**Experiment-2**

Jaiwin Shah, Manan Shah

2019130058, 2019130059

TE Comps

Batch C

Aim: To implement the maze solver using BFS and DFS

Theory:

• Depth-first Search (DFS):

1. DFS always expands DEPTH-FIRST the deepest node in the current frontier of the search tree.

2. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.

3. DFS uses a LIFO queue.

4. Visits children before siblings.

• Breadth-first Search (BFS):

1. BFS is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.

2. All the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

3. BFS uses a FIFO queue.

4. Visits siblings before children.

Problem Description:

Code:

from collections import deque

# to keep track of the blocks of maze

class Grid\_Position:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

# each block will have its own position and cost of steps taken

class Node:

    def \_\_init\_\_(self, pos: Grid\_Position, cost):

        self.pos = pos

        self.cost = cost

#BFS algo for the maze

def bfs(Grid, dest: Grid\_Position, start: Grid\_Position):

    # to get neighbours of current node

    adj\_cell\_x = [-1, 0, 0, 1]

    adj\_cell\_y = [0, -1, 1, 0]

    m, n = (len(Grid), len(Grid))

    visited\_blocks = [[False for i in range(m)]

                for j in range(n)]

    visited\_blocks[start.x][start.y] = True

    queue = deque()

    sol = Node(start, 0)

    queue.append(sol)

    cells = 4

    cost = 0

    while queue:

        current\_block = queue.popleft()  # Dequeue the front cell

        current\_pos = current\_block.pos

        if current\_pos.x == dest.x and current\_pos.y == dest.y:

            print("Algorithm used = BFS")

            print("Path found!!")

            print("Total nodes visited = ", cost)

            return current\_block.cost

        if current\_block not in visited\_blocks:

            visited\_blocks[current\_pos.x][current\_pos.y] = True

            cost = cost + 1

        x\_pos = current\_pos.x

        y\_pos = current\_pos.y

        for i in range(cells):

            if x\_pos == len(Grid) - 1 and adj\_cell\_x[i] == 1:

                x\_pos = current\_pos.x

                y\_pos = current\_pos.y + adj\_cell\_y[i]

            if y\_pos == 0 and adj\_cell\_y[i] == -1:

                x\_pos = current\_pos.x + adj\_cell\_x[i]

                y\_pos = current\_pos.y

            else:

                x\_pos = current\_pos.x + adj\_cell\_x[i]

                y\_pos = current\_pos.y + adj\_cell\_y[i]

            if x\_pos < 12 and y\_pos < 12 and x\_pos >= 0 and y\_pos >= 0:

                if Grid[x\_pos][y\_pos] == 1:

                    if not visited\_blocks[x\_pos][y\_pos]:

                        next\_cell = Node(Grid\_Position(x\_pos, y\_pos),

                                       current\_block.cost + 1)

                        visited\_blocks[x\_pos][y\_pos] = True

                        queue.append(next\_cell)

    return -1

def create\_node(x, y, c):

    val = Grid\_Position(x, y)

    return Node(val, c + 1)

#dfs algo for maze

def dfs(Grid, dest: Grid\_Position, start: Grid\_Position):

    adj\_cell\_x = [1, 0, 0, -1]

    adj\_cell\_y = [0, 1, -1, 0]

    m, n = (len(Grid), len(Grid))

    visited\_blocks = [[False for i in range(m)]

               for j in range(n)]

    visited\_blocks[start.x][start.y] = True

    stack = deque()

    sol = Node(start, 0)

    stack.append(sol)

    neigh = 4

    neighbours = []

    cost = 0

    while stack:

        current\_block = stack.pop()

        current\_pos = current\_block.pos

        if current\_pos.x == dest.x and current\_pos.y == dest.y:

            print("Algorithm used = DFS")

            print("Path found!!")

            print("Total nodes visited = ", cost)

            return current\_block.cost

        x\_pos = current\_pos.x

        y\_pos = current\_pos.y

        for i in range(neigh):

            if x\_pos == len(Grid) - 1 and adj\_cell\_x[i] == 1:

                x\_pos = current\_pos.x

                y\_pos = current\_pos.y + adj\_cell\_y[i]

            if y\_pos == 0 and adj\_cell\_y[i] == -1:

                x\_pos = current\_pos.x + adj\_cell\_x[i]

                y\_pos = current\_pos.y

            else:

                x\_pos = current\_pos.x + adj\_cell\_x[i]

                y\_pos = current\_pos.y + adj\_cell\_y[i]

            if x\_pos != 12 and x\_pos != -1 and y\_pos != 12 and y\_pos != -1:

                if Grid[x\_pos][y\_pos] == 1:

                    if not visited\_blocks[x\_pos][y\_pos]:

                        cost += 1

                        visited\_blocks[x\_pos][y\_pos] = True

                        stack.append(create\_node(x\_pos, y\_pos, current\_block.cost))

    return -1

def main():

    maze = [[0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],

            [0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0],

            [0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0],

            [0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0],

            [0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1],

            [0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0],

            [0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],

            [0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],

            [0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0],

            [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0],

            [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0],

            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

    destination = Grid\_Position(10, 0)

    starting\_position = Grid\_Position(4, 11)

    res = bfs(maze, destination, starting\_position)

    if res != -1:

        print("Shortest path steps = ", res)

    else:

        print("Path does not exit")

    print()

    res2 = dfs(maze, destination, starting\_position)

    if res2 != -1:

        print("Steps with backtracking = ", res2)

    else:

        print("Path does not exit")

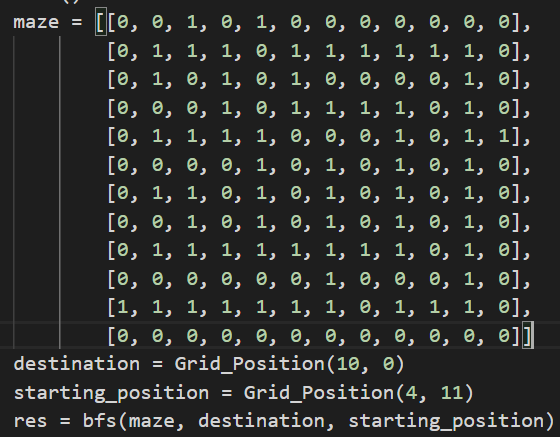
# Press the green button in the gutter to run the script.

if \_\_name\_\_ == '\_\_main\_\_':

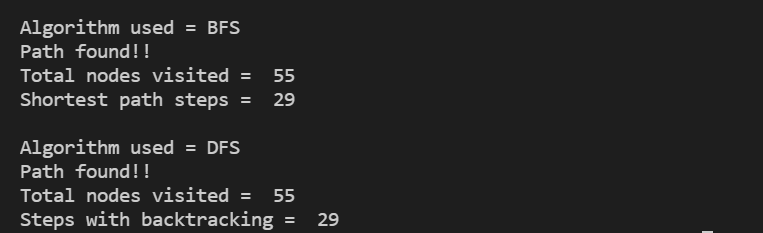
    print("main start\n")

    main()

Input:



Output:



Conclusion:

**Problem Description:**

• Available actions are Left, Right, Up, down.

• The termination criterion is that the Maze is completed by the agent. • Agent knows the size of Maze (grid 12 x 12), the content of the cell they land in and the location of the landing cell (coordinates).

• The performance of an agent is calculated after the termination criteria is met. The performance measure of an agent is the (number of steps used) to complete the maze. Maze is completed when agent reaches 11 x 1 block.

• The perception is given by the environment and includes, cell coordinates and if the current piece in the cell is empty or blocked.

So, we deduce that DFS searches until it reaches the leaves of the tree or the last node of the graph that has no further successors, whereas BFS investigates all the nodes of a specific frontier before moving on to the next frontier. BFS is complete, whereas DFS isn't. BFS looks better for nodes that are closer to the source and DFS is better for nodes that are further away from the source.