Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 02, May, 2025
Website: usajournals.org

MAZE ALGORITHM: IS IT SOLVABLE?

Zayniyev Sanjar Azmiddin ogli
Computer and Software Engineering Student at Inha University in Tashkent, Software Test Automation Engineer at The Adtech

Abstract:
This article presents a solution to the classic maze traversal problem, where the objective is to determine whether a valid path exists from a designated starting point ('S') to an exit point ('E') in a grid-based maze. The maze is composed of open paths ('.'), walls ('#'), a start, and an exit, with movement restricted to horizontal and vertical directions. The solution implements the Depth-First Search (DFS) algorithm to explore the maze and identify whether a path from the start to the exit exists, while avoiding revisits and obstacles. Key operations include maze input handling, algorithm initialization, execution, and result output. The implementation employs essential data structures such as a 2D array for maze representation, a stack for DFS traversal, and a boolean array to track visited cells. The article discusses the design decisions behind these structures, outlines the algorithm's limitations, and explores potential future enhancements. This work highlights the importance of algorithmic problem-solving and data structure selection in pathfinding applications.

Keywords: Maze Solving, Depth-First Search (DFS), Pathfinding Algorithm, Java Programming, Data Structures, Stack, Grid Traversal, Algorithmic Problem Solving, Search Algorithms, Optimal Pathfinding, 2D Maze Representation.

Introduction

We are given a maze represented as a grid, where each cell can be one of the following:

- ‘S’: Starting point (entrance).
- ‘E’: Exiting point (destination).
- ‘.’: Open path, representing a passable cell.
- ‘#’: Wall or obstacle, representing an impassable cell.

Our task is to determine whether there exists a valid path from the start to the exit within the maze. A valid path must obey the following rules:

1. We can only move horizontally or vertically (no diagonal movements).
2. We cannot pass through walls (‘#’).
3. We must navigate from the start (‘S’) to the exit (‘E’).

The problem is considered solved if a valid path from start to exit exists, otherwise, it is unsolvable.

Example (Consider the following maze)

```
S .  .  # #
. #  .  .  #
. #  #  .  #
. .  .  #  .
## #   .  E
```

In this maze, ‘S’ represents the start, ‘E’ represents the exit, ‘.’ represents an open path, and ‘#’ represents a wall. The question is whether there is a valid path from ‘S’ to ‘E’ in the maze.

Objective

Implement an algorithm, such as Depth-First Search (DFS), to determine if a valid path exists in the given maze from the start to the exit.

Operations included in the application

1. **Input Maze**: Accepting or loading the maze data, which represents the layout of the maze, including the start, exit, open path and walls.

2. **Initialize Algorithm**: Choosing and initializing the maze-solving algorithm, such as Depth-First Search (DFS).

3. **Maze Solving**: Executing the selected algorithm to determine whether a valid path exists from the start to the exit in the maze.

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 02, May, 2025
Website: usajournals.org

4. **Output Result**: Displaying the result of the maze-solving algorithm, indicating whether the maze is solvable or not.

Choice of Data Structures

1. 2D array:

● Purpose: Represents the maze layout.

● Usage: Each cell in the array corresponds to a location in the maze and contains information about whether it is an open path, wall, the start or the exit.

2. Stack (for DFS):

● Purpose: Used in Depth-First Search (DFS) algorithm.

● Usage: Keeps track of cells to be explored, DFS typically uses a recursive approach, but an explicit stack can be used to implement an iterative solution.

3. Hashset or boolean[][] (to track the visited cells)

● Purpose: Used to prevent revisiting the same cell.

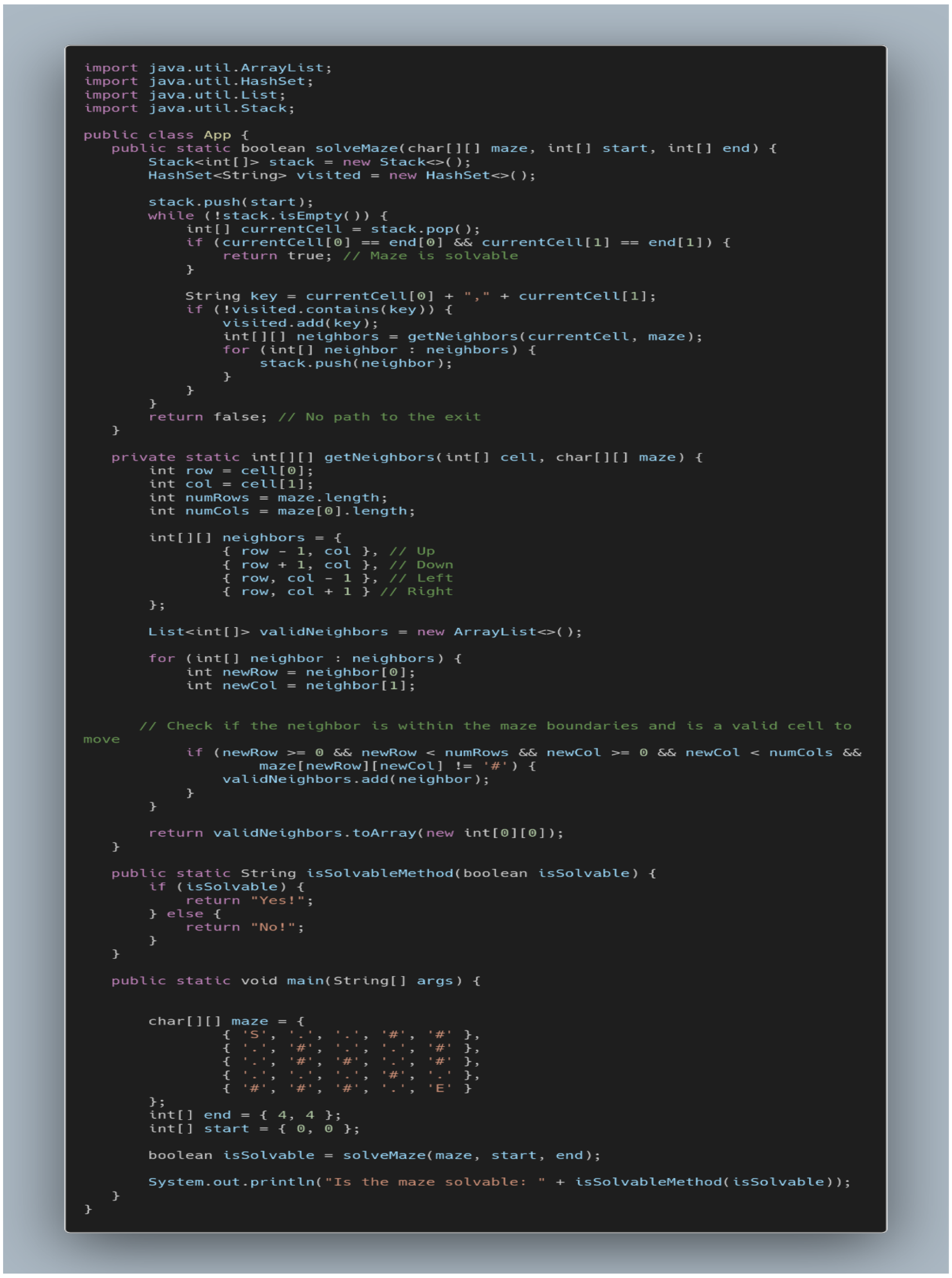
● Usage: Marks cells as visited to avoid infinite loops and unnecessary exploration. A ‘Hashset’ is useful when the maze can have any dimensions, while a ‘boolean[][]’ array is suitable for fixed-size mazes.

4. Array List:

● Purpose: Used to store valid neighbors

● Usage: Gets possible neighbors to visit to know before knowing whether there will continue passing, stores 1D array type of integer (int[]).

Code (Implementation in Java)



```java
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Stack;

public class App {
   public static boolean solveMaze(char[][] maze, int[] start, int[] end) {
       Stack<int[]> stack = new Stack<>();
       HashSet<String> visited = new HashSet<>();

       stack.push(start);
       while (!stack.isEmpty()) {
           int[] currentCell = stack.pop();
           if (currentCell[0] == end[0] && currentCell[1] == end[1]) {
               return true; // Maze is solvable
           }

           String key = currentCell[0] + "," + currentCell[1];
           if (!visited.contains(key)) {
               visited.add(key);
               int[][] neighbors = getNeighbors(currentCell, maze);
               for (int[] neighbor : neighbors) {
                   stack.push(neighbor);
               }
           }
       }
       return false; // No path to the exit
   }

   private static int[][] getNeighbors(int[] cell, char[][] maze) {
       int row = cell[0];
       int col = cell[1];
       int numRows = maze.length;
       int numCols = maze[0].length;

       int[][] neighbors = {
               { row - 1, col }, // Up
               { row + 1, col }, // Down
               { row, col - 1 }, // Left
               { row, col + 1 } // Right
       };

       List<int[]> validNeighbors = new ArrayList<>();

       for (int[] neighbor : neighbors) {
           int newRow = neighbor[0];
           int newCol = neighbor[1];


      // Check if the neighbor is within the maze boundaries and is a valid cell to
move
           if (newRow >= 0 && newRow < numRows && newCol >= 0 && newCol < numCols &&
                   maze[newRow][newCol] != '#') {
               validNeighbors.add(neighbor);
           }
       }

       return validNeighbors.toArray(new int[0][0]);
   }

   public static String isSolvableMethod(boolean isSolvable) {
       if (isSolvable) {
           return "Yes!";
       } else {
           return "No!";
       }
   }

   public static void main(String[] args) {


       char[][] maze = {
               { 'S', '.', '.', '#', '#' },
               { '.', '#', '.', '.', '#' },
               { '.', '#', '#', '.', '#' },
               { '.', '.', '.', '#', '.' },
               { '#', '#', '#', '.', 'E' }
       };
       int[] end = { 4, 4 };
       int[] start = { 0, 0 };

       boolean isSolvable = solveMaze(maze, start, end);

       System.out.println("Is the maze solvable: " + isSolvableMethod(isSolvable));
   }
}
```

I pushed my code to GitHub as well. You can find the link in the references section of this article.

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 02, May, 2025
Website: usajournals.org

Limitations:
1. Completeness: DFS algorithm can fail to find a solution if the maze is infinite or very large. In practical scenarios, limitations on memory and processing power can affect the completeness of the algorithm.
2. Optimality: The implemented algorithm (DFS) focuses on finding any valid path from the start to the exit. They do not necessarily find the shortest path. If finding the optimal path is crucial, more advanced algorithms like Dijkstra's may be considered.
3. Memory Usage: The algorithms use additional memory to store data structures (stack or queue) and to track visited cells. For very large mazes, memory usage can become a limitation.
4. Grid Representations: The implementation assumes a 2D grid representation of the maze. If the maze has a more complex structure or is represented differently, the algorithm may need modifications.

Future Scope:
1. Optimizing for Memory and Performance: Future improvements can focus on optimizing memory usage and performance. This might involve tweaking data structures, implementing pruning strategies or considering parallelization.
2. Shortest Path Algorithms: If finding the shortest path is a requirement, consider implementing algorithms like Djikstra's or A*(A-star). These algorithms take into account the cost of reaching each cell and aim for the most efficient path.
3. Dynamic Mazes: Extend the algorithm to handle dynamic mazes where the maze structure can change over time. This could involve real-time updates or modifications during pathfinding.
4. Visualizations and User Interaction: Enhance the application by adding visualizations to help users understand how the algorithm explores the maze. We will consider adding user interaction features, allowing users to interact with the maze dynamically.
5. Multiple Paths: We need to modify the algorithm to find multiple paths from the start to the exit, if they exit. This could involve extending the solution to consider alternative routes.

6. Handling 3D or Multi-agent Mazes: We can extend the algorithm to handle 3D mazes or mazes with multiple agents, introducing new challenges and opportunities for optimization.

Subject Importance

Pathfinding in Various Fields: Maze-solving algorithms play a crucial role in pathfinding applications across diverse industries. From guiding autonomous robots through physical environments to optimizing routing systems in logistics, these algorithms offer efficient solutions for navigating intricate mazes.

Application in Games: In the gaming industry, maze-solving algorithms serve as the backbone for creating captivating and challenging game scenarios. Game developers leverage these algorithms to design intricate mazes, adding depth and complexity to gameplay and enhancing the overall gaming experience.

Routing and Navigating systems: Maze-solving algorithms are fundamental to the development and optimization of routing and navigation systems. Whether in GPS applications or virtual map services, these algorithms enable efficient and reliable route planning, contributing to seamless navigation experiences for users in both physical and digital spaces.

Educational Significance: Maze-solving algorithms hold educational importance as practical examples for teaching fundamental concepts in computer science and algorithms. Their visibility and applicability make them valuable tools for students and educators alike, fostering a deeper understanding of algorithmic principles.

Real-World Impact: Maze-solving algorithms, once confined to theoretical realms, have now evolved into indispensable tools with tangible real-world impact. Their practical significance extends across diverse domains, revolutioning navigation, automation, and problem-solving methodologies.

Modern American Journal of Engineering, Technology, and Innovation
ISSN(E): 3067-7939
Volume 01, Issue 02, May, 2025
Website: usajournals.org

References

1. Stack overflow discussion on maze path searching using dfs. July, 2017.
2. Beauldung tutorial on maze problem implementation using java. January, 2024.
3. Source code of this algorithm, Github. May, 2025.