# Maze-Inator

A project presented to the
College of Computer Studies
Xavier University

Cabili, Jamsed R.
Rodriguez, Gecel Bonn G.
Gabales, Mariz Shalia L.

March 2018

# Table of Contents

## ABSTRACT

A lot of people has been fascinated by maze or a labyrinth since the 5th century, it was made by the Egyptian people and called it the Labyrinth. A maze is a collection of paths that contains an entrance and an end, in which a solver must find its route out of it and out of the false end path. What seems to be fascinating about mazes is that it is a puzzle that has countless of forms with countless solutions.

## CCS Concepts

CCS → Software and its engineering → Software notations and tools → General programming languages → Language features → Classes and objects

## 1. INTRODUCTION

Maze-inator is a program that both generates random Maze and finds a path from the selected Starting Wall to the selected Ending Wall. The generated maze can be saved and loaded for future use. It utilizes JavaFX for Graphical User Interface(GUI). The program will make use of 2-Dimensional Array to embody the maze, each element in the maze is either a Cell or Wall object. The program uses Recursive Backtracking in its algorithm for creating the maze as well as finding the correct path from Start to End.

## 2. OBJECTIVES

This program main aim is to generate maze randomly and finds a path from start to end by using a simple algorithm implementation.

1. Create Classes and Interface for components of Maze such as Walls and Cells as well as Node and Stack.
2. Search for simple algorithm implementation for Generating the Maze.
3. Program the algorithm for Generating the Maze using a 2-Dimensional Array.
4. Add a program/feature for user input such as the Dimension of the Maze and start and end of the Maze.
5. Program the algorithm for Searching the path from start to end of the Maze by using backtracking.
6. Add a File System for user to be able to save and load the generated Maze.
7. Implement JavaFX for GUI.

## 3. SCOPE AND LIMITATION

The dimension of the Maze is at most 96 depending on the RAM or else it will cause a StackOverFlow Error while searching for a path. User cannot zoom in and out from the maze, but user can decrease the size(pixel) of the Maze.

## 4. FUNCTIONALITIES

### 4.1 Dimension & Size Choice

#### *4.1.1 Customizable Cell/Wall Size*

User can decrease or increase the pixel size of Cell and Wall.

#### *4.1.2 Number of Cells*

User can input the dimension of maze.

### 4.2 Start and End Choice

#### *4.2.1 Right Click Cell for Start (Generating)*

User will be able to select the starting cell to generate the Maze by clicking **right mouse button** to selected cell which will change the color of the cell from Black to White.

#### *4.2.2 Left Click Wall for Start (Path Finding)*

User will be able to select the starting wall of the Maze by clicking **left mouse button** to the Wall selected which will change the color of Wall from Black to Light Green to indicate that it is the starting.

#### *4.2.3 Right Click Wall for End (Path Finding)*

User will be able to select the ending wall of the Maze by clicking **right mouse button** to the Wall selected which changes the color of Wall from Black to White.

### 4.3 Buttons

#### *4.3.1Decrease Size*

Decreases the pixel size of walls and cells.

#### *4.3.2 Increase Size*

Increases the pixel size of walls and cells.

#### *4.3.3 Generate*

Generates the maze according to the dimension inputted by the user.

#### *4.3.4 Create*

Create the maze starting from the cell chosen by the user.

#### *4.3.5 Find Path*

Finds the path from the starting wall chosen by the user to the ending wall chosen by the user.

#### *4.3.6 Save*

**S**aves the maze currently in the screen to the name file inputted by the user in the Text Field

#### *4.3.7 Load*

Load the maze from a file inputted by the user in the Text Field

### 4.4 Text Field

#### *4.4.1 Dimension*

User can customize the dimension of the maze

#### *4.4.2 File Name*

User can input the name of the file to be loaded or saved into. If it already exists it will be overwritten. The extension will be FIleName.**dat**, user don't need to input the extension on the text field in order to load or save the Maze.

## 5. PSEUDOCODE

### 5.1 Creating the Maze

```
createBackTrack(Cell current, StackCell stck) {
    if current has no unvisited neighbors it just return;
    index = random number from 0 to 3;
    tempIn;
    for(int i = 0; i  < 4; i++) {
        tempIn = (index + i) if the sum is less than 4 or else to (index + i – 4);
        tempX = moveX[tempIn] + current's row coordinate;
        tempY = moveY[tempIn] + current's col coordinate;
        check if coordinate tempX, tempY is not out of bounds{
            check if the cell in that coordinate has not been visited {
                push the cell into stack stck;
                remove the wall between the current cell and the next cell;
                initialize current to the next cell;
                set its visited data field to true;
                recursively call createBackTrack(current, stck);
            }
        }
    }
    check if stck is not empty {
        initialize current to stck.pop();
        recursively call createBackTrack(current, stck);
    }
}
```

## 5.2 Finds Path from Start to End

```
revX = {0, 1, 0, -1};
revY = {1, 0, -1, 0};
findPathBT(int x, int y) {
    check all sides of the cell in coordinate (x, y) if it is the End
        return true;
    for(int i = 0; i < 4; i++) {
        tempX  = (x + revX[i]);
        tempY = (y + revY[i]);
        check if the object in coordinate (tempX,tempY) hasn't been visited or  the Wall is opened {
            visit the object in coordinate (tempX,tempY);
             if(findPathBt(tempX, tempY)
                return true;
            else
                unvisit or leave the object in coordinate (tempX,tempY);
        }
    }
    return false;
}
```

# 6. DATA STRUCTURE AND ALGORITHM

## 6.1 Data Structure

The Data Structure involved in this program is a Stack that can store the Object Cell. The stack uses nodes instead of array to store Cell, because the dimension of the Maze is dynamic. Implementing stack using nodes makes the size of stack also dynamic. In this case we don't have to worry about the capacity of stack being too small or large that it takes unnecessary space in the memory. In the stack we made sure that every time we push a Cell it was added on the head so that when we pop it would be much faster because we don't have to traverse to tail.

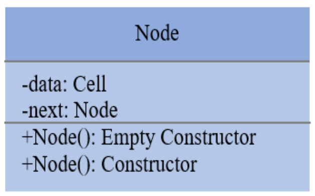


Figure 1. UML Node Diagram

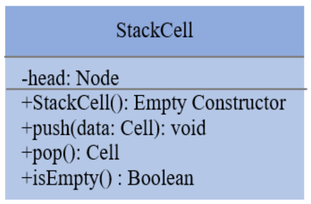


Figure 2. UML StackCell Diagram

Aside from Stack we also use 2-Dimensional Array in order embody the maze. The data type of the array is *Component* in order to store both objects Wall and Cell. They are stored alternatively, if the index of either row or column is divisible by 2 then a *Wall* is stored on it otherwise a *Cell*. Component is an interface that is implemented by both Wall and Cell in order for them to share common methods while extending the class Pane for GUI. Since a class can only extend one class we decided that Component will be an interface.

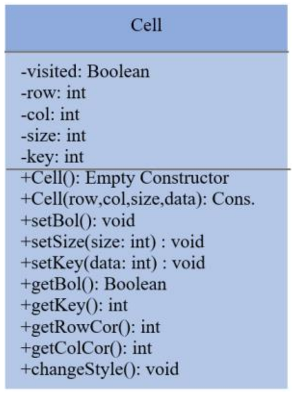


Figure 3. UML Cell Diagram

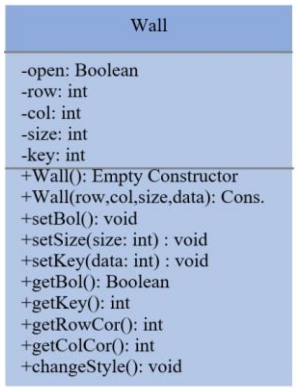


Figure 4. UML Wall Diagram

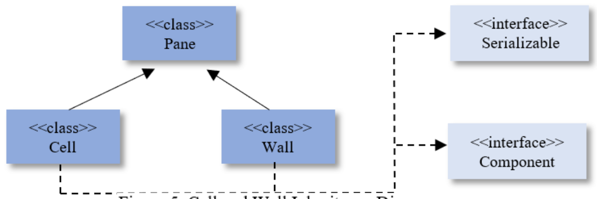


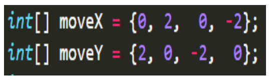
Figure 5. Cell and Wall Inheritance Diagram

The Cell and the Wall are both Abstract Data Type that extends *Pane* and implements *Component* and *Serializable*. They both have their own setter and getter methods in order for them to be encapsulated. We provided a data fields to represent the coordinate of Cell and Wall so that they can be used for future projects like other board games such as chess and others. The Cell and Wall has to implement the interface Serializable in order for them to be saved and loaded in a file using *ObjectOutputStream* and *ObjectInputStream* while maintaining their data fields and methods. They need to extend *Pane* for the GUI.

## 6.2 Algorithm

### *6.2.1 Traversing*

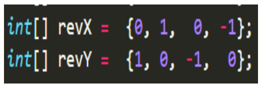
The main focus of the program is traversing through the Maze or in our case the 2D Array. With that we have to create the movements for traversing and we did this by creating two arrays that represent the x and y traversal. The array that we created is *moveX* and *moveY* which is for vertical and horizontal traversal respectively. For each element in *moveX* has pair in *moveY*, the pairing is according to their index. So the element of *moveX* in index 0 which is 0 is paired to index 0 in *moveY* which is 2. This pair(0,2) will perform the movement of traversing to right. The second pair of index will be for traversing down, third for traversing left, and the last one is for traversing up.



```
int[] moveX = {0, 2,  0, -2};
int[] moveY = {2, 0, -2,  0};
```

Movement for traversing to Cells

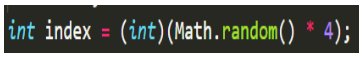
The program for creating the maze requires to visit a neighbor cell first and check it, before it can open the wall between them. That is why the elements in the *moveX* and *moveY* is 2 and -2 instead of 1 and -1 which is just to skip the wall between the two cells. Although this is the case it also need to traverse to the wall for it to open but only after checking the neighbor cell. That is why we created two new arrays just to traverse to the wall. The array is *revX* and *revY* which also follow the pairing paradigm of *moveX* and *moveY*. The algorithm has to keep track on which neighbor cell that the *current* cell has visited (left, right, top or bottom neighbor cell). It needs to keep track of this for it to know which wall to open. It does it by the index of the chosen movement of traversing to the neighbor cell. Let's say the chosen movement to traverse to neighbor cell is the index 0 which is the pair (0,2) in (*moveX, moveY)*, the index 0 will also be the movement to traverse to the wall between them which is the pair (0,1) in (*revX, revY*). This pair will just be added to the current's coordinate in order to traverse to the wall between the current and next cell. The arrays *revX* and *revY* will also be used in the algorithm for finding the right path. They will be used since the algorithm does not require to skip any elements in the 2-Dimensional Array.



```
int[] revX =  {0, 1,  0, -1};
int[] revY =  {1, 0, -1,  0};
```

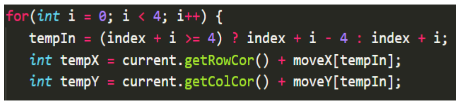
Movement for traversing to Walls and Cells

This traversal alone is not enough to create a random maze because the program needs to randomly select which neighbor to visit. If it is not random then it is going to visit the right neighbor always until it reaches the right most, then the bottom neighbors until it reaches the bottom most, then left, then top. Which basically create the same maze all the time with just different dimension depending on the user. This is the case because that is how we arranged the movements in the arrays. In order for this not to happen we have to generate a random number from 0 to 3 which is the variable *index*. This number will be the basis on which neighbor that the current cell will first visit and check. So, if the generated number is 2 then it will perform the movement in index 2 which is to visit the left neighbor cell, then index 3 which is to visit the top, then back to index 0 which is right, and then index 1 which is the bottom neighbor.



```
int index = (int)(Math.random() * 4);
```

Generating random number



```
for(int i = 0; i < 4; i++) {
    tempIn = (index + i >= 4) ? index + i - 4 : index + i;
    int tempX = current.getRowCor() + moveX[tempIn];
    int tempY = current.getColCor() + moveY[tempIn];
```

Initializing which neighbor to visit

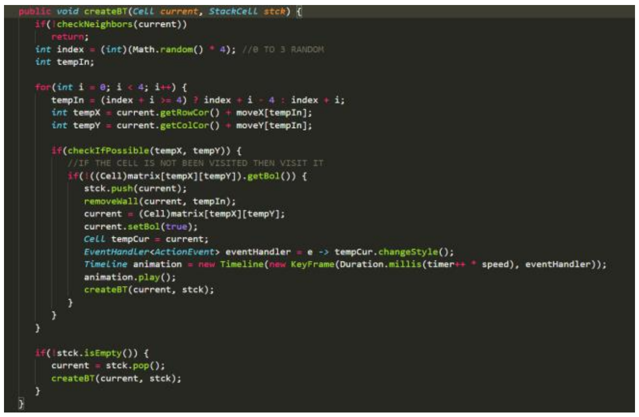
### *6.2.1BackTracking*

The main algorithm used in this program is recursive backtracking. It is used in both creating the Maze and finding the correct path from start to end.

For creating the Maze all it does:

1. Check if the current cell has unvisited neighbors if it has then visit it and push it to the stack
2. The visited cell now becomes the current cell.
3. Check if current cell has unvisited neighbors if not then backtrack.
4. Visit the other unvisited neighbors of previous cell.
5. If all has been visited, then check if the stack is empty if it's not then pop from stack and the popped cell now becomes the current.
6. Then go back to step 1

Although this algorithm is fast, easy to understand and straightforward for implementation. It does take a toll on the memory because it needs enough memory space to store all element in the maze. Not to mention that in our case there are many objects involve using javaFX such as the animation, timeline, keyframe, and others. So, a StackOverFlow Error is a common occurrence in our program.

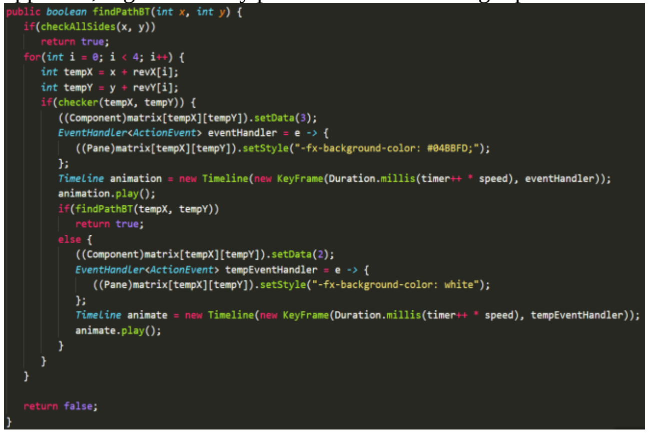


```
public void createBT(Cell current, StackCell stck) {
    if(!checkNeighbors(current))
        return;
    int index = (int)(Math.random() * 4); //0 TO 3 RANDOM
    int tempIn;

    for(int i = 0; i < 4; i++) {
        tempIn = (index + i >= 4) ? index + i - 4 : index + i;
        int tempX = current.getRowCor() + moveX[tempIn];
        int tempY = current.getColCor() + moveY[tempIn];

        if(checkIfPossible(tempX, tempY)) {
            //IF THE CELL IS NOT BEEN VISITED THEN VISIT IT
            if(!((Cell)matrix[tempX][tempY]).getBol()) {
                stck.push(current);
                removeWall(current, tempIn);
                current = (Cell)matrix[tempX][tempY];
                current.setBol(true);
                Cell tempCur = current;
                EventHandler<ActionEvent> eventHandler = e -> tempCur.changeStyle();
                Timeline animation = new Timeline(new KeyFrame(Duration.millis(timer++ * speed), eventHandler));
                animation.play();
                createBT(current, stck);
            }
        }
    }

    if(!stck.isEmpty()) {
        current = stck.pop();
        createBT(current, stck);
    }
}
```

For finding the path:

1. Check the walls of current cells if it is the end.
2. If it is then return true
3. If not, then visit the next cell.
4. Continue visiting the neighbor cell until you find the End in that path.
5. If the path that the program took does not locate End, then it backtracks and goes for another path.

This is not the most efficient algorithm for finding path, but it is the simplest to implement. This algorithm is a brute force approach, it goes to every path until it finds the right path.



```
public boolean findPathBT(int x, int y) {
    if(checkAllSides(x, y))
        return true;
    for(int i = 0; i < 4; i++) {
        int tempX = x + revX[i];
        int tempY = y + revY[i];
        if(checker(tempX, tempY)) {
            ((Component)matrix[tempX][tempY]).setData(3);
            EventHandler<ActionEvent> eventHandler = e -> {
                ((Pane)matrix[tempX][tempY]).setStyle("-fx-background-color: #04BBFD;");
            };
            Timeline animation = new Timeline(new KeyFrame(Duration.millis(timer++ * speed), eventHandler));
            animation.play();
            if(findPathBT(tempX, tempY))
                return true;
            else {
                ((Component)matrix[tempX][tempY]).setData(2);
                EventHandler<ActionEvent> tempEventHandler = e -> {
                    ((Pane)matrix[tempX][tempY]).setStyle("-fx-background-color: white");
                };
                Timeline animate = new Timeline(new KeyFrame(Duration.millis(timer++ * speed), tempEventHandler));
                animate.play();
            }
        }
    }

    return false;
}
```

## 7. CONCLUSION

Human have always been fascinated by mazes, it makes us think which helps us pass time. The program Maze-inator generates Maze randomly at the same time it can find the correct path from start to end. It does it by using recursive backtracking which is one of the simplest implementation, but it takes a toll on the memory. A 2-Dimensional Array that contains objects Wall and Cell will embody the maze. There are multiple arrays(*moveX¸moveY,revX,revY*) that will perform the traversal in the maze. The maze can be saved and loaded for future use such as to print the generated maze, so people can try to solve it, or it can be used for future programs. Overall, this program is a great project to demonstrate what Object-Oriented Programming is, not only because there are objects and data structure involved but also because of the reusability of the objects and algorithms.

## 8. APPENDICES

### 8.1 Project Proposal

A. Memberss

1. Gecel Bonn G. Rodriguez
2. Mariz Shalia Gabales
3. Jamsed R. Cabili

B. Title: Maze-inator

C. Description

Maze-inator is a program that generates Maze randomly by inputting the dimension and lets the user choose the *Start* and *End* of the Maze, it will then find the path from Start to End. It utilizes JavaFX for Graphic User Interface(GUI). We will use 2-Dimensional Array to represent the maze, each element in the array will be an object either a Cell or Wall.

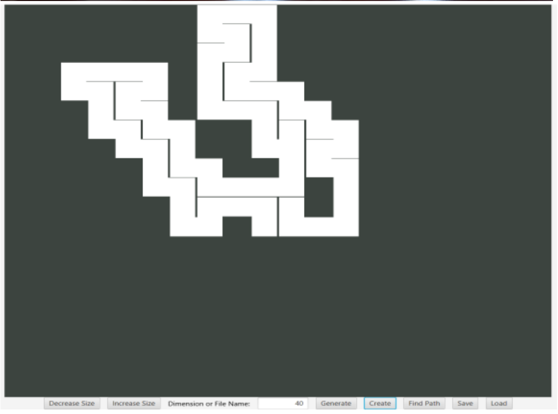
D. Functionalities and Features

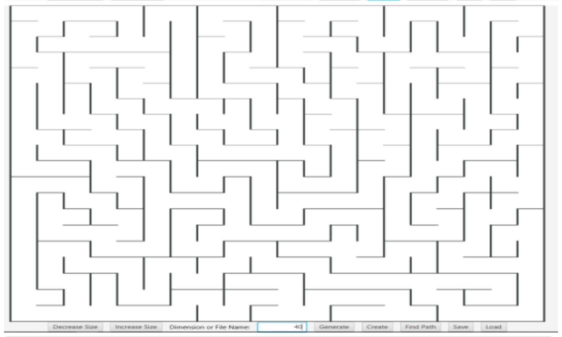
- Objects
  - Walls
  - Cells
- GUI
  - JavaFX
- User's Choice
  - Size
    - Customizable Cell and Wall Size
    - Customizable Maze Dimension
  - Start and End
    - Left mouse click to Wall for Start
    - Right mouse click to Wall for End
- Controls
  - Buttons
  - Text Field
- Save and Load
  - Save – user will be able to save the Generated Maze
  - Load – user will be able to load the Generated Maze

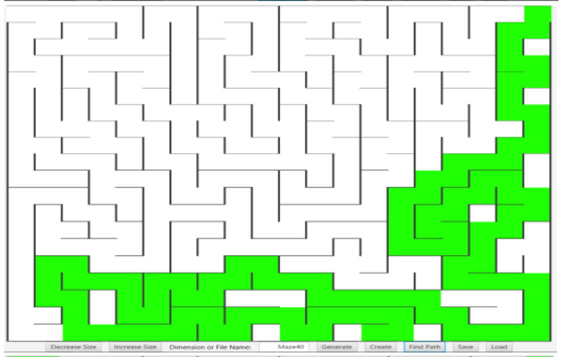
E. Possible Data Structure and Algorithm to be used

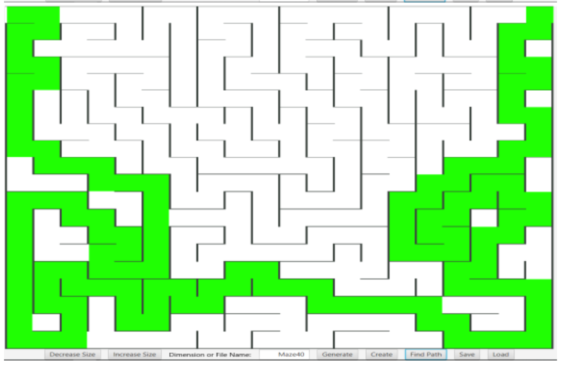
- Stack
- 2-Dimensional Array
- Recursive Backtracking
- Abstract Data Types
- File System

### 8.2 Photo Documentation


Decrease Size
Increase Size
Dimension or File Name:
40
Generate
Create
Find Path
Save
Load


Decrease Size
Increase Size
Dimension or File Name:
40
Generate
Create
Find Path
Save
Load


Decrease Size
Increase Size
Dimension or File Name:
Maze40
Generate
Create
Find Path
Save
Load


Decrease Size
Increase Size
Dimension or File Name:
Maze40
Generate
Create
Find Path
Save
Load