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]
```

```python
            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:

```
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)
```

Output:

```
Algorithm used = BFS
Path found!!
Total nodes visited =  55
Shortest path steps =  29

Algorithm used = DFS
Path found!!
Total nodes visited =  55
Steps with backtracking =  29
```

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.