Maze Solver - Depth-First Search (DFS)

Overview

This project implements a **maze solver** using the **Depth-First Search (DFS)** algorithm in Python. The maze is represented as a 2D grid, and the goal is to find a path from the **start** point (denoted by 3) to the **end** point (denoted by 2). The solver uses backtracking to explore possible paths, and when it encounters a dead end, it "backtracks" to find alternative paths.

Features

- **Depth-First Search (DFS)** algorithm.
- **Backtracking** to handle dead-ends.
- **Marking visited cells** to avoid re-exploring.
- **Valid move checking** to ensure the solver only moves on walkable paths.
- **Solution path output** showing the series of moves from start to end.

Maze Representation

The maze is a 2D grid (list of lists), where each element can have one of the following values:

- 1: Wall or blocked cell (cannot be traversed).
- 0: Open path or empty cell (can be traversed).
- 3: Start point.
- 2: End point.

Example maze:

```
maze = [[1,1,1,1,1,2,1,1,1,1],
        [1,0,0,0,0,0,0,0,0,1],
        [1,0,1,0,1,1,1,0,0,1],
        [1,0,1,0,0,0,0,0,0,1],
        [1,0,1,1,1,1,1,1,0,1],
        [1,0,0,0,0,0,1,1,0,1],
        [1,0,1,1,0,0,1,1,0,1],
        [1,1,1,1,3,0,1,1,0,1]
        ]
```

- The **start point** (3) is at position (7,4).
- The **end point** (2) is at position (0,5).

The Algorithm

The algorithm follows these steps to solve the maze:

1. **Find the Start and End Points**:
    - The function find_start_end(maze) scans the grid and identifies the coordinates of the start (3) and end (2) points.
2. **Depth-First Search (DFS) with Backtracking**:
    - The solve_maze(maze, start, end) function is the core of the algorithm. It uses a recursive function (dfs) to explore the maze.
    - The DFS function checks all possible directions (up, right, down, left) and attempts to reach the end point from the current position.
    - **Backtracking**: If the algorithm hits a dead-end (a position with no valid moves), it pops the current position from the path and backtracks to try another direction.
3. **Valid Move Check**:
    - The function is_valid_move(maze, visited, row, col) ensures that the algorithm only moves to unvisited, walkable cells (0 or 2), within the bounds of the maze.
4. **Solution Path**:
    - The solution path, if found, is a list of coordinates representing the cells from the start to the end point. If no path is found, the output is empty.

Code Explanation

Functions

1. **find_start_end(maze)**
    - This function scans the maze and returns the coordinates of the start and end points.

```python
def find_start_end(maze):
    start = None
    end = None

    for i, row in enumerate(maze):
        for j, col in enumerate(row):
            if col == 3:
                start = (i,j)
            if col == 2:
                end = (i,j)

    return start, end
```

2. **is_valid_move(maze, visited, row, col)**
    - This function checks if the current position is a valid move:
        - It ensures the position is within bounds of the maze.
        - It ensures the position is not a wall (1).
        - It ensures the position has not been visited before.

```
def is_valid_move(maze, visited, row, col):
    rows = len(maze)
    cols = len(maze[0])

    if (0 <= row < rows) and (0 <= col < cols) and maze[row][col] != 1 and not visited[row][col]:
        return True
    return False
```

3. **solve_maze(maze, start, end)**
    - This function contains the DFS logic to solve the maze. It recursively tries to move in all four directions (up, right, down, left).
    - If a valid path is found to the end point, it returns the solution path.
    - If a dead-end is encountered, it backtracks by popping the last cell from the path and trying another direction.

```
def solve_maze(maze, start, end):
    directions = [(-1,0),
                  (0,1),
                  (1,0),
                  (0,-1),
                  ]

    paths = []
    visited = [[False for _ in range(len(maze[0]))] for _ in range(len(maze))]

    def dfs(row, col):
        if (row, col) == end:
            paths.append((row, col))
            return True

        if is_valid_move(maze, visited, row, col):
            visited[row][col] = True
            paths.append((row, col))

            for x, y in directions:
                if dfs(row + x, col + y):
                    return True
            paths.pop()

        return False

    start_row, start_col = start
    dfs(start_row, start_col)
    return paths

start, end = find_start_end(maze)

solution_paths = solve_maze(maze, start, end)

print(solution_paths)
```

How to Run the Code

Prerequisites

- Python 3.x

Steps to Run:

1. Copy the code into a Python file (e.g., maze_solver.py).
2. Modify the maze variable if needed to test different mazes.
3. Run the script using the Python command:

```
python maze.py
```

4. The program will output the solution path, which is a list of coordinates. If no solution is found, it will print an empty list.

Example Output

For the given example maze, the output would be:

```
[(7, 4), (6, 4), (5, 4), (5, 3), (5, 2), (4, 2), (4, 1), (3, 1), (2, 1), (1, 1), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5)]
```

This list represents the path from the start point (7, 4) to the end point (0, 5).