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.
