Exp 1: Design and solve a Maze problem using Depth First Search and Breadth First Search

Program: Using DFS

```
class MazeSolver:
    def __init__(self, maze):
        self.maze = maze
        self.rows = len(maze)
        self.cols = len(maze[0])
        self.visited = [[False] * self.cols for _ in range(self.rows)]
        self.solution = [[0] * self.cols for _ in range(self.rows)]

    def is_valid_move(self, row, col):
        return 0 <= row < self.rows and 0 <= col < self.cols and not 
self.visited[row][col] and self.maze[row][col] == 1

    def depth_first_search(self, row, col):
        if row == self.rows - 1 and col == self.cols - 1:
            self.solution[row][col] = 1  # Mark the destination cell
            return True

        if self.is_valid_move(row, col):
            self.visited[row][col] = True
            self.solution[row][col] = 1

            # Explore in all four directions: up, down, left, right
            directions = [(-1, 0), (0, -1), (1, 0), (0, 1)]
            for dr, dc in directions:
                if self.depth_first_search(row + dr, col + dc):
                    return True

            # If no valid move found, backtrack
            self.solution[row][col] = 0
            return False

        return False

    def solve_maze(self):
        if not self.depth_first_search(0, 0):
            print("No solution exists.")
        else:
            self.print_solution()

    def print_solution(self):
        for row in self.solution:
            print(" ".join(map(str, row)))
```

```python
# Example usage:
maze = [
    [1, 0, 1, 1, 1],
    [1, 1, 1, 0, 1],
    [0, 0, 0, 1, 1],
    [1, 1, 1, 1, 1],
    [1, 1, 0, 0, 1]
]

solver = MazeSolver(maze)
solver.solve_maze()
```

Explanation:

Initialization (__init__ method):

The constructor takes a maze as an argument and initializes various attributes such as the maze itself, the number of rows and columns, a 2D array to keep track of visited cells (visited), and a 2D array to store the solution path (solution).

is_valid_move method:

Checks if a given cell (specified by its row and column) is a valid move. The move is considered valid if the cell is within the boundaries of the maze, has not been visited before, and contains a value of 1 (indicating an open path).

depth_first_search method:

Recursively explores the maze using depth-first search. It starts from the top-left corner and explores in all four directions (up, down, left, right) until it reaches the bottom-right corner.

If the destination is reached, the method marks the solution path in the solution matrix and returns True.

If a valid move is found, it marks the current cell as visited and continues the exploration in all possible directions.

If no valid move is found in any direction, it backtracks by marking the current cell as unvisited and returns False.

solve_maze method:

Calls the depth_first_search method to attempt to solve the maze starting from the top-left corner.

If no solution is found, it prints "No solution exists." Otherwise, it calls the print_solution method.

print_solution method:

Prints the solution path by iterating through the solution matrix and printing each row.

Example usage:

An example maze is provided, and an instance of the MazeSolver class is created with this maze.

The solve_maze method is then called to attempt to solve the maze.

Program: Using DFS with visualization

```
class MazeSolver:
    def __init__(self, maze):
        self.maze = maze
        self.rows = len(maze)
        self.cols = len(maze[0])
        self.visited = [[False] * self.cols for _ in range(self.rows)]
        self.solution = [[0] * self.cols for _ in range(self.rows)]

    def is_valid_move(self, row, col):
        return 0 <= row < self.rows and 0 <= col < self.cols and not
self.visited[row][col] and self.maze[row][col] == 1

    def depth_first_search(self, row, col):
        if row == self.rows - 1 and col == self.cols - 1:
            self.solution[row][col] = 1  # Mark the destination cell
            return True

        if self.is_valid_move(row, col):
            self.visited[row][col] = True
            self.solution[row][col] = 1

            # Explore in all four directions: up, down, left, right
            directions = [(-1, 0), (0, -1), (1, 0), (0, 1)]
            for dr, dc in directions:
                if self.depth_first_search(row + dr, col + dc):
                    return True

            # If no valid move found, backtrack
            self.solution[row][col] = 0
            return False

        return False

    def solve_maze(self):
        if not self.depth_first_search(0, 0):
            print("No solution exists.")
        else:
            self.print_solution()

    def print_solution(self):
        for row in self.solution:
            print(" ".join(map(str, row)))


# Example usage:
maze = [
    [1, 0, 1, 1, 1],
    [1, 1, 1, 0, 1],
    [0, 0, 0, 1, 1],
    [1, 1, 1, 1, 1],
    [1, 1, 0, 0, 1]
]

solver = MazeSolver(maze)
solver.solve_maze()

# Create a figure and axes object.
fig, ax = plt.subplots()

# Plot the maze.
ax.imshow(maze, cmap="Greys")

# Plot the solution.
ax.imshow(solver.solution, cmap="Greens")

# Show the plot.
plt.show()
```

Explanation:

Creating an instance of MazeSolver:

An instance of the MazeSolver class is created with the provided maze.

Solving the maze:

The solve_maze method is called to attempt to solve the maze using the depth-first search algorithm.

Visualization using matplotlib:

A figure and axes object are created using plt.subplots() from the matplotlib library.

The original maze is plotted using ax.imshow(maze, cmap="Greys"), where cmap="Greys" sets the color map to grayscale.

The solution path is plotted using ax.imshow(solver.solution, cmap="Greens"), where cmap="Greens" sets the color map to green.

The resulting plot is displayed using plt.show().

Program: Using BFS with visualization

```
from collections import deque
import matplotlib.pyplot as plt

class MazeSolverBFS:
    def __init__(self, maze):
        self.maze = maze
        self.rows = len(maze)
        self.cols = len(maze[0])
        self.visited = [[False] * self.cols for _ in range(self.rows)]
        self.solution = [[0] * self.cols for _ in range(self.rows)]

    def is_valid_move(self, row, col):
        return 0 <= row < self.rows and 0 <= col < self.cols and not
self.visited[row][col] and self.maze[row][col] == 1

    def breadth_first_search(self, start_row, start_col, end_row,
end_col):
        queue = deque([(start_row, start_col)])
        self.visited[start_row][start_col] = True

        while queue:
            row, col = queue.popleft()

            if row == end_row and col == end_col:
                self.construct_solution(start_row, start_col, end_row,
end_col)
                return True

            directions = [(-1, 0), (0, -1), (1, 0), (0, 1)]

            for dr, dc in directions:
                new_row, new_col = row + dr, col + dc
                if self.is_valid_move(new_row, new_col):
                    queue.append((new_row, new_col))
                    self.visited[new_row][new_col] = True

        return False

    def construct_solution(self, start_row, start_col, end_row,
end_col):
        row, col = end_row, end_col
        while row != start_row or col != start_col:
            self.solution[row][col] = 1
            for dr, dc in [(-1, 0), (0, -1), (1, 0), (0, 1)]:
                new_row, new_col = row + dr, col + dc
                if 0 <= new_row < self.rows and 0 <= new_col <
self.cols and self.solution[new_row][new_col] == 0:
```

```
                    row, col = new_row, new_col
                    break

    def solve_maze(self, start_row, start_col, end_row, end_col):
        if not self.breadth_first_search(start_row, start_col, end_row,
end_col):
            print("No solution exists.")
        else:
            self.print_solution()

    def print_solution(self):
        for row in self.solution:
            print(" ".join(map(str, row)))


# Example usage:
maze = [
    [1, 0, 1, 1, 1],
    [1, 1, 1, 0, 1],
    [0, 0, 0, 1, 1],
    [1, 1, 1, 1, 1],
    [1, 1, 0, 0, 1]
]

solver_bfs = MazeSolverBFS(maze)
start_row, start_col = 0, 0
end_row, end_col = len(maze) - 1, len(maze[0]) - 1
solver_bfs.solve_maze(start_row, start_col, end_row, end_col)

# Create a figure and axes object.
fig, ax = plt.subplots()

# Plot the maze.
ax.imshow(maze, cmap="Greys")

# Plot the solution.
ax.imshow(solver_bfs.solution, cmap="Greens")

# Show the plot.
plt.show()
```

Explanation:

MazeSolverBFS Class:

Initialization (__init__ method):

The constructor initializes the MazeSolverBFS object with the provided maze.

It sets attributes such as maze, rows, cols, visited (a 2D array to keep track of visited cells), and solution (a 2D array to store the solution path).

is_valid_move method:

Checks if a given cell (specified by its row and column) is a valid move.

A move is valid if the cell is within the boundaries of the maze, has not been visited before, and contains a value of 1 (indicating an open path).

breadth_first_search method:

Implements the Breadth-First Search algorithm to explore the maze.

It starts from the specified start point and explores in all four directions until it reaches the destination.

Uses a deque from the collections module to maintain a queue for BFS.

If the destination is reached, it calls construct_solution to mark the solution path and returns True.

If no solution is found, it returns False.

construct_solution method:

Constructs the solution path by backtracking from the destination to the start point.

solve_maze method:

Calls the breadth_first_search method to attempt to solve the maze.

If no solution is found, it prints "No solution exists." Otherwise, it calls print_solution.

print_solution method:

Prints the solution path.

Example Usage:

A sample maze is defined as a 2D list (maze).

An instance of the MazeSolverBFS class is created with the maze.

The solve_maze method is called with the start and end points.

The solution path is visualized using matplotlib:

A figure and axes object are created.

The maze is plotted in grayscale (cmap="Greys").

The solution path is plotted in green (cmap="Greens").

The plot is displayed.

Visualization:

matplotlib is used to visualize the maze and its solution:

The original maze is shown in grayscale.

The solution path is overlaid in green.

The visualization helps to see the solution path through the maze. The imshow function is used to display the 2D arrays (maze and solver_bfs.solution) as images.