**A Design and Analysis of Algorithms Lab Project**

**Report on**

**MAZE SOLVING ALGORITHM**

Submitted to

The Department of Computer Science and Engineering

Bachelor of Technology

In

Computer Science and Engineering

(2022 – 2026)

Kota Kameshwari Sripriya 22R11A0523

Under the Guidance of

Dr. A. Sree Lakshmi & G. Krishna Lava Kumar

****

Department of Computer Science and Engineering

**Geethanjali College of Engineering and Technology (UGC Autonomous)**

(Affiliated to J.N.T.U.H, Approved by AICTE, New Delhi, Accredited by NBA)

Cheeryal (V), Keesara (M), Medchal.Dist.-501 301.

**June-2024**

I

**CONTENTS**

**S.No. Particulars Page No**

1. Abstract 1

2. Introduction 2

2.1. Example with Explanation 3

3. System Configuration (H/W and S/W) 4

4. Code/Implementation 5

5. Result/Output Screens 11

6. Conclusion 14

Bibliography 15

II

**ABSTRACT**

Dynamic Maze Solver is an interactive Java application developed with the help of the Swing framework to visualize and solve mazes. The interface is educative where one can dynamically set their start and end points by clicking on the grid; the grid resizes when the window size is adjusted for an interactive and adaptive interface. It is a 2D array representation where different integer values have meaning as walls, paths, start and endpoint, and the shortest path.

User has a window to set start and end point, and on pressing the Enter key, the Breadth-First Search (BFS) algorithm is executed. Since the algorithm is finding its way to the shortest path, it will then highlight the path on the grid. Mouse listeners have been used effectively to make it user interactive, where the left-clicked mouse is going to be the start point, and right-clicked mouse is the end point. The process was made lively and engaging with the real-time path finding by the BFS algorithm.

This project has been developed in such a way that it would work on the majority of the screen sizes and window resolutions as the cell size gets adjusted with the changing widow size; the maze looks even more beautiful with the color-coded things. Understanding graph traversal algorithms and exploration and path finding is a piece of cake with the Dynamic Maze Solver.

**Keywords**  Java Swing, Maze Solving, BFS Algorithm, Interactive GUI, Path finding

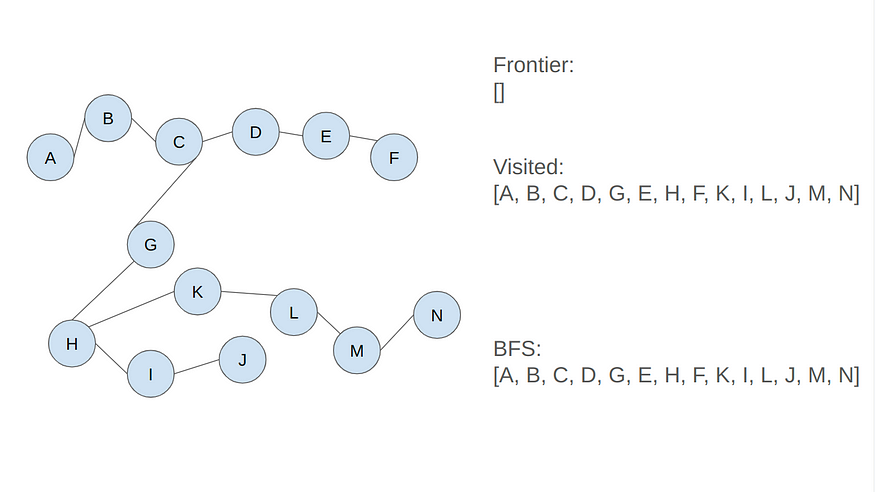
**INTRODUCTION**

The Dynamic Maze Solver is an interactive Java application that offers a graphical representation of maze-solving algorithms, particularly Breadth-First Search (BFS), using the Swing framework of Java. Users can set a start and end point in a grid by simply clicking. The coloring of the walls, path, and shortest path found helps in understanding the concept effectively. Users allow BFS to explore and highlight the shortest path in the initiated solving process by pressing Enter. The size of the grid cells is dynamically reset according to the scaling of the window, making this application responsive and user-friendly. More importantly, the application serves as an educational tool that helps in both graph traversal and path finding algorithms by directly solving relevant problems on the user end.

**EXAMPLE WITH EXPLANATION**

In maze solving with Breadth-First Search (BFS), the method starts with the definition of the maze graph, start and goal nodes, and auxiliary data structures: a frontier queue, a visited list, and a parent map. The start is added to the frontier and visited list. Exploration begins by dequeuing nodes from the frontier. If the current node is the goal, the path from the start to the goal is reconstructed using the parent map and returned as the solution. Otherwise, the node is marked to be part of the solution path and its neighbors are considered part of the search. If it has a neighbor that hasn't been visited yet, then this node is added to the frontier and visited list, and its parent is reflected within the parent map. The process repeats until either the frontier is empty or the goal has been found. When the goal is not found and the frontier is empty, then there is no solution available.

We took an example Medium article. In the provided Medium article, the author delves into solving maze problems using the Breadth-First Search (BFS) algorithm. They provide a full explanation of BFS, emphasizing how it explores nodes level by level, guaranteeing it locates the shortest path. The article describes the initialization process, exploring nodes, and conditions of termination of BFS, making parallel steps of any algorithm. The author provides clear examples and visualizations of how BFS traverses a maze in a pretty effective way to find the optimal path. Overall, the article presents a great source for perceiving and applying BFS in maze-solving problems.

**Source**

https://medium.com/@luthfisauqi17\_68455/artificial-intelligence-search-problem-solve-maze-using-breadth-first-search-bfs-algorithm-255139c6e1a3#:~:text=According%20to%20geeksforgeeks.org%3A,at%20the%20next%20depth%20level.

**SYSTEM CONFIGURATION**

**HARDWARE CONFIGURATION**

* **Processor**- A modern CPU capable of running Java applications efficiently.
* **Memory**- Adequate RAM to support the execution of the Java program without performance issues.
* **Storage**- Enough storage space to store the codebase and any necessary libraries or resources.
* **Graphics Card-** A standard graphics card, as the program does not have intensive graphical requirements.

**SOFTWARE CONFIGURATION**

* **Operating System**- Any modern operating system compatible with Java development.
* **Programming Languages**- Java for coding the application.
* **Development Environment-** A Java IDE or text editor for writing and debugging code.
* **Additional Software-** Java Development Kit (JDK) installed for compiling and running Java programs.

**CODE AND IMPLEMENTATION**

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;

import java.awt.event.KeyAdapter;

import java.awt.event.KeyEvent;

import java.util.\*;

public class DynamicMazeSolver extends JFrame {

private static final Color WALL\_COLOR = Color.BLACK;

private static final Color PATH\_COLOR = Color.GREEN;

private static final Color EMPTY\_COLOR = Color.WHITE;

private static final Color VISITED\_COLOR = Color.CYAN;

private static final Color START\_COLOR = Color.ORANGE;

private static final Color END\_COLOR = Color.BLUE;

private static final Color SHORTEST\_PATH\_COLOR = Color.PINK;

private int[][] maze;

private int rows;

private int cols;

private int startX = -1;

private int startY = -1;

private int endX = -1;

private int endY = -1;

public DynamicMazeSolver(int rows, int cols) {

this.rows = rows;

this.cols = cols;

this.maze = new int[rows][cols];

setWalls(); // Set static walls

setTitle("Dynamic Maze Solver");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setResizable(true);

setLayout(new BorderLayout());

MazePanel mazePanel = new MazePanel();

mazePanel.addMouseListener(new MazeMouseListener());

add(mazePanel, BorderLayout.CENTER);

addKeyListener(new KeyAdapter() {

@Override

public void keyPressed(KeyEvent e) {

if (e.getKeyCode() == KeyEvent.VK\_ENTER) {

solveMaze();

}

}

}

);

setExtendedState(JFrame.MAXIMIZED\_BOTH);

setVisible(true);

}

private void setWalls() {

for (int i = 0; i < rows; i++) {

if (i % 2 == 0) {

maze[i][3] = 1; // vertical wall

maze[i][7] = 1; // vertical wall

}

}

for (int j = 0; j < cols; j++) {

if (j % 2 == 0) {

maze[3][j] = 1; // horizontal wall

maze[7][j] = 1; // horizontal wall

}

}

}

class MazePanel extends JPanel {

@Override

protected void paintComponent(Graphics g) {

super.paintComponent(g);

int width = getWidth();

int height = getHeight();

int cellSize = Math.min(width / cols, height / rows);

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

Color color = getColorForCell(i, j);

g.setColor(color);

g.fillRect(j \* cellSize, i \* cellSize, cellSize, cellSize);

g.setColor(Color.BLACK);

g.drawRect(j \* cellSize, i \* cellSize, cellSize, cellSize);

}

}

}

}

class MazeMouseListener extends MouseAdapter {

@Override

public void mousePressed(MouseEvent e) {

int width = getWidth();

int height = getHeight();

int cellSize = Math.min(width / cols, height / rows);

int y = e.getX() / cellSize;

int x = e.getY() / cellSize;

if (SwingUtilities.isLeftMouseButton(e)) {

if (isValid(x, y) && maze[x][y] == 0 && startX == -1 && startY == -1) {

startX = x;

startY = y;

}

} else if (SwingUtilities.isRightMouseButton(e)) {

if (isValid(x, y) && maze[x][y] == 0 && endX == -1 && endY == -1) {

endX = x;

endY = y;

}

}

repaint();

}

}

private void solveMaze() {

if (startX == -1 || startY == -1 || endX == -1 || endY == -1) {

JOptionPane.showMessageDialog(this, "Please set start and end points.", "Error", JOptionPane.ERROR\_MESSAGE);

return;

}

java.util.List<Point> shortestPath = findShortestPath();

if (shortestPath != null) {

markShortestPath(shortestPath);

repaint();

} else {

JOptionPane.showMessageDialog(this, "No path found.", "Error", JOptionPane.ERROR\_MESSAGE);

}

}

private java.util.List<Point> findShortestPath() {

Queue<Point> queue = new LinkedList<>();

boolean[][] visited = new boolean[rows][cols];

Map<Point, Point> parentMap = new HashMap<>();

queue.offer(new Point(startX, startY));

visited[startX][startY] = true;

while (!queue.isEmpty()) {

Point current = queue.poll();

int x = current.x;

int y = current.y;

if (x == endX && y == endY) {

java.util.List<Point> shortestPath = new ArrayList<>();

Point node = new Point(endX, endY);

while (!node.equals(new Point(startX, startY))) {

shortestPath.add(node);

node = parentMap.get(node);

}

shortestPath.add(new Point(startX, startY));

Collections.reverse(shortestPath);

return shortestPath;

}

for (int i = 0; i < 4; i++) {

int newX = x + dx[i];

int newY = y + dy[i];

if (isValid(newX, newY) && maze[newX][newY] == 0 && !visited[newX][newY]) {

queue.offer(new Point(newX, newY));

visited[newX][newY] = true;

parentMap.put(new Point(newX, newY), new Point(x, y));

}

}

}

return null; // No path found

}

private void markShortestPath(java.util.List<Point> shortestPath) {

for (Point p : shortestPath) {

maze[p.x][p.y] = 4; // Mark the shortest path

}

}

private boolean isValid(int x, int y) {

return x >= 0 && x < rows && y >= 0 && y < cols;

}

private Color getColorForCell(int x, int y) {

if (maze[x][y] == 1) {

return WALL\_COLOR;

} else if (maze[x][y] == 2) {

return PATH\_COLOR;

} else if (maze[x][y] == 3) {

return VISITED\_COLOR;

} else if (x == startX && y == startY) {

return START\_COLOR;

} else if (x == endX && y == endY) {

return END\_COLOR;

} else if (maze[x][y] == 4) {

return SHORTEST\_PATH\_COLOR;

} else {

return EMPTY\_COLOR;

}

}

public static void main(String [] args) {

int rows = 20; // Increase number of rows

int cols = 20; // Increase number of columns

SwingUtilities.invokeLater(() -> new DynamicMazeSolver(rows, cols));

}

private static final int[] dx = {0, 1, 0, -1};

private static final int[] dy = {1, 0, -1, 0};

}

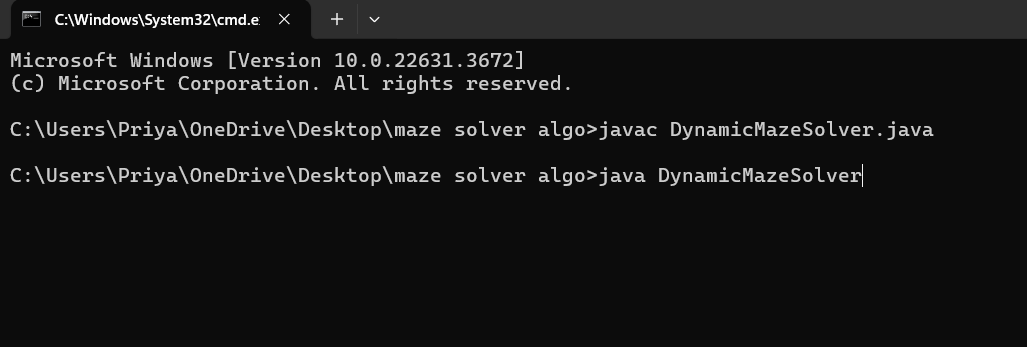
**RESULT/OUTPUT SCREENS**

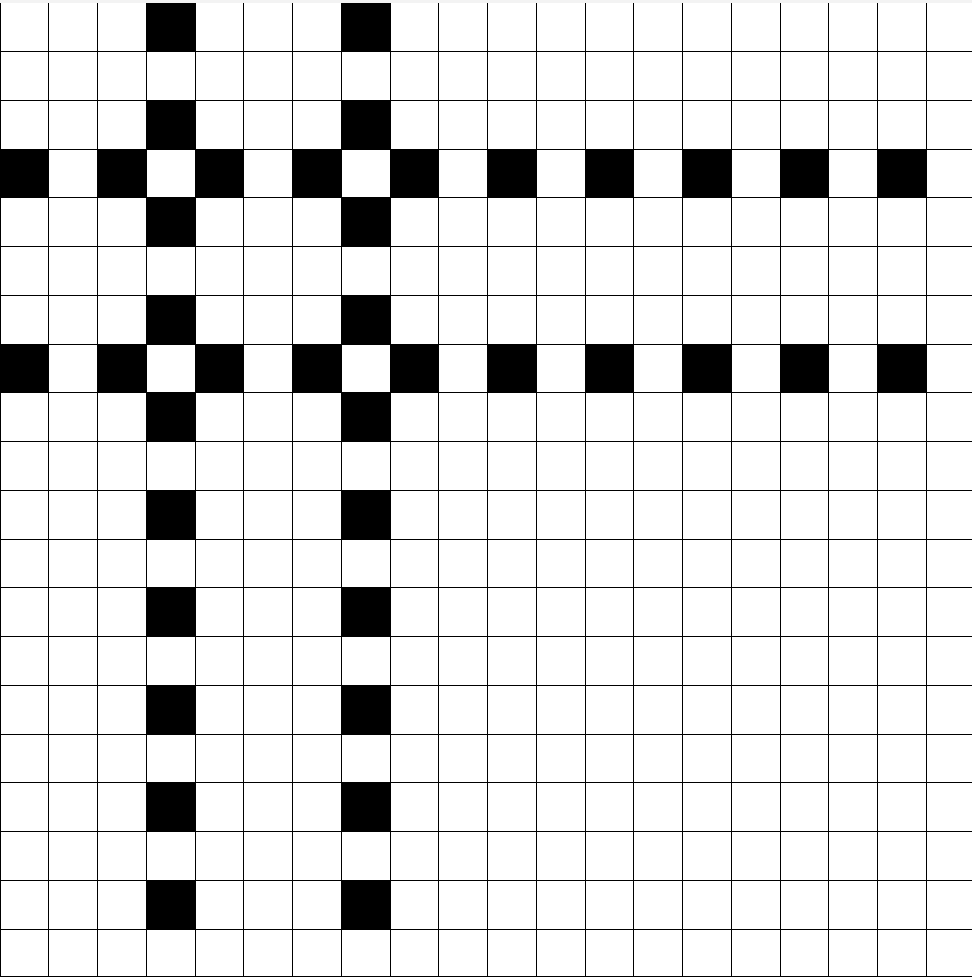
**F**

Fig.1 – Command Prompt

Fig.2 – The Maze

The White Grids signify where the paths can be formed and the Black Grids are the walls; the path cannot be formed on the walls.

****

****

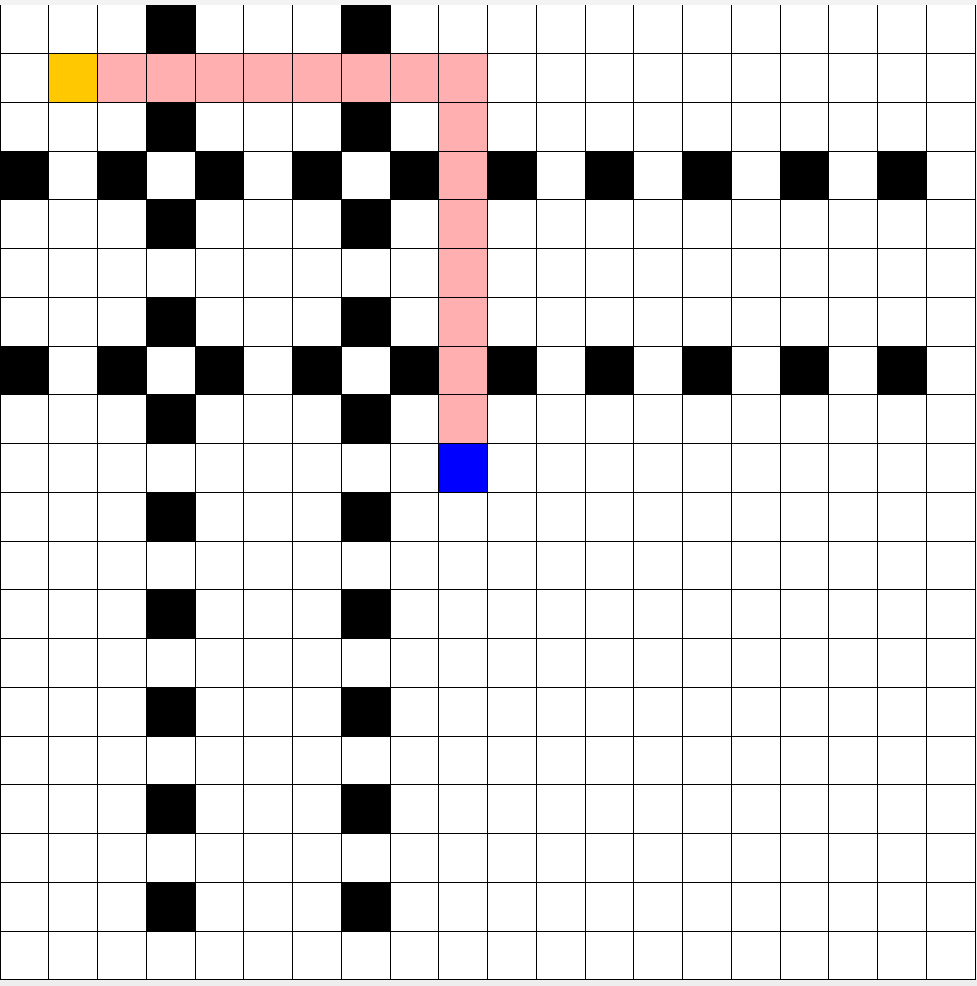
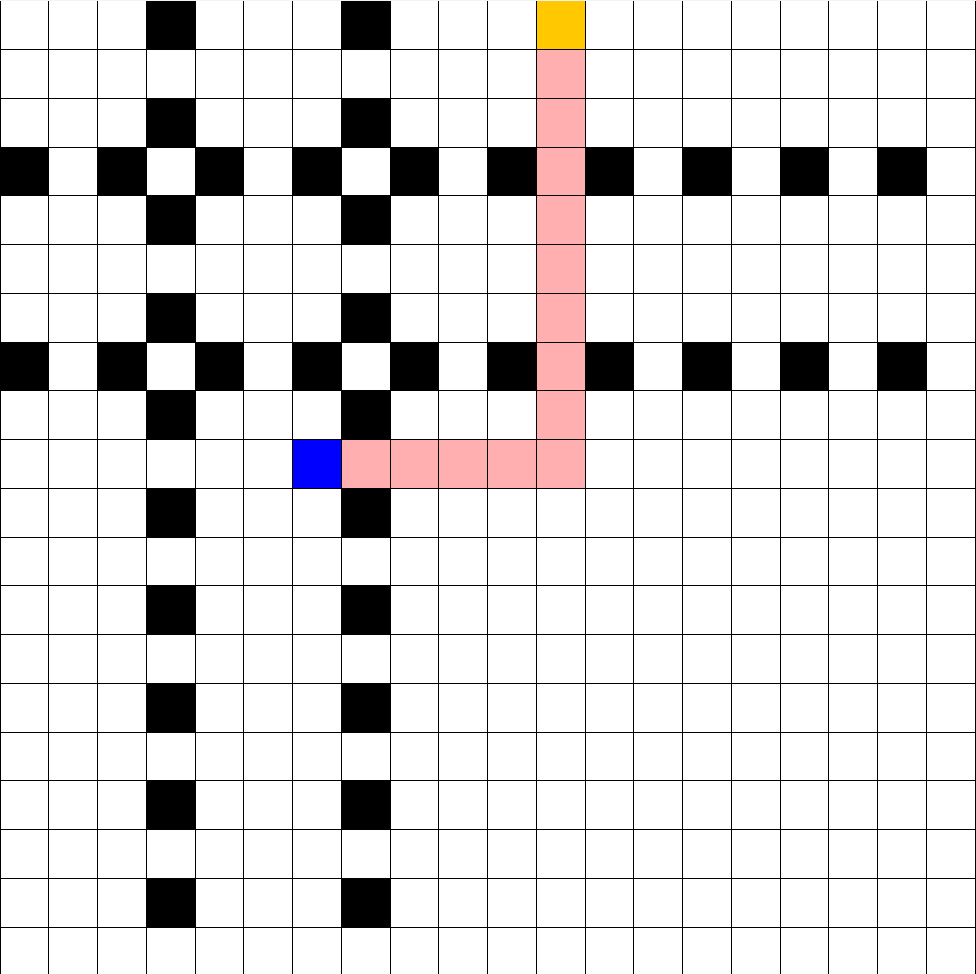
Fig.3 – The Orange Grid signifies the starting point of the maze. It can be selected dynamically by Left Clicking the mouse. The Blue Grid signifies the end of the maze, which can be dynamically selected by Right Clicking the mouse. The Pink Grids signify the SHORTEST PATH between the start and the end of the maze.

Fig.4 – Another Example



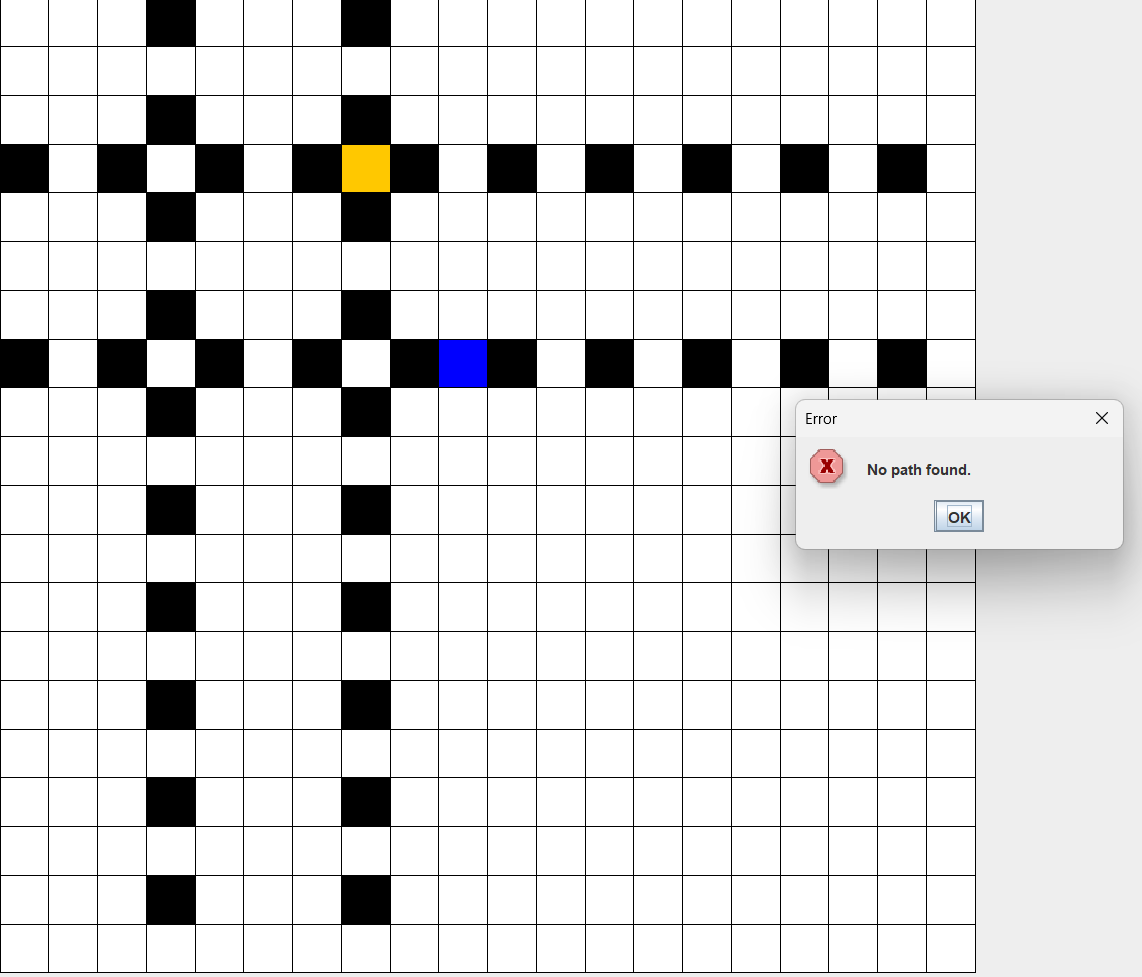
****

Fig.5 – When the points are given but are restricted by walls, a pop-up is given showing ‘No path found.’

**CONCLUSION**

In brief, this revised Dynamic Maze Solver increases the maze's scale and adds extra walls, enhancing its level of difficulty yet retaining its rich graphical nature. Users are allowed to reach for the best path between the predefined starting and finishing points within the maze by implementing the breadth-first searching algorithm. The user interactive behavior of solver allows him to dynamically provide these points to visualize their desired path at run time. So, this application provides a way to be professional and engaging in the development of maze-solving algorithms within a graphical context.

**BIBILOGRAPHY**

* https://medium.com/@luthfisauqi17\_68455/artificial-intelligence-search-problem-solve-maze-using-breadth-first-search-bfs-algorithm-255139c6e1a3#:~:text=According%20to%20geeksforgeeks.org%3A,at%20the%20next%20depth%20level.
* https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/#breadth-first-search-or-bfs-for-a-graph