| | | |
|---|---|---|
| | Marathwada Mitra Mandal's | |
| | **Institute of Technology, Lohgaon Pune - 47** | |
| | **Department of Artificial Intelligence and Data Science** | |

Semester -I A.Y.2025-26 Sub.: - Artificial Intelligence Lab Class: SE

---

**Assignment 07:** Implementing a Maze Solver using AI Search Algorithms (BFS & DFS).

**Objective:** Solve AI search problems using Graph Search Algorithms..

---

**Explanation:**

**Maze Representation:**

- **The maze is represented as a list of lists (a 2D grid).**
- **Each inner list represents a row.**
- **'#' indicates a wall.**
- **' ' indicates an open path.**
- **'S' is the starting point.**
- **'E' is the ending point.**

**Common Elements for Both BFS and DFS:**

1. **find_start_end(maze) (implicit in the code):**
    - **The first step is to iterate through the maze to locate the start ('S')**

and end ('E') coordinates.

2. directions:

   ○ A list of tuples [(0, 1), (0, -1), (1, 0), (-1, 0)] represents possible movements: right, left, down, up.

3. is_valid(r, c, rows, cols, maze, visited) (implicit in the code):

   ○ Checks if a given cell (r, c) is within the maze boundaries, is not a wall ('#'), and has not been visited yet.

**Breadth-First Search (BFS):**

- Goal: Find the shortest path from the start to the end.
- Data Structure: collections.deque (double-ended queue).
- How it works:
  - ○ Starts at the start node.
  - ○ Explores all its immediate neighbors.
  - ○ Then explores all unvisited neighbors of those neighbors, and so on. ○ It expands layer by layer, ensuring that the first time it reaches the end node, it has found the shortest path.
- queue = collections.deque([(start, [start])]):
  - ○ Each item in the queue is a tuple: (current_position, path_taken_to_reach_here). This is crucial for reconstructing the path. ● visited = set([start]):
  - ○ Keeps track of all cells that have been added to the queue to prevent cycles and redundant processing.
- queue.popleft(): Removes the element from the front of the queue (FIFO - First-In, First-Out).

**Depth-First Search (DFS):**

- **Goal: Find *any* path from the start to the end. It doesn't guarantee the shortest path.**
- **Data Structure: A list used as a stack.**
- **How it works:**
  - **Start at the start node.**

○ **Explores as far as possible along each branch before backtracking. ○ It goes deep into one path before trying another.**

- **stack = [(start, [start])]:**

○ **Similar to BFS, each item in the stack is (current_position, path_taken_to_reach_here).**

- **visited = set([start]):**

○ **Keeps track of visited cells.**

- **stack.pop(): Removes the element from the end of the list (LIFO - Last-In, First-Out), simulating a stack.**

**print_path(maze, path):**

- **This helper function takes the original maze and the found path, then prints the maze with the path marked by '*'.**

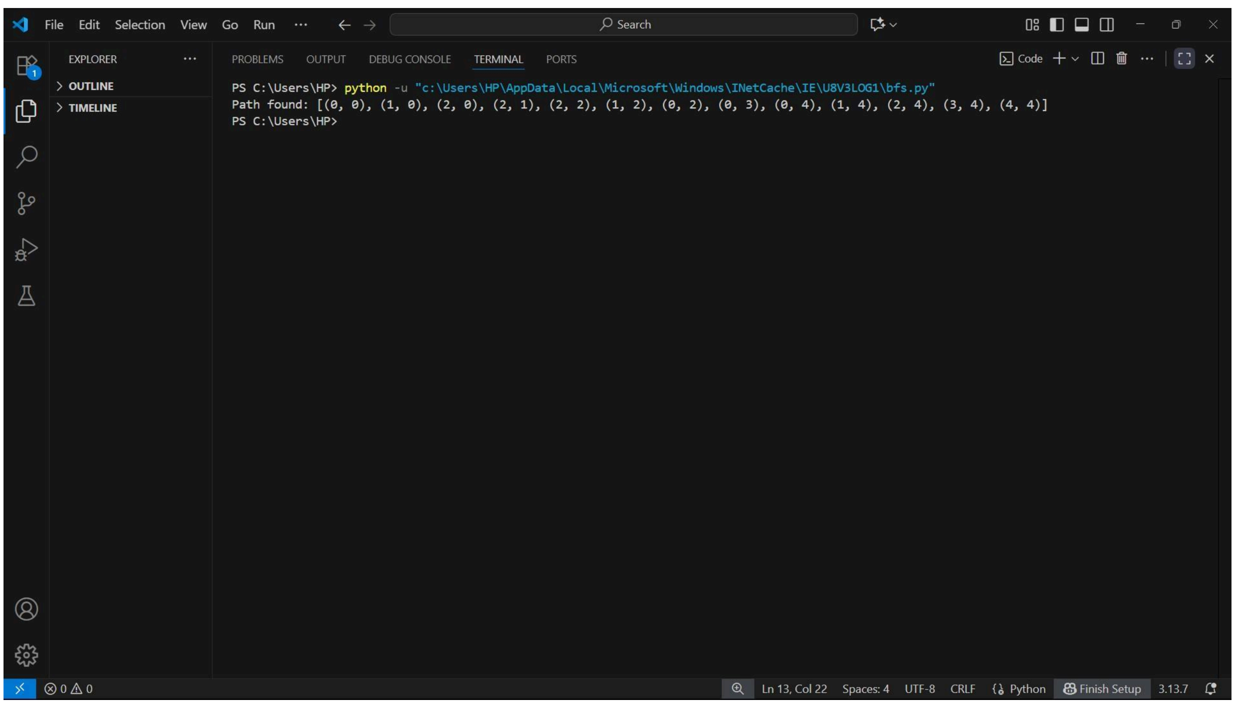
**Choosing Between BFS and DFS for Maze Solving:**

- BFS is generally preferred for maze solving when you need the shortest path because it explores evenly in all directions from the start.
- DFS is simpler to implement recursively (though the iterative stack version is shown here). It can find *a* path quickly, but not necessarily the shortest. If the maze has a very long, winding path to the solution while a shorter one exists, DFS might explore the longer one first.

```
def dfs(maze, start, end):
    stack = [start] # Initialize stack with start position
    visited = set() # Track visited positions
    parent = {start: None} # To reconstruct the path

    while stack:
        position = stack.pop()
        x, y = position

        # Check if we've reached the end
        if position == end:
                    path = []
            while position is not None:
                path.append(position)
                position = parent[position]
            path.reverse()
            return path

        # Mark the current cell as visited
        visited.add((x, y))

        # Explore neighbors (up, down, left, right)
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            new_x, new_y = x + dx, y + dy
            new_pos = (new_x, new_y)

            # Check bounds and conditions
            if (0 <= new_x < len(maze) and
                0 <= new_y < len(maze[0]) and
                maze[new_x][new_y] == 0 and
                new_pos not in visited and
                new_pos not in stack): # prevent duplicates
                stack.append(new_pos)
                parent[new_pos] = position
    return None # No path found


# Example maze: 0 -> open path, 1 -> wall
maze = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
```

```
]

# Start and end positions
start = (0, 0)
end = (4, 4)

# Solve the maze
path = dfs(maze, start, end)
if path:
    print("Path found:", path)
else:
    print("No path exists.")
```

## Output :



```
from collections import deque

def bfs(maze, start, end):
    # Directions: up, right, down, left
    directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]
    queue = deque([start])
    visited = set([start])      parent = {start: None}
    while queue:
        current = queue.popleft()
        if current == end:
                    path = []
            while current is not None:
                path.append(current)
                current = parent[current]
            path.reverse()
            return path

        for direction in directions:
            next_cell = (current[0] + direction[0], current[1] + direction[1])

            if (0 <= next_cell[0] < len(maze) and
                0 <= next_cell[1] < len(maze[0]) and
                maze[next_cell[0]][next_cell[1]] != '#' and
                next_cell not in visited):
                queue.append(next_cell)
                visited.add(next_cell)
                parent[next_cell] = current

    return None


# Example maze
maze = [
    ['S', '.', '.', '#', '.', '.', '.'],
    ['.', '#', '.', '#', '.', '#', '.'],
    ['.', '#', '.', '.', '.', '.', '.'],
    ['.', '.', '#', '#', '#', '.', '.'],
    ['.', '#', '.', '.', '.', '#', '.'],
    ['.', '#', '#', '#', '.', '#', '.'],
    ['.', '.', '.', '.', '.', '.', 'E'],
]

start = (0, 0) # Starting position
end = (6, 6)    # Ending position (exit)
```

```
# Run BFS to find the path
path = bfs(maze, start, end)

if path:
    print("Path found:", path)
else:
    print("No path exists.")
```

## Output :