MAZE GENEARATION

PROJECT REPORT

Submitted by :

B.E.PRANAV KUMAAR : CB.EN.U4AIE20052

DIVI ESWAR CHOWDARY: CB.EN.U4AIE20012

DINESH KUMAR M.R : CB.EN.U4AIE20011

HARSHA DABBRA : CB.EN.U4AIE20010

of

2nd SEM B.Tech CSE-AI

For the Completion of

19AIE111 – DATA STRUCTURES AND ALGORITHMS

CSE - AI

AMRITA VISHWA VIDYAPEETHAM, ETTIMADAI

Submitted on:

06th July 2021

ACKNOWLEDGEMENT

Nammah Shivaya, firstly of all of us express our gratitude to Amma as finally we were able to finish our assignment that has been given by our Data Structures and Algorithms teacher to us.

Next, we would like to express our special thanks to our project guide and teacher Mr.Sachin Kumar S who gave us the golden opportunity to do this wonderful project on the topic "Maze Generation", which in turn helped us in doing a lot of research in fields we were less familiar with, via which we learned so many new things. We are really thankful to them.

Lastly, we would also like to thank our parents and friends who helped us a lot in finalizing this project within the limited time frame and keeping us motivated throughout the process.

We would like to declare that we never knowingly discredited anyone of their work nor have we made efforts to plagiarize and for those whose work we referred to suitably equip ourselves to complete this project we acknowledge their effort and thank them for it.

TABLE OF CONTENTS

ABSTRACT

Procedural content generation is widely used in game development, although it does not give an opportunity to generate the whole game level. However, for some game artifacts procedural generation is the preferred way of creation. Mazes are a good example of such artifacts: manual generation of mazes does not provide a variety of combinations and makes games too predictable. Our project gives an overview of four maze generation algorithms: Binary Tree, Side Winger Algorithm, Hunt and Kill Algorithm, Depth-first search. These algorithms describe four conceptually different approaches to maze generation. The completion time of the algorithms are compared, and advantages and disadvantages of each algorithm are given.

1. INTRODUCTION

1.1 What is a maze?

A network of paths and hedges designed as a puzzle through which one has to find a way. The word is used to refer both to branching tour puzzles through which the solver must find a route, and to simpler non-branching patterns that lead unambiguously through a convoluted layout to a goal.

1.2 What is a perfect maze?

A so called 'perfect' maze **has every path connected to every other path**, so there are no unreachable areas. Also, there are no path loops or isolated walls. There is always one unique path between any two points in the maze.

1.3 How can we achieve a maze?

Maze generation

– By removing walls of the cells in a grid.

– By sequentially building the walls within a boundary.

- There are few algorithms that can generate maze:

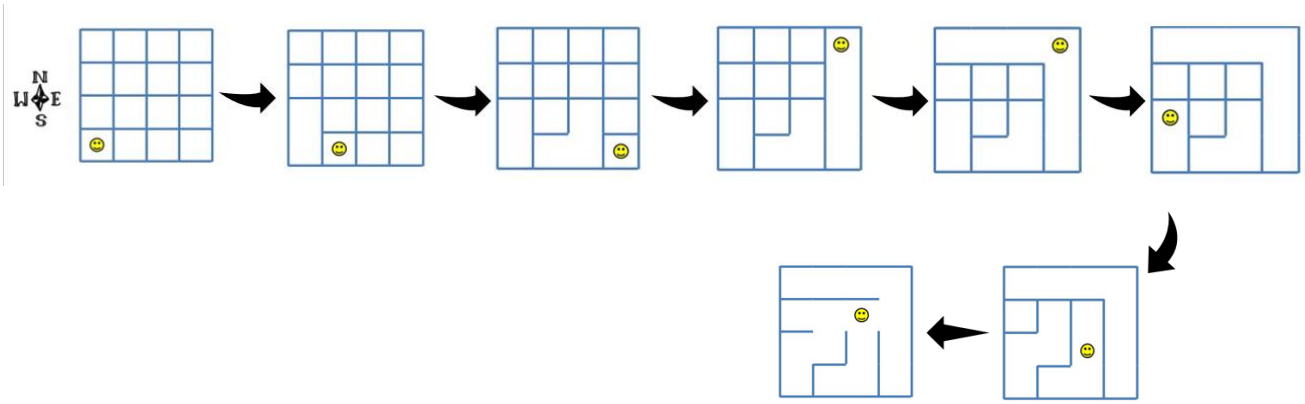
1. Binary Tree
2. Side Winger Algorithm
3. Hunt and Kill Algorithm
4. Recursive Backtracker

2. MAZE GENERATION METHODS

2.1 – 2.4 generation methods 2.5 – 2.6 solving methods

(MAZE GENERATION)
2.1 BINARY TREE

Choose a grid with required number of cells.
Choose a cell and remove either north or east wall at random.
Do not remove walls for active cell change. (Linking Vs Active Cell)



Steps Involved:

- For each existing cell in the grid:

1. Get if they exist, north or west neighbors
2. Now, let's flip a coin. If it's "heads", we carve the right wall, otherwise, the upper wall.
3. Toss a coin to connect with one of them.
4. It is already done!

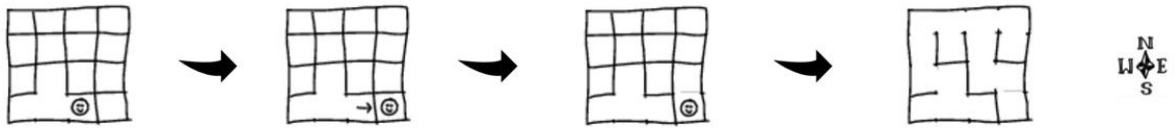
2.2 SIDE WINGER ALGORITHM

Unlike Binary Tree, the Sidewinder won't easily let us start carving anywhere we like.

It has a strong preference for beginning in the western column, so that's where we'll start.

Once again, tails will mean "carve east," but we'll see that heads means something a little different this time around.

When the coin comes up heads, we look back on the path we just made, at that group of cells that were most recently joined by carving through walls.

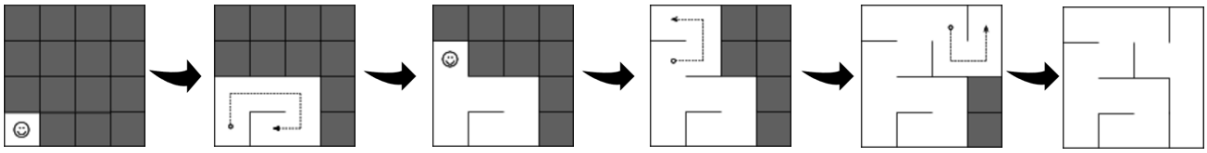


Required Conditions:

1. Coin toss required
2. Requires to start from west most column of cells.
3. Linking ≠ Active Cell

2.3 HUNT AND KILL ALGORITHM

- Hunt and Kill is based on Random Walk.
- It allows only to random walk unvisited cells.



Steps Involved:

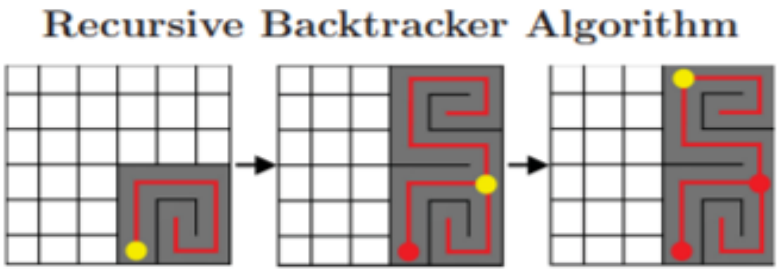
1. Choose a starting location.
2. Perform a random walk, carving passages to unvisited neighbors, until the current cell has no unvisited neighbors.
3. Enter “hunt” mode, where you scan the grid looking for an unvisited cell that is adjacent to a visited cell. If found, carve a passage between the two and let the formerly unvisited cell be the new starting location.
4. Repeat steps 2 and 3 until the hunt mode scans the entire grid and finds no unvisited cells.

2.4 DEPTH-FIRST SEARCH

- It is similar to Hunt and Kill.
- A large grid of cells is the space for MAZE
- Each cell has 4 walls [top, right, bottom, left].

Steps Involved:

1. Starting from the current cell, the program randomly selects neighboring cells and removes the walls between them. This cell is mark visited and filled to the stack (Backtracking)
2. The program keeps on randomly selecting neighbor until it reaches the cell with no neighbor. (dead-end)
3. After it encounters dead-end it backtracks through the path until it reaches a cell with an unvisited neighbor.
4. This process continues until the stack is empty.



Recursive Backtracker algorithm example. Yellow points depict cells where the new walk started from.

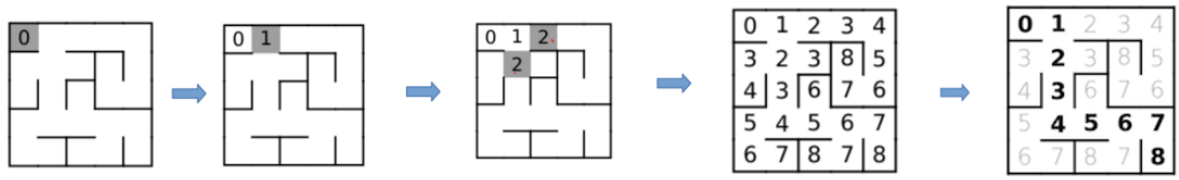
- If we are to perfectly follow the steps mentioned in each algorithm, we will be able to generate mazes with corresponding properties.

(MAZE SOLVING)

2.5 DIJKSTRA'S ALGORITHM

An interesting and efficient method to solve a maze

Solves a maze by finding the **shortest path** between a given **starting and an end point.**



Steps Involved:

1. Determines the starting point of the grid (commonly the northwestern-most cell).
2. Find that cell's navigable neighbors.

Repeating the above steps, taking care not to revisit already-visited cells.

2.6 MAZE SOLVING USING BACKTRACKING

We can creatively modify the Recursive backtracking to identify the start and end point of maze and allow us to find the solution.

Steps Involved:

- Repeat the steps of 2.4 while trying to identify the start and end points of maze using dimension interpretation.

3. DESCRIPTION

3.1 ALGORITHM ARE WE USING FOR MAZE GENERATION AND SOLVING

ALGORITHMS WE ARE USING:

- Recursive backtracking implementation of randomized Depth-first search algorithm**. (2.4 in method 1) – for generation**
- **(2.6 in method 2) – for solving**

CONCEPTS INCLUDED:

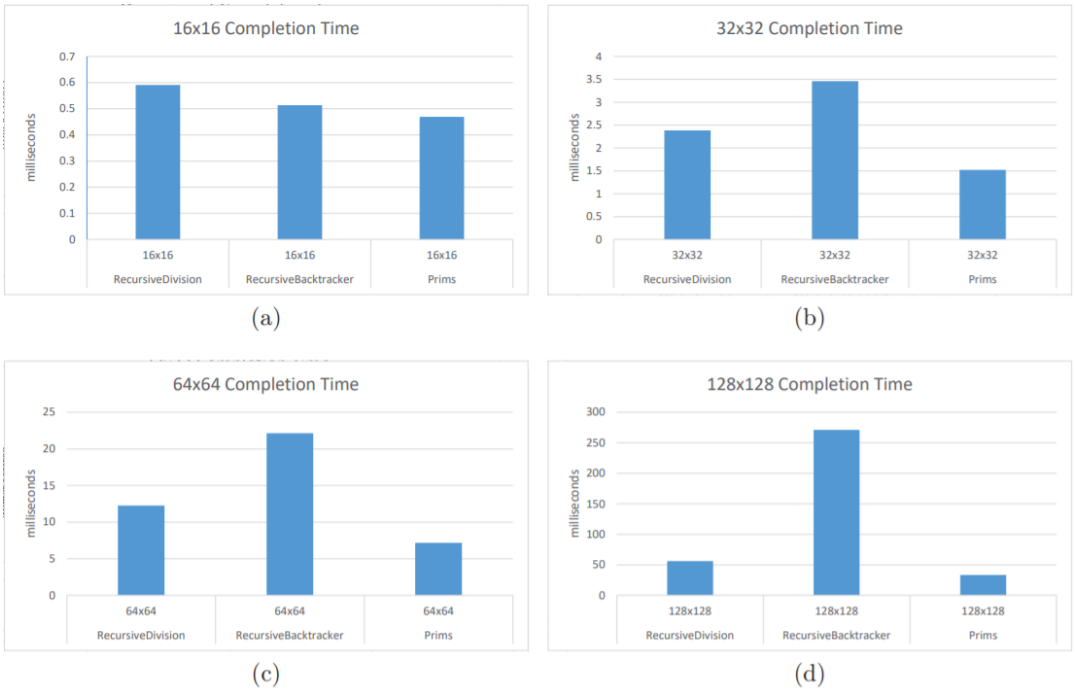
1. Arrays – to maintain features of each cell like state, walls etc
2. Recursion – process employed to implement backtracking

3. Stack – keeps track of the history of steps to get to the current cell from first cell.

3.2 REASONS FOR CHOOSING RECURSIVE BACKTRACKING

1. This algorithm finds minimal path among the all.
2. The major advantage of the backtracking algorithm is the ability to find and count all the possible solutions rather than just one while offering decent speed. In fact, this is the reason it is so widely used.
3. Mazes generated with a depth-first search have a low branching factor and contain many long corridors, because the algorithm explores as far as possible along each branch before backtracking.
4. Thereby algorithm also handles dead ends.

3.3 TIME COMPLEXITY OF DFS



OBSERVATIONS:

At resolution 16x16: RecBack has intermediate completion time.

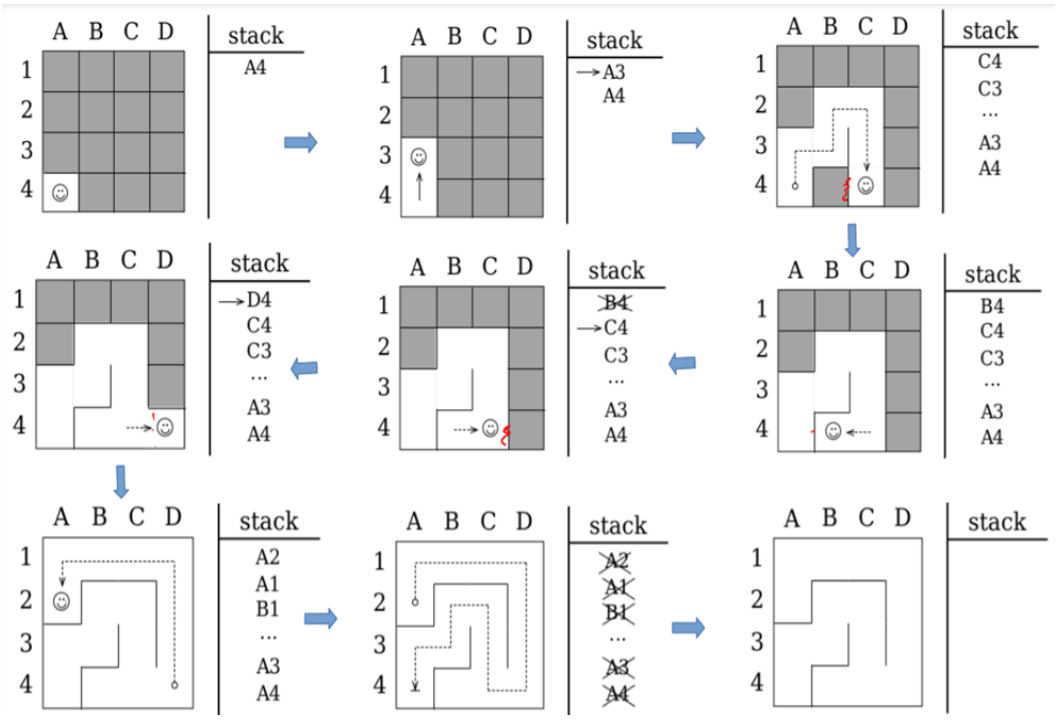
At all other resolutions: RecBack has the longest completion time, RecDiv has intermediate completion time, and Prims has the shortest completion time.

3.4 DETAILED ALGORITHM OF DFS WE USED IN CODE

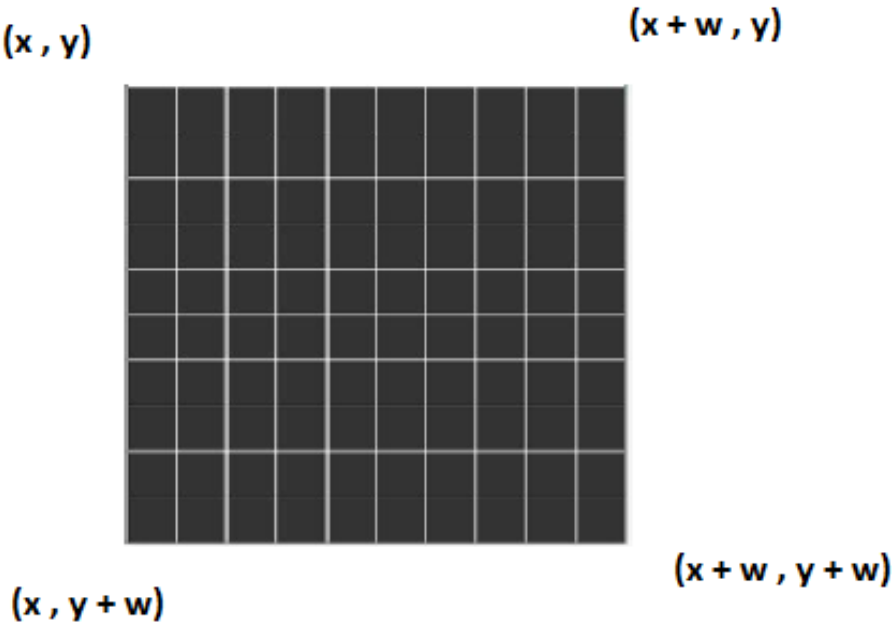
1. Make the initial cell the current cell and mark it as **visited.**

2. **While there are unvisited cells:**
    - If the current cell has neighbors which have not been visited.
    - Randomly choose any one of the unvisited neighbors.
    - Push the current cell to the stack.
    - Remove the wall between the current cell and chosen cell.
    - Make the chosen cell the current cell and mark it as visited.

3. **Else if the stack is not empty**
    - Pop a cell from the stack
    - Make it the current cell.

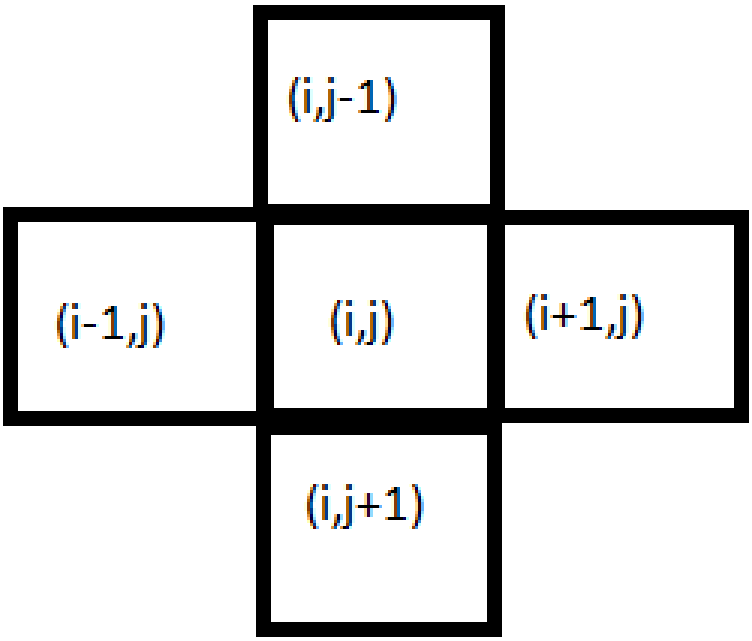


3.5 GRID INTERPRETATION



3.6 NEIGHBOUR CALCULATION

- Neighbors of (i,j)
- Difference = (Current Cell - Neighbor)



3.7 TRADE OFFS AND BLOWS

1) It needs to maintain a stack.

2) In worst case may require twice as memory.

3) Backtracking Approach is not efficient for solving strategic Problem.

4. IMPLEMENTATION

4.1 METHOD 1: MAZE GENERATION

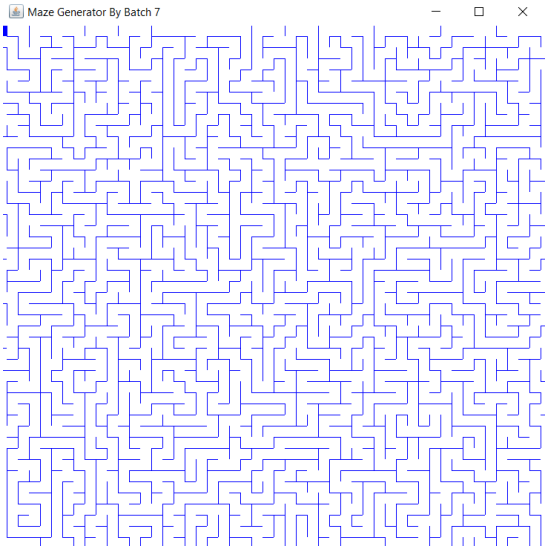
CODE

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Main extends JFrame implements Runnable {
    public static final int WIDTH = 400, HEIGHT = 200;

    public static Main ins;

    private Thread thread;
    private boolean running;

    JButton start = new JButton("Begin");
    JButton exit = new JButton("Quit");

    public Main() {
        // region Window Settings
        super("Maze Generator by Batch 7");

        running = false;

        setBounds(0, 0, WIDTH, HEIGHT);
        setResizable(true);
        setLocationRelativeTo(null);

        setLayout(null);
```

```
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    //endregion

    // region Start Button
    start.setSize(150, 100);
    start.setLocation(WIDTH / 2 - 25, HEIGHT - 175);
    start.setBackground(Color.GREEN);
    start.setForeground(Color.BLACK);
    add(start);
    start.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            ins.setVisible(false);
            new Maze();
        }
    });
    //endregion

    // region Quit Button
    exit.setSize(150, 100);
    exit.setLocation(WIDTH / 3 - 75, HEIGHT - 175);
    exit.setBackground(Color.RED);
    exit.setForeground(Color.BLACK);
    add(exit);
    exit.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {

            System.out.println("Leaving us!!!");
            System.exit(0);
        }
    });




    setVisible(true);
    this.start();
}

private synchronized void start() {
    thread = new Thread(this);
    thread.start();

    running = true;
}
```

```
    private synchronized void stop() {
        try {
            thread.join();
            running = false;
        } catch(Exception e) { e.printStackTrace(); }
    }

    private void tick() { }

    private void renderComponent() {
        repaint();
    }

    public void run() {
        long past = System.currentTimeMillis();
        while(running) {
            long now = System.currentTimeMillis();
            tick();
            if((now - past) / 1000 == 60)
                renderComponent();
        }
        stop();
    }
    public static void main(String[] args) {
        System.out.println("Welcome");
        ins = new Main();
    }
}
```

FINAL OUTPUT



the generated maze using 2.

4.2 METHOD 2: MAZE SOLVING

CODE

```
import java.awt.*;
import javax.swing.*;


public class Maze extends JPanel implements Runnable {


    public static void main(String[] args) {
        JFrame window = new JFrame("Maze Solver By Batch 7 ");
        window.setContentPane(new Maze());
        window.pack();
        window.setLocation(120, 80);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setVisible(true);
    }


    int[][] maze;

    final static int backgroundCode = 0;
    final static int wallCode = 1;
    final static int pathCode = 2;
    final static int emptyCode = 3;
    final static int visitedCode = 4;


    Color[] color;
    int rows = 31;
    int columns = 41;
    int border = 0;
    int sleepTime = 5000;
    int speedSleep = 1;
    int blockSize = 12;

    int width = -1;
    int height = -1;

    int totalWidth;
    int totalHeight;
    int left;
    int top;

    boolean mazeExists = false;
```

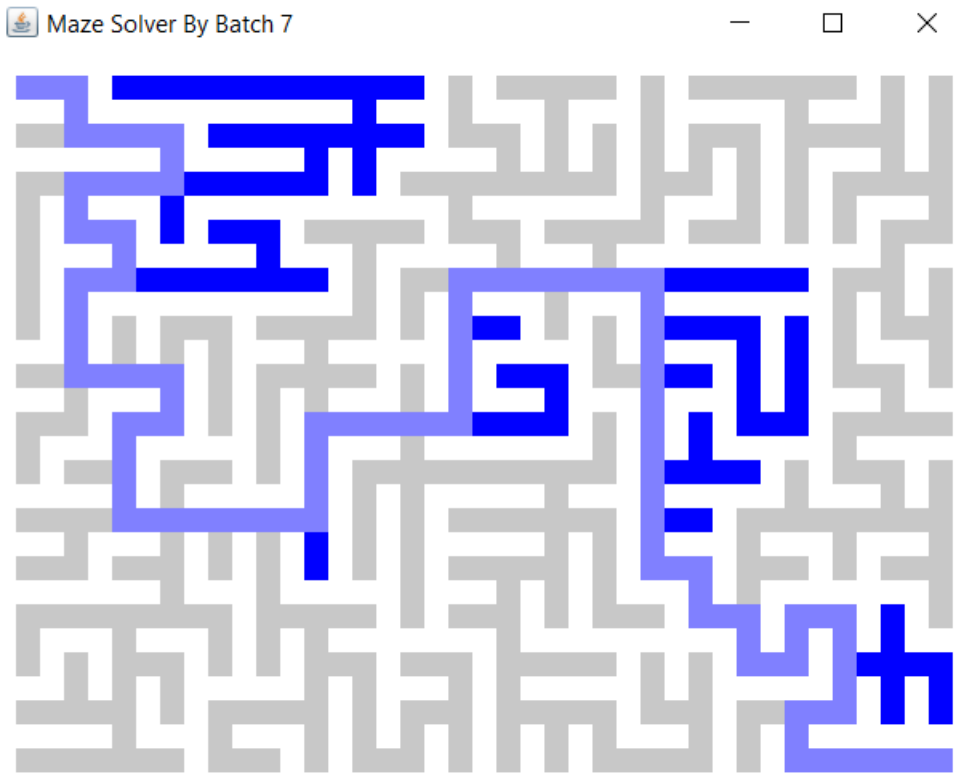
```
    public Maze() {
        color = new Color[] {
            Color.BLACK,
            Color.WHITE,
            new Color(128,128,255),
            Color.BLUE,
            new Color(200,200,200)
        };
        setBackground(color[backgroundCode]);
        setPreferredSize(new Dimension(blockSize*columns, blockSize*rows));
        new Thread(this).start();
    }

    void checkSize() {

        if (getWidth() != width || getHeight() != height) {
            width  = getWidth();
            height = getHeight();
            int w = (width - 2*border) / columns;
            int h = (height - 2*border) / rows;
            left = (width - w*columns) / 2;
            top = (height - h*rows) / 2;
            totalWidth = w*columns;
            totalHeight = h*rows;
        }
    }

    synchronized protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        checkSize();
        redrawMaze(g);
    }

    void redrawMaze(Graphics g) {

        if (mazeExists) {
            int w = totalWidth / columns;
            int h = totalHeight / rows;
            for (int j=0; j<columns; j++)
                for (int i=0; i<rows; i++) {
                    if (maze[i][j] < 0)
                        g.setColor(color[emptyCode]);
                    else
                        g.setColor(color[maze[i][j]]);
                    g.fillRect( (j * w) + left, (i * h) + top, w, h );
                }
        }
    }
```

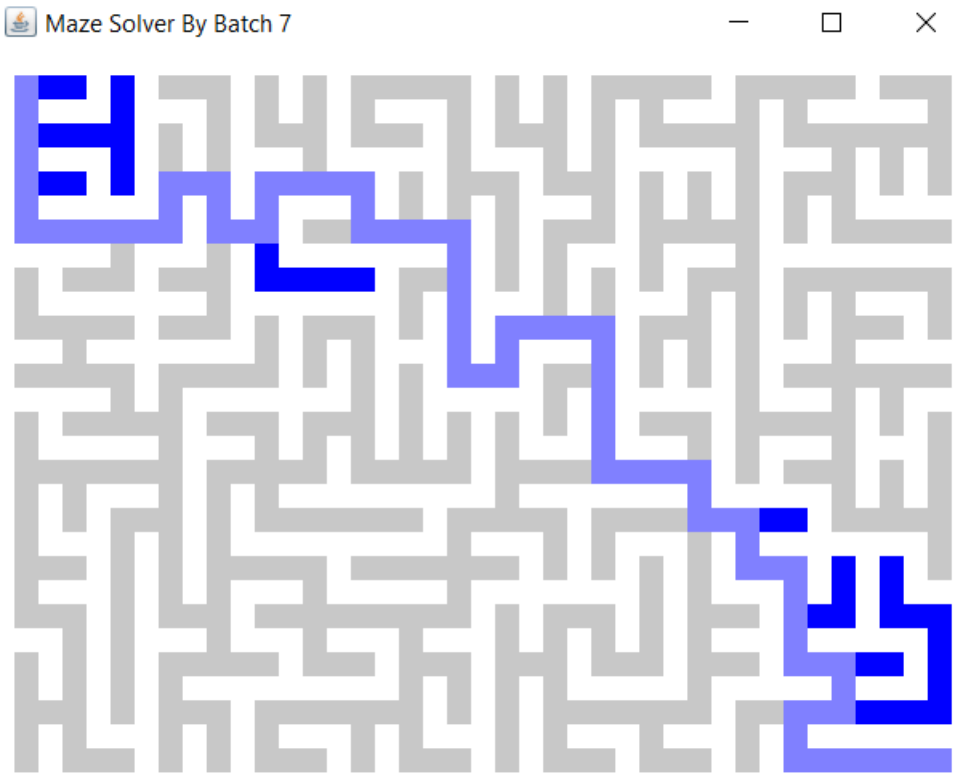
```
public void run() {

    try { Thread.sleep(1000); }
    catch (InterruptedException e) { }
    while (true) {
        makeMaze();
        solveMaze(1,1);
        synchronized(this) {
            try { wait(sleepTime); }
            catch (InterruptedException e) { }
        }
        mazeExists = false;
        repaint();
    }
}

void makeMaze() {

    if (maze == null)
        maze = new int[rows][columns];
    int i,j;
    int emptyCt = 0;
    int wallCt = 0;
    int[] wallrow = new int[(rows*columns)/2];
    int[] wallcol = new int[(rows*columns)/2];
    for (i = 0; i<rows; i++)
        for (j = 0; j < columns; j++)
            maze[i][j] = wallCode;
    for (i = 1; i<rows-1; i += 2)
        for (j = 1; j<columns-1; j += 2) {
            emptyCt++;
            maze[i][j] = -emptyCt;
            if (i < rows-2) {
                wallrow[wallCt] = i+1;
                wallcol[wallCt] = j;
                wallCt++;
            }
            if (j < columns-2) {
                wallrow[wallCt] = i;
                wallcol[wallCt] = j+1;
                wallCt++;
            }
        }
    mazeExists = true;
    repaint();
    int r;
    for (i=wallCt-1; i>0; i--) {
```

```
            r = (int)(Math.random() * i);
            tearDown(wallrow[r],wallcol[r]);
            wallrow[r] = wallrow[i];
            wallcol[r] = wallcol[i];
        }
        for (i=1; i<rows-1; i++)
            for (j=1; j<columns-1; j++)
                if (maze[i][j] < 0)
                    maze[i][j] = emptyCode;
    }

    synchronized void tearDown(int row, int col) {

        if (row % 2 == 1 && maze[row][col-1] != maze[row][col+1]) {

            fill(row, col-1, maze[row][col-1], maze[row][col+1]);
            maze[row][col] = maze[row][col+1];
            repaint();
            try { wait(speedSleep); }
            catch (InterruptedException e) { }
        }
        else if (row % 2 == 0 && maze[row-1][col] != maze[row+1][col]) {

            fill(row-1, col, maze[row-1][col], maze[row+1][col]);
            maze[row][col] = maze[row+1][col];
            repaint();
            try { wait(speedSleep); }
            catch (InterruptedException e) { }
        }
    }

    void fill(int row, int col, int replace, int replaceWith) {

        if (maze[row][col] == replace) {
            maze[row][col] = replaceWith;
            fill(row+1,col,replace,replaceWith);
            fill(row-1,col,replace,replaceWith);
            fill(row,col+1,replace,replaceWith);
            fill(row,col-1,replace,replaceWith);
        }
    }

    boolean solveMaze(int row, int col) {

        if (maze[row][col] == emptyCode) {
            maze[row][col] = pathCode;
            repaint();
            if (row == rows-2 && col == columns-2)
```

```
                return true;
            try { Thread.sleep(speedSleep); }
            catch (InterruptedException e) { }
            if ( solveMaze(row-1,col)  ||
                    solveMaze(row,col-1)  ||
                    solveMaze(row+1,col)  ||
                    solveMaze(row,col+1) )
                return true;

            maze[row][col] = visitedCode;
            repaint();
            synchronized(this) {
                try { wait(speedSleep); }
                catch (InterruptedException e) { }
            }
        }
        return false;
    }

}
```

FINAL OUTPUT





- The light blue colour depicts the solution to the maze whereas the dark blue depicts the paths from which the program backtracked before reaching the solution.
- We **get different outputs** each time. The above are some sample outputs from our Code.

CONCLUSION

We have successfully been able to complete a simulation of Maze Generation Algorithm and solved it.

FOR FUTURE WORK:

As an additional feature we can calculate the time required to generate the maze and to solve the maze.

Other researchers could try replicating the test results which would be beneficial for the relatively sparse field of study of mazes. Perhaps a metric for complexity better than DFS solver completion time could be found.

A search algorithm specifically crafted to more closely simulate human behavior (like having an imperfect memory, making it retread already walked paths).

No search-algorithm that considers human has been found, therefore it would perhaps be beneficial if such an algorithm was developed, perhaps by closely observing humans solving mazes, and then translate the observed behavior to algorithms. It would provide more accurate results when measuring complexity of mazes and could perhaps prove to have other uses.

BIBLIOGRAPHY

- the code for the implementation of the algorithms is in java

https://www.google.com – for definitions and information (cost and weight table)

ARTICLES

http://ipsitransactions.org/journals/papers/tir/2019jan/p5.pdf - Article for Maze Generation

https://en.wikipedia.org/wiki/Maze_generation_algorithm - Article for Maze Generation