Green University of Bangladesh
Department of Computer Science and Engineering (CSE)

Faculty of Sciences and Engineering

Semester: (Spring, Year: 2025), B.Sc. in CSE (Day)

Lab Report NO #01

Course Title: Artificial Intelligence Lab

Course Code: CSE-316 Section: 222_D2

Student Details

Name		ID
1.	Md. Zehadul Islam	222902069

Lab Date : 02/03/2025

Submission Date : 09.03.2025

Course Teacher's Name : Md. Abu Rumman Refat

Lab Report Status

Marks: …… **Signature:.....**

Comments: ..... **Date:.....**

1. TITLE OF THE LAB REPORT EXPERIMENT

- ✓ Write a program where a robot traverses on a 2D plane using the BFS algorithm and goes from start to destination point. Now print the path it will traverse to reach the destination.
- ✓ Write a program to find the topological order of node traversal of the robot on the 2D graph plane (DFS).

2. OBJECTIVES/AIM

The main purpose of this HTML code is to create a personal class routine using basic HTML. The key objectives are:

- **Robot Traversal System on a 2D Plane:**

  => Implement a system where a robot navigates a 2D grid.
  => Use Breadth-First Search (BFS) to find the shortest path from a starting point to a destination.
  => Print the shortest path found by BFS.

- **Topological Sorting of Nodes in a Directed Graph:**

  => Represent the 2D plane as a directed graph.
  => Use Depth-First Search (DFS) to determine the topological order of node traversal.
  => Print the topological order obtained from DFS.

3. PROCEDURE / ANALYSIS / DESIGN

- **BFS (Breadth-First Search):** It is used to find the shortest path in a 2D grid by exploring all possible moves level by level. The algorithm starts from the source, marks visited cells, and tracks the path using a parent dictionary. Once the goal is reached, the shortest path is reconstructed by backtracking through the parent dictionary.
- **DFS (Depth-First Search) for Topological Sorting:** The 2D grid is converted into a directed graph of walkable cells. DFS is used to explore nodes deeply before backtracking, marking each node as visited. The nodes are added to the topological order list after all their neighbors are visited. Once DFS completes, the final topological order is returned.

4. IMPLEMENTATION

1) BFS Code =>

```
from collections import deque

class Location:
    def __init__(self, x, y, distance):
        self.x = x
        self.y = y
        self.distance = distance

class MazeSolver:
    def __init__(self):
        self.found_goal = False
        self.total_moves = 0
        self.trail = {}
        self.destination = None
        self.start_point = None
        self.directions = [(1, -1), (-1, 0), (-1, 0), (0, -1), (1,
1),(0, 1)]
        self.grid_size = 0


    def initialize(self):


        maze = [
            [1, 0, 1, 0, 1],
            [1, 0, 1, 1, 1],
            [1, 1, 1, 0, 1],
            [1, 1, 1, 1, 1],
            [1, 0, 1, 0, 1],
            [1, 0, 1, 0, 0]
        ]
        self.grid_size = len(maze)
        start_x, start_y = 2, 0
        self.start_point = Location(start_x, start_y, 0)
        goal_x, goal_y = 4, 4
        self.destination = Location(goal_x, goal_y, 0)

        self.bfs_traversal(maze)

        if self.found_goal:
            print("Destination reached are moves => ",
self.total_moves)
            self.display_path()
        else:
            print("Destination not reachable")

    def move_direction(self, dx, dy):
        if dx == 1 and dy == 0:
            return "Move Down"
        elif dx == -1 and dy == 0:
```

```
            return "Move Up"
        elif dx == 0 and dy == 1:
            return "Move Right"
        elif dx == 0 and dy == -1:
            return "Move Left"

    def bfs_traversal(self, maze):
        queue = deque()
        queue.append(self.start_point)
        self.trail[(self.start_point.x, self.start_point.y)] = None

        while queue:
            current = queue.popleft()
            for dx, dy in self.directions:
                new_x, new_y = current.x + dx, current.y + dy
                if 0 <= new_x < self.grid_size and 0 <= new_y <
self.grid_size and maze[new_x][new_y] == 1:
                    new_distance = current.distance + 1
                    if (new_x, new_y) not in self.trail:
                        self.trail[(new_x, new_y)] = (current.x,
current.y)

                    if new_x == self.destination.x and new_y ==
self.destination.y:
                        self.found_goal = True
                        self.total_moves = new_distance
                        return
                    maze[new_x][new_y] = 0
                    next_location = Location(new_x, new_y,
new_distance)
                    queue.append(next_location)

    def display_path(self):
        route = []
        current = (self.destination.x, self.destination.y)
        while current is not None:
            route.append(current)
            current = self.trail[current]
        route.reverse()
        print("Here Path -> to -> goal=", route)

if __name__ == "__main__":
    solver = MazeSolver()
    solver.initialize()
```

2) DFS Code =>

```
def robot_topological_sort(matrix):

    def dfs_sort(graph):
```

```
        def explore(node):
            if node in visited_nodes:
                return
            visited_nodes.add(node)
            for neighbor in graph[node]:
                explore(neighbor)

            sorted_order.insert(0, node)

        visited_nodes = set()
        sorted_order = []

        for node in graph.keys():
            if node not in visited_nodes:
                explore(node)

        return sorted_order

    total_rows, total_cols = len(matrix), len(matrix[0])
    adjacency_list = {}

    for row in range(total_rows):
        for col in range(total_cols):
            if matrix[row][col] == 1:
                current_cell = (row, col)
                adjacent_cells = []

                if row - 1 >= 0 and matrix[row - 1][col] == 1:

                    adjacent_cells.append((row - 1, col))

                if row + 1 < total_rows and matrix[row + 1][col] == 1:


                    adjacent_cells.append((row + 1, col))

                if col - 1 >= 0 and matrix[row][col - 1] == 1:

                    adjacent_cells.append((row, col - 1))

                if col + 1 < total_cols and matrix[row][col + 1] == 1:

                    adjacent_cells.append((row, col + 1))

                adjacency_list[current_cell] = adjacent_cells

    return dfs_sort(adjacency_list)


matrix = [
    [1, 1, 0, 1],
    [1, 0, 1, 0],
    [1, 0, 0, 1],
    [0, 1, 0, 1],
```

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

]


topological_order = robot_topological_sort(matrix)
print("Topological Order:", topological_order)
```

5. TEST RESULT / OUTPUT

1. BFS Pathfinding Output:

```
PS E:\University\7th Semester\AI\AI Lab> python -u "e:\University\7th Semester\AI\AI Lab\lab1.py"
Destination reached are moves =>  4
Here Path -> to -> goal= [(2, 0), (3, 1), (3, 2), (3, 3), (4, 4)]
PS E:\University\7th Semester\AI\AI Lab>
```

2. DFS Topological Sort Output:

```
PS E:\University\7th Semester\AI\AI Lab> python -u "e:\University\7th Semester\AI\AI Lab\dfs.py"
Topological Order: [(5, 3), (3, 1), (4, 1), (4, 2), (5, 1), (5, 0), (4, 0), (2, 3), (3, 3), (1, 2), (0, 3), (0, 0), (0, 1), (1, 0), (2, 0)]
PS E:\University\7th Semester\AI\AI Lab>
```

6. ANALYSIS AND DISCUSSION

The BFS algorithm finds the shortest path by exploring the grid level by level. It ensures the robot reaches the destination with minimal moves. In contrast, DFS performs a deep traversal to generate a topological order of nodes. Both algorithms explore 2D grids efficiently but differ in their approach: BFS focuses on distance, while DFS prioritizes node dependencies.

7. SUMMARY

The code solves two separate problems: pathfinding using BFS and topological sorting using DFS on a 2D grid. The BFS algorithm ensures the shortest path to the destination in a maze, while the DFS ensures a valid node traversal order in a grid. Both algorithms are fundamental in navigating and processing grids or graphs efficiently.