 2), (11, 11), (3, 3), (3, 4), (3, 5), (3, 6), (3, 7), (14, 3), (3, 8), (13, 7), (14, 4), (12, 10), (14, 0), (14, 5), (2, 0), (15, 1), (13, 8), (2, 1), (2, 2), (11, 12), (2, 3), (2, 4), (10, 12), (2, 5), (9, 12), (2, 6), (8, 12), (15, 2), (2, 7), (7, 12), (14, 6), (6, 12), (2, 8), (15, 3), (13, 9), (12, 11), (5, 12), (14, 7), (1, 0), (1, 1), (15, 4), (1, 2), (1, 3), (11, 13), (1, 4), (10, 13), (1, 5), (9, 13), (15, 0), (4, 12), (13, 10), (1, 6), (8, 13), (15, 5), (14, 8), (7, 13), (1, 7), (12, 12), (6, 13), (1, 8), (15, 6), (0, 0), (0, 1), (0, 2), (14, 9), (13, 11), (0, 3), (11, 14), (1, 9), (5, 13), (0, 4), (3, 12), (15, 7), (0, 5), (3, 11), (12, 13), (0, 6), (3, 10), (14, 10), (4, 10)]

iii. Map2:
Path: None

Visited: [(4, 0), (3, 0), (4, 1), (3, 1), (2, 0), (4, 2), (2, 1), (4, 3), (3, 3), (4, 4)]
Since there is no path from inti to goal state the algorithm returns path not found.

4.2 (10 pts) Now implement a Manhattan distance heuristic. Run it on the same three maps and perform the same analysis.

i. **Map0:**

Path:

[(4, 0), (4, 1), (4, 2), (4, 3), (4, 4), (3, 4), (2, 4), (1, 4), (0, 4)]

Visited:

[(4, 0), (3, 0), (4, 1), (4, 2), (4, 3), (4, 4), (3, 4), (2, 4), (1, 4), (0, 4)]

ii. **Map1**

Path: [(10, 1), (10, 2), (10, 3), (11, 3), (11, 4), (11, 5), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), (11, 11), (11, 12), (10, 12), (9, 12), (8, 12), (7, 12), (6, 12), (5, 12), (4, 12), (3, 12), (3, 11), (3, 10), (4, 10)]

Visited: [(10, 1), (10, 2), (10, 3), (9, 2), (8, 2), (11, 1), (10, 0), (11, 2), (9, 0), (8, 0), (7, 0), (6, 0), (6, 1), (6, 2), (6, 3), (5, 3), (6, 4), (4, 3), (5, 4), (5, 2), (5, 1), (5, 0), (7, 1), (8, 1), (11, 3), (11, 4), (11, 5), (10, 5), (4, 4), (4, 2), (5, 5), (4, 1), (4, 0), (9, 5), (11, 6), (10, 6), (4, 5), (5, 6), (9, 6), (8, 5), (11, 7), (10, 7), (4, 6), (8, 6), (5, 7), (9, 7), (7, 5), (11, 8), (11, 9), (10, 8), (4, 7), (7, 6), (6, 6), (7, 7), (7, 8), (6, 7), (8, 7), (10, 9), (6, 8), (6, 9), (5, 8), (9, 8), (11, 10), (4, 8), (9, 9), (9, 10), (8, 10), (7, 10), (8, 9), (10, 10), (7, 9), (8, 8), (6, 10), (3, 8), (3, 3), (12, 10), (11, 0), (12, 3), (12, 2), (12, 1), (12, 4), (7, 4), (12, 5), (11, 11), (3, 4), (3, 2), (3, 1), (3, 0), (9, 4), (3, 5), (12, 6), (12, 9), (12, 7), (3, 6), (12, 8), (3, 7), (2, 4), (11, 12), (13, 6), (13, 5), (2, 6), (13, 10), (12, 0), (13, 7), (13, 9), (12, 11), (2, 8), (13, 1), (13, 3), (2, 0), (13, 2), (2, 3), (2, 7), (2, 5), (13, 8), (2, 1), (2, 2), (13, 4), (10, 12), (9, 12), (8, 12), (7, 12), (6, 12), (5, 12), (4, 12), (1, 4), (14, 1), (12, 12), (1, 6), (11, 13), (14, 6), (14, 5), (14, 10), (1, 8), (13, 11), (13, 0), (14, 7), (14, 9), (1, 0), (14, 3), (14, 2), (1, 3), (14, 8), (1, 7), (1, 5), (1, 1), (14, 4), (1, 2), (10, 13), (9, 13), (8, 13), (7, 13), (6, 13), (5, 13), (3, 12), (3, 11), (3, 10), (4, 13), (1, 9), (4, 10)]

iii. **Map2:**

Path: None

Visited: [(4, 0), (3, 0), (4, 1), (4, 2), (4, 3), (4, 4), (3, 3), (3, 1), (2, 1), (2, 0)]

Since there is no path from inti to goal state the algorithm returns path not found.

4.3 (10 pts) Now extend the set of actions to allow the robot to move diagonally on the grid. Make diagonal action cost 1.5 as opposed to 1 for lateral and vertical moves. Run the same tests and analysis from 4.1 and 4.2 using both heuristics. Are these heuristics still admissable for our new problem? Why or why not? What is the effect of the new actions being added compared to the previous problems? What is the effect of the heuristic being admissable or not on the solutions your implementation gives?
Ans:
No, the heuristics are not admissible, because irrespective of the Euclidean or Manhattan distance, the algorithm takes the exact same path to reach the goal state.
The new set of actions, reduce the length of the path and the number of nodes visited is also reduced as the algorithm can move diagonally.
When the new actions are added, the algorithm takes a shorter path to reach the goal and less number of states are visited in this process.
The only effect using this approach is that the number of nodes explored by the algorithm reduces substantially if we use Manhattan and will be a bit more in Euclidean.

i. Eucledian:

a. Map0:
Path:
[(4, 0), (4, 1), (4, 2), (3, 3), (2, 4), (1, 4), (0, 4)]
Visited:
[(4, 0), (3, 1), (3, 0), (4, 1), (2, 1), (4, 2), (2, 0), (3, 3), (2, 4), (1, 4), (0, 4)]

b. Map1
Path:
[(10, 1), (10, 2), (10, 3), (11, 4), (11, 5), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), (11, 11), (10, 12), (9, 12), (8, 12), (7, 12), (6, 12), (5, 12), (4, 12), (3, 11), (4, 10)]
Visited:
[(10, 1), (9, 2), (10, 2), (10, 3), (9, 4), (8, 5), (8, 2), (7, 6), (9, 5), (6, 7), (8, 6), (5, 8), (7, 7), (6, 8), (7, 5), (9, 6), (6, 6), (8, 7), (7, 8), (11, 2), (5, 7), (6, 9), (9, 7), (11, 3), (11, 1), (8, 8), (4, 8), (5, 6), (10, 0), (7, 9), (9, 0), (11, 4), (10, 5), (8, 1), (9, 8), (4, 7), (6, 10), (11, 5), (8, 9), (7, 4), (10, 6), (8, 0), (7, 1), (7, 10), (4, 6), (11, 6), (9, 9), (12, 3), (11, 0), (10, 7), (3, 8), (6, 2), (6, 3), (12, 2), (6, 4), (7, 0), (12, 4), (8, 10), (12, 1), (5, 3), (5, 4), (5, 5), (11, 7), (3, 7), (6, 1), (10, 8), (12, 5), (9, 10), (4, 5), (4, 4), (5, 2), (3, 6), (6, 0), (11, 8), (12, 6), (4, 3), (5, 1), (10, 9), (3, 5), (12, 0), (13, 4), (2, 8), (13, 3), (4, 2), (13, 2), (12, 7), (5, 0), (11, 9), (3, 4), (2, 7), (13, 1), (13, 5), (2, 6), (10, 10), (4, 1), (3, 3), (12, 8), (13, 6), (2, 5), (11, 10), (4, 0), (3, 2), (2, 4), (13, 0), (13, 7), (3, 1), (12, 9), (1, 8), (14, 4), (1, 9), (14, 5), (14, 3), (1, 7), (2, 3), (14, 2), (14, 1), (1, 6), (3, 0), (11, 11), (13, 8), (2, 2), (14, 6), (1, 5), (12, 10), (1, 10), (1, 4), (2, 1), (10, 12), (9, 12), (14, 7), (8, 12), (13, 9), (7, 12), (1, 3), (14, 0), (2, 0), (11, 12), (6, 12), (0, 8), (0, 7), (15, 4), (15, 3), (1, 2), (12, 11), (15, 5), (15, 2), (0, 9), (0, 6), (1, 11), (14, 8), (15, 6), (15, 1), (5, 12), (9, 13), (0, 5), (1, 1), (13, 10), (0, 10), (8, 13), (10, 13), (0, 4), (7, 13), (15, 7), (1, 0), (4, 12), (14, 9), (0, 3), (6, 13), (11, 13), (12, 12), (15, 0), (0, 2), (13, 11), (1, 12), (0, 11), (5, 13), (15, 8), (0, 1), (3, 11), (4, 10)]

c. Map2:
Path: [(4, 0), (4, 1), (4, 2), (3, 3), (2, 4), (1, 4), (0, 4)]

Visited: [(4, 0), (3, 1), (3, 0), (4, 1), (2, 1), (4, 2), (2, 0), (3, 3), (2, 4), (1, 4), (0, 4)]

ii. Manhattan:

a. **Map0:**

Path: [(4, 0), (4, 1), (4, 2), (3, 3), (2, 4), (1, 4), (0, 4)]

Visited: [(4, 0), (3, 1), (2, 1), (3, 0), (4, 1), (4, 2), (3, 3), (2, 4), (1, 4), (0, 4)]

b. **Map1**

Path: [(10, 1), (10, 2), (10, 3), (11, 4), (11, 5), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), (11, 11), (10, 12), (9, 12), (8, 12), (7, 12), (6, 12), (5, 12), (4, 12), (3, 11), (4, 10)]

Visited:

Visited: [(10, 1), (9, 2), (8, 2), (10, 2), (10, 3), (9, 4), (8, 5), (7, 6), (6, 7), (5, 8), (4, 8), (5, 7), (6, 8), (4, 7), (6, 9), (6, 10), (7, 7), (6, 6), (5, 6), (4, 6), (7, 8), (7, 9), (7, 10), (8, 6), (7, 5), (8, 7), (8, 8), (8, 9), (8, 10), (9, 5), (9, 6), (3, 8), (9, 7), (9, 8), (9, 9), (9, 10), (5, 5), (4, 5), (3, 7), (7, 4), (3, 6), (6, 4), (5, 4), (4, 4), (10, 6), (10, 7), (10, 10), (10, 5), (7, 1), (6, 2), (5, 3), (4, 3), (6, 3), (5, 2), (4, 2), (8, 1), (10, 9), (10, 8), (6, 1), (5, 1), (4, 1), (2, 8), (9, 0), (11, 4), (11, 3), (11, 2), (3, 4), (8, 0), (11, 5), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), (3, 5), (7, 0), (6, 0), (5, 0), (4, 0), (11, 1), (10, 0), (2, 7), (3, 3), (3, 2), (2, 6), (12, 8), (12, 7), (2, 5), (12, 10), (12, 9), (3, 1), (12, 6), (12, 3), (1, 9), (12, 5), (12, 4), (1, 10), (11, 0), (1, 8), (3, 0), (11, 11), (2, 4), (12, 2), (12, 1), (1, 7), (2, 3), (2, 2), (13, 9), (13, 8), (1, 6), (13, 7), (13, 10), (13, 5), (13, 4), (0, 10), (13, 6), (12, 11), (1, 5), (2, 1), (13, 3), (0, 9), (1, 11), (10, 12), (9, 12), (8, 12), (7, 12), (6, 12), (5, 12), (4, 12), (12, 0), (11, 12), (2, 0), (0, 8), (1, 4), (13, 2), (13, 1), (1, 3), (0, 7), (14, 5), (14, 10), (14, 9), (14, 8), (14, 6), (14, 7), (0, 11), (13, 11), (1, 2), (0, 6), (14, 4), (9, 13), (3, 11), (4, 10)]

c. **Map2:**

Path: [(4, 0), (4, 1), (4, 2), (3, 3), (2, 4), (1, 4), (0, 4)]

Visited: [(4, 0), (3, 1), (2, 1), (3, 0), (4, 1), (4, 2), (3, 3), (2, 4), (1, 4), (0, 4)]

Since there is no path from inti to goal state the algorithm returns path not found.

4.4 (5 pts) Run Uniform Cost Search with diagonal actions described in 4.3. Compare the Uniform Cost Search and A* with diagonal actions.

Ans:
The path found by both A* and UCS are the same, whereas the number of nodes visited by UCS is substantially large when compared to the nodes visited by A*.

5 Self Analysis (5 pts)

5.1 (1 pt) What was the hardest part of the assignment for you?

Getting the UCS to work. It took me a lot of debugging to finally understand where I went wrong and rectify it.

5.2 (1 pt) What was the easiest part of the assignment for you?

Modifying the dfs algorithm to work as a bfs algorithm. I just had to convert the stack to a queue and it worked.

5.3 (1 pt) What problem(s) helped further your understanding of the course material?

Practically working on these algorithms helped me understand all the algorithms in a much better way.

5.4 (1 pt) Did you feel any problems were tedious and not helpful to your understanding of the material?

Not at all.

5.5 (1 pt) What other feedback do you have about this homework?

None!