DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
Discover. Learn. Empower.

A PROJECT REPORT

CLI Based Maze Solver in Java using DFS Algorithm

Design and Analysis of Algorithms
23CSH-301

Submitted by

Himanshu Gupta (23BCS10889)

Submitted to

Mr. Mohammad Shaqlain (E17211)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE & ENGINEERING

CHANDIGARH UNIVERSITY
Discover. Learn. Empower.

Chandigarh University

November 2025

BONAFIDE CERTIFICATE

Certified that this project report "**CLI Based Maze Solver in Java using DFS Algorithm"** is the bonafide work of "**Himanshu Gupta"** who carried out the project work under my/our supervision.

Mr. Mohammad Shaqlain (E17211)
SIGNATURE

TABLE OF CONTENTS

Sr. No.	Topic	Title / Description	Page
1.	Aim	To develop and analyze the complexity of a program to solve a maze using the Depth-First Search (DFS) and backtracking approach.	4
2.	Objective	To implement a Maze Solver that finds a valid path from a start point (S) to an end point (E) in a 10×10 maze using recursive DFS and backtracking.	4
3.	Input / Apparatus Used	Programming language (Java), IDE (IntelliJ IDEA), and a 10×10 maze represented as a 2D character array with S, E, ., and	5
4.	Procedure / Algorithm	Step-by-step explanation of the Maze Solver algorithm using recursive DFS and the solveMaze(), isValidMove(), and printMaze() functions.	6
5.	Functions Used	Description of all main functions: solveMaze(), isValidMove(), and printMaze().	6
6.	Flow of Control (Step-by-Step)	Explains the logical flow of program execution from input to output using backtracking.	7
7.	Code	Complete Java implementation of the CLI-based Maze Solver using DFS algorithm.	7
8.	Example Output	Displays sample input and the corresponding solved maze with the valid path marked using '*'.	10
9.	Time and Space Complexity Analysis	Analyzes time complexity $O(N^2)$ and space complexity $O(N^2)$ for the DFS approach.	11
10.	Advantages	Highlights the simplicity, visual clarity, and effectiveness of the recursive approach.	11
11.	Disadvantages	Discusses limitations such as non-optimal pathfinding and high recursion depth for large mazes.	11
12.	Applications	Lists practical uses in robotics, AI, games, and educational problem-solving.	12
13.	Conclusion / Result	Summarizes the success of the CLI-based Maze Solver, emphasizing recursion, backtracking, and algorithmic design principles in Java.	12

1. Aim

To develop and analyse the complexity of a program to solve a maze using the **Depth-First Search (DFS)** and **backtracking** approach.

2. Objective

The objective is to implement a **Maze Solver** that finds a valid path from a **start point (S)** to an **end point (E)** in a 10×10 maze using **recursive DFS**.

The program ensures:

- Only valid cells (. or E) are visited.
- Walls (|) are avoided.
- The solver **backtracks** when it reaches a dead end.
- The path found is marked with *.

3. Input / Apparatus Used

Programming Language: Java

IDE Used: IntelliJ IDEA

Input:

A **10×10 maze** represented as a 2D character array:

- S → Start position
- E → End position
- . → Path (movable cell)
- | → Wall (blocked cell)

```
char[][] maze = {
        {'S', '.', '.', '|', '.', '.', '.', '.', '.', '.'},
        {'|', '|', '.', '|', '.', '|', '|', '|', '|', '.'},
        {'.', '.', '.', '.', '.', '.', '.', '.', '|', '.'},
        {'.', '|', '|', '|', '|', '|', '.', '|', '|', '.'},
        {'.', '|', '.', '.', '.', '|', '.', '.', '.', '.'},
        {'.', '|', '.', '|', '.', '|', '|', '|', '|', '.'},
        {'.', '|', '.', '|', '.', '.', '.', '.', '|', '.'},
        {'.', '|', '.', '|', '|', '|', '|', '.', '|', '.'},
        {'.', '.', '.', '.', '.', '.', '|', '.', '.', 'E'},
        {'|', '|', '|', '|', '|', '|', '|', '|', '|', '|'}
};
```

4. Procedure / Algorithm

Algorithm: Maze Solver using Backtracking

1. **Start**
2. Read or define the 2D maze grid.
3. Locate the **Start ('S')** position in the maze.
4. Call the recursive function solveMaze(maze, startRow, startCol).
5. Inside solveMaze():
    - If the current cell is the end cell (E), return success.
    - If the current cell is a valid move (. or S):
        - Mark it with * (visited path).
        - Recursively explore all four directions using isValidMove():
            - Move **Up:** (row - 1, col)
            - Move **Down:** (row + 1, col)
            - Move **Left:** (row, col - 1)
            - Move **Right:** (row, col + 1)
        - If any recursive call returns true, the path is found.
        - Otherwise, backtrack by restoring the original cell value.
6. Continue until the maze is solved or no valid path exists.
7. Display the maze using printMaze().
8. **Stop**

5. Functions Used

Function Name	Description
solveMaze(char[][] maze, int row, int col)	Recursively explores all possible paths using DFS and marks visited cells.
isValidMove(char[][] maze, int row, int col)	Checks if the next cell is within maze boundaries and not a wall (`
printMaze(char[][] maze)	Prints the maze grid neatly before and after solving.

6. Flow of Control (Step-by-Step)

1. Start program execution.
2. Display the original maze using printMaze().
3. Find the position of S.
4. Call solveMaze(maze, startRow, startCol).
5. In solveMaze():
    - Check all four directions (up, down, left, right).
    - If a valid path is found, mark and continue.
    - If dead end → backtrack.
6. Stop when E is reached or no path exists.
7. Display final maze showing the solved path (*).

7. Code

```
public class MazeSolver {

    static int rows, cols;

    public static void main(String[] args) {
        System.out.println("=== MAZE SOLVER ===");
        System.out.println("S = Start, E = End, | = Wall, . = Path\n");

        // maze
        char[][] maze = {
                {'S', '.', '.', '|', '.', '.', '.', '.', '.', '.'},
                {'|', '|', '.', '|', '.', '|', '|', '|', '|', '.'},
                {'.', '.', '.', '.', '.', '.', '.', '.', '|', '.'},
                {'.', '|', '|', '|', '|', '|', '.', '|', '|', '.'},
                {'.', '|', '.', '.', '.', '|', '.', '.', '.', '.'},
                {'.', '|', '.', '|', '.', '|', '|', '|', '|', '.'},
                {'.', '|', '.', '|', '.', '.', '.', '.', '|', '.'},
                {'.', '|', '.', '|', '|', '|', '|', '.', '|', '.'},
                {'.', '.', '.', '.', '.', '.', '|', '.', '.', 'E'},
                {'|', '|', '|', '|', '|', '|', '|', '|', '|', '|'}
        };

        rows = maze.length;
        cols = maze[0].length;

        System.out.println("--- Original Maze ---");
        printMaze(maze);

        // starting position (S)
        int startRow = -1, startCol = -1;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (maze[i][j] == 'S') {
                    startRow = i;
                    startCol = j;
                    break;
```

```
                    }
                }
            }

            System.out.println("\nSolving maze...");
            boolean pathFound = solveMaze(maze, startRow, startCol);

            if (pathFound)
                System.out.println("\nPath Found!");
            else
                System.out.println("\nNo Path Found!");

            System.out.println("\n--- Final Maze ---");
            printMaze(maze);
        }

        // Recursive DFS Maze Solver

        public static boolean solveMaze(char[][] maze, int row, int col) {
            // base case
            if (maze[row][col] == 'E') return true;


            if (maze[row][col] == '.' || maze[row][col] == 'S') {
                char original = maze[row][col];
                maze[row][col] = '*'; // mark as part of path

                // Move UP
                if (isValidMove(maze, row - 1, col)) {
                    if (solveMaze(maze, row - 1, col)) return true;
                }

                // Move DOWN
                if (isValidMove(maze, row + 1, col)) {
                    if (solveMaze(maze, row + 1, col)) return true;
                }

//              // Move RIGHT
//              if (isValidMove(maze, row, col + 1)) {
//                  if (solveMaze(maze, row, col + 1)) return true;
//              }

                // Mai right phele explore kar raha hu ya baad me isse path ka
phrk pd raha hai since
                // more than one paths are existing

                // Move LEFT
                if (isValidMove(maze, row, col - 1)) {
                    if (solveMaze(maze, row, col - 1)) return true;
                }

                // Move RIGHT
                if (isValidMove(maze, row, col + 1)) {
                    if (solveMaze(maze, row, col + 1)) return true;
                }


                // Backtrack
                maze[row][col] = original;
            }
```

```
        return false;
    }


    public static boolean isValidMove(char[][] maze, int row, int col) {
        return row >= 0 && row < rows &&
                col >= 0 && col < cols &&
                (maze[row][col] == '.' || maze[row][col] == 'E');
    }


    public static void printMaze(char[][] maze) {
        for (char[] line : maze) {
            for (char cell : line) {
                System.out.print(cell + " ");
            }
            System.out.println();}}}
```

8. Example Output

```
"C:\Program Files\Java\jdk-25\bin\java.exe" "-javaagen
=== MAZE SOLVER ===
S = Start, E = End, | = Wall, . = Path

--- Original Maze ---
S . . | . . . . . .
| | . | . | | | | .
. . . . . . . . | .
. | | | | | . | | .
. | . . . | . . . .
. | . | . | | | | .
. | . | . . . . | .
. | . | | | | . | .
. . . . . . | . . E
| | | | | | | | | |
```

```
Path Found!

--- Final Maze ---
* * * | . . . . . .
| | * | . | | | | .
* * * . . . . . | .
* | | | | | . | | .
* | * * * | . . . .
* | * | * | | | | .
* | * | * * * * | .
* | * | | | | * | .
* * * . . . | * * E
| | | | | | | | | |

Process finished with exit code 0
```

9. Time and Space Complexity Analysis

Time Complexity:

Each cell is visited once in the worst case →

$$O(N^2)$$

Space Complexity:

The recursion stack may go up to all accessible cells →

$$O(N^2)$$

where N × N is the maze size.

10. Advantages

- Simple and intuitive recursive approach.
- Easy to visualize and implement in CLI.
- Demonstrates the concept of **backtracking** effectively.

11. Disadvantages

- Not guaranteed to find the *shortest* path.
- High recursion depth for large mazes.
- May explore redundant paths before reaching the end.

12. Applications

- Pathfinding in robotics and games.
- AI-based maze and puzzle solvers.
- Network routing and navigation algorithms.
- Teaching recursion and backtracking concepts.

13. Conclusion / Result

The **CLI-based Maze Solver** successfully identifies a valid path from the start (S) to the end (E) using the **Depth-First Search (DFS)** technique.
The recursive function solveMaze() effectively explores all possible moves, validates each step with isValidMove(), and visualizes the result with printMaze().
This project demonstrates the power of **recursion, backtracking, and structured problem-solving** in Java.