Maze Solver Using Backtracking

This presentation explains a Java-based maze solver implemented with backtracking. The program uses a graphical interface to visualize the maze and the solution path. It demonstrates how recursive backtracking can find a path from the start to the end of a maze.

MEET OUR TEAM

LEADER: SHADAAN ALI

ROLL NO: 23CS2021068

MEMBERS AND THEIR ROLL NOS:

- RAZA KHAN : 23CS2021051
- RAHUL KUMAR THAKUR : 23CS2021048
- SAKIL AHAMED : 23CS2021057
- SAMIR KUMAR : 23CS2021059

Maze Representation and Setup

Maze Grid

The maze is a 10x10 grid represented by a 2D array. Cells with 0 are paths, and 1 are walls. The start is at the top-left corner, and the goal is at the bottom-right corner.

GUI Components

The interface uses Java Swing with a panel to draw the maze and a button to trigger the solver. The window size is based on cell size and grid dimensions.

Backtracking Algorithm Overview

1 | Step 1: Check Boundaries

Ensure the current cell is within maze limits and not a wall or already visited.

2 | Step 2: Mark Cell

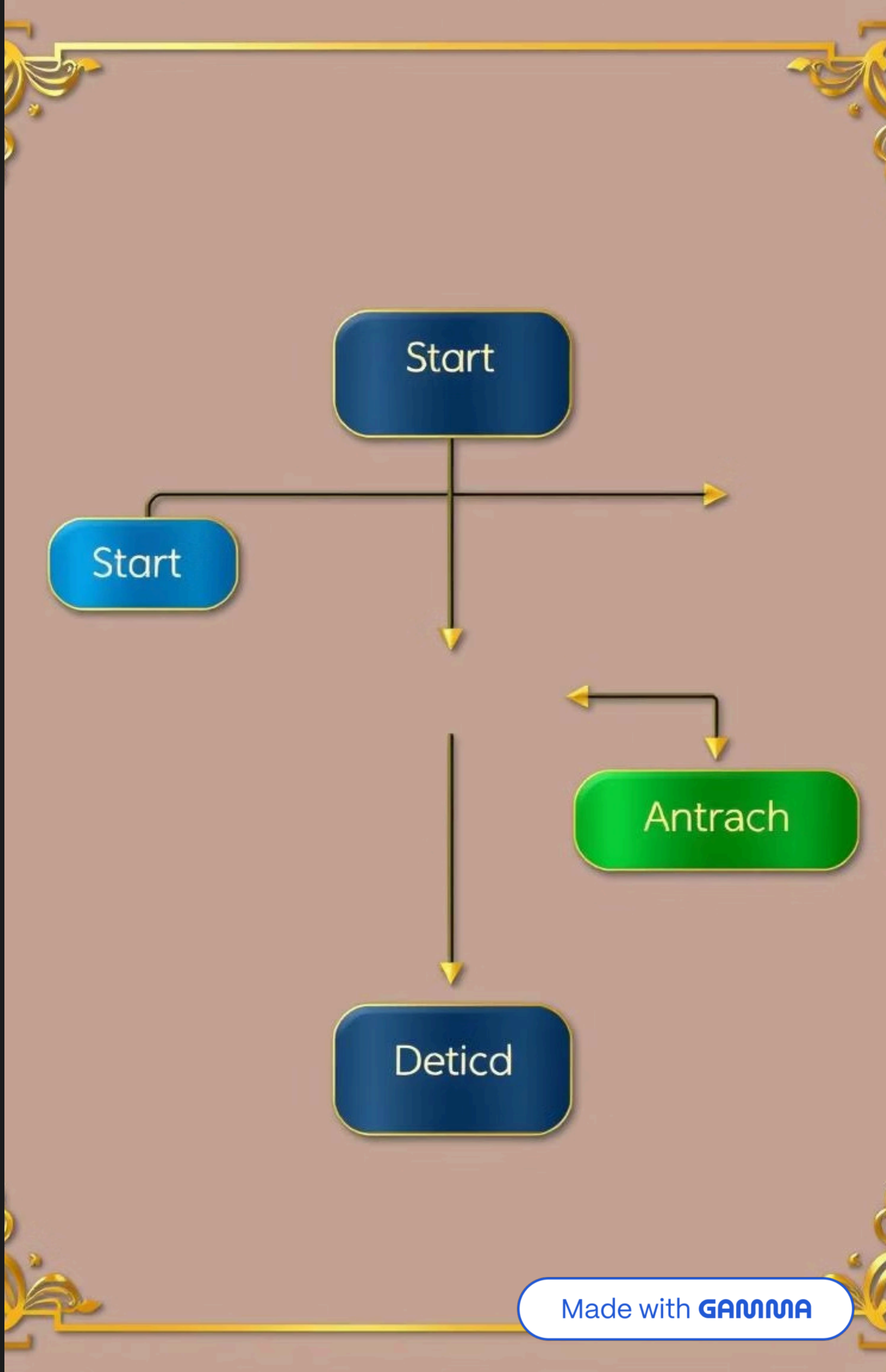
Mark the current cell as part of the solution path.

3 | Step 3: Explore Neighbors

Recursively try moving right, down, left, and up to find the path to the goal.

4 | Step 4: Backtrack

If no path is found, unmark the cell and backtrack to try other routes.



Recursive Maze Solving Method

Function Signature

The method takes the current row and column as parameters and returns true if a path to the goal is found.

Base Cases

Checks for out-of-bounds, walls, or visited cells. Returns true when the goal cell is reached.

Graphical Visualization of Maze

Walls

Drawn as black squares representing obstacles.

Paths

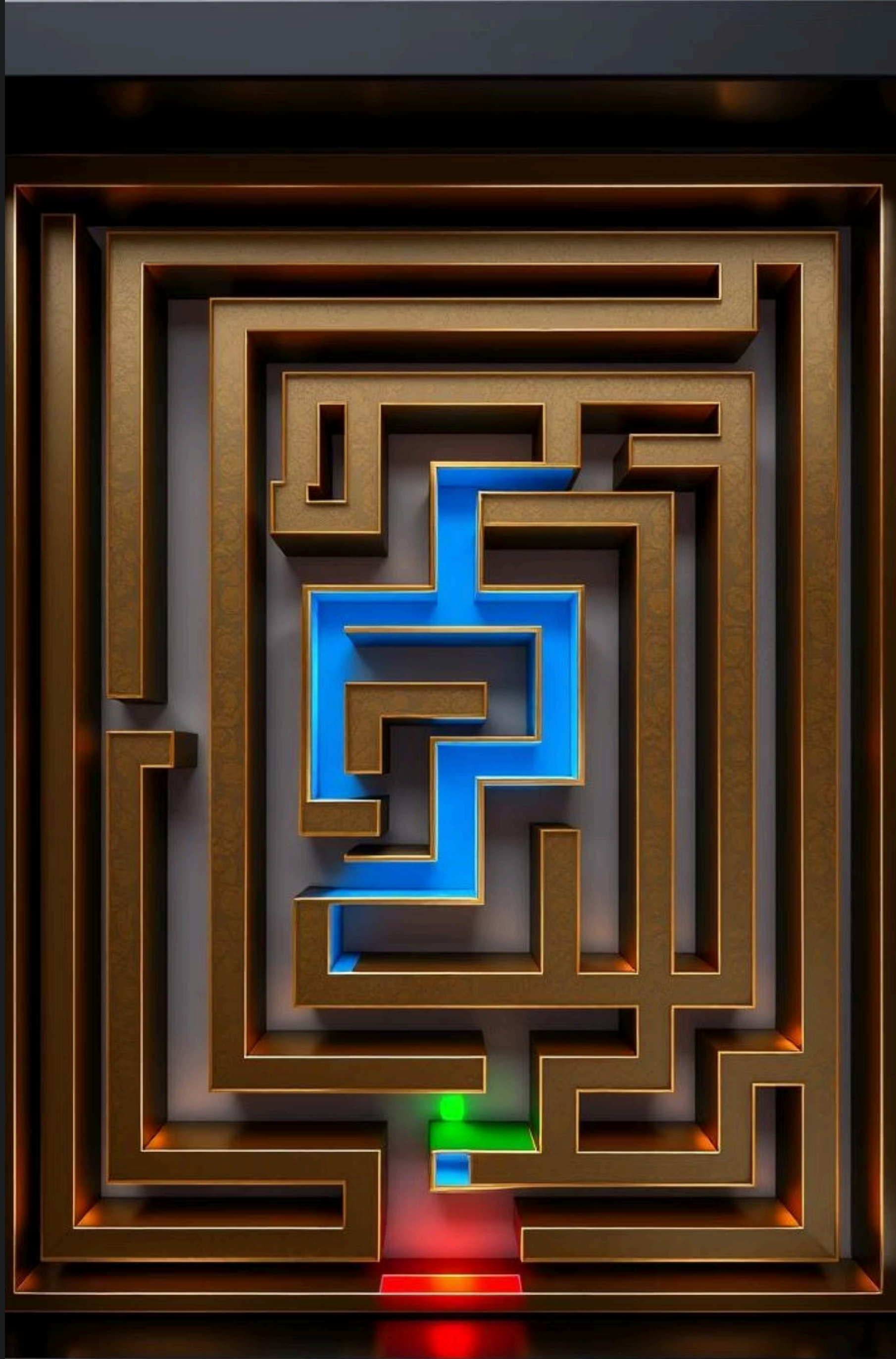
White squares indicate open paths where movement is possible.

Solution Path

Cyan highlights the cells forming the successful route from start to finish.

Start and End

Green marks the start cell, and red marks the goal cell.



User Interaction and Controls

Solve Button

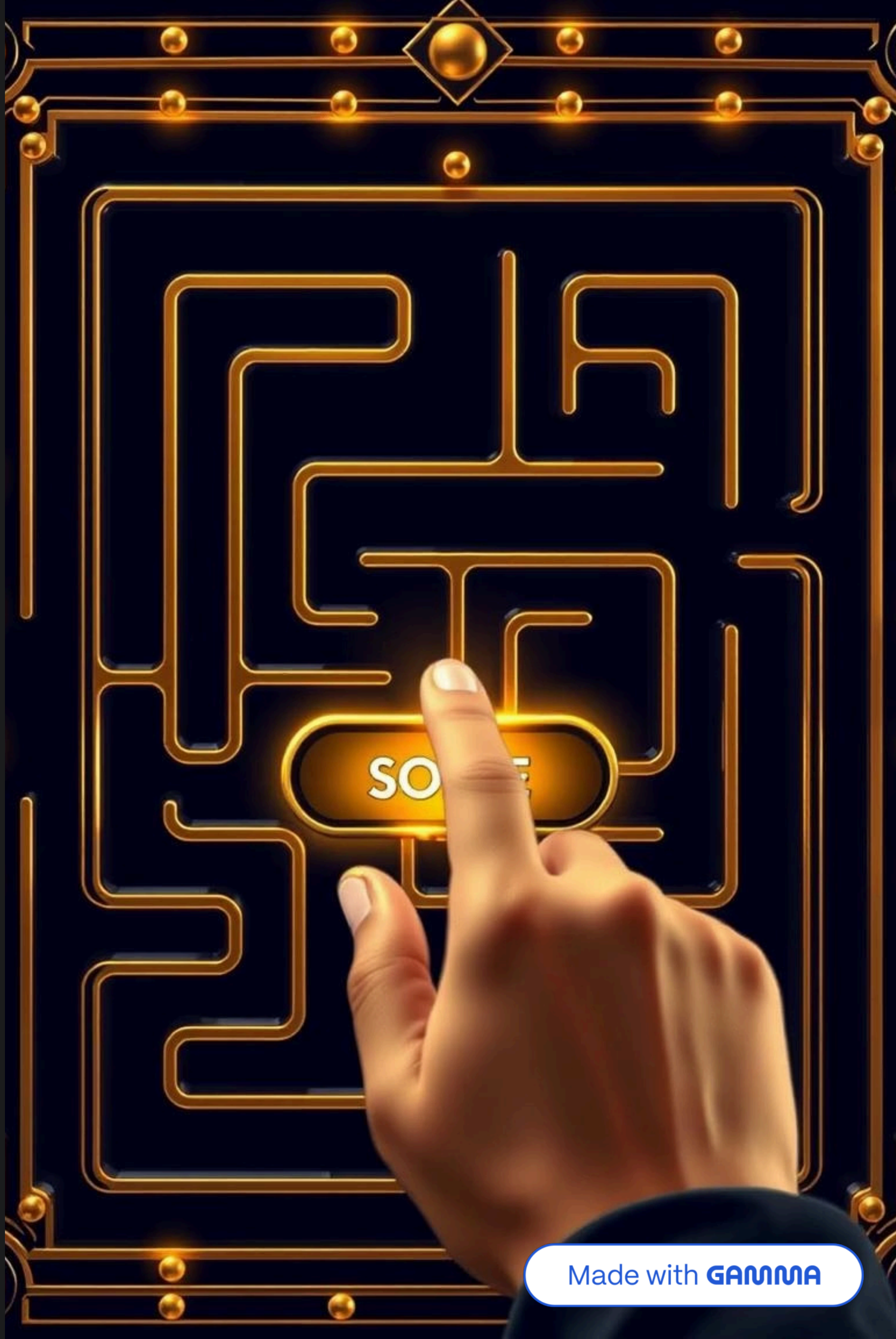
When clicked, it resets the solution and starts the backtracking algorithm.

Feedback

Displays a message dialog indicating whether the maze was solved or no solution was found.

Real-time Update

The maze panel repaints to show the solution path immediately after solving.



Handling No Solution Cases

If the algorithm cannot find a path, it backtracks completely and returns false. The user is notified with a dialog stating "No solution found!" ensuring clear communication of failure cases.

This prevents the program from hanging and allows users to try different mazes or configurations.

Code Structure and Modularity

Main Class

Handles GUI setup, event handling, and maze initialization.

MazePanel Class

Responsible for drawing the maze, walls, paths, and solution on the screen.

Recursive Solver

Encapsulated in a private method that performs the backtracking search.

SOURCE CODE

import javax.swing.;

import java.awt.;

import java.awt.event.*;

public class MazeSolverGUI extends JFrame {

private static final int ROWS = 10;

private static final int COLS = 10;

private static final int CELL_SIZE = 40;

```
// Maze representation: 0 = path, 1 = wall
private int[][] maze = {
    {0, 1, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 1, 0, 1, 0, 1, 0, 1, 1, 0},
    {0, 0, 0, 1, 0, 0, 0, 1, 0, 0},
    {1, 1, 0, 1, 1, 1, 0, 1, 0, 1},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {0, 1, 1, 1, 1, 1, 1, 1, 0, 1},
    {0, 1, 0, 0, 0, 0, 0, 1, 0, 0},
    {0, 1, 0, 1, 1, 1, 0, 1, 1, 0},
    {0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
    {1, 1, 0, 1, 0, 1, 1, 1, 1, 0}
};

private boolean[][] solution = new boolean[ROWS][COLS];
private MazePanel mazePanel;
private JButton solveButton;

public MazeSolverGUI() {
    setTitle("Maze Solver with Backtracking");
    setSize(COLS * CELL_SIZE + 20, ROWS * CELL_SIZE + 100);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLocationRelativeTo(null);

    mazePanel = new MazePanel();
    solveButton = new JButton("Solve Maze");

    solveButton.addActionListener(e -> {
        for (int i = 0; i < ROWS; i++)
            for (int j = 0; j < COLS; j++)
                solution[i][j] = false; // reset solution

        if (solveMaze(0, 0)) {
            JOptionPane.showMessageDialog(this, "Maze solved!");
        } else {
            JOptionPane.showMessageDialog(this, "No solution found!");
        }
        mazePanel.repaint();
    });

    add(mazePanel, BorderLayout.CENTER);
    add(solveButton, BorderLayout.SOUTH);
}

private boolean solveMaze(int row, int col) {
    // Check bounds
    if (row < 0 || col < 0 || row >= ROWS || col >= COLS) return false;
    // Check if it's a wall or already part of solution
    if (maze[row][col] == 1 || solution[row][col]) return false;
    // Mark this cell as part of solution
    solution[row][col] = true;
    // Check if destination reached (bottom right cell)
    if (row == ROWS - 1 && col == COLS - 1) return true;

    // Move Right
    if (solveMaze(row, col + 1)) return true;
    // Move Down
    if (solveMaze(row + 1, col)) return true;
    // Move Left
    if (solveMaze(row, col - 1)) return true;
    // Move Up
    if (solveMaze(row - 1, col)) return true;

    // Backtrack - unmark this cell
    solution[row][col] = false;
    return false;
}

private class MazePanel extends JPanel {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        for (int row = 0; row < ROWS; row++) {
            for (int col = 0; col < COLS; col++) {
                int x = col * CELL_SIZE;
                int y = row * CELL_SIZE;

                if (maze[row][col] == 1) {
                    g.setColor(Color.BLACK); // wall
                    g.fillRect(x, y, CELL_SIZE, CELL_SIZE);
                } else {
                    g.setColor(Color.WHITE); // path
                    g.fillRect(x, y, CELL_SIZE, CELL_SIZE);
                }

                if (solution[row][col]) {
                    g.setColor(Color.CYAN); // solution path
                    g.fillRect(x, y, CELL_SIZE, CELL_SIZE);
                }

                // Cell border
                g.setColor(Color.GRAY);
                g.drawRect(x, y, CELL_SIZE, CELL_SIZE);
            }
        }

        // Start and End
        g.setColor(Color.GREEN);
        g.fillRect(0, 0, CELL_SIZE, CELL_SIZE); // start
        g.setColor(Color.RED);
        g.fillRect((COLS - 1) * CELL_SIZE, (ROWS - 1) * CELL_SIZE, CELL_SIZE, CELL_SIZE); // end
    }
}

public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        new MazeSolverGUI().setVisible(true);
    });
}
```

}

Key Takeaways and Next Steps

Backtracking Works

It is an effective method for solving maze problems by exploring all possible paths.

Visual Feedback

Graphical representation helps users understand the solution process clearly.

Extend and Improve

Future work could include dynamic maze generation, multiple algorithms, or animation of the solving process.

Thank You

We appreciate your time today. We hope this presentation illuminated the power of recursive backtracking for efficient maze solving.

Questions?

We welcome your inquiries and feedback on the maze solver.

Contact Us

Reach out for further discussions or collaboration opportunities.

Explore the Code

Find the full project code and additional details on GitHub.