**MAZE SOLVER**

**MICRO PROJECT REPORT**

**22IST32 – DATA STRUCTURES**

**Submitted by**

**AKILESH G A - 23ISR002**

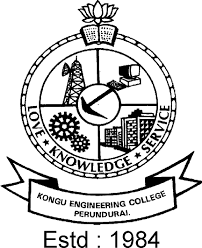
**BHARATH P - 23ISR008**

**SREE SETHU MADHAVAN DS -23ISR054**

**MSc SOFTWARE SYSTEMS**

**THIRD SEMESTER**

**DEPARTMENT OF COMPUTER TECHNOLOGY - PG**



**KONGU ENGINEERING COLLEGE**

**(AUTONOMOUS)**

**PERUNDURAI, ERODE - 638 060**

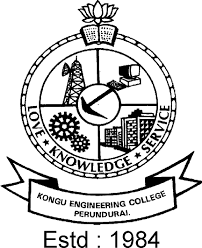
**OCT 2024**

**DEPARTMENT OF COMPUTER TECHNOLOGY - PG**

**KONGU ENGINEERING COLLEGE**

**(AUTONOMOUS)**

**PERUNDURAI, ERODE - 638 060**



**BONAFIDE CERTIFICATE**

This is to certify that the Micro Project Report entitled **MAZE SOLVER** is the bonafide record of **22IST32 - DATA STRUCTURES** done by **AKILESH GA** - 23ISR002, **BHARATH P** - 23ISR008, **SREE SETHU MADHAVAN DS** 23ISR054 of III Semester II and MSc(Software Systems) during the academic year of 2024-2025.

Date:

Faculty In-Charge Head of the Department

**Table of Contents**

| **S.No.** | **Content** | **Page No.** |
| --- | --- | --- |
|  | Abstract | **1** |
|  | Problem Statement | **2** |
|  | Data Structure Concepts Involved in Implementation | **3-5** |
|  | Source Code | **6-8** |
|  | Output Screenshots | **9-10** |
|  | Conclusion | **11** |

**ABSTRACT**

The primary objective is to develop a maze solver using HTML CSS JS and Python. The maze solver is designed to navigate from a source point to a sink point, achieving the goal by crossing the minimum number of paths. The maze is represented as a graph data structure. The maze solver is a puzzle or game where the challenge is to move from a source point to a sink point within a maze. The maze is represented as a graph, consisting of vertices (nodes) connected by edges (paths). The goal is to find the shortest path from the source to the sink.

The approach to solving the maze involves using backtracking. Backtracking is a method used to solve puzzles by exploring all possible paths in the graph. It systematically searches for a solution by trying out different paths and backtracking when a path does not lead to thEsolution. The maze is represented using a graph data structure. A graph consists of a set of vertices connected by edges. To find the shortest and most feasible path to solve the maze, the Breadth-First Search (BFS) traversal algorithm is employed. BFS starts at the source node and explores all its adjacent nodes before moving on to their adjacent nodes.

A website developed to visualize the working of backtracking algorithms using HTML, CSS, and JavaScript. The website clearly explains the mechanism of backtracking and workflow of game.The steps are highlighted to define the path and exploration the backtracking algorithm navigates through the maze. After, Exploring all the paths it will stop.

**PROBLEM STATEMENT**

The maze solver problem involves navigating through a 2D grid, where each cell represents either a walkable path or a blocked wall. The task is to find a path from a given start point to a goal point. The grid is composed of 0s, representing walkable cells, and 1s, representing blocked cells. Movement is allowed in four directions—up, down, left, and right—but not diagonally. The goal is to find the shortest possible path from the start to the goal, avoiding any blocked cells.

If such a path exists, the solution should return the sequence of steps or coordinates that lead to the goal. If no valid path exists, the output should indicate that the destination is unreachable. This problem can be solved using algorithms like BFS (Breadth-First Search) for unweighted mazes.

The grid may also include special elements, such as teleportation portals or traps, which add complexity to the problem. These additions can make the algorithm more challenging and exciting as it needs to handle various elements while finding the optimal path. The inclusion of these features requires the algorithm to adapt to different scenarios dynamically, making the maze-solving task even more intriguing.

**DATA STRUCTURE CONCEPTS INVOLVED IN IMPLEMENTATION**

Data structures are crucial elements in computer science, as they allow for the efficient organization, manipulation, and retrieval of data in various computational problems. In the context of the \*\*Maze Solver Problem, the use of different data structures becomes highly significant for representing the maze as a grid or graph, managing the exploration process, and finding an optimal solution to navigate from the start point to the destination. Each data structure serves a specific purpose: storing the maze’s layout, keeping track of visited locations, and determining the best path to avoid obstacles or dead ends. By implementing the appropriate data structures, the system can systematically explore the maze, optimize the search process, minimize unnecessary steps, and ultimately identify the shortest or most efficient route to the goal. This ensures the problem is solved both effectively and efficiently, even in complex mazes.

### Graph Representation:

In the maze solver problem, the maze is treated as a graph. Each walkable cell in the grid is considered a node, and the valid movements between adjacent cells (up, down, left, right) form the edges of the graph. This graph-based approach allows for a structured and systematic exploration of the maze. By representing the maze as a graph, the system can easily identify possible paths, detect obstacles, and understand the connections between different parts of the grid. This representation is essential for solving the problem using various search algorithms.

### Queue (for BFS):

A queue is an essential data structure used in Breadth-First Search (BFS) to traverse the maze. BFS works by exploring the maze level by level, meaning it visits all of the neighboring cells of a node before moving to the next layer of cells. The queue ensures that exploration happens in a first-in, first-out (FIFO) manner, meaning the first cell to be discovered is the first to be explored. This order of exploration is crucial for finding the shortest path, as BFS naturally prioritizes paths that require fewer steps. By systematically exploring every neighboring cell before moving deeper into the maze, BFS guarantees the shortest path to the goal, provided one exists.

**Backtracking Methodology:**

Backtracking is a powerful algorithmic technique used to solve problems incrementally by trying partial solutions and then abandoning them if they are found to be invalid. This method is particularly effective for solving problems that require a systematic exploration of various configurations, making it a suitable approach for a wide range of computational challenges, including the Maze Solver Problem. In the context of maze solving, backtracking can be employed to explore potential paths through the maze by systematically trying each possible route until a solution is found or all options are exhausted.

**Steps in BFS Implementation for Maze Solving:**

1. Setup and Initialization

The algorithm begins by preparing a queue and marking the starting position (typically the entrance of the maze) as visited. This step establishes the initial conditions for systematically exploring the maze.

2. Adding Cells to the Queue

The initial cell is enqueued into the queue. As the algorithm processes each cell, it identifies any adjacent, unvisited cells and adds them to the queue. This layer-by-layer approach facilitates thorough exploration of the maze.

3. Cell Processing Loop

BFS operates within a continuous loop where cells are dequeued from the front of the queue for processing. For every cell that is dequeued, the algorithm inspects all neighboring cells. If a neighboring cell has not been visited, it is marked as visited and enqueued for future examination.

4. Detecting Loops

During the cell processing phase, if the algorithm encounters a neighboring cell that has already been visited (excluding the immediate predecessor), it indicates a potential cycle within the maze. This detection is crucial to ensure the algorithm does not get trapped in infinite loops while navigating.

5. Finalization and Results

The BFS traversal continues until all reachable cells have been processed. At the conclusion of the traversal, the results reveal whether a path to the exit has been located or if the algorithm has reached a dead end. This outcome is essential for assessing the maze's solvability.

**Visualization of Maze:**

The visualization of the maze-solving project uses Python Turtle Graphics to display a graphical representation of the maze and the solution path. The maze appears as a grid with different colors representing various elements. Black is used for the walls, creating barriers the turtle cannot cross, while gray or white represents open paths where the turtle can move. A key feature is the blue path, which shows the successful route the algorithm has found to solve the maze. The green color likely marks either the start or the end point of the maze.

The turtle dynamically traces the path, visualizing how the maze-solving algorithm works in real time, giving insight into its movement and decision-making process. Once the solution is found, the message "Maze Solved" appears, signaling the algorithm has completed its task. The overall layout is clean and focused, with the maze centrally positioned in the window and minimal additional elements, allowing viewers to focus on the solving process

**SOURCE CODE**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Maze Solver</title>

<script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.4.0/p5.js"></script>

<style>

body {

display: flex;

justify-content: center;

align-items: center;

height: 100vh;

margin: 0;

background: url(v3-img.png) no-repeat center center;

background-size: cover;

flex-direction: column;

}

h2 {

font-family: 'Segoe UI';

text-align: center;

font-size: 35px;

color: white;

margin-top: 0;

margin-bottom: 100px;

}

canvas {

border: 5px solid white;

border-radius: 25px;

}

</style>

</head>

<body>

<h2>MAZE SOLVER<br>BACKTRACKING VISUALIZATION</h2>

<script>

let maze = [

[1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],

[1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1],

[1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1],

[1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1],

[1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1],

[1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1],

[1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1],

[1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1],

[1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1],

[1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1],

[1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1],

[1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1],

[1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 2],

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

];

let palette = ["#FFFFFF", "#000000", "#00ff00", "#0000ff", "#AAAAAA", "#0000ff"];

let boxSize = 25;

let delay = 100;

function setup() {

createCanvas(maze[0].length \* boxSize, maze.length \* boxSize);

noLoop();

drawMaze();

exploreMaze(0, 1);

}

function drawMaze() {

background(255);

for (let i = 0; i < maze.length; i++) {

for (let j = 0; j < maze[i].length; j++) {

fill(palette[maze[i][j]]);

rect(j \* boxSize, i \* boxSize, boxSize, boxSize);

}

}

}

function exploreMaze(row, col) {

setTimeout(() => {

if (maze[row][col] == 2) {

fill(0);

alert("Maze Solved", 10, 230);

} else if (maze[row][col] == 0) {

maze[row][col] = 5;

drawMaze();

redraw();

if (row < maze.length - 1 && exploreMaze(row + 1, col)) {

return true;

}

if (row > 0 && exploreMaze(row - 1, col)) {

return true;

}

if (col < maze[row].length - 1 && exploreMaze(row, col + 1)) {

return true;

}

if (col > 0 && exploreMaze(row, col - 1)) {

return true;

}

maze[row][col] = 4;

drawMaze();

redraw();

}

}, delay);

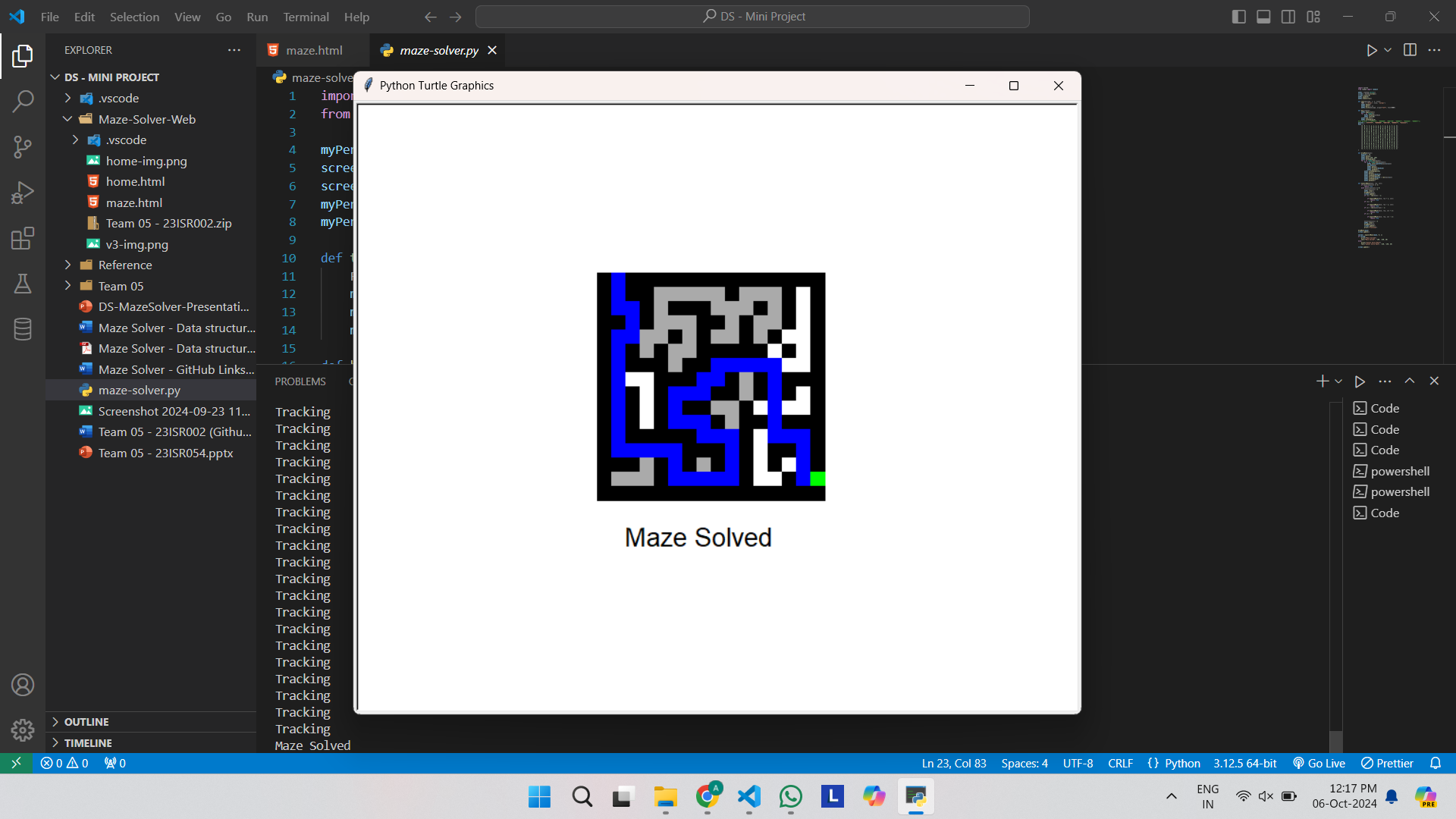
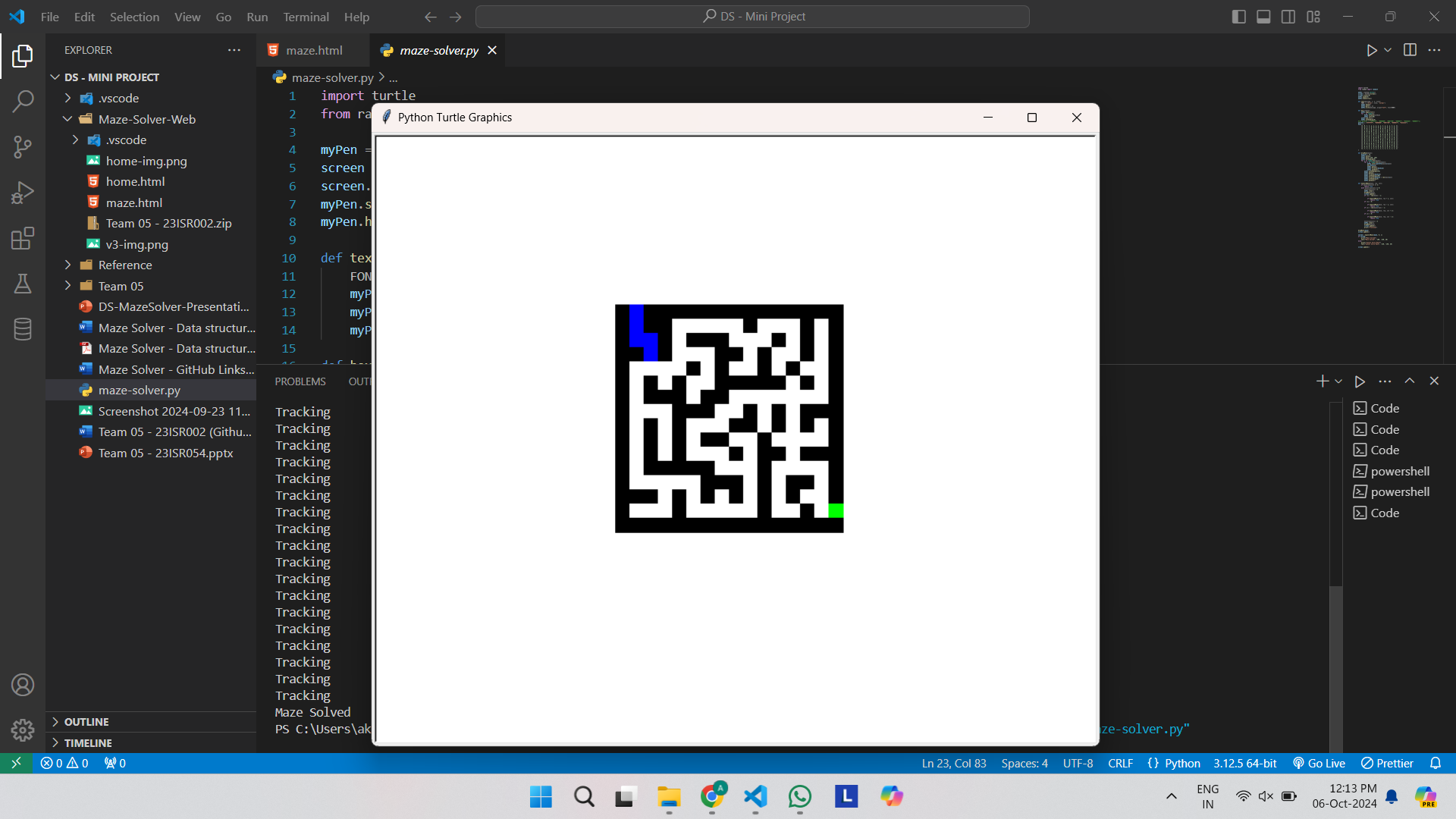
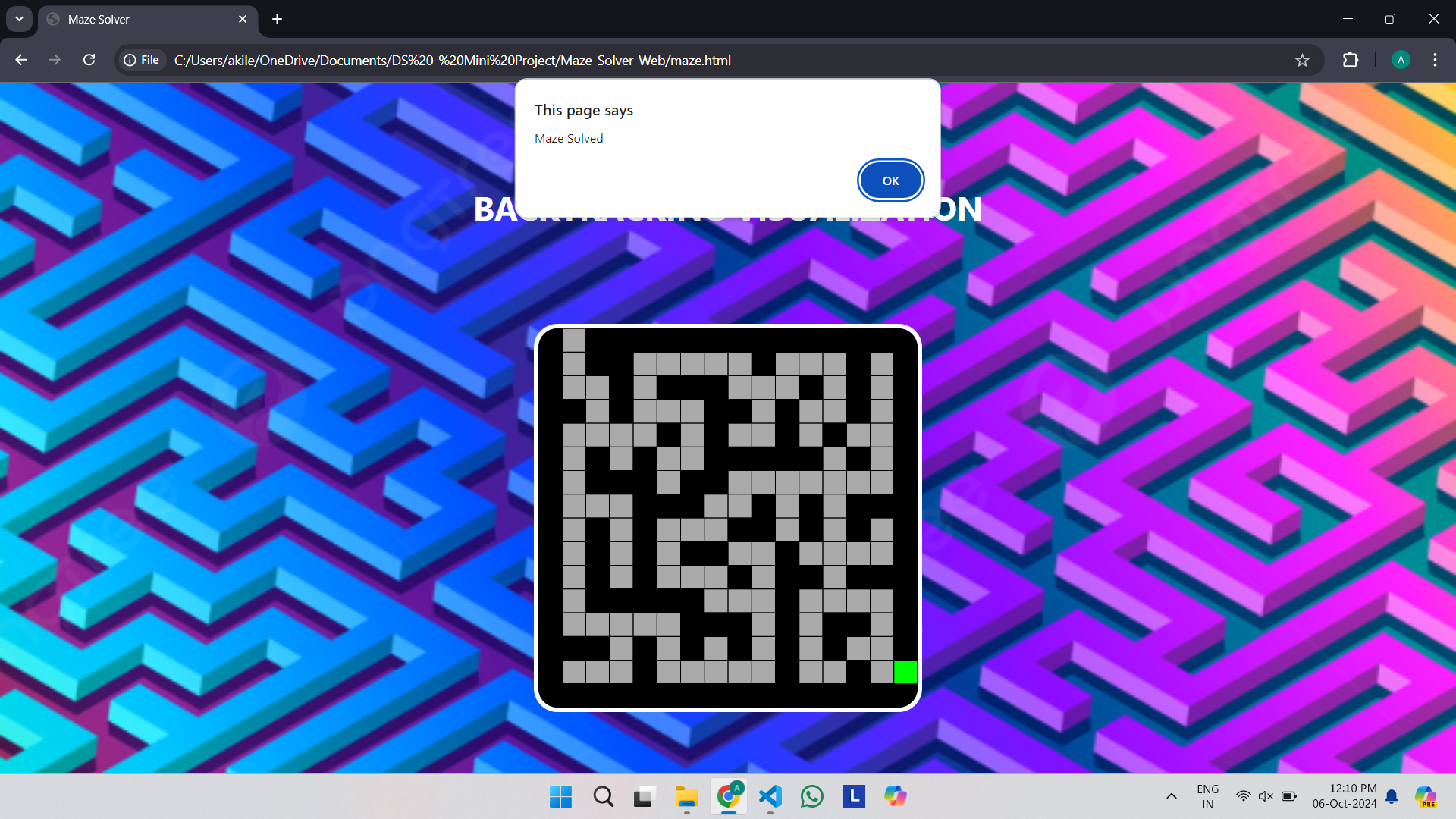
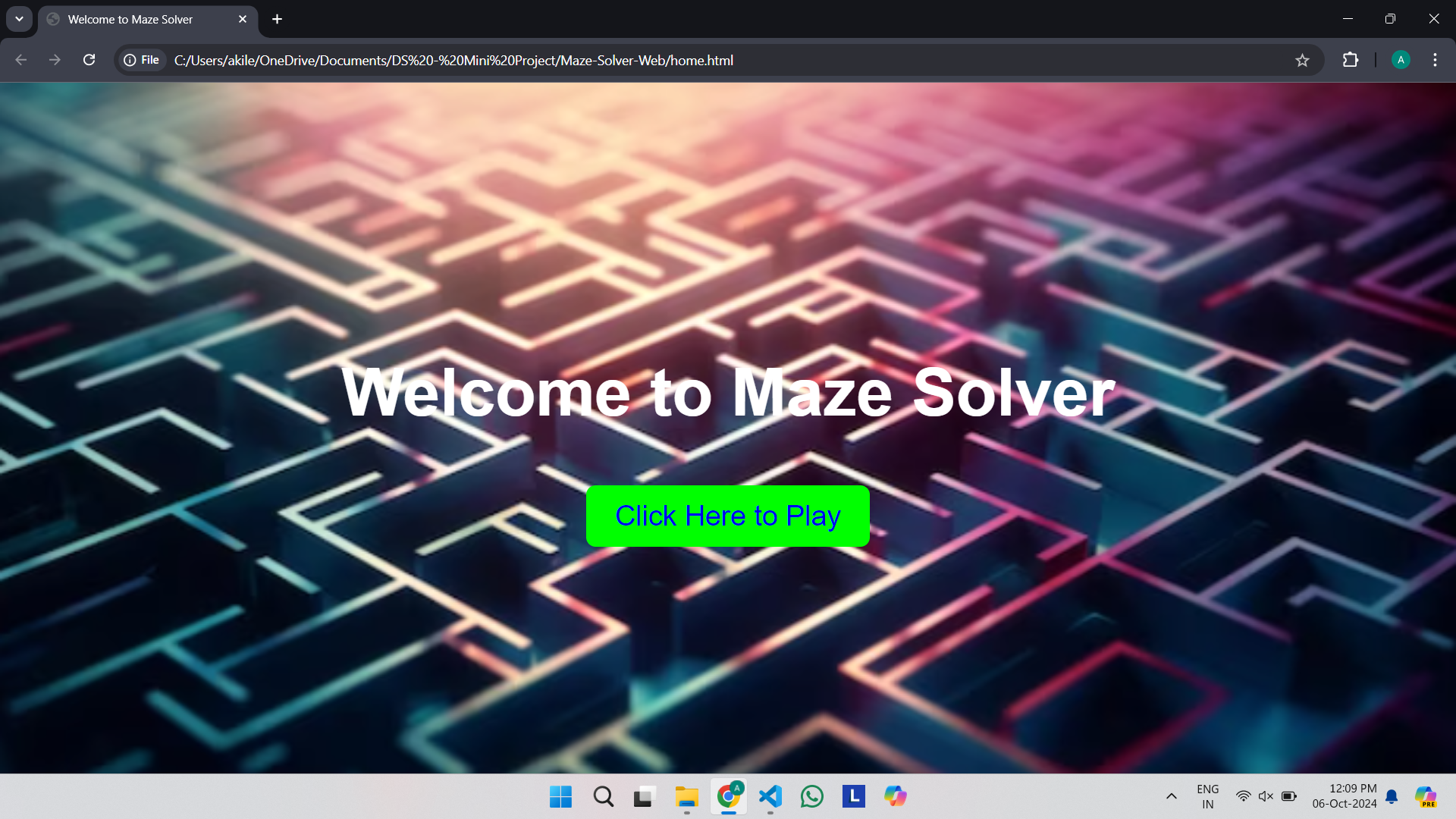
}

</script>

</body>

</html>

**OUTPUT**

****

**CONCLUSION**

The “**Maze Solver Problem**” is a fundamental example of algorithmic problem-solving in computer science, showcasing the intricacies of navigating complex structures. Techniques such as Backtracking and Breadth-First Search (BFS) demonstrate the diversity of approaches available for tackling maze-related challenges. Backtracking systematically explores every possible path through the maze by incrementally building potential solutions and discarding those that lead to dead ends. This exhaustive search is particularly useful for intricate mazes where multiple solutions may exist. Conversely, BFS excels in finding the shortest path in unweighted mazes by employing a level-order traversal mechanism, ensuring that all routes are explored layer by layer. This guarantees that the first time the exit is reached, it is through the shortest route.

Beyond theoretical exploration, the Maze Solver Problem has practical implications in diverse fields, including robotics, computer games, and network routing. Algorithms like BFS and backtracking can enhance robot navigation, allowing machines to traverse unknown environments while optimizing their routes. In video games, efficient pathfinding algorithms improve player experience by creating dynamic, engaging gameplay. Moreover, the principles behind maze-solving techniques inform optimizations in data packet routing in computer networks. Ultimately, the study of these algorithms fosters critical thinking and problem-solving skills, equipping individuals with valuable tools for addressing real-world challenges and contributing to innovations in algorithm design and artificial intelligence.