Computer Science NEA

By

Gabriel Guimaraes

“Maze Visualiser”

[Glossary of terms 5](#_Toc133575597)

[Analysis 6](#_Toc133575598)

[Background Information 6](#_Toc133575599)

[Research 6](#_Toc133575600)

[User Interface 6](#_Toc133575601)

[Current Solutions 6](#_Toc133575602)

[Problems with Current Solutions 7](#_Toc133575603)

[Image Input 8](#_Toc133575604)

[Width / Height 11](#_Toc133575605)

[Generation Algorithm 11](#_Toc133575606)

[Solving Algorithms 12](#_Toc133575607)

[Dead End Remover 13](#_Toc133575608)

[Maze entry 15](#_Toc133575609)

[Background / Maze / Solve colour 15](#_Toc133575610)

[End-user 16](#_Toc133575611)

[Interview with end user 16](#_Toc133575612)

[Questions & Answers 16](#_Toc133575613)

[Prototype 16](#_Toc133575614)

[Feedback from end-user 18](#_Toc133575615)

[Requirements from end-user 18](#_Toc133575616)

[New prototype 18](#_Toc133575617)

[Objectives 19](#_Toc133575618)

[Primary Objectives 19](#_Toc133575619)

[Secondary Objectives 19](#_Toc133575620)

[Data Flow Diagram 22](#_Toc133575621)

[Data Flow Diagram Glossary 22](#_Toc133575622)

[Level 0 Data Flow Diagram 23](#_Toc133575623)

[Level 1 Data Flow Diagram 24](#_Toc133575624)

[Design Data Dictionary (DDD) 28](#_Toc133575625)

[Public Variables [Maze Form] 28](#_Toc133575626)

[Class ‘Circular Queue’ Attributes and Methods 29](#_Toc133575627)

[Class ‘PriorityQueue’ Attributes and Methods 29](#_Toc133575628)

[Class ‘Cell’ Attributes and Methods 30](#_Toc133575629)

[initailizeMaze 30](#_Toc133575630)

[setMazeEntryExit 31](#_Toc133575631)

[resetMaze 31](#_Toc133575632)

[animationLock 31](#_Toc133575633)

[interpolateColour 31](#_Toc133575634)

[randomisedDFS 31](#_Toc133575635)

[randomisedPrims 32](#_Toc133575636)

[kurskals 32](#_Toc133575637)

[aldousBorder 32](#_Toc133575638)

[distanceCalc 32](#_Toc133575639)

[dijkstra 32](#_Toc133575640)

[BFS 33](#_Toc133575641)

[Astar 33](#_Toc133575642)

[wallFollower 33](#_Toc133575643)

[deadEndRemover 33](#_Toc133575644)

[generateBtn\_Click 34](#_Toc133575645)

[downloadMaze 34](#_Toc133575646)

[componentAnalysis 34](#_Toc133575647)

[imageInputBtn\_Click 34](#_Toc133575648)

[Public Variables [Helper Form] 34](#_Toc133575649)

[Design 35](#_Toc133575650)

[System Overview 35](#_Toc133575651)

[Flow Chart 36](#_Toc133575652)

[Flow Chart Glossary 36](#_Toc133575653)

[Level 0 Flowchart 37](#_Toc133575654)

[Level 1 Flowchart 39](#_Toc133575655)

[Inputs / Processing / Storage / Output (IPSO) 51](#_Toc133575656)

[General IPSO 51](#_Toc133575657)

[Depth First Search Backtracker IPSO 51](#_Toc133575658)

[Randomised Prims IPSO 51](#_Toc133575659)

[Kruskal's IPSO 52](#_Toc133575660)

[Aldous-Border IPSO 52](#_Toc133575661)

[Dijkstra’s IPSO 52](#_Toc133575662)

[Breath First Search IPSO 53](#_Toc133575663)

[A\* IPSO 53](#_Toc133575664)

[Wall Follower IPSO 54](#_Toc133575665)

[Algorithms 55](#_Toc133575666)

[Cell Class 55](#_Toc133575667)

[Image Input 56](#_Toc133575668)

[Generation Algorithms 58](#_Toc133575669)

[Dead End Remover 62](#_Toc133575670)

[Download 63](#_Toc133575671)

[Chronological Design 63](#_Toc133575672)

[Implementation 64](#_Toc133575673)

[RLE Encoding 64](#_Toc133575674)

[Height/Width 64](#_Toc133575675)

[Changing RGB to HEX 64](#_Toc133575676)

[Converting to .JPG 64](#_Toc133575677)

[Saving a copy of the image 64](#_Toc133575678)

[Information button 64](#_Toc133575679)

[Removing Wave Function Collapse 64](#_Toc133575680)

[Naming Conventions 65](#_Toc133575681)

[Technical Solution 66](#_Toc133575682)

[Maze Form 66](#_Toc133575683)

[Helper Form {NOT COMPLETED} 93](#_Toc133575684)

[Testing 95](#_Toc133575685)

[Testing URL 95](#_Toc133575686)

[Testing maze generation and initialisation 95](#_Toc133575687)

[Testing maze solving 95](#_Toc133575688)

[Testing dead-end removal 96](#_Toc133575689)

[Testing maze entry-exit 97](#_Toc133575690)

[Testing maze colours 97](#_Toc133575691)

[Testing maze animations 98](#_Toc133575692)

[Image to maze testing 99](#_Toc133575693)

[General Testing 100](#_Toc133575694)

[Evaluation 101](#_Toc133575695)

[Personal Feedback 101](#_Toc133575696)

[Objective Evaluation 101](#_Toc133575697)

[End User Feedback 103](#_Toc133575698)

[Reflections on end-user feedback 108](#_Toc133575699)

[Were the previous feedback points met? 108](#_Toc133575700)

[New requirements from end-user 108](#_Toc133575701)

[Further improvements 108](#_Toc133575702)

[Responsive application 108](#_Toc133575703)

[Different maze types 108](#_Toc133575704)

[Making this a web or mobile application 109](#_Toc133575705)

[Objective Mapping 109](#_Toc133575706)

[Appendix 1 110](#_Toc133575707)

[Unhighlighted code 110](#_Toc133575708)

[Maze Form 110](#_Toc133575709)

[Helper Form 136](#_Toc133575710)

[Appendix 2 137](#_Toc133575711)

[Youtube Testing Links 137](#_Toc133575712)

[Playlist 137](#_Toc133575713)

[Feature showing and bug fixing 137](#_Toc133575714)

[1.0 137](#_Toc133575715)

[2.0 137](#_Toc133575716)

[3.0 137](#_Toc133575717)

[4.0 137](#_Toc133575718)

[5.0 138](#_Toc133575719)

# Glossary of terms

|  |  |
| --- | --- |
| **Name / Term** | **Description** |
| RGB | Stands for Red, Green, and Blue, is a colour model that combines these primary colours to represent a vast spectrum of hues. |
| Grayscale | Often used in image processing, refers to the representation of an image using shades of grey. |
| Luminosity | The intrinsic brightness of an object. |
| RLE Encoding | Run-Length Encoding is a compression technique that efficiently represents repetitive data by storing the length of consecutive identical elements. |
| Stack | Is a versatile data structure that follows the Last-In, First-Out (LIFO) principle. |
| Tiles | Used to represent the units that are used to generate coherent patterns. |
| Superposition | Where all cells exist in multiple tiles simultaneously, unit measurement forces the cell to adopt a final tile. |
| Adjacency Rules | This refers to the constraints that dictate how neighbouring tiles can be arranged. |
| Entropy | As a measure of the number of tiles that in superposition, for a single cell. |
| Graph | Is a structure representing a set of objects that are connected. |
| Node | A distinct object within a graph. |
| Tree | A connected, non-looping graph structure that presents a hierarchical organization of nodes and. |
| Walkable Neighbour | A neighbouring cell that has not been visited to before by the algorithm. |
| HEX | Short for hexadecimal, is a base-16 numeral system commonly used in computing to represent binary data in a more compact format. |
| Babyproofing | Creating a safe user interface that does not break when erroneous data is inputted. |
| GUI | Stands for Graphical User Interface is a way for the user to interact with software using graphical elements. |
| Weave Maze | A variation of a maze where paths can pass over or under each other. |
| Mobius-strip | Is a one-sided surface that can be constructed by joining 2 ends of a rectangle with a half twist. |

# Analysis

## Background Information

Maze generation and solving is a popular area within computer science. However, its uses can be seen in other areas, like in game development where new levels must be generated effectively whilst still making the level fun. Or in children’s development where it can help build cognitive function.

My goal is to create a program which allows the user to generate a maze whilst giving them the ability to customise the maze. They will be able to pick which generation / solving algorithm they’d like and watch the maze be generated / solved with an animation. The maze and solved maze will be available to download. Statistics will also be available to see, this would include statistics like generation / solving speed and dead-end counter.

## Research

User Interface

### Current Solutions

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | [www.mazegenerator.net](http://www.mazegenerator.net) is a very popular maze generation website, their user interface (UI) is simplistic with minimal colour. They have a range of customisation options with help button if the user doesn’t understand what each option does.  They allow you to view the maze unsolved or solved.  I would add a choice to see an animation of the maze being solved to be able to visualise how the algorithm does it. Building on the earlier point, a way to choose what generation and what solving algorithm to use.  This is a web application which means it can be accessed from any device that has an internet connection and a modern browser. However, if the user does not have an internet connection, they will not be able to access the service.   |  |  | | --- | --- | | **Pros** | **Cons** | | It’s a web application | Doesn’t explain what each option does | | Explains what input they expect from user | No generation / solving visualisation | | Range of options to alter the maze | No colour customisation | |  | Can’t choose what generation and solving algorithm to use | |  | There is no statistics for how long it takes to generate / solve. | |
|  | [keesiemeijer.github.io/maze-generator/](http://keesiemeijer.github.io/maze-generator/) is another web application made by the GitHub user ‘keesiemeijer’. It has limited customisation options but some more unique. Like being able to change the colour of the: background, maze and solver.  This suffers from the same issues as [www.mazegenerator.net](http://www.mazegenerator.net) with no way to pick the solving and generation algorithms and no animations.   |  |  | | --- | --- | | **Pros** | **Cons** | | It’s a web application | Doesn’t explain what input they expect form user | | Colour customisation for the background, maze walls and solve | No generation / solving visualisation | | Ability to remove dead ends | Can’t choose what generation and solving algorithm to use | |  | No generation / solving visualisation | |  | There is no statistics for how long it takes to generate / solve. | |

### Problems with Current Solutions

1. No way to pick the type of generation and the type of solving algorithm.
2. No way of visualising how the algorithm generates or solves the maze.
3. All web-based applications so no way of accessing without an internet connection.

#### Fixing Problems with Current Solutions

1. I will include multiple generation and solving algorithms.
2. The user will be able to choose whether they would like to see the maze being generated/solved.
3. I will create a desktop application which will eliminate the need for an internet connection.

#### Customisation Options

|  |  |  |
| --- | --- | --- |
| **Name** | **Options** | **Description** |
| Image Input | Hello.jpeg, Upload your own photo | This will create a maze inside the image selected.  If this option is selected it will make the and the Width and Height option = the Width and Height of the input image. |
| Width | Integer number between 2 - 200 | This will control the width of a rectangular maze |
| Height | Integer number between 2 - 200 | This will control the height of a rectangular maze |
| Generation Algorithm | Randomised backtracker, Wave function collapse, Randomised Prims algorithm | This will control how the maze will be generated |
| Solving Algorithm | Dijkstra’s algorithm, Breadth first search | This will control how the maze will be solved |
| Dead end remover | Rational number between 0 - 1 | This will control how many dead ends the maze will have |
| Maze entry | None, Top – Bottom, Diagonal, Right – Left | This will control where the entry and exit points are for the maze |
| Background colour | Any RGB value or Hexadecimal equivalent | This will control the colour behind the maze |
| Maze colour | Any RGB value or Hexadecimal equivalent | This will control the colour of the maze walls |
| Solve colour | Any RGB value or Hexadecimal equivalent | This will control the colour of the path the solving algorithm will take |

### Image Input

The aim to be able to take and image input and generate a maze as our output.

#### Objectives

##### Taking the Image Input

Once the user selects the ‘Upload your own photo’ option, a file explorer window will open and they will navigate to the file directory of the image. That image will then be copied to the dedicated storage of the program.

###### Why are we uploading the photo to dedicated storage?

This will allow the user to reselect the image after closing the software as it will already be stored locally. Also, the user will then be allowed to move, alter, and rename the original photo without causing any issues with the software.

##### Converting the Image to .JPEG

I will use bit manipulation to alter the value of each bit within the image.

###### Why .JPEG over other file types?

One of the reasons we use .JPEG because it’s a raster image file [Figure 1.0]. A raster image file type displays static images where each pixel will have their own colour, position and proportion based on their resolution. This makes raster files easy to analyse as each grid location has a pixel assigned. This is useful for 2 reasons: it is easier to alter the colour of pixels to black or white, and the structure of our maze is similar to the structure of a raster file (one maze grid location has a cell assigned), therefore there won’t have to be any extra manipulation to map a maze cell to a pixel.

|  |  |
| --- | --- |
|  |  |
| Figure 1.0 | Figure 1.1 |
| Raster Image | Vector Image |

Another reason we use .JPEG over other raster image files is because it’s widely used over the internet, so when a user is uploading an image, that has been downloaded from the internet, chances are it’s going to be a .JPEG file and there won’t be any need to convert the image. Additionally, .JPEG will have a lower file size over other raster image types like .PNG due to the type of compression the image undergoes [Figure 1.3].

|  |  |
| --- | --- |
|  |  |
| Figure 1.2 | Figure 1.3 |
| .PNG Image (100% file size) | .JPEG Image (51% file size) |

##### Turning the Image into Black and White Pixels Only

Firstly, we will have to turn the input image into grayscale and then alter the threshold of black and white pixels.

###### Why are we turning the image into grayscale in the first place?

Grayscale is a collection of monochromatic shades of grey, where pure white is on the brightest end and pure black is on the darkest end. Grayscale only holds information about a pixel's brightness but not its colour - this is important as we want the dimmer pixels to be our maze border whilst the brighter pixels will become maze cells.

Turning the image into grayscale:

A pixel’s colour is determined by its RGB value. RGB is an additive colour model which means it takes a pixel’s red component, blue component and green component and adds them together to accurately recreate the colour of the pixel. [Figure 1.4]

|  |  |
| --- | --- |
|  |  |
| Figure 1.4 | |
| Original Image | Original’s Image Red, Green, Blue Components |

We can use a pixel’s RGB value to determine its brightness and then map this pixel to the grayscale spectrum. The equation to find a pixel’s brightness is:

*ΡL= (r \* 0.2126)γ +(g \* 0.7152)γ +(b \* 0.0722)γ*

Where: *ΡL* = Pixel luminosity, *r, g, b* = Red component, green component, blue component, *γ* = image gamma.

The gamma value will help control how bright or how dark each grayscale output will be. Lower gamma = dimmer output and vice versa.

The constants: 0.2126, 0.7152, 0.0722 are there to even the luminosity distribution from each component. Different components affect luminosity more due to how sensitive the human eye is to each colour. Humans are more sensitive to the colour green (see for yourself in [Figure 1.4]). To create the grayscale effect accurately we must give the green component a higher weighting than the rest, this is called evening the distribution.

Altering the gradient threshold:

The grayscale spectrum is like a gradient from black to white. Just like a gradient, we can change the threshold of when it begins to fade and ends to fade. This will be possible by setting up a threshold (in this case its 125 [Figure 1.6). If a pixel’s luminosity is >125 it will be white and be a maze cell, if a pixel’s luminosity is ≤125 it will be black and will become a maze border.

|  |  |
| --- | --- |
|  |  |
| Figure 1.5 | Figure 1.6 |
| Normal Grayscale Gradient | Grayscale Gradient with Altered Threshold |

##### Masking the Image

Masking is when we turn our black and white image into our maze [Figure 1.7]. We will use a Run Length Encoding (RLE) algorithm which is a form of image compression to help us map our maze faster. All this data will be stored within a 2D array and from here we can let the maze generation algorithm (that the user has chosen) to create the maze within the image.

|  |  |
| --- | --- |
|  |  |
| Figure 1.7 | |
| User Image as a Mask | User Image as a Maze |

###### Why are we using Run Length Encoding?

After compression, RLE will allow us to see how many pixels of the same colour are next to each other. This will allow us to easily map the correct number of pixels to the correct cell type. This is a more efficient method than just checking each pixel one by one.

RLE Encoding Algorithm:

We will start from the top left of our image. We will check each pixel, going left to right. If the next cell has the same colour as the cell before it, we will increment our counter. This will happen until we reach a pixel with a different colour.

The data is stored as NΡc, where N is the value stored in our counter and Ρc is the pixels colour.

|  |  |
| --- | --- |
|  |  |
| Figure 1.8 | Figure 1.9 |
| A Row of a Maze | The Same Row after RLE Compression |

### Width / Height

The user will input an integer that is between the 2-1000. This will decide the width / height of the maze. +1 = an extra maze cell added to the width / height.

###### Why between 2-1000?

We are using 2 as our lowest value because for a maze to be a maze there must be a start cell and an end cell, therefore the smallest number of cells the width / height can be is 2

We are using 1000 as our highest value because we don’t want the maze to be so big it takes too long to generate and solve the maze. Also, if the user would like the download the maze, we don’t want the file size to be too big.

### Generation Algorithm

#### DFS Backtracker

A random point on the maze will be chosen as the start cell and that cell will be recorded into a backtracking-stack. From the start cell we will check any non-visited cells that are: left, right, above or below it (if the maze border is in any of those positions, we will ignore it). At random, we will pick an unvisited cell to enter, and we will record this cell into our backtracking-stack. This process will carry on until the cell has no unvisited cells: left, right, above or below it, we will then go through our backtracking-stack until we find a cell that has an unvisited cell: left, right, above or below it. This algorithm will finish when all cells are visited, and our backtracking stack will have no cells in it.

|  |  |
| --- | --- |
|  |  |
| Figure 2.0 | Figure 2.1 |
| The maze at the beginning of the algorithm where only the source node is recorded in our backtracking-stack | The maze before being fully completed and before we back track to check if all cells have been visited. |

#### Wave Function Collapse

All cells will have states (tiles) that they could be (superposition) and we must also know what cell tiles can be next to other tiles (adjacency rules). An entropy level will be given too all cells depending on how many tiles a cell could be. The cell with the lowest entropy level will be chosen to pick a tile (this will be picked out of the adjacency rules) if there are multiple cells with the same lowest entropy level one of them will be chosen at random. This information will be sent to other neighbouring cells. As we know the adjacency rules, we can cut some of the possible states neighbouring cells can be. We will then repeat this algorithm until all cell states are determined/collapsed.

|  |  |
| --- | --- |
|  |  |
| Figure 2.3 | Figure 2.4 |
| All possible states that a cell can be in and the numbers tell us what cells can be next to each other | The image generated |

#### Randomised Prim’s Algorithm

We will fill our maze with wall cells. A wall will be picked at random and marked as a path, the walls: left, right, above and below will be recorded in a wall-priority-queue (**WQ**), if one of the walls recorded is a maze border it will not be recorded. A walls’ priority within the **WQ** is given by how close it is to the maze border, the closer the lower the priority. Starting with the highest priority wall in the **WQ**, it will visit the cell opposite it. If the visited cell is a wall, we change it to a path and remove that wall from the **WQ**. If the visited cell is a path, we will keep the wall but still remove it from the **WQ**.

|  |
| --- |
|  |
| Figure 2.5 |
| This is the Prim’s algorithm running and producing a maze path |

### Solving Algorithms

Any maze generated will have to be represented as a graph. The graph can then be traversed.

#### Dijkstra’s Algorithm

For Dijkstra’s Algorithm ‘weight’ must be given to each edge of the graph, where the effective weight (**EW**) is how far each node is from the source node.

First, we must know the source node and give it an **EW** of 0 because the path to itself is equal to 0. All nodes will be set with a weight of ‘infinity’. Then it will look at all unvisited nodes that are neighbours to the previously visited node and give it an **EW** of +1. This will repeat until all nodes have been given an **EW**.; It will then choose the lowest weighted path from the source node to the terminate node.

|  |  |
| --- | --- |
|  |  |
| Figure 2.6 | Figure 2.7 |
| The maze being fully weighted | The shortest path picked |

#### Breadth-First Search

Breadth-first search is type of tree transversal which searches each level of a tree before moving on to the next.

What this means in a maze is that, at each corner the where there are 2 or more connections the search will split to search each connection.

|  |  |
| --- | --- |
|  |  |
| Figure 2.8 | Figure 2.9 |
| First group of cells with 2 connections splitting. | Second group of cells with 2 connections splitting. |

### Dead End Remover

The aim is to be able to control the number of dead ends the generated maze has.

#### Objectives

##### Count the Amount of Dead Ends

A dead end is a path cell that only has 1 walkable neighbour. Therefore, we can check each cell to see how many walkable neighbours it has. If the number of walkable neighbours is >1 then it is not a dead end. We will also store the positions of each dead end within an array (position-array).

##### Calculate the Correct Amount of Dead Ends

The user will input a rational number that is between 0-1. The equation needed to be able to correctly calculate the number of dead ends that need to be removed is:

ω*= Nd \* P*

Where: ω= Number of dead ends that need to be removed, *Nd* = The number of dead ends within the generated maze, *P* = The user input.

###### Why is the input a rational number between 0-1?

This will give us a decimal number that can be used as a percentage multiplier. This means the user is picking a percentage of dead ends to be removed. This is important as each maze will have a different number of dead ends and the user might input a number that is > than the number of dead ends that exists in the maze.

###### What if the number calculated has a decimal part?

If this occurs, the number will simply be rounded to the closest whole number.

##### Randomly Delete the Dead Ends

We will randomly pick an item within the position-array and find that cell within the maze. If a dead end is surrounded by 3 maze walls [Figure 3.0], one of the walls will be picked at random to be removed. If a dead end has maze boundaries and maze walls surrounding it [Figure 3.1], only the maze walls will be randomly chosen to be removed.

|  |  |
| --- | --- |
| \\FBK-SVR-ST01\16GuimG11$\Downloads\Untitled.png | \\FBK-SVR-ST01\16GuimG11$\Downloads\Untitled (2).png |
| Figure 3.0 | Figure 3.1 |
| A dead end that is surrounded by maze walls | A dead end is surrounded by maze cells and maze boundaries |
| Black cells are maze walls. Cyan cells are maze boundaries. Red cells are cells that have been selected to be randomly removed. White cells are cells within the position-array. | |

###### Why is the removal process random?

Randomness allows for more variation with each maze generated. Also, it will mean that the removal of dead ends will be spread out across the maze.

The only way that this process will change is if the user inputs 1 or 0. If they input 0 [Figure 3.2] then we won’t do anything to the maze as they want to remove 0% of the dead ends. If they input 1 [Figure 3.3] (100%) then we won’t randomly pick items within the position-array, we will go through one by one and removed the dead ends on the maze

|  |  |
| --- | --- |
|  |  |
| Figure 3.2 | Figure 3.3 |
| Maze When User Input = 0 | Maze When User Input = 1 |

### Maze entry

This will change where the start and end cell are positioned on the maze

#### Top – Bottom

Start cell at the top middle of the maze – end cell at the bottom middle of the maze. [Figure 3.4]

#### Diagonal

Start cell at the top left of the maze – end cell at the bottom right of the maze. [Figure 3.5]

#### Right – Left

Start cell at the right middle – end cell at the left middle of the maze [Figure 3.6]

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Figure 3.4 | Figure 3.5 | Figure 3.6 |
| Top – Bottom Entry | Diagonal Entry | Right – Left Entry |

### Background / Maze / Solve colour

The user will input a RGB value or HEX equivalent.

###### How to convert RGB values to HEX?

Each component in the RGB model is represented by a number that goes up to 255 (i.e white = (255,255,255)). Each components deanery value can be converted to hexadecimal. For example, the colour purple can be represented in deanery = (255,0,255) and the HEX equivalent = FF00FF.

#### Background Colour

This will change the background colour of the maze. [Figure 3.7]

#### Maze Colour

This will change the colour of the maze walls and maze boundaries. [Figure 3.8]

#### Solve Colour

Once the solving algorithm is complete, the shortest path calculated will be highlighted in this colour. [Figure 3.9]

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | Figure 3.7 | Figure 3.8 | Figure 3.9 |
| A Maze with No Colour Customisation | A Maze with a Changed Background Colour | A Maze with a Changed Maze Colour | A Solved Maze with a Changed Solve Colour |

## End-user

My end-user will be Jonathan Kenworthy. He is a first-year computer science student at the University of Winchester.

## Interview with end user

I’ve met with my end-user in person and conducted this interview. I showed Mr Kenworthy my research and the current solutions are available.

This was conducted on the 27/03/23.

### Questions & Answers

#### 1. What real world application would the software have?

When it comes to architectural design, you can use this software to create eye-catching layouts for both building interiors and city streets. In the video game world, it helps create levels that change every time you play, making it more fun and repayable. Plus, for AI research, random mazes make great testing grounds for self-driving systems.

#### 2. What are the main requirements do you expect from a maze generator?

What I'm looking for is: being able to make and solve a maze, change how big the maze is, switch up its appearance, and change the entry and exit points. These are the basic things the software should be able to do.

#### 3. As a user, what would be the most intuitive way to layout the user-interface?

It'd be intuitive to have maze customization options in a column on one side of the UI, and then put the maze interaction options near where the actual maze image shows up.

#### 4. From the current solutions that are available, are there any features that I should include?

Current solutions meet the main requirments [discussed before], but they’re missing some features that would be nice to have. Adding maze animations and different algorithm options would be great and could it more useful in real-life situations.

#### 5. How many solving and generation algorithms would you expect to be included?

3 generation algorithms and 3 solving algorithms.

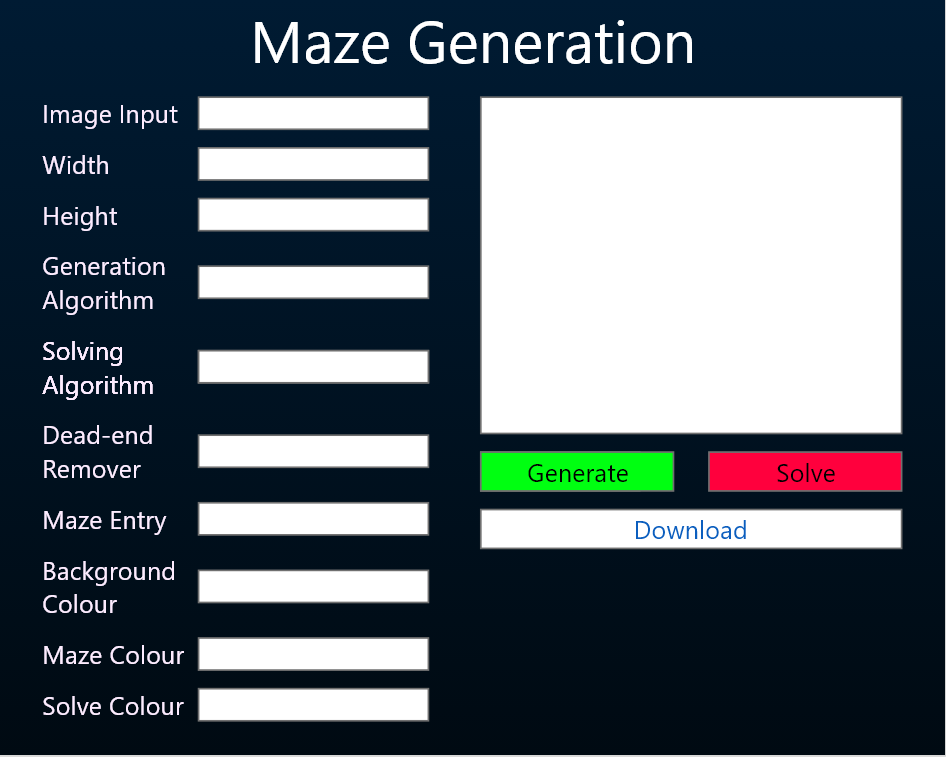
#### 6. Are there any algorithm you expect to be included?

I feel Randomised Depth First Search (DFS Backtracker) is a go-to for maze generation. And when it comes to maze solving, Dijkstra’s is probably the first algorithm that pops into my head.

## Prototype

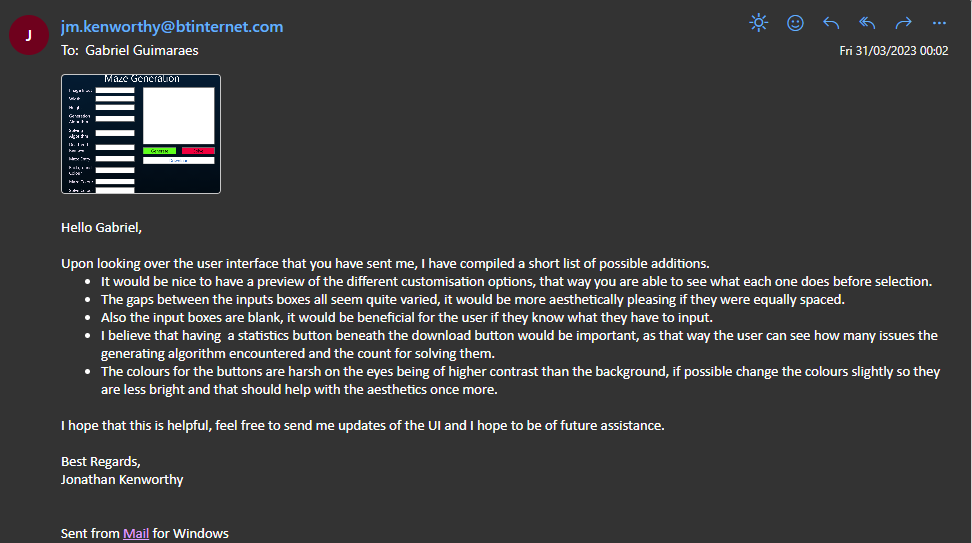
My prototype will be an image of how my user interface will look like. I considered information from my own research and the interview that was conducted with Mr Kenworthy.

I made sure to use the UI layout suggested by Mr Kenworthy and I made sure to have the download button easy to access to allow the maze image to be exported to be used real-world applications.





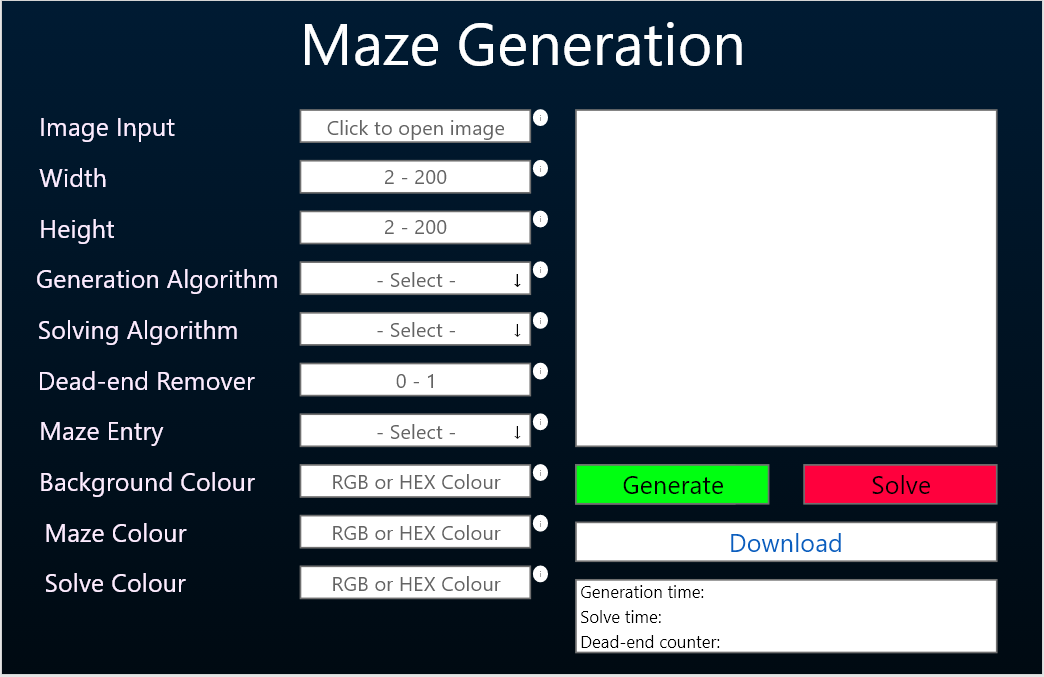
## Feedback from end-user



### Requirements from end-user

|  |  |
| --- | --- |
| **Feedback Point** | **What I’m going to do to improve** |
| Add a way for the user to see what each customisation option does. | Add a small (i) that when hovered will bring up a box that explains what each option does |
| Create equal spacing between each input box. | Add more horizontal space so all the option names will be flat. |
| Add what type the input should be. | Inside of the input box have grey text that explains what input it requires. |
| Add statistics section below download button. | Add a statistics box below the download button |

### New prototype

Requirements and Limitations

The minimum requirement for my program to be classed as successful is if it can generate and maze and solve the generated maze, in a reasonable time span.

Some limitations would be the user's memory speed. As some mazes could be quite large. If the user does not upload an image, their computer should be able to process the maze as the average user has DDR4 RAM that runs at 2666 MHz. However, if the user uploads a large photo in the ‘Image Input’ section it could slow down the average computer.

## Objectives

### Primary Objectives

#### A - Create a maze grid

1. Represent maze as a 2d array of cell objects.
2. Based on user input for width and height, create a 2D array to represent the maze.
3. Make sure that the dimensions are within the limits such as (55,89) or (7,99)
4. Initalise cell objects within the 2D array.
   1. Loop though each element of the array and create new instances of the cell object to represent the corresponding position in the maze.
   2. Assign the correct properties to each object. Such as wall values and coordinates.

#### B - Generate a maze

1. Implement the Randomised DFS algorithm for maze generation.
2. Code the algorithim to work with the maze array.
3. Show the generated maze in the maze display box.
4. Use the Visual Basic Graphics library to draw the maze correctly.
5. Update the maze display to show the correctly draw maze.

#### C - Solve a maze

1. Use Dijkstra’s algorithm to solve the generated maze and create the shortest path.
2. Code the algorithim to work with the maze array and calculate the shortest path from the entry to the start.
3. Show the path of the solved maze in the maze display box.
4. Use the Visual Basic Graphics library to highlight the shortest path found.
5. Update the maze display to show the solved path clearly.

#### D - Customise the maze colours

1. Take the colours the user has inputted.
2. Provide a user interface to select the desired colours for the maze.
3. Update the colour accordingly.
4. Use the Visual Basic Graphics library to apply the selected colours to the background, maze walls, and solved path.

#### E - Download the maze

1. Ensure the maze is generated or solved before downloading.
2. Check if the maze is generated or solved before allowing the download process to start.
3. Have the user to input the desired file location for the image to download.
4. Use the file browser dialog to allow the user to choose the destination folder.
5. Convert maze the maze to an image.
6. Create a new image with the same dimensions as the maze.
7. Match the colour of each pixel in the image to the corresponding colour in the maze image.

### Secondary Objectives

#### F - Animate the maze

1. Create an image of the maze
2. Using the Visual Basic Graphics library to draw the maze
3. Update the display as the maze changes
4. Refresh the picture box that is used to display the maze when there is a change in the maze such as when a wall breaks or a solve path has been found.

#### G - Implement more generation algorithms

1. Implement additional generation algorthims: Wave Function Collapse,Randomised Prims, Kurskals, Aldous-Border.
   1. Incorporate these algorthims into the existing software.
2. Enable the user to choose the generation algorithm.
   1. Provide a user interface option for selecting the algorithm from the available options.

#### H - Implement more solving algorithms

1. Implement additional solving algorithms: Breadth-First Search, A\*, Wall Follower (Right-Hand & Left-Hand Rule).
2. Incorporate these algorithms into the existing software.
3. Enable the user to choose the solving algorithm.
4. Provide a user interface option for selecting the algorithm from the available options.

#### I - Change the maze entry and exit

1. Use the user input to determine where the start and the end cells are on the maze.
2. Mark the start and end cells within the 2d array.
3. Use the Visual Basic Graphics library to draw the start and end cells in green and red respectively.

#### J - Remove dead ends

1. Remove the correct number of dead ends using user percentage.
2. Determine the appropriate number of dead ends to remove.
3. Calculate the total number of dead ends within the maze.
4. Randomly remove dead ends throughout the maze.

#### K - Make an aesthetic an efficient GUI

1. Develop a user-friendly interface with clear measurements.
2. Input boxes should indicate the expected format or example input (e.g., “Enter a number between 1 and 100”).
3. Buttons should have descriptive text labels (e.g., “Generate Maze” instead of “Start”).
4. Allocate a minimum display area size for the maze (e.g., 500 x 500 pixels).
5. Display tooltips or help text explaining the effect of each input parameter on the maze (e.g., “Increasing the value will result in a more complex maze.”).
6. Implement visual feedback for input box interactions, such as changing the background color to a 20% darker shade when hovered or selected.
7. Design a good-looking user interface with specific criteria.
8. Use a consistent color palette of [40,60,86] primary colors [28, 88, 199] secondary colors, and [110, 83, 150] accent colors.
9. Apply the 70-20-10 rule to color distribution, with 70% dominant, 20% secondary, and 10% accent colors.
10. Utilize the chosen font Segeo UI with three specific font sizes: 14.5px, 21.5px, and 45px, ensuring readability and visual appeal.
11. Ensure consistent spacing between interface elements (e.g., 10 pixels between each element).
12. Optimize the user interface to minimize unnecessary processing.
13. Compress images using JPEG compression with a target quality of 75% to balance quality and file size.
14. Use elements made in the designer whenever possible to reduce the need for updates during run time.

#### L - Implement Statistics

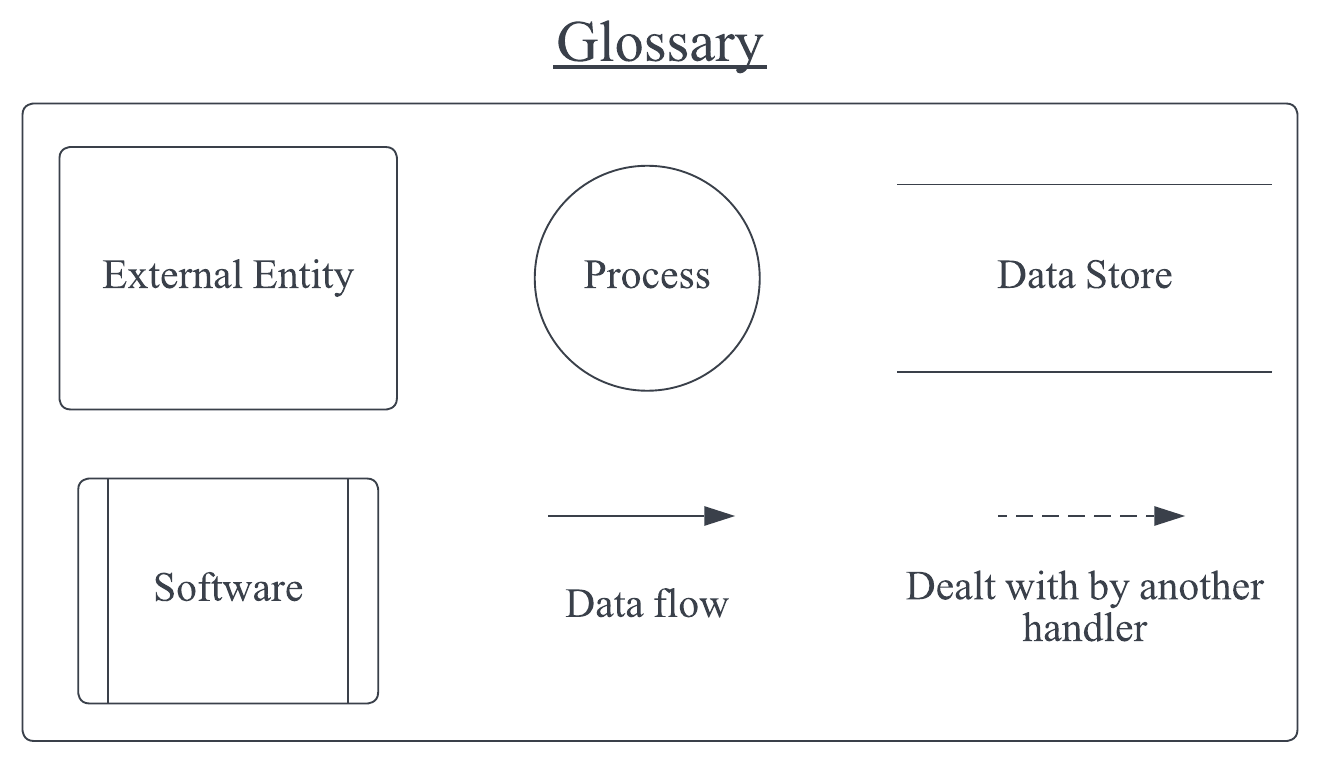
1. Record the generation time of the maze.
2. Start the timer when any generation algorithm begins.
3. Stop the timer when any generation algorithm finishes.
4. Display the recorded time after completion.
5. Record the solving time of the maze.
6. Start the timer when any solving algorithm begins.
7. Stop the timer when any solving algorithm finishes.
8. Display the recorded time after completion.
9. Count the number of dead ends.
10. Count the number of dead ends in the maze after each generation or dead-end removal process.
11. Display the number of dead ends after counting.

#### M - Be able to input an image to turn into a maze

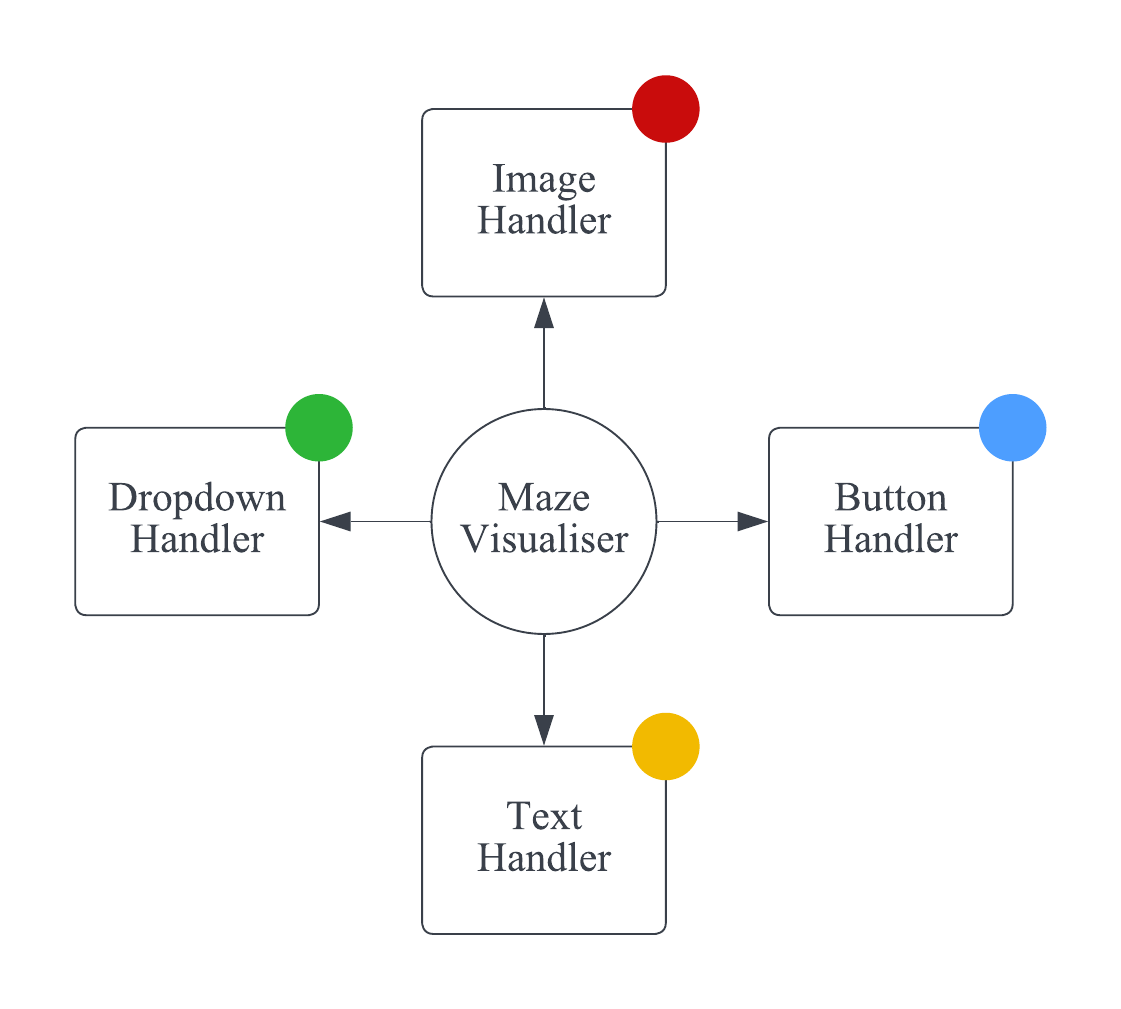
1. Accept a user's input image.
   1. Allow users to select an image file using the 'fileDirectoryDialog' class in VB.Net and store the file path.
   2. Copy the selected image file into a specific storage location within the code directory.
2. Convert the image into a grayscale format.
   1. Calculate each pixel's RGB value by iterating through the image using the Bitmap class in VB.Net.
   2. Convert the RGB value of each pixel to a luminosity value based on the researched luminosity equation.
3. Adjust the grayscale threshold to create a black and white image.
   * 1. Determine whether each pixel should be black or white based on the luminosity value: set pixels with values below 125 to black, and those with higher values to white.
4. Transform the black and white image into a maze.
5. Create an array containing the positions of all black pixels. During maze initialization, treat cell positions within this array as maze walls.

## Data Flow Diagram

### Data Flow Diagram Glossary

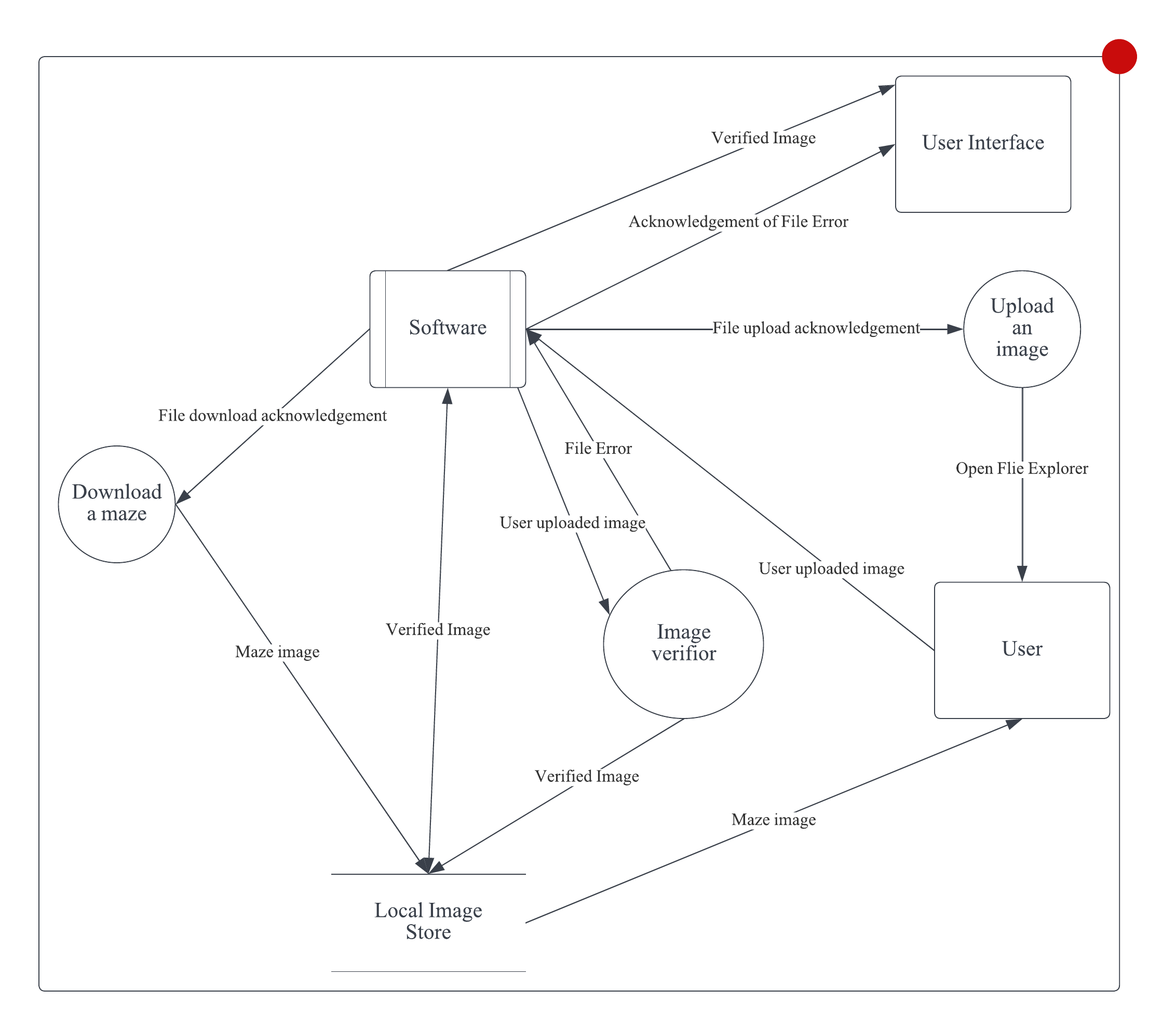


### Level 0 Data Flow Diagram

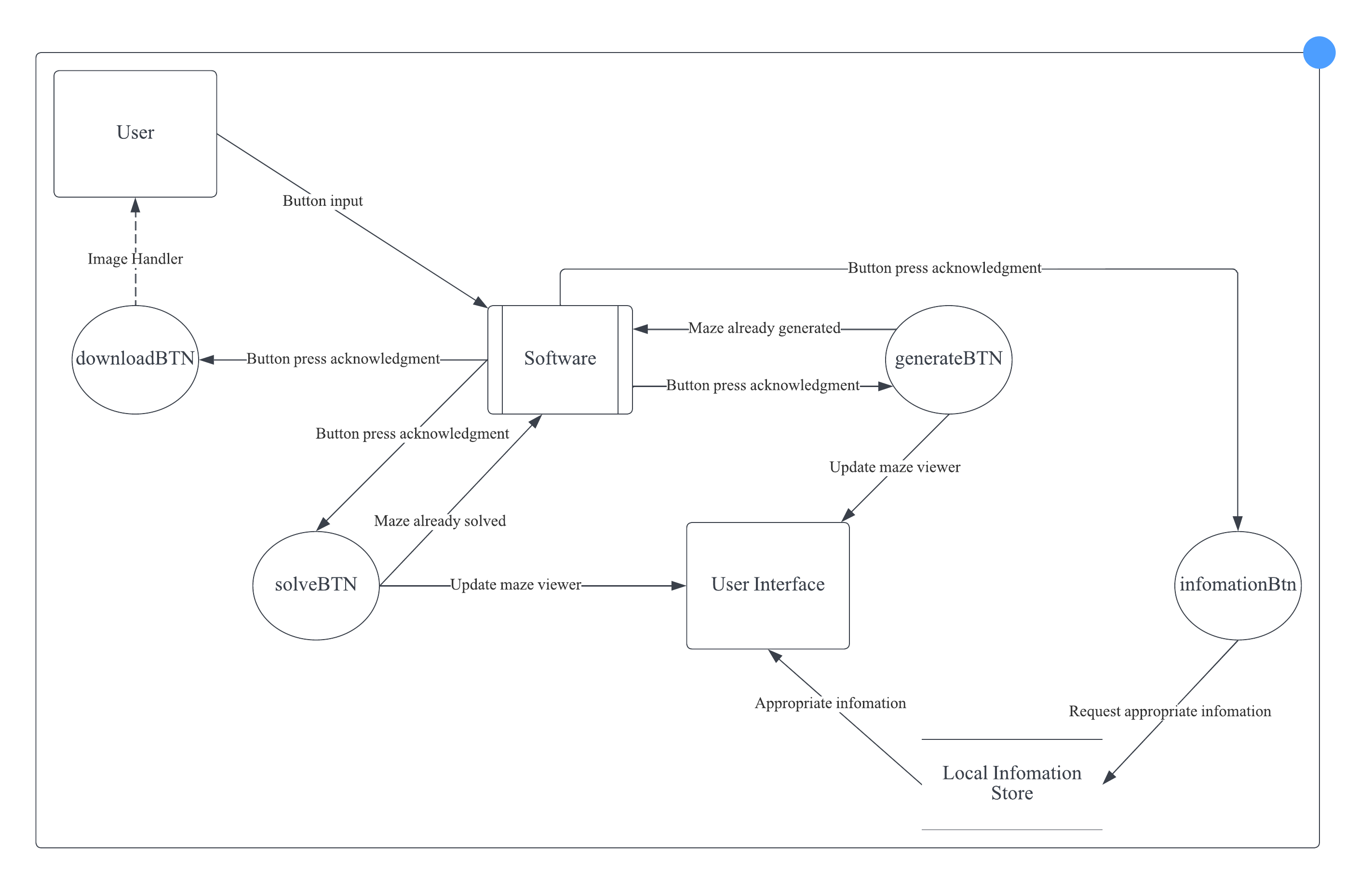


### Level 1 Data Flow Diagram

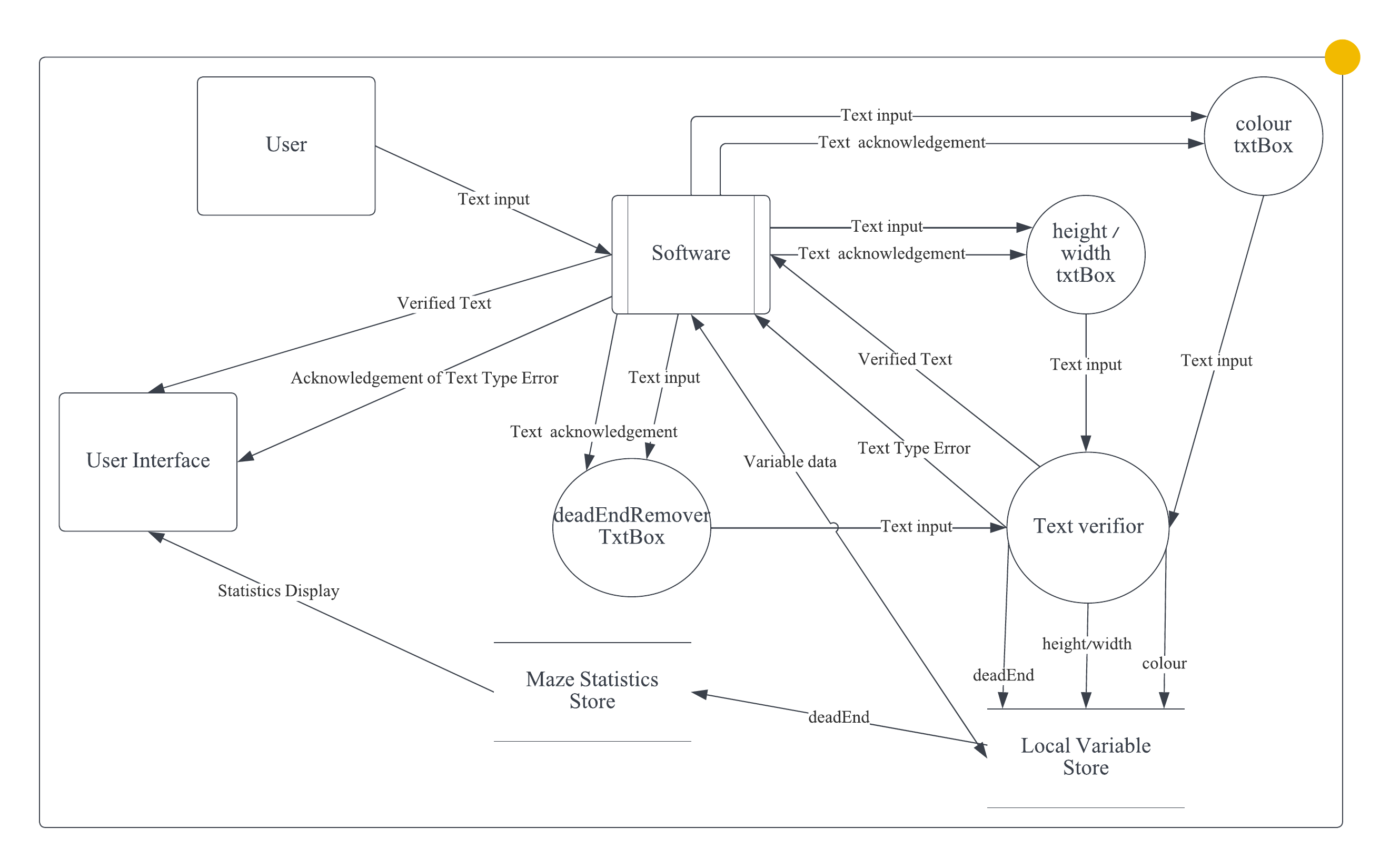
#### Image Handler



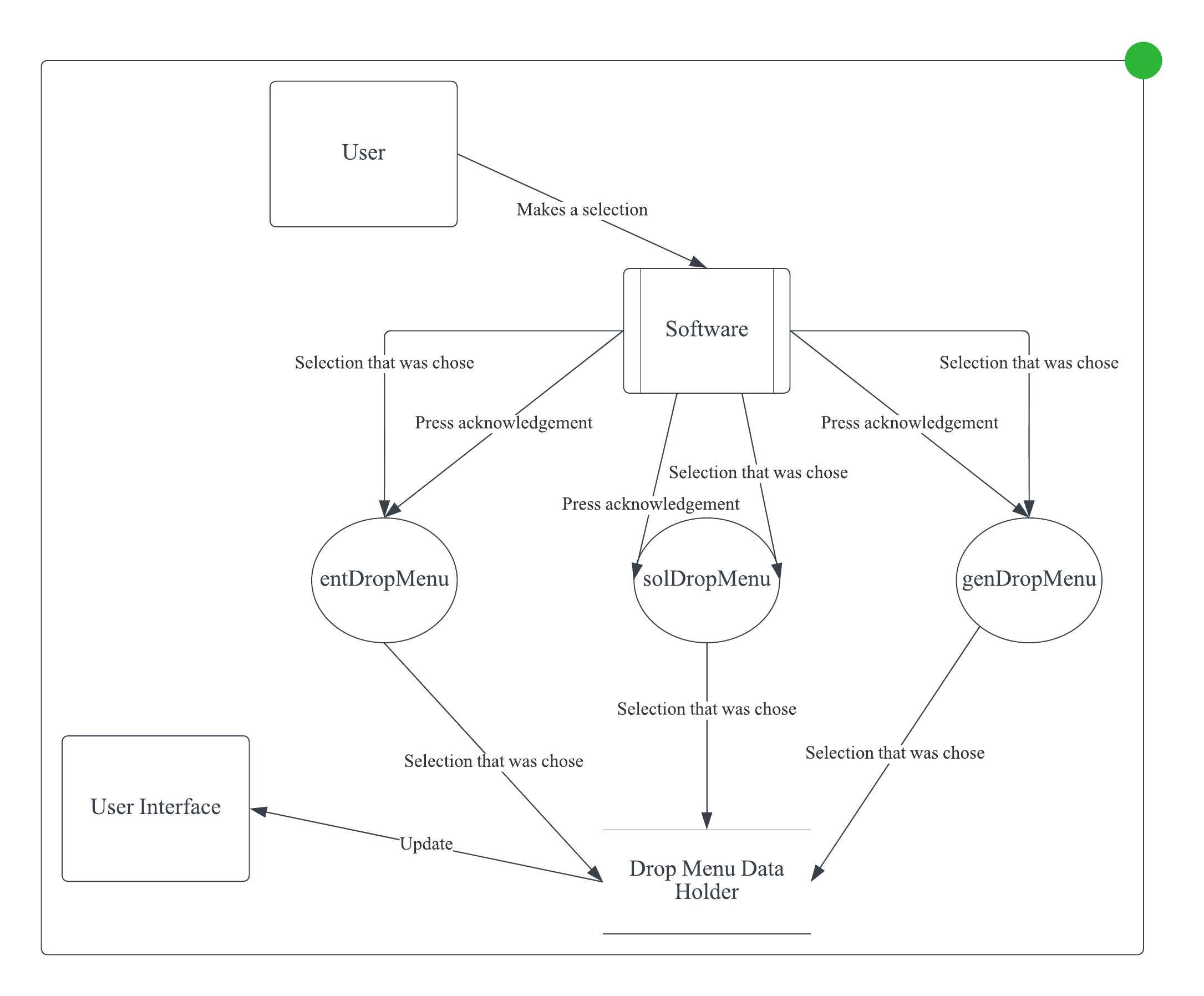
#### Button Handler



#### Text Handler



#### Dropdown Handler



## Design Data Dictionary (DDD)

### Public Variables [Maze Form]

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| rnd | Random | To randomise numbers. | N/A |
| PEN\_SIZE | Integer | Thickness of the walls. | 2 |
| m | Integer | Size multiplier. | 3 |
| maze | 2D Array of Cells | 2D Array representing the maze. | N/A |
| width | Integer | Width of the maze. | 123 |
| height | Integer | Height of the maze. | 456 |
| deadEndPercent | Double | Percentage of dead-ends that need to be removed. | 0.34 |
| mazeEntryType | String | Type of maze entry. | ‘Random’ |
| mazeEntry | Point | Position of maze entry. | (1,1) |
| mazeExit | Point | Position of maze exit. | (123,456) |
| deadEndPos | List of Points | Positions of dead-ends. | N/A |
| bgColour | Color | Colour of background. | N/A |
| mazeColour | Color | Colour of maze. | N/A |
| solveColour | Color | Colour of solved path. | N/A |
| pinkColor | Color | Colour of the start of heat path gradient. | N/A |
| purpleColor | Color | Colour of the end of heat path gradient. | N/A |
| T | Integer | Control interval between each frame of animation. | 50 |
| passedPath | List of Points | Controls what paths get highlighted in animations. Used in solving algorithms that don’t use queues. | N/A |
| solvedVisited | Queue of Points | Controls what paths get highlighted in animations. Used in solving algorithms that do use queues. | N/A |
| visitedCells | List of Cells | Controls what paths get highlighted in animations. Used in solving algorithms that don’t use queues and use instances of Cell. | N/A |
| maxWeight | Integer | Maximum weight a cell can be | 1792 |
| cancelAnimation | Boolean | Controls if user wants to stop animations. | True |
| resetType | String | Controls if user cancelled during generation or solving. | ‘G’ |
| generationAlgorithm | String | What generation algorithm the user picked. | ‘Randomised Backtracker’ |
| generationStack | Stack of Points | The cell that has been visited in the past. | N/A |
| primsWalls | List of Tuples of Points and Integers | Stores position of cells and when it got stores. Used in Randomised Prims. | N/A |
| kurskWalls | List of Tuples of Points and Integers | Stores position of cells and when it got stores. Used in Kurskal’s. | N/A |
| kursSets | Dictionary with key of Point and value of Integers | Stores the position of cells and what setID they belong too. Used in Kurskal’s. | N/A |
| kursNeighbourCoords | Point | Stores the position of the neighbours. Used in Kurskal’s. | (5,8) |
| kursCurrentWallIndex | Integer | The index of the wall. Used in Kurskal’s. | 2 |
| abTotalCells | Integer | Total cell that will be visited. Used in Aldous-Border. | 444 |
| abVisitedCells | Integer | Number of cells that have been visited. Used in Aldous-Border. | 321 |
| currentX | Integer | The x component of the current cell. | 77 |
| currentY | Integer | The y component of the current cell. | 66 |
| directions | Array of Points | The 2D vector of how a cell can move on the board. | N/A |
| abHasBeenAnimated | Boolean | Checks if Aldous-Border has been animated. | False |
| gWeights | Dictionary with key of Point and value of Doubles | Stores the position of cell and its corresponding weight. | N/A |
| branchingPoints | List of Points | Store the positions of cells that branch into sub trees. | N/A |
| solveAlgorithm | String | What solving algorithm the user picked. | “A\*” |
| mazeWallCount | Integer | Number cells that are mazeWalls. | 55 |
| totalCells | Integer | Total number of cells. | 465 |
| path | Queue of Points | Position of the solved path. | N/A |
| helperPath | Queue of Points | Position of the shortest solved path. | N/A |
| mazeImage | Bitmap | Image used to control the maze. | N/A |
| mazeImageGraphics | Graphics | Graphics interface linked the maze image. | N/A |
| downlaodGenerated | DialogResult | Result of an MSG box. | ‘OK’ |
| downlaodSolved | DialogResult | Result of an MSG box. | ‘CANCEL’ |
| deadEndToShow | Integer | Number of dead-ends that should be shown in statistics. | 238 |
| solveTimer | Stopwatch | Time to solve. | 00:00:00:74 |
| generationTimer | Stopwatch | Time to generate. | 00:00:03:40 |
| drawTimer | Stopwatch | Time to draw. | 00:00:26:32 |
| deadEndTimer | Stopwatch | Time to remove dead-ends. | 00:01:59:59 |
| mazeGenerated | Boolean | Checks if a maze is generated. | True |
| imageInputted | Boolean | Checks if an image is inputted. | False |
| inputImage | Bitmap | Stores the inputted image. | N/A |
| mazeWallList | List of Points | Position of the maze walls. | N/A |
| luminosity | Double | Luminosity of an image. | 80.91 |
| GAMMA | Double | Controls how bright the image becomes. | 1.1 |
| R | Double | Controls how much red effects luminosity. | 0.2126 |
| G | Double | Controls how much green effects luminosity. | 0.7152 |
| B | Double | Controls how much blue effects luminosity. | 0.0722 |
| imgComponents | List of List of Points | Stores the cell position of each component of an image. | N/A |
| largestComponent | List of Points | Stores the cell position of the largest component. | N/A |

### Class ‘Circular Queue’ Attributes and Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| currentIndex | Integer | Stores the current index of the data items. | 4 |
| items | List of [inputted type] | Store the data. | N/A |
| New | Protocol | Initializes items inputted into the items list. | N/A |
| turnRight | Function | Increments currentIndex and returns the item in that index. | N/A |
| turnLeft | Function | Decrements currentIndex and returns the item in that index. | N/A |

### Class ‘PriorityQueue’ Attributes and Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| dictionary | Dictionary with key of [inputted priority] and value a Queue of [inputted value] | Stores the priority and the data. | N/A |
| New | Protocol | Initializes items inputted into the dictionary. | N/A |
| Enqueue | Protocol | Adds a value in to the priority queue. | N/A |
| Dequeue | Function | Removes a value from the priority queue. And returns that value. | N/A |
| isEmpty | Function | Return a Boolean if the priority queue is empty. | N/A |
| Count | Function | Returns the number of items in the priority queue. | N/A |
| Contains | Function | Returns a Boolean if a value is within the priority queue. | N/A |

### Class ‘Cell’ Attributes and Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| x | Integer | The x position of the cell. |  |
| y | Integer | The y position of the cell. |  |
| walls | List of Boolean | Represents the 4 walls of the cell. | N/A |
| wallPos | Point | The drawing positions of each wall of the cell. | (10,100) |
| mazeWallBool | Boolean | Controls if a cell is a maze wall. | True |
| mazeEntryBool | Boolean | Controls if a cell is a entry cell. | False |
| mazeExitBool | Boolean | Controls if a cell is an exit cell. | True |
| mazeSolved | Boolean | Controls if a cell is a part of the solved path. | False |
| visited | Boolean | Controls if a cell is visited or not. | True |
| connectedCell | List of Point | Stores the connected cells to a given cell. | N/A |
| drawWalls | Protocol | Draws the walls of a cell. Used to speed up animations. | N/A |
| deadEndFinder | Protocol | Checks if it’s a dead-end, if it is then it will add it’s x, y into the deadEndPos list. | N/A |
| breakWall | Function | Breaks the wall of a cell given a direction. Changes the connectedCell list. Returns the newest cell in connectedCell. | N/A |
| checkUnvistedNeighbours | Function | Checks what cells around it is unvisited. Returns a list of boolean. | N/A |
| checkConnectedCell | Function | Checks if a cell is connected given a direction. Returns a boolean. | N/A |

### initailizeMaze

This adds all the necessary cells into the maze 2d array and adds the properties to each cell depending on their position.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| mazeWallCount | Integer | Counts the amount of maze walls there are. | PUBLIC |
| totalCells | Integer | Counts the amount of cell there are. | PUBLIC |
| posi | Integer | This value controls the i component of the drawing positions of each wall in the cell. | 10 |
| posj | Integer | This value controls the j component of the drawing positions of each wall in the cell. | 100 |

### setMazeEntryExit

This uses the user input to generate positions of the entry and exit cells.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| imageInputted | Boolean | Checks if an image is uploaded. | PUBLIC |
| largestComponent | List of Points | Stores the cell position of the largest component. | N/A |
| orderedList | List of Points | Orders the list from biggest to smallest. | N/A |
| mazeEntryType | String | Type of maze entry. | PUBLIC |
| mazeEntry | Point | This point is generated depending on the user input. This will be the entry to the maze. | PUBLIC |
| mazeExit | Point | This point is generated depending on the user input. This will be the exit to the maze. | PUBLIC |

### resetMaze

This resets the variables used in animation and finishes the animation when cancelled.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| solvedVisited | Queue of Points | Resets the queue. | N/A |
| helperPath | Queue of Points | Resets the queue. | N/A |
| path | Queue of Points | Resets the queue. | N/A |
| passedPath | List of Points | Resets the list. | N/A |
| resetType | String | Stores when the animation got cancelled. | PUBLIC |
| generationAlgorithm | String | If animation got cancelled during generation this is used to know what algorithm to use. | PUBLIC |
| solveAlgorithm | String | If animation got cancelled during solving this is used to know what algorithm to use. | PUBLIC |

### animationLock

Locks the UI that shouldn’t be in use whilst in animation.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| Lock | Boolean | To know if it will lock the UI or unlock the UI. | True |

### interpolateColour

Uses linear interpellation between 3, 3D vectors.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| colour1 | Color | The first colour we want to interoperate in between. | N/A |
| colour2 | Color | The first last colour we want to interoperate in between. | N/A |
| ratio | Double | This is the ‘step variable’. It decides how far we go from colour1 to colour2 | 0.1 |

### randomisedDFS

A generation algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| generationStack | Stack of Points | The cell that has been visited in the past. | PUBLIC` |
| direction | Integer | Controls what cell it will break into. | 3 |
| currentCell | Point | Store the position of the current cell. | (1,2) |
| unvisitedNeighbors | List of Points | Stores the position of unvisited neighbouring cells. | N/A |
| validNeigbours | List of Points | Stores the neighbours that won’t return empty. | N/A |

### randomisedPrims

A generation algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| primsWalls | List of Tuples of Points and Integers | The walls that could be the broken. | N/A |
| visitedCell | List of Points | The position of the cells that have been visited. | N/A |
| direction | Integer | Controls what cell it will break into. | 3 |
| neighbour | List of Points | Stores the position of unvisited neighbouring cells. | N/A |

### kurskals

A generation algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| kurskWalls | List of Tuples of Points and Integers | Stores the walls around a set. | N/A |
| kursSets | Dictionary with key of Point and value of Integers | Assigns a set to each cell. | N/A |
| setId | Integer | Value that controls the set number of the next group | 43 |
| direction | Integer | Controls what cell it will break into. | 3 |
| setIdToReplace | Integer | Stores the value of setID that needs to be replaced, | 32 |
| setIdToKeep | Integer | Stores the value of the setID that needs to be kept. | 31 |

### aldousBorder

A generation algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| abTotalCells | Integer | Total cell that will be visited. | 435 |
| abVisitedCells | Integer | Number of cells that have been visited. | 421 |
| currentX | Integer | The x component of the current cell. | 44 |
| currentY | Integer | The y component of the current cell. | 2 |
| newX | Integer | The x component of the new cell. | 43 |
| newY | Integer | The y component of the new cell. | 3 |

### distanceCalc

Calculates the Manhattan distance between any given points.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| a | Point | The first point | (2,5) |
| b | Point | The second point | (3,5) |

### dijkstra

A solving algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| gWeights | Dictionary with key of Point and value of Doubles | Stores the position of cell and its corresponding weight. | N/A |
| solvedVisited | Queue of Point | Stores the cells visited during runtime. | N/A |
| parents | Dictionary with key of Point and value of Point | Stores the parent-child relation of nodes. | N/A |
| pQueue | Priority Queue of Double, Point | Stores what cells need to be weighted next. | N/A |
| path | Queue of Point | Position of the solved path. | N/A |

### BFS

A solving algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| queue | Queue of Point | To store what cells, need to be checked next | N/A |
| parents | Dictionary with key of Point and value of Point | Stores the parent-child relation of nodes. | N/A |
| currentNode | Point | Store the position of the current Node. | (1,2) |
| branchingPoints | List of Points | Store the positions of cells that branch into sub trees. | N/A |

### Astar

A solving algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| gWeights | Dictionary with key of Point and value of Doubles | Stores the position of cell and its corresponding weight. | N/A |
| solvedVisited | Queue of Point | Stores the cells visited during runtime. | N/A |
| parents | Dictionary with key of Point and value of Point | Stores the parent-child relation of nodes. | N/A |
| pQueue | Priority Queue of Double, Point | Stores what cells need to be weighted next. | N/A |
| path | Queue of Point | Position of the solved path. | N/A |
| fWeights | Double | The value produced by weight + heuristic weight | 5.3 |
| heuristicWeight | Double | Is the Manhattan weight of two given cells. | 0.3 |

### wallFollower

A solving algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| node | Point | To store the position of the current node. | (1,2) |
| directionQueue | Circular Queue of Integer | To store what wall index to go to. | N/A |
| type | String | To choose if it was Right Hand Rule or Left Hand Rule. | ‘RHR’ |

### deadEndRemover

Removes a percentage of dead-ends

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| numToBeRemoved | Integer | This the number of dead-ends that needs to be removed. | 43 |
| deadEnd | Point | Stores the position of a dead-end | (5,5) |
| direction | Integer | Controls what cell it will break into. | 3 |
| removed | Integer | This is the number of dead-ends removed. | 41 |
| validIndexs | List of Integer | This store what indexes are valid to break through. | N/A |
| deadEndPos | List of Points | Positions of dead-ends. | N/A |

### generateBtn\_Click

Where the majority of the ‘baby-proofing’ is and adds the user inputs into their appropriate variables. Calls the generation functions

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| width | Integer | Width of the maze. | 123 |
| height | Integer | Height of the maze. | 456 |
| mazeGenerated | Boolean | Checks if a maze is generated. | True |
| mazeEntryType | String | Type of maze entry. | PUBLIC |
| generationAlgorithm | String | If animation got cancelled during generation this is used to know what algorithm to use. | PUBLIC |
| imageInputted | Boolean | Checks if an image is inputted. | False |

### downloadMaze

Downloads the maze

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| mazeGenerated | Boolean | Checks if a maze is generated. | True |
| openFile | DialogResult | Result of an MSG box. | ‘CANCEL’ |
| mazeImage | Bitmap | Image used to control the maze. | N/A |

### componentAnalysis

Finds the components of a black and white image

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| components | List of List of Points | Stores the cell position of each component of an image. | N/A |
| largestComponent | List of Points | Stores the cell position of the largest component. | N/A |
| visited | 2D Array of Boolean | Tracks which cells have been visted | N/A |
| generationStack | Stack of Points | Stores the pixels that need to be cheked | N/A |

### imageInputBtn\_Click

Takes in the user image and does the majority of the image processing.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| inputImage | Bitmap | Stores the inputted image. | N/A |
| newWidth | Integer | New width when border is added. | 32 |
| newHeight | Integer | New height when border is added. | 32 |
| luminosity | Double | Luminosity of an image. | 80.91 |
| GAMMA | Double | Controls how bright the image becomes. | 1.1 |
| R | Double | Controls how much red effects luminosity. | 0.2126 |
| G | Double | Controls how much green effects luminosity. | 0.7152 |
| B | Double | Controls how much blue effects luminosity. | 0.0722 |

### Public Variables [Helper Form]

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Purpose** | **Example** |
| currentIndex | Integer | Stores the current index of the image. | 4 |
| MAX\_NUMBER\_OF\_IMAGES | Integer | Controls the maximum number of images that can be uploaded. | 10 |

# Design

## System Overview

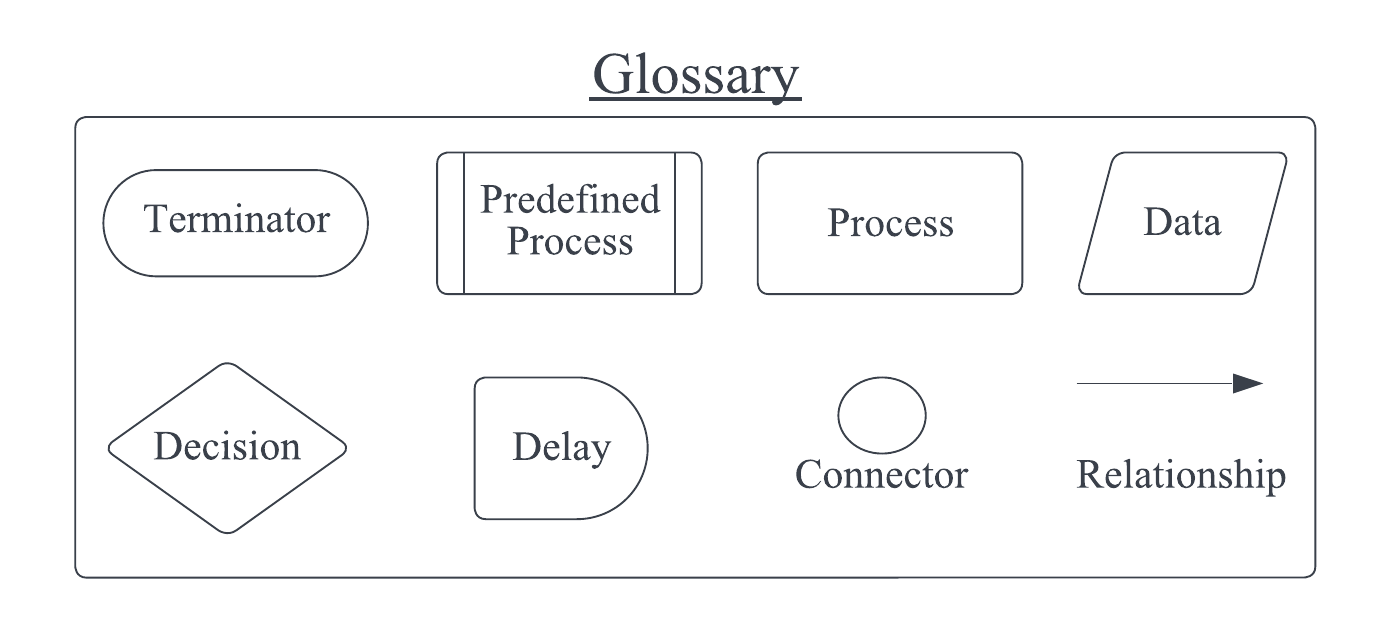
The following flow chart outlines the core functionalities that the solution will have, including image manipulation, maze entry and exit points, maze statistics, customization options, dead end removal, maze generation, and maze solving. Each of these predefined functions is further detailed below, with some more specific sub-procedures represented in pseudo-code.

The choice of VB.Net as the programming language is primarily driven by the ease of UI modification offered by Windows Forms. This allows me to prioritize the solution's performance and efficiency. The platform supports various input methods such as buttons, drop-down menus, text inputs, and toggle boxes, enabling a versatile and user-friendly experience. Furthermore, VB.Net's event handler system ensures seamless execution of specific procedures and functions in response to user interactions.

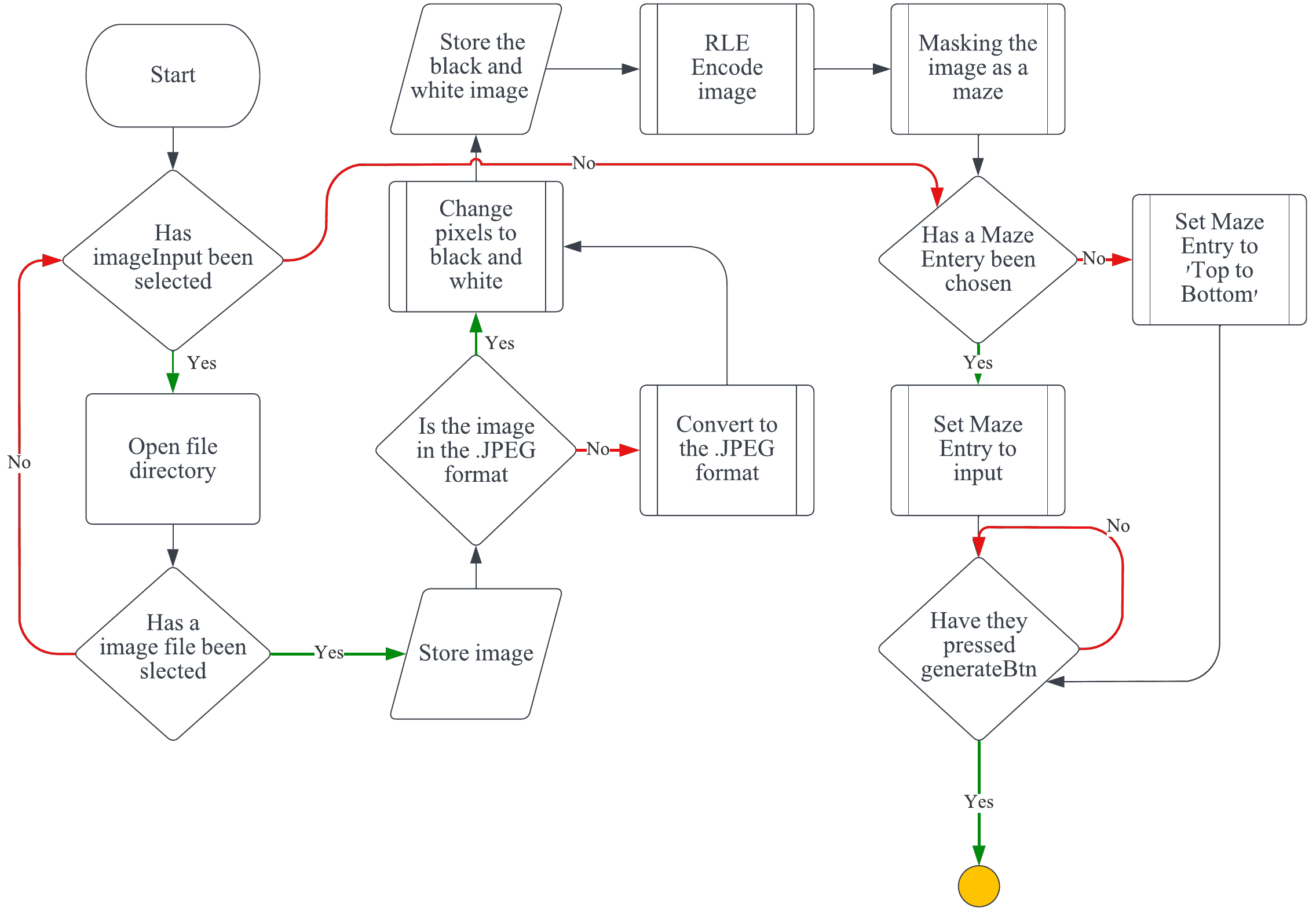
As I move forward with the development process, I'll carefully consider each function's implementation, ensuring that the final solution meets both primary and secondary objectives whilst ensuring easy usability.

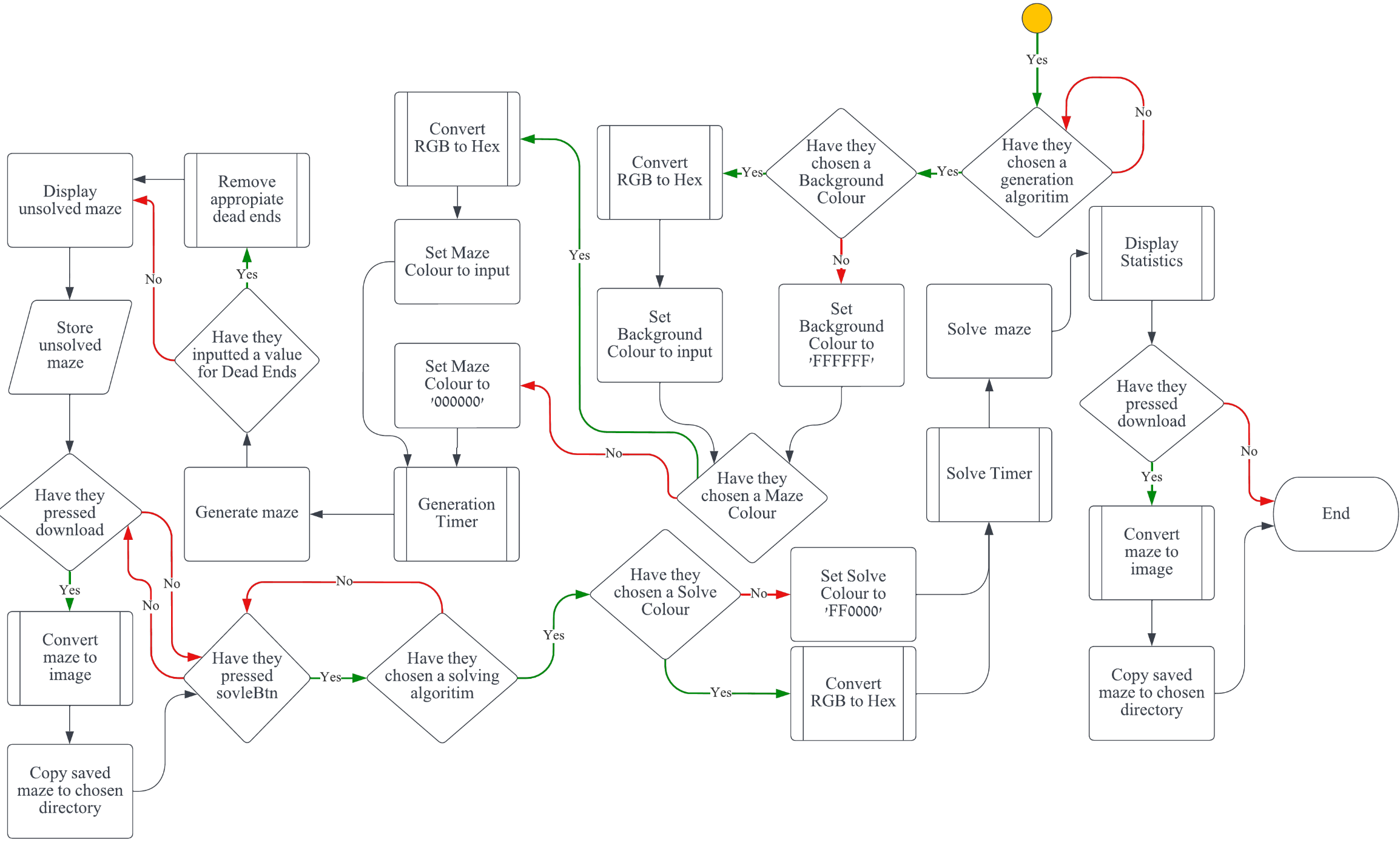
## Flow Chart

### Flow Chart Glossary



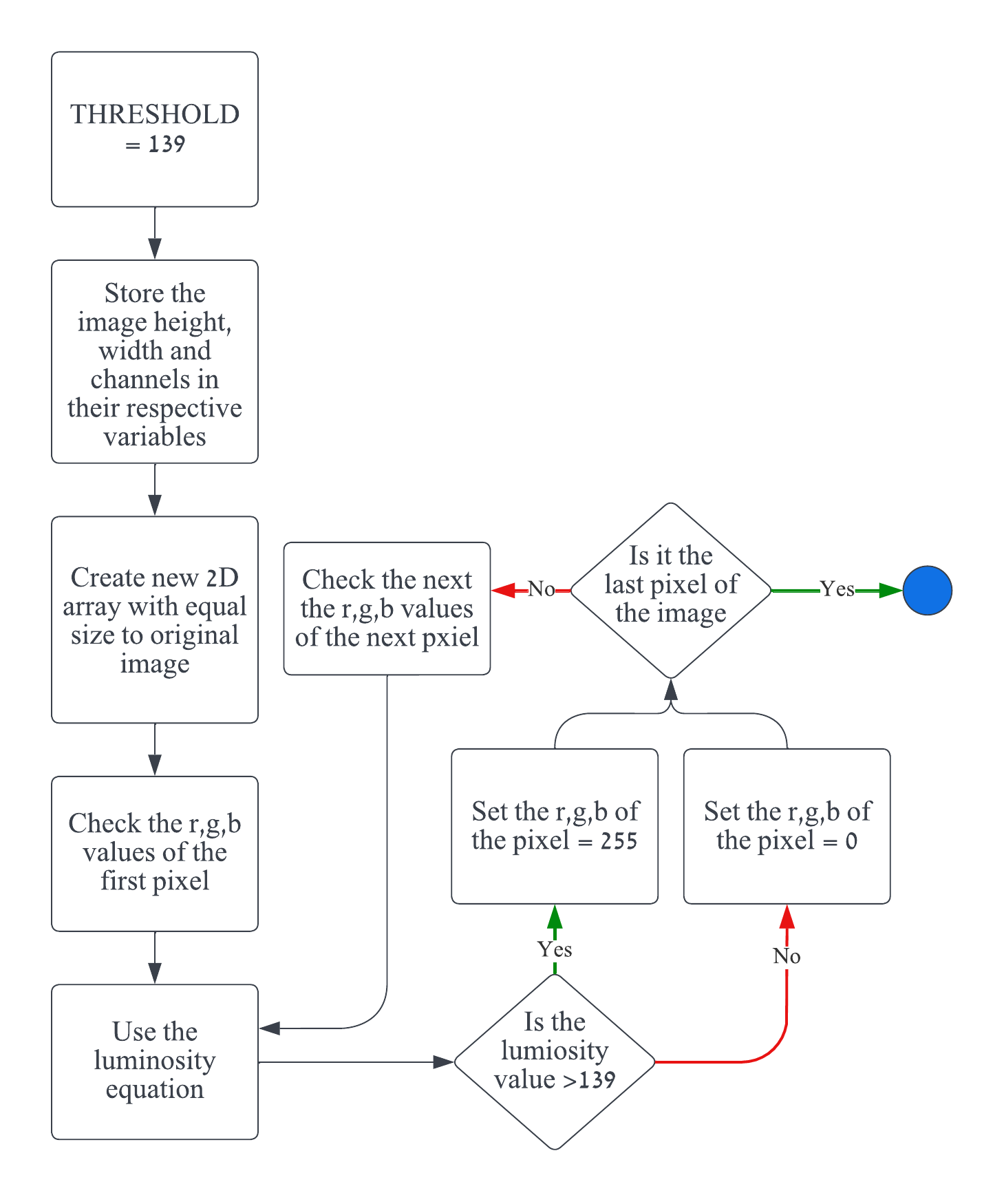
### Level 0 Flowchart



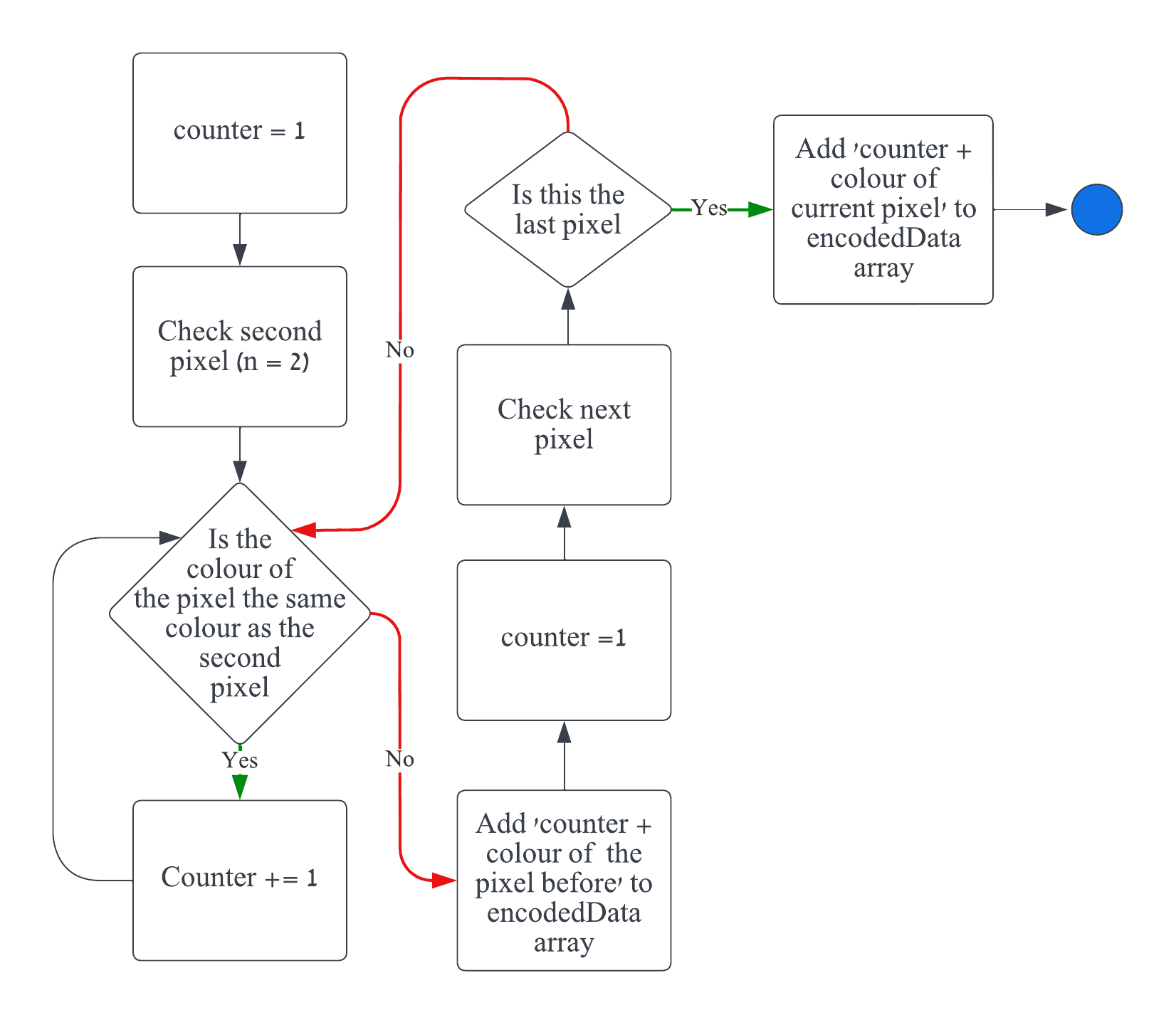


### Level 1 Flowchart

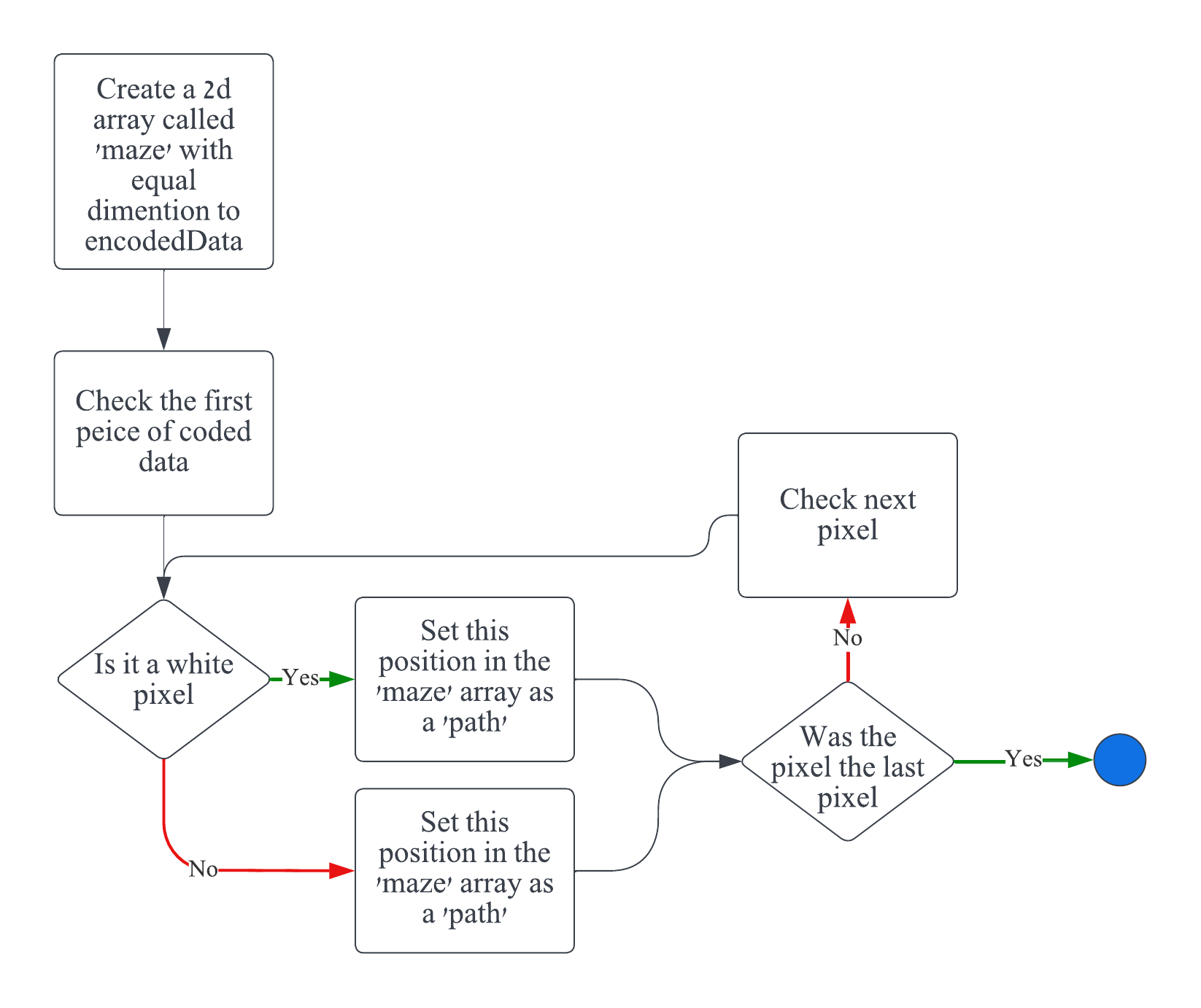
#### Change pixels to black and white



#### RLE Encode image



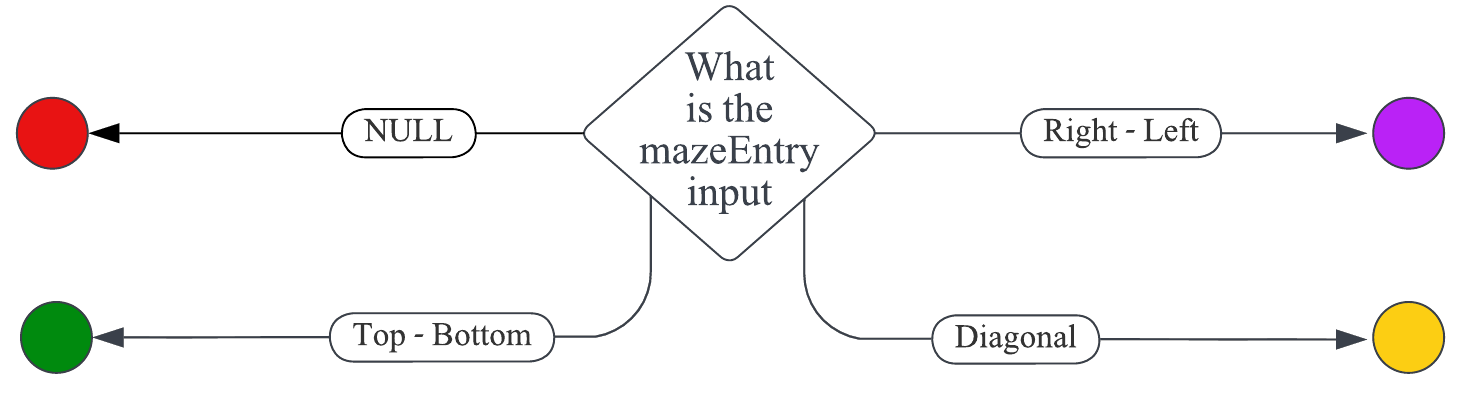
#### Making the image as a maze

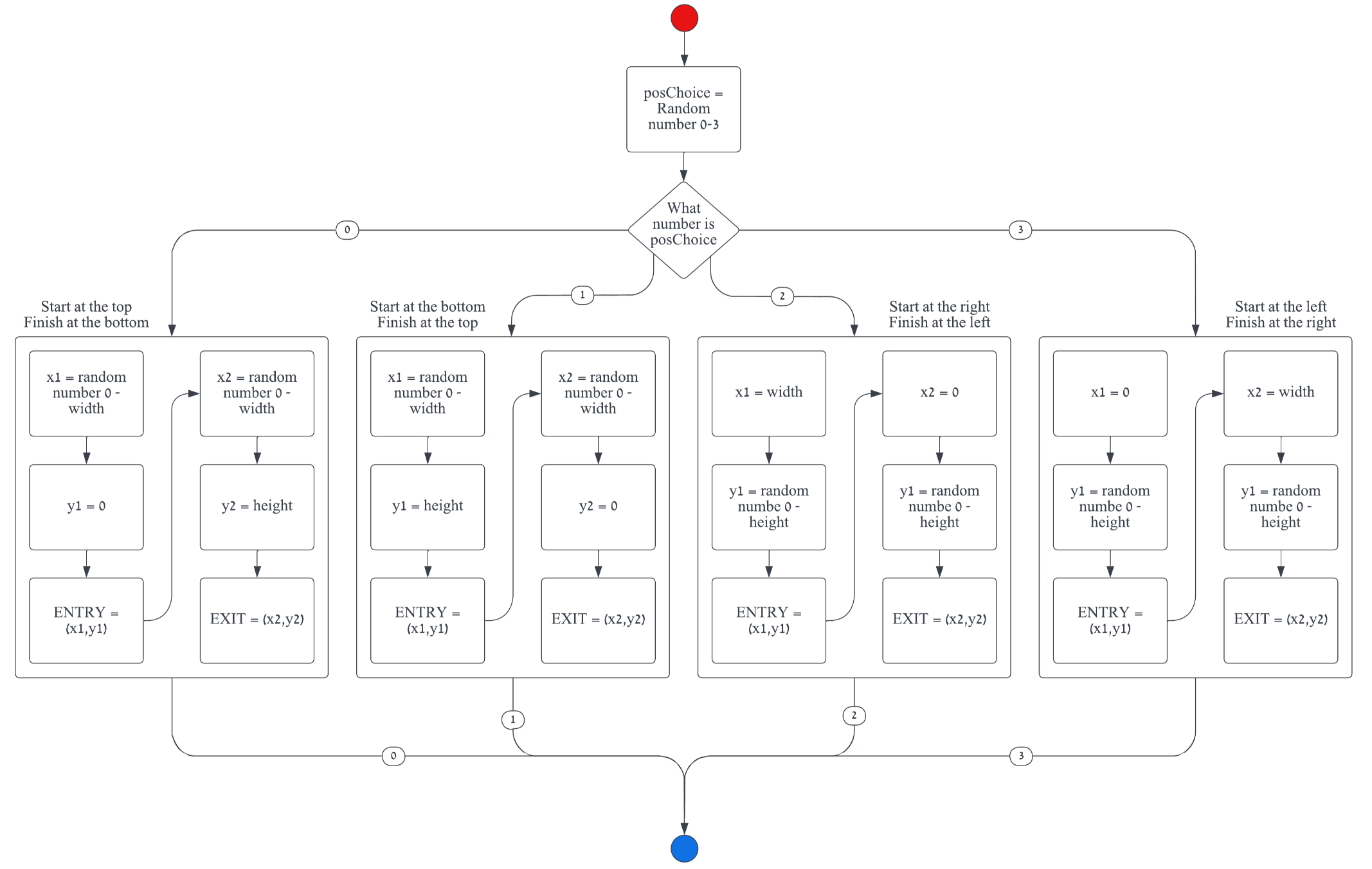


#### Set Maze Entry to ‘Top to Bottom’



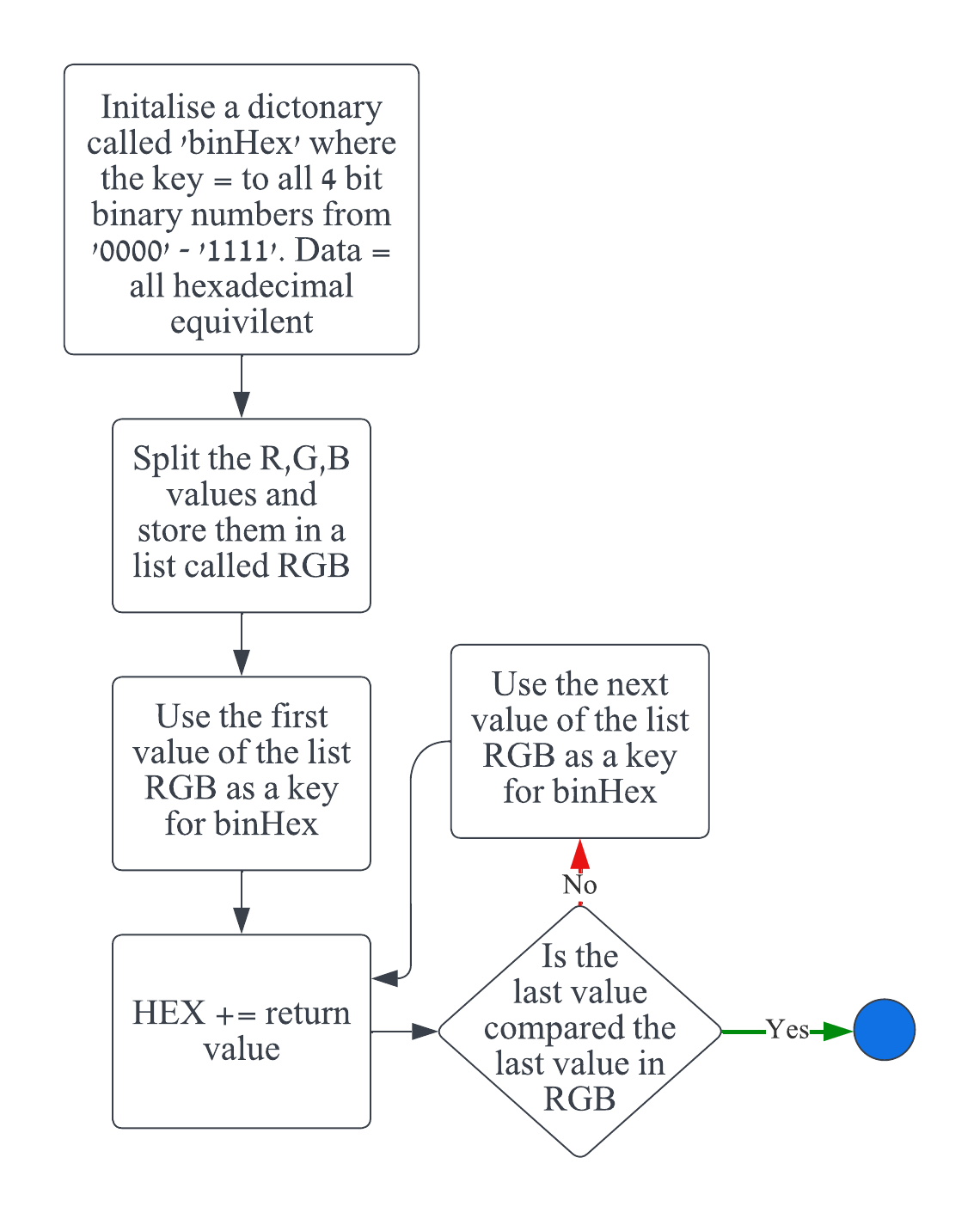
#### Set Maze Entry to input



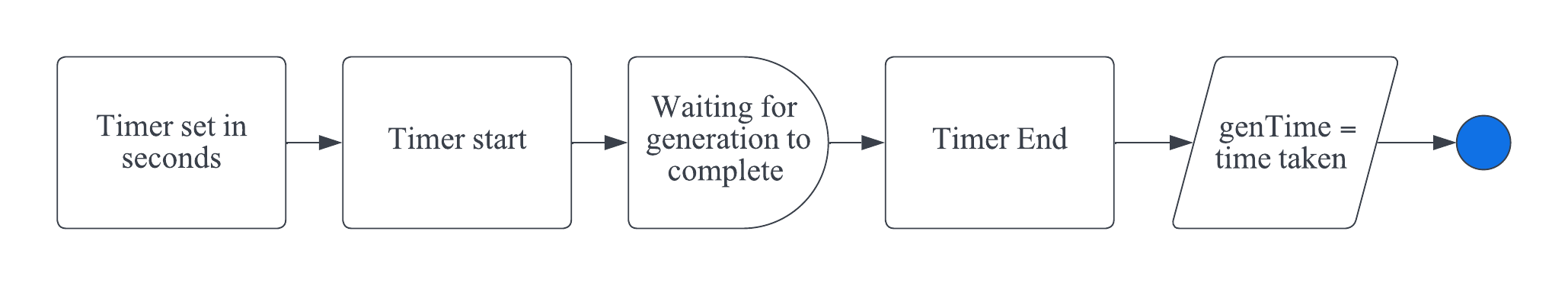


|  |  |  |
| --- | --- | --- |
|  |  |  |

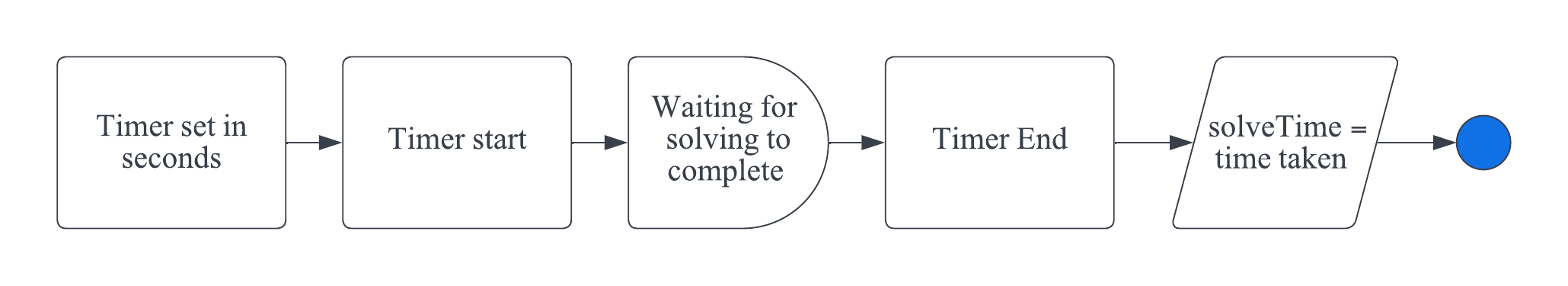
#### Convert RGB to HEX



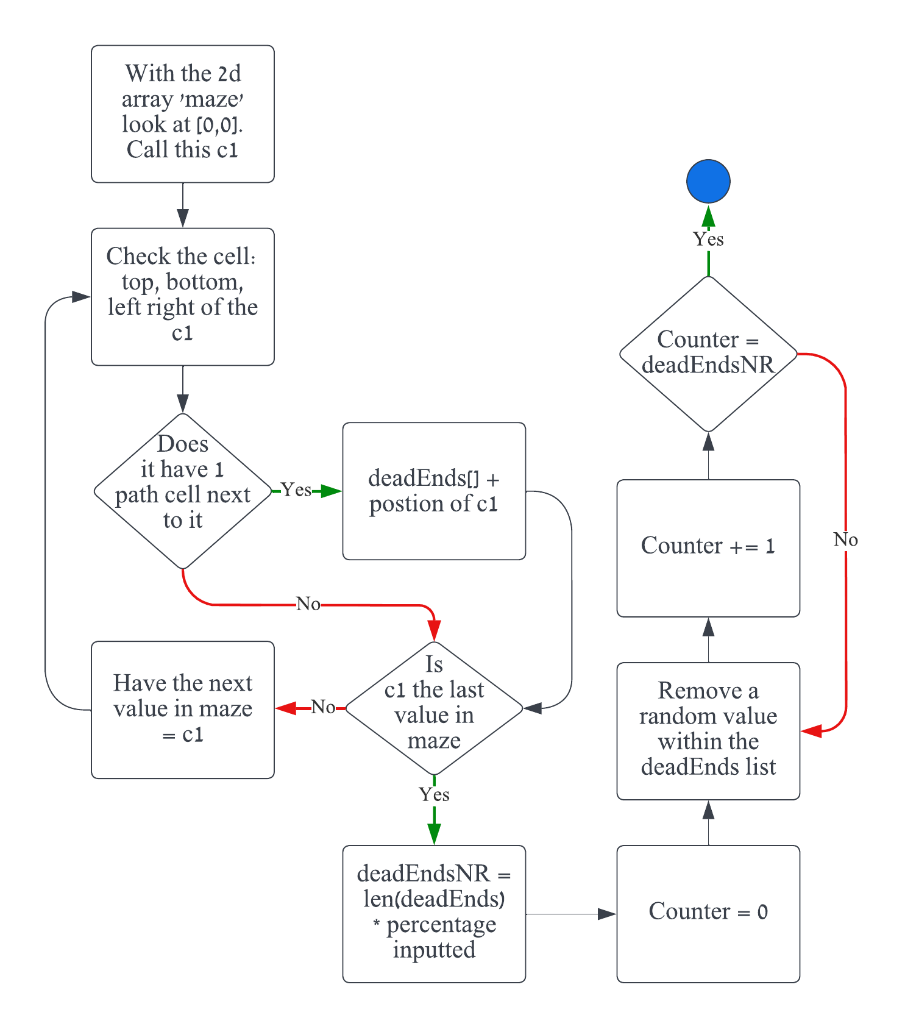
#### Generation Timer



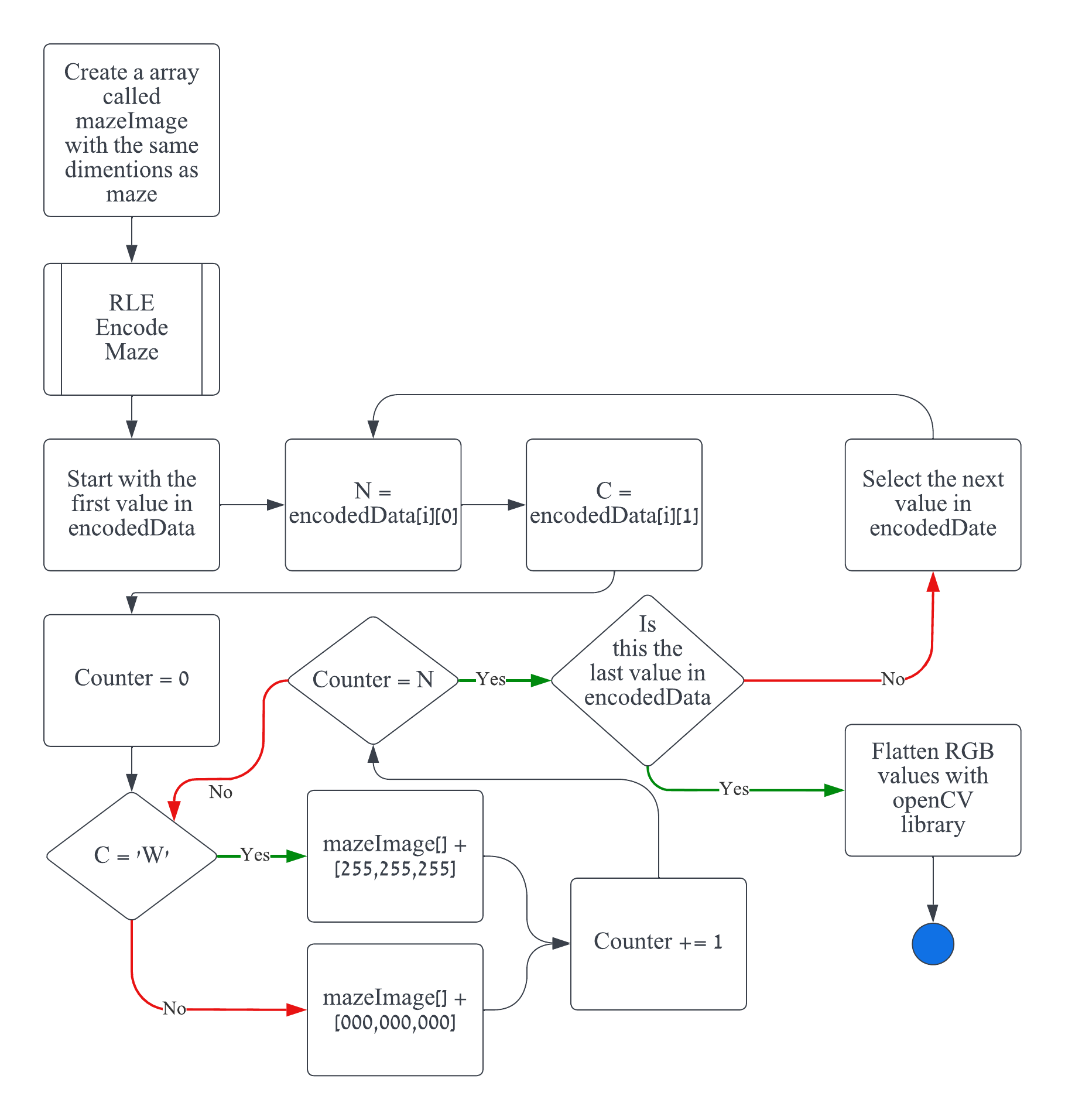
#### Solve timer -



Remove appropriate dead ends



#### Convert maze to image



#### Display Statistics



## 

## Inputs / Processing / Storage / Output (IPSO)

### General IPSO

|  |  |
| --- | --- |
| **Inputs** | **Processing** |
| * File path * Maze Width * Maze Height * Generation Algorithm * Solving Algorithm * Dead End Remover * Maze Entry * Maze/Background/Solve Colour | * Generate Maze * Solve Maze * Turing input image to black and white * Remove Dead Ends * Change Maze Entry * Change Maze/Background/Solve Colour * Animate Maze Generation/Solving and Dead-End Removal * Downloading Maze |
| **Storage** | **Output** |
| * Maze Image * Solved Path * Number of Dead Ends * Time to Remove Dead Ends * Time to Generate Maze * Time to Solve Maze * Time to Draw Maze * Instant Animation Boolean * Status | * Maze Generation/Solving and Dead-End Removal Animation * Maze Statistics * Maze Ready to be Downloaded * Maze Displayed |

### Depth First Search Backtracker IPSO

|  |  |
| --- | --- |
| **Inputs** | **Processing** |
| * N/A * X * Y | * Backtracking Stack will store all pervious nodes. * Direction will store which wall will be broken. * Unvisited Neighbours will store which neighbours to a cell is unvisited. * Valid Neighbours will store which neighbours are valid to enter. * X will store the start X position * Y will store the start Y position |
| **Storage** | **Output** |
| * Backtracking Stack * Direction * Unvisited Neighbours * Valid Neighbours * X * Y | * Maze Statistics * Maze Displayed * Generation Animation |

### Randomised Prims IPSO

|  |  |
| --- | --- |
| **Inputs** | **Processing** |
| * N/A * X * Y | * Visited Cells will store what cell have been visited. * Walls will store the wall information of the current cell. * Direction will store which wall will be broken. * Current Cell will store the information of the current cell. * X will store the start X position * Y will store the start Y position |
| **Storage** | **Output** |
| * Visited Cells * Walls * Direction * Current Cell * X * Y | * Maze Statistics * Maze Displayed * Generation Animation |

### Kruskal's IPSO

|  |  |
| --- | --- |
| **Inputs** | **Processing** |
| * N/A | * Walls will store the wall information of the current set. * Set will store the 2 cells that are current being looked at. * Direction will store which wall will be broken. * Neighbours will store information of the sets neighbours. |
| **Storage** | **Output** |
| * Walls * Set * Direction * Neighbours | * Maze Statistics * Maze Displayed * Generation Animation |

### Aldous-Border IPSO

|  |  |
| --- | --- |
| **Inputs** | **Processing** |
| * X * Y | * Total Cells will count the number of total cells that can be visited * Visited Cells will store what cell have been visited. * Direction will store which wall will be broken. * X will store the start X position * Y will store the start Y position |
| **Storage** | **Output** |
| * Total Cells * Visited Cells * Direction * X * Y | * Maze Statistics * Maze Displayed * Generation Animation |

### Dijkstra’s IPSO

|  |  |
| --- | --- |
| **Inputs** | **Processing** |
| * N/A | * Weights will store the current weight to assigned to the next group of cells, * Queue will store the cells next to be assigned a weight. * Current Cell will store the information of the current cell. * Visited Cells will store what cell have been visited. * Neighbours will store information of the neighbouring cells. |
| **Storage** | **Output** |
| * Weights * Queue * Current Cell * Visited Cells * Neighbours | * Maze Statistics * Maze Displayed * Solving Animation |

### Breath First Search IPSO

|  |  |
| --- | --- |
| **Inputs** | **Processing** |
| * N/A | * Queue will store the cells that need their sub tree traversed. * Parents will store the parents of a given node. * Current Cell will store the information of the current cell. |
| **Storage** | **Output** |
| * Queue * Parents * Current Cell | * Maze Statistics * Maze Displayed * Solving Animation |

### A\* IPSO

|  |  |
| --- | --- |
| **Inputs** | **Processing** |
| * N/A | * Weights will store the current weight to assigned to the next group of cells. * Heuristic Weight will store the additional weight that needs to add to each group of cells. * Queue will store the cells next to be assigned a weight. * Current Cell will store the information of the current cell. * Visited Cells will store what cell have been visited. * Neighbours will store information of the neighbouring cells. |
| **Storage** | **Output** |
| * Weights * Heuristic Weight * Queue * Current Cell * Visited Cells * Neighbours | * Maze Statistics * Maze Displayed * Solving Animation |

### Wall Follower IPSO

|  |  |
| --- | --- |
| **Inputs** | **Processing** |
| * N/A | * Current Cell will store the information of the current cell. * Direction Queue will store the valid directions of a given cell. |
| **Storage** | **Output** |
| * Current Cell * Direction Queue | * Maze Statistics * Maze Displayed * Solving Animation |

## Algorithms

###### Quick Note

Each algorithm will have an explanation of what it does, followed by the pseudo-code, then and explanation of how it works.

### Cell Class

#### Dead-End Finder

The goal is to be able to find the position of each dead-end. A dead end is a cell that have 3 walls around it.

|  |
| --- |
| wallCount = 0  for each wall in walls  if wall = True And mazeWallBool = False then  wallCount += 1  end  if wallCount = 3 and Not deadEndPos.Contains(Point(x, y)) then  deadEndPos.add(Point(x, y)) |

This works by looping each wall in the cell property ‘walls’, ‘walls’ stores whether a wall is True or False. If the wall is true and the cell isn’t a maze edge, then we increment ‘wallCount’. If wall count is 3 and isn’t already in the dead-end list, then we add.

#### Break Wall

The goal is to break a wall of a cell in each direction ‘d’.

|  |
| --- |
| d = [INPUT]  if mazeWallBool = True then  return Point.Empty  Select Case d  Case 0 # break top wall  if maze(x, y - 1).mazeWallBool = True then  return Point.Empty  Exit Function    walls(d) = False  maze(x, y - 1).walls(d + 2) = False  maze(x, y - 1).connectedCell.Add(Point(x, y))    … # break right and bottom walls  Case 3 # break left walls  if maze(x - 1, y).mazeWallBool = True then  return Point.Empty  Exit Function    walls(d) = False  maze(x - 1, y).walls(d + 2) = False  maze(x - 1, y).connectedCell.Add(Point(x, y)) |

If the cell is a maze edge, return nothing because we never want to break the walls around the maze edges. Depending on ‘d’, we check if it breaks the top = 0, right = 1, bottom = 2, left = 3. We then set the wall on that cell = False, we must also set the wall of the connecting cell = False. So, if we break the right wall of cell (1,1), we break the right wall of cell (1,1) we must also break the left wall of the cell (2,1).

#### Check Unvisited Neighbours

The goal is having a list of positions of neighbouring cells which have not been visited.

|  |
| --- |
| neighbours as List (Point)  if mazeWallBool = True then  return List.Fill(3, Point.Empty)  Exit Function  # check above  if maze(x, y - 1).visited = False then  neighbours.Add(Point(x, y -1))  … # check to the right and below  # check to the left  if maze(x - 1, y).visited = False then  neighbours.Add(Point(x - 1, y))  return neighbours |

If the cell is a maze edge, we will the list with empty point objects. Else, we check top, right, bottom, left, it any of the cells are NOT visited we add its position into the list.

#### Check Connected Cell

The goal is to get the point of a connected cell given a direction ‘d’. A connected cell is a cell that doesn’t have a wall between them.

|  |
| --- |
| d = [INPUT]  Select Case d  Case 0 # check above  if connectedCell.Contains(Point(x, y - 1)) then  return Point(x, y-1)    … # check to the right and below  Case 3 # check to the left  if connectedCell.Contains(Point(x - 1, y)) then  return Point(x - 1, y) |

Depending on ‘d’, we check if it breaks the top = 0, right = 1, bottom = 2, left = 3. If the point is within the connected cell list, we can return that point.

Image Input

#### Turing Image to Grayscale

The goal is to find the luminosity of each pixel in an image, this will set by the ability to alter the grayscale threshold.

|  |
| --- |
| GAMMA = 1  filePath = fileDialog  img = OpenFile filePath  for x = 0 to img.Width - 1  for y = 0 to img.Height - 1  currentPixel = inputImage.Height -1  luminosity = (currentPixel.R \* R) ^ GAMMA + (currentPixel.G \* G) ^ GAMMA + (currentPixel.B \* B) ^ GAMMA  CloseFile |
| |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | | Original Image |  | Output Image | |

I loop though each pixel in the image and get their R,G,B values. I then use the luminosity equation to find the luminosity of each pixel.

Altering the Grayscale Threshold  
The goal is to set each pixel as black or white, this will allow me to mask the maze.

|  |
| --- |
| GAMMA = 1  filePath = fileDialog  img = OpenFile filePath  for x = 0 to img.Width - 1  for y = 0 to img.Height - 1  currentPixel = inputImage.Height -1  luminosity = (currentPixel.R \* R) ^ GAMMA + (currentPixel.G \* G) ^ GAMMA + (currentPixel.B \* B) ^ GAMMA  if luminosity <= 125 then  img.SetPixel(x, y, Color.Black)  else  img.SetPixel(x, y, Color.White)  CloseFile |
| |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | | Original image | Grayscale Image | Output Image | |

If the luminosity of a pixel is less or equal to 125 then set the colour of that pixel = to black. Else set it to white.

Masking the Image

The goal to be able to store the position of these the black pixels. Then when I’m initializing the maze I can set any cell that has a position = to a black pixel set it as a maze edge.

|  |
| --- |
| GAMMA = 1  filePath = fileDialog  img = OpenFile filePath  for x = 0 to img.Width - 1  for y = 0 to img.Height - 1  currentPixel = inputImage.Height -1  luminosity = (currentPixel.R \* R) ^ GAMMA + (currentPixel.G \* G) ^ GAMMA + (currentPixel.B \* B) ^ GAMMA  if luminosity <= 125 then  img.SetPixel(x, y, Color.Black)  mazeWallList.Add(Point(x, y))  else  img.SetPixel(x, y, Color.White)  CloseFile |
| # INITAILIZE MAZE  for i = 0 to maze.width  for j = 0 to maze.height  … # setting up the maze cells  # setting the maze wall cells with mazeWallBool  if i = 0 Or j = 0 Or i = width Or j = height Or mazeWallList.Contains(Point(i, j))  maze(i, j).mazeWallBool = True  … # Setting the rest of cell properties |

When setting the pixel to black, I can store its position in a list (mazeWallList).  
When initializing maze edges, they are on the edges (i=0 or i=width, j=0 or j=height) or they are in the black pixel list (mazeWallList).

Component Analysis

The goal is to be able to store the position of each component of the image, a component is group of connecting white pixels. This allows to give my generation algorithims these lists, and maze will be generated in every component.

|  |
| --- |
| components as List(List(Point))  visited as 2DArray(Boolean)({img.Width - 1, img.Height – 1}  for y = 0 to img.Height - 1  for x = 0 to img.Width - 1  if Not visited(x, y) And img.GetPixel(x, y) = Color.White  component as List(Point)  stack as Stack(Point)  stack.push(Point(x, y))    while stack.Count > 0  pixel = stack.Pop    if Not visited(pixel.x, pixel.y) and img.GetPixel(pixel.X, pixel.Y) = Color.White  visited(pixel.X, pixel.Y) = True  component.add(pixel)    if pixel.Y > 0 then # add cell on top to the stack  stack.Push(Point(pixel.X, pixel.Y - 1))  … # add cell on the right and below to the stack  if pixel.X > 0 then # add cell to the left to the stack  stack.Push(Point(pixel.X - 1, pixel.Y))    components.Add(component) |

I loop though each pixel in the image, if the pixel isn’t visited and its white we create a new ‘component list and crate a temporary stack and push the position of the pixel. We then check each pixel in the stack, if the pixel position at the top of the stack is not visited and it’s white, we add it into the current ‘component’ list. I then push the cells on top, on right, on bottom, on left to the stack. I do the same checks, until all pixels surrounding a component are black = the stack being empty.

### Generation Algorithms

The goal for all of the generation algorithms to generate a maze.

#### Depth First Search Backtracker

|  |
| --- |
| x, y = [INPUT, INPUT]  stack as Stack(Point)  stack.push(x,y)  while stack.Count > 0  currentCell = stack.Peek  cell = maze(currentCell.x, currentCell.Y)  cell.visited = True    unvistedNeighbours = cell.checkUnvistedCells  if unvistedNeighbours.All( Point.Empty)= True then  stack.Pop  Continue While    validNeighbours as List(Point)  for each point in unvistedNeighbours  if point <> Point.Empty then  validNeighbours.Add(point)  direction = unvistedNeighbours.IndexOf(validNeighbours(rnd.Next(0, validNeighbours.Count)))  randomNeighbour = cell.breakWall(direction)  stack.push(randomNeighbour) |

We push the start cell into the stack. Until the stack is empty, we check the top value of the stack. If the cell has no unvisited cells, we remove the cell from the stack. Else, we check for valid neighbours. A valid neighbour is a cell that doesn’t = nothing. We randomly pick a valid neighbour, find the direction of that neighbour in relation to the current cell. Break the wall between the cell in that given direction and push the random neighbour to the stack.

#### Randomised Prim’s

|  |
| --- |
| x, y = [INPUT, INPUT]  currentCell = maze(x,y)  visitedCells as List(Cell)  walls as List(Tuple(Point, Integer))  currentCell.Visited = True  visitedCells.Add(currentCell)  for i = 0 to 3  if Not currentCell.walls(i) then Continue For  walls.Add(Tuple.Create(Point(x, y), i))  While visitedCells.Count < totalCells AndAlso walls.Count > 0  randomIndex = rnd.Next(walls.count)  randomWall as Tuple(Point, Integer) = walls(randomIndex)  cellCoords as Point = randomWall.Item1  direction as Point = randomWall.Item2  cell as Cell = maze(cellCoords.X, cellCoords.Y)    neighbour as Point = cell.checkUnvistedNeighbours()(direction)  if neighbours <> Point.Empty AndAlso Not maze(neighbour.X,  neighbour.Y).visited then  cell.breakWall(direction)  maze(neighbour.X, neighbour.Y).visited = True  visitedCell.Add(maze(neighbour.X, neighbour.Y))    for i = 0 to 3  if Not maze(neighbours.X, neighbours.Y).walls(i) then Continue For  walls.Add(Tuple.Create(Point(neighbour.X, neighbour.Y), i))  walls.removeAt(randomIndex) |

I will mark the currentCell as visited and add it to the visitedCells list. I will add the currentCells walls into a list. While there are still unvisited cells and the walls list is not empty, I will randomly pick a cell that shares a wall with currentCell. If that cell in unvisited break the wall between them and mark it as visited and add it to the vistitedCell list. Create a new walls list for the new cell and remove the old wall from that list.

Kurskal’s

|  |
| --- |
| sets as Dictionary(Point, Integer)  setID = 0  walls as List(Tuple(Point, Integer))  for x = 0 to width - 1  for y = 0 to height - 1  sets(Point(x, y)) = setID  setID += 1  for each wall as tuple(point, integer) in walls  cellCoords as Point = wall.Item1  direction = wall.Item2  cell as Cell = maze(cellCoords.X, cellCoords.Y)  If direction = 0 Then                neighbourCoords = Point(cellCords.X, cellCords.Y - 1)  else  neighbourCoords = Point(cellCords.X + 1, cellCords.Y)  if neighbourCoords.X >= 0 AndAlso neighbourCoords.X < width AndAlso neighbourCoords.Y >= 0 AndAlso neighbourCoords.Y < height then  if sets(cellCoords) <> sets(neighbourCoords) then  cell.breakwall(direction)  setIdToReplace = sets(neighbourCoords)  setIdToKeep = sets.Keys.ToList  for each key in sets.Key.ToList then  if sets(key) = setIdToReplace then  sets(key) = setIdToKeep 5 |

First, we assign a unique setID to each cell in the maze. Iterate through each wall in the maze, for each wall find the coordinates of the cell associated with the wall ‘cellCoords’ and find the neighbouring cells ‘neighbourCoords’. Check if though neighbour cells are not maze edges. If the ‘cellCoords’ and ‘neighbourCoords’ belong to different sets then, break the wall between them, replace the setID’s. Update the ‘sets’ dictionary.

#### Aldous-Border

|  |
| --- |
| while visitedCells < totalCells  randomDirection = rnd.Next(directions  if randomDirection.X > 0 And randomDirection.X < width And randomDirection.Y > 0 And randomDirection.Y < height then  if Not maze(randomDirection.X, randomDirection.Y).visted then  maze(currentX, currentY).breakWall(randomDirection)  maze(randomDirection.X, randomDirection.Y).visited = true  visitedCells += 1    currentX = randomDirection.X  currentY = randomDirection.Y |

First choose a random direction from the available directions. Check if the coordinates of the cell in the chosen direction is a maze edge. If the neighbour has not been visited, break the wall between, mark the cell as visited and increment visitedCells += 1. Update the currentX and currentY.  
  
Solving Algorithms

The goal is to create algorithms that its able to find some coordinate on the maze grid and take into account the maze walls. Then produce a list of coordinates that is the path that the algorithim took to find said coordinate.

#### Dijkstra’s

|  |
| --- |
| **weights as Dictionary (Point, Integer)**  **parents as Dictionary (Point, Point)**  **queue as PriorityQueue(Double, Point)**    **weights(mazeEntry) = 0**  **queue.enqueue(0, mazeEntry)**    **while Not is.Empty**  **current = queue.dequeue**  **if current = mazeExit then**  **Exit While**  **for each neighbour in maze(current.x, current.y).connectedCell**  **weight = weights(current) + 1**  **if not weights.ContatainsKey(neighbour) then**  **weights(neighbour) = Double.MaxValue**  **parents(neighbour) = current**  **if Not pQueue.Contains(neighbour) then**  queue.enqueue(weight, neighbour) |

Set the weight to the mazeEntry cell to 0. Loop through the priority queue until its empty. Dequeue the smallest weight from the priority queue, if the current cell is the mazeExit then exit loop as the shortest path has been found. Else, calculate a new weight = weight + 1, if the neighbour is not in the dictionary set its weight = infinity and add the current cell as its parent in the ‘parents’ dictionary, add the neighbour to the priority queue with the calculated weight.  
  
Breadth-First Search

|  |
| --- |
| queue as Queue(Point)  visitedNode as List(Point)  parents and Dictionary (Point, Point)  queue.enqueue(mazeEntry)  While queue.Count <> 0  currentNode = queue.dequeue    if currentNode = mazeExit then  Exit While  for each point in maze(currentNode.X, currentNode.Y).connectedCell  if Not visitedNodes.Contains(Point)  visitedNodes.Add(point)  parents(point) = currentNode  queue.enqueue(point) |

Add mazeEntry to the queue. Until the queue is not empty, remove the first item in the queue. If the cell is the ‘mazeExit’ then exit the loop as the path has been constructed. Else, for each connected neighbour, that has not been visited, add it to the ‘visitedNode’ list. Add the neighbour node to the ‘parent’ dictionary and set the current node as the parent. Add neighbour to the queue.

#### A\*

|  |
| --- |
| gWeights as Dictionary (Point, Double)  parents as Dictionary (Point, Point)  queue as PriorityQueue(Double, Point)  gWeights(Maze Entry) = 0  queue.enqueue(distanceCalc(mazeEntry, MazeExit), mazeEntry)  While Not pQueue.isEmpty()  current = pQueue.dequeue    if current = mazeExit then  Exit While    for each neighbour in maze(current.X, current.Y).ConnectedCell  heuristicWeight = gWeights(current) + distanceCalc(current, neighbour)    if Not gWeight.Contains(neighbour) then  gWeights(neighbour) = Double.MaxValue    if heuristicWeight < gWeights(neighbour) then  parents(neighbour) = current  gWeights(neighbour) = heuristicWeight  fWeight = gWeights(neighbour) + distanceCalc(neighbour, mazeExit)  if Not pQueue.Contains(neighbour) then  pQueue.Enqueue(fWeight) |

A\* has the same idea a Dijkstra’s but when we calculate the new weight we use the heuistic weight (the Manhattan distance)

##### Heuristic Distance Calculation

|  |
| --- |
| function distanceCalc(a as Point, b as Point) as Double  return Math.Abs(a.X - b.X) + Math.Abs(b.Y - b.X) |

This allows me to consider the Manhattan distance from one node to another. This is only used in A\*. The Manhattan distance will prioritise moves that are directly adjacent to the given cell. I chose this because, in this maze, the solving algorithims will no be able to move in diagonals.

#### Wall Follower LHR

|  |
| --- |
| directionQueue as CircularQueue(Integer)({0,1,2,3})  node = mazeEntry  while node <> mazeExit  index directionQueue.turnLeft  if maze(node.X, node.Y).checkConnectedCell(index) <> Point.Empty then  node = maze(node.X, node.Y).checkConnectedCell(index)  else  do  index = directionQueue.turnRight  loop until maze(node.x, node.y).checkConnectedCell(index) <> Point.Empty    if node <> mazeEntry and node <> mazeExit then  path.Enqueue(node) |

#### Wall Follower RHR

|  |
| --- |
| directionQueue as CircularQueue(Integer)({0,1,2,3})  node = mazeEntry  while node <> mazeExit  index directionQueue.turnRight  if maze(node.X, node.Y).checkConnectedCell(index) <> Point.Empty then  node = maze(node.X, node.Y).checkConnectedCell(index)  else  do  index = directionQueue.turnLeft  loop until maze(node.x, node.y).checkConnectedCell(index) <> Point.Empty    if node <> mazeEntry and node <> mazeExit then  path.Enqueue(node) |

Start at ‘mazeEntry’, until the current node doesn’t equal ‘mazeExit’, turn left by dequeing the first direction from the ‘directionQueue’. Check if there is a connected cell in that direction. If there is, update node to that direction, else continue to turn left until a connected cell is found.  
The only thing that changes in LHR to RHR is that you right instead of left.

#### Reconstruct Path

|  |
| --- |
| **for each node in path**  **maze(node.X, node.Y).mazeSolved = True** |

This allows me to mark the correct cells as solved.

### Dead End Remover

|  |
| --- |
| deadEndPercent = [USER INPUT]  for each cell in maze  cell.deadEndFinder()  numToBeRemoved = Math.Round(deadEndPos.Count() \* deadEndPercent)  removed = 0  while removed <> numToBeRemoved  deadEnd = deadEndPos(rnd.Next(0, deadEndPos.Count))    validIndexs as List (Integers)  for i = 0 to 3  if maze(deadEnd.X, deadEnd.Y).walls(i) then  validIndexs.add(i)  do  direction = validIndexs(rnd.Next(0, validIndexs.Count))  node = maze(deadEnd.x, deadEnd.Y).breakWall(direction)  loop while node.IsEmpty  deadEndPos.Remove(deadEnd)  if deadEndPos.Contains(node) then  deadEndPos.Remove(node)  removed += 1  removed += 1 |

For each cell we identify if it’s a dead end or not. Calculate the number of dead ends to remove. Until the number of removed dead ends doesn’t equal the number of dead ends to be removed then select a random dead end. Choose a random, valid, wall to break from that dead end. Remove dead end from the ‘deadEndPos’ list. If the current dead end selected now has a connected cell that is in the ‘deadEndPos’ list, that means we have removed 2 dead ends, therefore we increment ‘removed’ by 1. Increment ‘removed’ by 1 even if we removed 1 or 2 dead ends.

### Download

|  |
| --- |
| **if mazeGenerated then**  **fileLocatoin = open saveFileDialog**  **fileName = Nothing**  **filter = .JPG**  **mazeImg.Save -> fileLocation**  **else**  exit Sub |

We first check if a maze has been generated, if yes then we open the file explorer, set the name to nothing (this can be changed by the user). Then it will save the image at the same

## Chronological Design

1. The level 0 flowchart was first developed to break down the problem into smaller, more manageable components, which provided an overview of the entire system and its processes. This high-level representation allowed for a better understanding of the projects scope and the necessary steps for implementation.

2. Next, the level 1 flowchart was designed to further deconstruct the smaller problems into solutions and formulate the main algorithms, diving deeper into the processes and sub-processes of the system. This detailed breakdown enabled a thorough analysis of the individual components, ensuring a more efficient and targeted approach to problem-solving.

3. With the guidance of the flowcharts, the main algorithms were developed, laying the foundation for the technical solution and providing a clear roadmap for the implementation. This step was crucial in transforming the conceptual design into a practical and functional solution that could be coded and executed.

4. The level 0 data flow diagram was created to visualize the four key locations where data would be sent in the solution, providing a clear understanding of how data flows within the system. By mapping the data flow, potential bottlenecks and inefficiencies could be identified and addressed early in the development process.

5. The level 1 data flow diagram was developed to dissect each location, enabling a more detailed view of data flow between processes and components, which helped in refining the data handling and processing. This granular perspective allowed for a comprehensive analysis of data interaction, ensuring optimal performance and data integrity.

6. The IPSO (Input/Processing/Storage/Output) analysis was produced to examine how the user interacts with the solution and identify areas that required “babyproofing” for enhanced usability and safety. By considering the user's perspective, the solution could be tailored to meet their needs and expectations, resulting in a more user-friendly and efficient system.

7. With the design resources and research at hand, the coding of the technical solution began, implementing the algorithms and processes developed in the previous steps. This stage involved translating the theoretical design into a functional application, ensuring that the solution was both efficient and effective in addressing the problem at hand.

8. Finally, the DDD (Design Data Dictionary) was created to outline all public variables, sub-procedures, and classes developed in the technical solution, providing a comprehensive reference for future development and maintenance. This documentation ensured that any future modifications or updates to the solution could be carried out efficiently, with a clear understanding of the underlying structure and design elements.

## Implementation

### RLE Encoding

RLE Encoding was deemed unnecessary, as it was not required to store any information other than the black pixels and implementing it would result in a waste of memory.

### Height/Width

The previous boundaries for the maze display have change due to the GUI redesign. The new boundaries are now 4-1000 for both height and width. After testing, the maximum value was 1000, which was decided due to memory constraints.

If the user wants to display the maze instead of downloading it, the boundaries for width and height are 4-406 and 4-230 respectively, as the magnifier variable (m) must be greater than or equal to 3.

### Changing RGB to HEX

Conversion to HEX is not necessary since RGB values are easier to comprehend (for colours) and it’s utilised by the VB.Net Color class.

### Converting to .JPG

Conversion to .JPG is no longer required, as the file explorer can be filtered to display only .JPG images.

### Saving a copy of the image

Manipulating the maze in memory is faster than in secondary storage, which is especially evident when handling a large image that requires constant reading and writing.

### Information button

The GUI redesign allowed for more space to display the maze statistics, and as a result, the text labels are updated every time a button is pressed.

### Removing Wave Function Collapse

During the process of implementing the algorithm, it became evident that using Wave Function Collapse for maze generation would be highly time-consuming.

Wave Function Collapse excels in problems where: the tile-set can be easily defined, and tiles have distinct adjacency rules. When considering mazes, there are numerous tile-sets that exist, with each one of these having their distinct adjacency rules.

There are two possible approaches that I considered: one would be investing many hours in exploring various tile-sets and their adjacency rules, or I could use machine learning models to identify suitable tile-sets and adjacency rules. Unfortunately, neither option is within the project’s scope.

## Naming Conventions

I used camel casing on all my variables and functions. I break away from camel casing when: naming classes, when naming a constant or when naming a constructor.

Classes will always start with a capital letter, if its two words there will be no connecting punctuation and all words will have a capital letter. When a class as a method that has the same name as a inbuilt method I will name it the same as the inbuilt method

Constants will always be in all capitals, if its two words they will be connected by an underscore.

Constructors will always be named ‘New’.

|  |  |
| --- | --- |
| **Type** | **Example** |
| Naming normal variable or function | ‘branchingPoints’ or ‘randomisedPrims’ |
| Naming classes | ‘CircularQueue’ or ‘Cells’ |
| Naming a constant | ‘M’ or ‘PEN\_SIZE’ |
| Naming a constructor | ‘New’ |

# Technical Solution

###### Quick note

Code that is highlighted yellow are followed by and explaintion of its purpose. Code that is highlighted in green are part of Objective Mapping references, this is further below in the document.

A unhighlighted version of the technical solution will be in Apendix 1.

## Maze Form

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Public** **Class** Form1  *' Class instanse used to randomize numbers*  **Private** rnd **As** **New** Random  *' Drawing Variables*  **Private** **Const** PEN\_SIZE **As** Integer = 2  **Private** M **As** Integer = 3  *' Maze properties*  **Private** maze **As** Cell(,)  **Private** width **As** Integer  **Private** height **As** Integer  **Private** deadEndPercent **As** Double  **Private** mazeEntryType **As** String  **Private** mazeEntry **As** Point  **Private** mazeExit **As** Point  **Private** deadEndPos **As** **New** List(**Of** Point)  *' Maze Colour Customisation*  **Private** bgColour **As** Color = Color.White  **Private** mazeColour **As** Color = Color.Black  **Private** solveColour **As** Color = Color.Silver  **Private** pinkColor **As** Color = Color.FromArgb(255, 0, 220) *' Pink color*  **Private** purpleColor **As** Color = Color.FromArgb(0, 0, 124) *' Purple color*  *' Animation Variables*  **Private** **Const** T **As** Integer = 50  **Private** passedPath **As** **New** List(**Of** Point)  **Private** solvedVisited **As** **New** Queue(**Of** Point)  **Private** maxWeight **As** Integer  **Public** cancelAnimation **As** Boolean = **False**  **Public** resetType **As** String  *' Maze Generation/Solving Inputs*  **Private** generationAlgorithm **As** String  *' Used in DFSbacktracker*  **Dim** generationStack **As** **New** Stack(**Of** Point)  *' Used in randomizedPrims*  **Private** visitedCells **As** **New** List(**Of** Cell)  **Private** primsWalls **As** **New** List(**Of** Tuple(**Of** Point, Integer))  *' Used in kurskals*  **Private** kurskWalls **As** **New** List(**Of** Tuple(**Of** Point, Integer))  **Private** kursSets **As** **New** Dictionary(**Of** Point, Integer)  **Private** kursNeighbourCoords **As** Point  **Private** kursCurrentWallIndex **As** Integer = 0  *' Used in aldousBroder*  **Private** abTotalCells **As** Integer  **Private** abVisitedCells **As** Integer = 0  **Private** random **As** **New** Random()  **Private** currentX **As** Integer  **Private** currentY **As** Integer  **Private** directions() **As** Point = {**New** Point(0, -1), **New** Point(1, 0), **New** Point(0, 1), **New** Point(-1, 0)}  **Private** abHasBeenAnimated **As** Boolean = **False**  *' Used in Astar()*  **Public** gWeights **As** **New** Dictionary(**Of** Point, Double)  *' Used in BFS()*  **Public** branchingPoints **As** **New** List(**Of** Point)  **Private** solveAlgorithm **As** String  **Private** mazeWallCount **As** Integer = 0  **Private** totalCells **As** Integer = 0  **Public** path **As** **New** Queue(**Of** Point)  **Public** helperPath **As** **New** Queue(**Of** Point)  *' Controls when the from draws*  **Private** mazeImage **As** Bitmap  **Private** mazeImageGraphics **As** Graphics  **Private** downlaodGenerated **As** DialogResult  **Private** downlaodSolved **As** DialogResult  *' Stats Variables*  **Private** deadEndToShow **As** Integer  **Private** solveTimer **As** **New** Stopwatch  **Private** generationTimer **As** **New** Stopwatch  **Private** drawTimer **As** **New** Stopwatch  **Private** deadEndTimer **As** **New** Stopwatch  *' Babyproofing Variables*  **Private** mazeGenerated **As** Boolean = **False**  *' Image to maze Variables*  **Private** imageInputted **As** Boolean = **False**  **Private** inputImage **As** Bitmap  **Private** mazeWallList **As** **New** List(**Of** Point)  **Private** luminosity **As** Double  **Private** **Const** GAMMA **As** Double = 1.0  **Private** **Const** R **As** Double = 0.2126  **Private** **Const** G **As** Double = 0.7152  **Private** **Const** B **As** Double = 0.0722  **Private** imgComponents **As** **New** List(**Of** List(**Of** Point))  **Private** largestComponent **As** **New** List(**Of** Point)  *' Circular Queue Class*  **Class** CircularQueue(**Of** T)  **Private** **ReadOnly** items **As** List(**Of** T)  **Private** currentIndex **As** Integer    *' Constructor*  **Public** **Sub** **New**(i **As** IEnumerable(**Of** T))  *' This assigns the items in the queue*  items = **New** List(**Of** T)(i)  currentIndex = 0  **End** **Sub**    *' This function return value will be same type and the input*  **Public** **Function** turnRight() **As** T  **If** items.Count = 0 **Then**  *' This helps when debugging*  **Throw** **New** InvalidOperationException("The queue is empty")  **End** **If**      currentIndex = (currentIndex + 1) **Mod** items.Count  **Dim** i **As** T = items(currentIndex)  **Return** i  **End** **Function**    *' This function return value will be same type and the input*  **Public** **Function** turnLeft() **As** T  **If** items.Count = 0 **Then**  *' This helps when debugging*  **Throw** **New** InvalidOperationException("The queue is empty")  **End** **If**    currentIndex = (currentIndex - 1 + items.Count) **Mod** items.Count  **Dim** i **As** T = items(currentIndex)    **Return** i  **End** **Function**  **End** **Class**  *' Priority Queue Class*  **Public** **Class** PriorityQueue(**Of** priority **As** IComparable, value)  **Private** **ReadOnly** dictionary **As** SortedDictionary(**Of** priority, Queue(**Of** value))    *' Constructor*  **Public** **Sub** **New**()  *' This assigns the items in the dictionary*  dictionary = **New** SortedDictionary(**Of** priority, Queue(**Of** value))()  **End** **Sub**    *' Adds values to the queue*  **Public** **Sub** Enqueue(priority **As** priority, value **As** value)  *' If we have a new priority we create a new queue*  **If** **Not** dictionary.ContainsKey(priority) **Then**  dictionary(priority) = **New** Queue(**Of** value)()  **End** **If**  *' Add value to queue*  dictionary(priority).Enqueue(value)  **End** **Sub**    *' Removes values to the queue*  **Public** **Function** Dequeue() **As** value  *' This helps when debugging*  **If** dictionary.Count = 0 **Then**  **Throw** **New** InvalidOperationException("The priority queue is empty.")  **End** **If**    **Dim** firstPair **As** KeyValuePair(**Of** priority, Queue(**Of** value)) = dictionary.First()  **Dim** value **As** value = firstPair.Value.Dequeue()    **If** firstPair.Value.Count = 0 **Then**  dictionary.Remove(firstPair.Key)  **End** **If**    **Return** value  **End** **Function**    *' Checks if the whole queue is empty*  **Public** **Function** isEmpty() **As** Boolean  **Return** dictionary.Count = 0  **End** **Function**    *' Returns the number of items in the queue*  **Public** **Function** Count() **As** Integer  **Dim** totalCount **As** Integer = 0  **For** **Each** q **In** dictionary.Values  totalCount += q.Count  **Next**  **Return** totalCount  **End** **Function**    *' Checks if a value is in the queue*  **Public** **Function** Contains(v **As** value) **As** Boolean  **For** **Each** q **In** dictionary.Values  **If** q.Contains(v) **Then**  **Return** **True**  **End** **If**  **Next**  **Return** **False**  **End** **Function**  **End** **Class**  *' Cell Class*  **Public** **Class** Cell  *' Postion Properties*  **Public** x **As** Integer  **Public** y **As** Integer  *' Wall Properties*  **Public** walls **As** **New** List(**Of** Boolean)({**True**, **True**, **True**, **True**})  **Public** wallPos(3, 1) **As** Point  *' Cell Type*  **Public** mazeWallBool **As** Boolean = **False**  **Public** mazeEntryBool **As** Boolean = **False**  **Public** mazeExitBool **As** Boolean = **False**  **Public** mazeSolved **As** Boolean = **False**  *' Generate/Solve Properties*  **Public** visited **As** Boolean = **False**  **Public** connectedCell **As** **New** List(**Of** Point)    *' Method to draw walls*  **Public** **Sub** drawWalls()  **For** wall **As** Integer = 0 **To** 3  **If** walls(wall) = **True** **And** Form1.mazeColour = Color.Empty **Then**  Form1.mazeImageGraphics.DrawLine(**New** Pen(Color.Black, PEN\_SIZE), wallPos(wall, 0), wallPos(wall, 1))  Form1.mazeImageGraphics.DrawLine(**New** Pen(Color.Black, PEN\_SIZE), wallPos(wall, 0), wallPos(wall, 1))  **ElseIf** walls(wall) = **True** **Then** *' If user hasnt selected colour*  Form1.mazeImageGraphics.DrawLine(**New** Pen(Form1.mazeColour, PEN\_SIZE), wallPos(wall, 0), wallPos(wall, 1))  Form1.mazeImageGraphics.DrawLine(**New** Pen(Form1.mazeColour, PEN\_SIZE), wallPos(wall, 0), wallPos(wall, 1))  **End** **If**  **Next**  **End** **Sub**  *' Method to find dead-ends*  **Public** **Sub** deadEndFinder()  **Dim** wallCount **As** Integer = 0  *' Checks each wall*  **For** **Each** wall **In** walls  **If** wall = **True** **And** mazeWallBool = **False** **Then**  wallCount += 1  **End** **If**  **Next**  **If** wallCount = 3 **And** **Not** Form1.deadEndPos.Contains(**New** Point(x, y)) **Then**  Form1.deadEndPos.Add(**New** Point(x, y))  **End** **If**  **End** **Sub**  *' Method to break wall*  **Public** **Function** breakWall(**ByVal** d **As** Integer)  *' Makes sure the cell isn't a maze wall*  **If** mazeWallBool = **True** **Then**  **Return** Point.Empty  **Exit** **Function**  **End** **If**  *' Breaks wall depending on the d (direction)*  **Select** **Case** d  *' The wall selected must be broken but also the neighbours wall*  **Case** 0 *' Breaking the top wall*  **If** Form1.maze(x, y - 1).mazeWallBool = **True** **Then**  **Return** Point.Empty  **Exit** **Function**  **End** **If**  walls(d) = **False**  Form1.maze(x, y - 1).walls(d + 2) = **False**  connectedCell.Add(**New** Point(x, y - 1))  Form1.maze(x, y - 1).connectedCell.Add(**New** Point(x, y))  **Case** 1 *' Breaking the right wall*  **If** Form1.maze(x + 1, y).mazeWallBool = **True** **Then**  **Return** Point.Empty  **Exit** **Function**  **End** **If**  walls(d) = **False**  Form1.maze(x + 1, y).walls(d + 2) = **False**  connectedCell.Add(**New** Point(x + 1, y))  Form1.maze(x + 1, y).connectedCell.Add(**New** Point(x, y))  **Case** 2 *' Breaking the bottom wall*  **If** Form1.maze(x, y + 1).mazeWallBool = **True** **Then**  **Return** Point.Empty  **Exit** **Function**  **End** **If**  walls(d) = **False**  Form1.maze(x, y + 1).walls(d - 2) = **False**  connectedCell.Add(**New** Point(x, y + 1))  Form1.maze(x, y + 1).connectedCell.Add(**New** Point(x, y))  **Case** 3 *' Breaking the left wall*  **If** Form1.maze(x - 1, y).mazeWallBool = **True** **Then**  **Return** Point.Empty  **Exit** **Function**  **End** **If**  walls(d) = **False**  Form1.maze(x - 1, y).walls(d - 2) = **False**  connectedCell.Add(**New** Point(x - 1, y))  Form1.maze(x - 1, y).connectedCell.Add(**New** Point(x, y))  **End** **Select**  **Return** connectedCell(connectedCell.Count() - 1)  **End** **Function**    *' Method to check unvisted neighbours*  **Public** **Function** checkUnvistedNeighbours() **As** List(**Of** Point)  **Dim** neighbours **As** **New** List(**Of** Point)    **If** mazeWallBool = **True** **Then**  **Return** {Point.Empty, Point.Empty, Point.Empty, Point.Empty}.ToList  **Exit** **Function**  **End** **If**    **If** Form1.maze(x, y - 1).visited = **False** **Then**  neighbours.Add(**New** Point(x, y - 1))  **Else**  neighbours.Add(Point.Empty)  **End** **If**    **If** Form1.maze(x + 1, y).visited = **False** **Then**  neighbours.Add(**New** Point(x + 1, y))  **Else**  neighbours.Add(Point.Empty)  **End** **If**    **If** Form1.maze(x, y + 1).visited = **False** **Then**  neighbours.Add(**New** Point(x, y + 1))  **Else**  neighbours.Add(Point.Empty)  **End** **If**    **If** Form1.maze(x - 1, y).visited = **False** **Then**  neighbours.Add(**New** Point(x - 1, y))  **Else**  neighbours.Add(Point.Empty)  **End** **If**  **Return** neighbours  **End** **Function**  *' Returns a boolean if cell has a connecter neighbour*  **Public** **Function** checkConnectedCell(d **As** Integer)  **Select** **Case** d  **Case** 0 *' Check above*  **If** connectedCell.Contains(**New** Point(x, y - 1)) **Then**  **Return** **New** Point(x, y - 1)  **End** **If**  **Case** 1 *' Check Right*  **If** connectedCell.Contains(**New** Point(x + 1, y)) **Then**  **Return** **New** Point(x + 1, y)  **End** **If**  **Case** 2 *' Check Below*  **If** connectedCell.Contains(**New** Point(x, y + 1)) **Then**  **Return** **New** Point(x, y + 1)  **End** **If**  **Case** 3 *' Check Left*  **If** connectedCell.Contains(**New** Point(x - 1, y)) **Then**  **Return** **New** Point(x - 1, y)  **End** **If**  **End** **Select**  **Return** Point.Empty  **End** **Function**  **End** **Class**    **Public** **Sub** **New**()  *' This call is required by the designer.*  InitializeComponent()  *' Add any initialization after the InitializeComponent() call.*  **AddHandler** helperBtn.Click, **AddressOf** helperBtn\_Click  **AddHandler** imageInputBtn.Click, **AddressOf** imageInputBtn\_Click  **AddHandler** bgColourBtn.Click, **AddressOf** bgColourBtn\_Click  **AddHandler** mazeColourBtn.Click, **AddressOf** mazeColourBtn\_Click  **AddHandler** solveColourBtn.Click, **AddressOf** solveColourBtn\_Click  **AddHandler** generateBtn.Click, **AddressOf** generateBtn\_Click  **AddHandler** solveBtn.Click, **AddressOf** solveBtn\_Click  **AddHandler** deadEndRemoverBtn.Click, **AddressOf** deadEndRemoverBtn\_Click  **AddHandler** downloadBtn.Click, **AddressOf** downloadBtn\_Click  **AddHandler** cancelAnimationBtn.Click, **AddressOf** cancelAnimationBtn\_Click  **AddHandler** solvedPathAnimationTimer.Tick, **AddressOf** solvedPathAnimationTimer\_Tick  **AddHandler** heatMapAnimationTimer.Tick, **AddressOf** heatMapAnimationTimer\_Tick  **End** **Sub**  **Private** **Sub** Form1\_Load(sender **As** Object, e **As** EventArgs) **Handles** **MyBase**.Load      generationCombo.SelectedIndex = 0 *' Makes index 0 default displayed on the combo list(so currently shows "DFS Backtracker" initially*  solveCombo.SelectedIndex = 0 *' Default displays (Dijkstra's Algortimn)*  mazeEntryCombo.SelectedIndex = 0 *' Default displays "Random"*    cancelAnimationBtn.Enabled = **False**    solvedPathAnimationTimer.Interval = T  heatMapAnimationTimer.Interval = T  generationPointTimer.Interval = T  heatMapAnimationTimer.Enabled = **False**  solvedPathAnimationTimer.Enabled = **False**  generationPointTimer.Enabled = **False**    **End** **Sub**  **Private** **Sub** initializeMaze()  mazeWallCount = 0    *' Resets old timer, Starts new timer, Upates Status*  statusLbl.Text = "Status: Initializing Maze"  statusLbl.Update()  *' Initialize each cell with correct: Type and Wall Position*  maze = **New** Cell(width, height) {}  mazeImage = **New** Bitmap(((width + 1) \* M) + M, ((height + 1) \* M) + M)  mazeImageGraphics = Graphics.FromImage(mazeImage)   |  | | --- | | Objective: A |   **For** i **As** Integer = 0 **To** width  **For** j **As** Integer = 0 **To** height  *' Giving each cell their index*  maze(i, j) = **New** Cell  maze(i, j).x = i  maze(i, j).y = j    *' Setting the maze wall cells with the mazeWallBool*  **If** i = 0 **Or** j = 0 **Or** i = width **Or** j = height **Or** mazeWallList.Contains(**New** Point(i, j)) **Then**  maze(i, j).mazeWallBool = **True**  maze(i, j).visited = **True**  mazeWallCount += 1  **Else**  totalCells += 1  **End** **If**    *' Giving each cell wall a start(0), end(1) and position on the screen*  **Dim** posi **As** Integer = i \* M  **Dim** posj **As** Integer = j \* M  **For** **Each** wall **In** maze(i, j).walls  *' Top Wall*  maze(i, j).wallPos(0, 0) = **New** Point(posi, posj)  maze(i, j).wallPos(0, 1) = **New** Point(posi + M, posj)  *' Right Wall*  maze(i, j).wallPos(1, 0) = **New** Point(posi + M, posj)  maze(i, j).wallPos(1, 1) = **New** Point(posi + M, posj + M)  *' Bottom Wall*  maze(i, j).wallPos(2, 0) = **New** Point(posi, posj + M)  maze(i, j).wallPos(2, 1) = **New** Point(posi + M, posj + M)  *' Left Wall*  maze(i, j).wallPos(3, 0) = **New** Point(posi, posj)  maze(i, j).wallPos(3, 1) = **New** Point(posi, posj + M)  **Next**  **Next**  **Next**    *' Setting Maze Entry and Exit*  setMazeEntryExit()  **End** **Sub**   |  | | --- | | Objective: I |   **Private** **Sub** setMazeEntryExit()  **If** imageInputted = **True** **Then**  *' Randomly picks a side*  **Dim** side **As** Integer = rnd.**Next**(1, 2)    **If** side = 1 **Then**  mazeEntry = **New** Point(largestComponent.OrderByDescending(**Function**(p) p.X).First())  mazeExit = **New** Point(largestComponent.OrderByDescending(**Function**(p) p.X).Last())  **Else**  **Dim** orderedList **As** List(**Of** Point) = largestComponent.OrderByDescending(**Function**(p) p.Y)  mazeEntry = **New** Point(orderedList.First())  mazeExit = **New** Point(orderedList.Last())  **End** **If**   |  | | --- | | This block of code generates a maze entry and exit point for maze images. It will order the largest component of an image; the variable slide will randomly pick if the entry or exit will be the first or last cell of the ordered list |   **Else**  **Select** **Case** mazeEntryType  **Case** "Random"  Randomize()  **Dim** randomType **As** Integer = rnd.**Next**(0, 4)  *' Chooses randomly what type of maze entry it will be*  **Select** **Case** randomType  **Case** 0 *' Start at a random top postion, finish at a random bottom position*  mazeEntry = **New** Point(rnd.**Next**(1, width), 1)  mazeExit = **New** Point(rnd.**Next**(1, width), height - 1)  **Case** 1 *' Start at a random bottom postion, finish at a random top position*  mazeEntry = **New** Point(rnd.**Next**(1, width), height - 1)  mazeExit = **New** Point(rnd.**Next**(1, width), 1)  **Case** 2 *' Start at a random right postion, finish at a random left positon*  mazeEntry = **New** Point(1, rnd.**Next**(1, height))  mazeExit = **New** Point(width - 1, rnd.**Next**(1, height))  **Case** 3 *' Start at a random left postion, finish at a random right positon*  mazeEntry = **New** Point(width - 1, rnd.**Next**(1, height))  mazeExit = **New** Point(1, rnd.**Next**(1, height))  **End** **Select**  **Case** "Top - Bottom"  mazeEntry = **New** Point(Math.Round(width / 2), 1)  mazeExit = **New** Point(Math.Round(width / 2), height - 1)  **Case** "Right - Left"  mazeEntry = **New** Point(1, Math.Round(height / 2))  mazeExit = **New** Point(width - 1, Math.Round(height / 2))  **Case** "Diagonal"  mazeEntry = **New** Point(1, 1)  mazeExit = **New** Point(width - 1, height - 1)  **End** **Select**  **End** **If**  *' Setting the entry cell with the mazeEntryBool*  maze(mazeEntry.X, mazeEntry.Y).mazeEntryBool = **True**  maze(mazeEntry.X, mazeEntry.Y).mazeWallBool = **False**  *' Setting the exit cell with the mazeExitBool*  maze(mazeExit.X, mazeExit.Y).mazeExitBool = **True**  maze(mazeExit.X, mazeExit.Y).mazeWallBool = **False**  **End** **Sub**  **Private** **Sub** drawMaze() *' If False is passed through then the background cells will be drawn*  *' Resets old timer, Starts new timer, Upates Status*  drawTimer.Reset()  drawTimer.Start()  statusLbl.Text = "Status: Drawing Maze"  statusLbl.Update()  *' Create brush objects for each color*  **Dim** bgBrush **As** **New** SolidBrush(bgColour)  **Dim** mazeBrush **As** **New** SolidBrush(mazeColour)  **Dim** solvedBrush **As** **New** SolidBrush(solveColour)  **Dim** entryBrush **As** **New** SolidBrush(Color.Green)  **Dim** exitBrush **As** **New** SolidBrush(Color.Red)    **For** **Each** cell **In** maze  *' Determine the fill color based on cell properties*  **Dim** fillBrush **As** Brush = bgBrush  **If** cell.mazeWallBool **Then**  fillBrush = mazeBrush  **End** **If**  **If** cell.mazeEntryBool **Then**  fillBrush = entryBrush  **End** **If**  **If** cell.mazeExitBool **Then**  fillBrush = exitBrush  **End** **If**  **If** cell.mazeSolved **Then**  fillBrush = solvedBrush  **End** **If**    *' Draw the cell background and fill*  mazeImageGraphics.FillRectangle(fillBrush, cell.wallPos(0, 0).X, cell.wallPos(1, 0).Y, M, M)    *' Draw the walls*  cell.drawWalls()  **Next**    *' Dispose of the brush objects*  bgBrush.Dispose()  mazeBrush.Dispose()  entryBrush.Dispose()  exitBrush.Dispose()  solvedBrush.Dispose()  drawTimer.**Stop**()  **End** **Sub**  **Private** **Sub** resetMaze()  instantAnimationBtn.Checked = **True**  *' Clear the collections*  solvedVisited.Clear()  helperPath.Clear()  path.Clear()  passedPath.Clear()    *' Reset the mazeSolved property for each cell*  **For** x **As** Integer = 0 **To** width - 1  **For** y **As** Integer = 0 **To** height - 1  maze(x, y).mazeSolved = **False**  **Next**  **Next**    **If** resetType = "G" **Then**  **If** generationAlgorithm = "DFS Backtracker" **Then**  **If** imageInputted **Then**  **For** **Each** component **In** imgComponents  **Dim** point **As** Point = component(rnd.**Next**(0, component.Count))  DFSbacktracker(point.X, point.Y)  **Next**  **Else**  DFSbacktracker(rnd.**Next**(1, width), rnd.**Next**(1, height))  **End** **If**  **ElseIf** generationAlgorithm = "Randomised Prims" **Then**  **If** imageInputted **Then**  **For** **Each** component **In** imgComponents  **Dim** point **As** Point = component(rnd.**Next**(0, component.Count))  randomisedPrims(point.X, point.Y)  **Next**  **Else**  randomisedPrims(rnd.**Next**(1, width), rnd.**Next**(1, height))  **End** **If**  **ElseIf** generationAlgorithm = "Kruskal 's" **Then**  **If** imageInputted **Then**  MsgBox("You can't use this algorithm with mazes." & vbCrLf & "Try another one!", MsgBoxStyle.OkOnly, "Invalid Input")  **Exit** **Sub**  **Else**  kruskals()  **End** **If**  **ElseIf** generationAlgorithm = "Aldous-Border" **Then**  **If** imageInputted **Then**  MsgBox("You can't use this algorithm with mazes." & vbCrLf & "Try another one!", MsgBoxStyle.OkOnly, "Invalid Input")  **Exit** **Sub**  **Else**  width += 1  height += 1  aldousBroder(rnd.**Next**(1, width), rnd.**Next**(1, height))  width -= 1  height -= 1  **End** **If**  **End** **If**  **ElseIf** resetType = "S" **Then**  *' Checks what solving algorithm user has chosen*  **If** solveAlgorithm = "Dijkstra's" **Then**  dijkstra()  **ElseIf** solveAlgorithm = "Breath Frist Search" **Then**  BFS()  **ElseIf** solveAlgorithm = "A\*" **Then**  Astar()  **ElseIf** solveAlgorithm = "Wall Follower LHR" **Then**  wallFollower("LHR")  **ElseIf** solveAlgorithm = "Wall Follower RHR" **Then**  wallFollower("RHR")  **End** **If**  **End** **If**        *' Redraw the maze*  drawMaze()  mazeBox.Image = mazeImage  mazeBox.Update()    instantAnimationBtn.Checked = **False**  **End** **Sub**  **Private** **Sub** animationLock(Lock **As** Boolean) *' Locks all inputs to prevent backlogging and crashes*  **If** Lock = **True** **Then**  *' Generate Button*  generateBtn.BackColor = Color.FromArgb(18, 73, 18)  generateBtn.Enabled = **False**  *' Solve Button*  solveBtn.BackColor = Color.FromArgb(112, 22, 22)  solveBtn.Enabled = **False**  *' Dead End Remover Button*  deadEndRemoverBtn.BackColor = Color.FromArgb(0, 73, 73)  deadEndRemoverBtn.Enabled = **False**  *' Download Button*  downloadBtn.BackColor = Color.FromArgb(19, 28, 40)  downloadBtn.Enabled = **False**  *' Rest Button*  imageInputBtn.BackColor = Color.FromArgb(19, 28, 40)  imageInputBtn.Enabled = **False**  bgColourBtn.BackColor = Color.FromArgb(19, 28, 40)  bgColourBtn.Enabled = **False**  mazeColourBtn.BackColor = Color.FromArgb(19, 28, 40)  mazeColourBtn.Enabled = **False**  solveColourBtn.BackColor = Color.FromArgb(19, 28, 40)  solveColourBtn.Enabled = **False**  cancelAnimationBtn.BackColor = Color.MediumSlateBlue  cancelAnimationBtn.Enabled = **True**  *' Rest TextBoxs*  widthTxtBox.BackColor = Color.FromArgb(19, 28, 40)  widthTxtBox.Enabled = **False**  heightTxtBox.BackColor = Color.FromArgb(19, 28, 40)  heightTxtBox.Enabled = **False**  deadEndRemoverTxtBox.BackColor = Color.FromArgb(19, 28, 40)  deadEndRemoverTxtBox.Enabled = **False**  *' Rest ComboBoxs*  generationCombo.BackColor = Color.FromArgb(19, 28, 40)  generationCombo.Update()  solveCombo.BackColor = Color.FromArgb(19, 28, 40)  solveCombo.Update()  mazeEntryCombo.BackColor = Color.FromArgb(19, 28, 40)  mazeEntryCombo.Update()  **Else**  *' Generate Button*  generateBtn.BackColor = Color.ForestGreen  generateBtn.Enabled = **True**  *' Solve Button*  solveBtn.BackColor = Color.Firebrick  solveBtn.Enabled = **True**  *' Dead End Remover Button*  deadEndRemoverBtn.BackColor = Color.DarkCyan  deadEndRemoverBtn.Enabled = **True**  *' Download Button*  downloadBtn.BackColor = Color.PaleVioletRed  downloadBtn.Enabled = **True**  *' Rest Button*  imageInputBtn.BackColor = Color.FromArgb(40, 60, 86)  imageInputBtn.Enabled = **True**  bgColourBtn.BackColor = Color.FromArgb(40, 60, 86)  bgColourBtn.Enabled = **True**  mazeColourBtn.BackColor = Color.FromArgb(40, 60, 86)  mazeColourBtn.Enabled = **True**  solveColourBtn.BackColor = Color.FromArgb(40, 60, 86)  solveColourBtn.Enabled = **True**  cancelAnimationBtn.BackColor = Color.DarkSlateBlue  cancelAnimationBtn.Enabled = **False**  *' Rest TextBoxs*  widthTxtBox.BackColor = Color.FromArgb(40, 60, 86)  widthTxtBox.Enabled = **True**  heightTxtBox.BackColor = Color.FromArgb(40, 60, 86)  heightTxtBox.Enabled = **True**  deadEndRemoverTxtBox.BackColor = Color.FromArgb(40, 60, 86)  deadEndRemoverTxtBox.Enabled = **True**  *' Rest ComboBoxs*  generationCombo.BackColor = Color.FromArgb(40, 60, 86)  generationCombo.Enabled = **True**  solveCombo.BackColor = Color.FromArgb(40, 60, 86)  solveCombo.Enabled = **True**  mazeEntryCombo.BackColor = Color.FromArgb(40, 60, 86)  mazeEntryCombo.Enabled = **True**  **End** **If**  **End** **Sub**  *' Interpolate between two colors based on a ratio (0.0 to 1.0)*  **Function** interpolateColour(color1 **As** Color, color2 **As** Color, ratio **As** Double) **As** Color  **Dim** r **As** Double = Int(color1.R) + (Int(color2.R) - Int(color1.R)) \* ratio  **Dim** g **As** Double = Int(color1.G) + (Int(color2.G) - Int(color1.G)) \* ratio  **Dim** b **As** Double = Int(color1.B) + (Int(color2.B) - Int(color1.B)) \* ratio  **Return** Color.FromArgb((r), (g), (b))  **End** **Function**  *' Generating*  **Private** **Sub** DFSbacktracker(**ByVal** x **As** Integer, **ByVal** y **As** Integer)  generationStack.Push(**New** Point(x, y))   |  | | --- | | Objective: B |     *' Check if they want animations*  **If** instantAnimationBtn.Checked = **False** **Then** *' Want Animations*  animationLock(**True**)  generationPointTimer.Enabled = **True**  **Else**  **Dim** direction **As** Integer    **While** generationStack.Count > 0  **Dim** currentCell = generationStack.Peek()  **Dim** cell = maze(currentCell.X, currentCell.Y)    *' Mark current cell as visited*  cell.visited = **True**    *' Get a list of unvisited neighbors*  **Dim** unvisitedNeighbors = cell.checkUnvistedNeighbours()    **If** unvisitedNeighbors.All(**Function**(p) p.Equals(Point.Empty)) = **True** **Then**  generationStack.Pop()  **Continue** **While**  **End** **If**    *' Make a new list that only contains the non empty values from neighbour*  **Dim** validNeigbours **As** **New** List(**Of** Point)  **For** **Each** point **In** unvisitedNeighbors  **If** point <> Point.Empty **Then**  validNeigbours.Add(point)  **End** **If**  **Next**    *' Randomly pick a valid neighbour. Find the index of that point within the orginal neighbour list and set that to direction*  Randomize()  direction = unvisitedNeighbors.IndexOf(validNeigbours(rnd.**Next**(0, validNeigbours.Count())))    *' Break the wall between the current cell and the chosen neighbor*  **Dim** randomNeighbor = cell.breakWall(direction)    *' Add the neighbor to the generationStack*  generationStack.Push(randomNeighbor)  **End** **While**  **End** **If**   |  | | --- | | Objective: G |   **End** **Sub**    **Public** **Sub** randomisedPrims(**ByVal** x **As** Integer, **ByVal** y **As** Integer)  *' Clear visitedCells and primsWalls to prepare for the new maze generation*  visitedCells.Clear()  primsWalls.Clear()    *' Start with an arbitrary cell*  **Dim** currentCell **As** Cell = maze(x, y)  currentCell.visited = **True**  visitedCells.Add(currentCell)    *' Add the walls of the initial cell to the list*  **For** i **As** Integer = 0 **To** 3  **If** **Not** currentCell.walls(i) **Then** **Continue** **For**  primsWalls.Add(Tuple.Create(**New** Point(x, y), i))  **Next**    *' Check if they want animations*  **If** instantAnimationBtn.Checked = **False** **Then** *' Want Animations*  animationLock(**True**)  generationPointTimer.Enabled = **True**  **Else**  *' Continue until all cells are visited and there are no more walls to process*  **While** visitedCells.Count < totalCells **AndAlso** primsWalls.Count > 0  *' Choose a wall connected to the visited cells uniformly at random*  **Dim** randomIndex **As** Integer = rnd.**Next**(primsWalls.Count)  **Dim** randomWall **As** Tuple(**Of** Point, Integer) = primsWalls(randomIndex)  **Dim** cellCoords **As** Point = randomWall.Item1  **Dim** direction **As** Integer = randomWall.Item2  **Dim** cell **As** Cell = maze(cellCoords.X, cellCoords.Y)  *' If the wall separates a visited cell from an unvisited cell*  **Dim** neighbour **As** Point = cell.checkUnvistedNeighbours()(direction)  **If** neighbour <> Point.Empty **AndAlso** **Not** maze(neighbour.X, neighbour.Y).visited **Then**  *' Remove the wall and mark the unvisited cell as visited*  cell.breakWall(direction)  maze(neighbour.X, neighbour.Y).visited = **True**  visitedCells.Add(maze(neighbour.X, neighbour.Y))  *' Add the neighboring walls of the cell to the walls list*  **For** i **As** Integer = 0 **To** 3  **If** **Not** maze(neighbour.X, neighbour.Y).walls(i) **Then** **Continue** **For**  primsWalls.Add(Tuple.Create(**New** Point(neighbour.X, neighbour.Y), i))  **Next**  **End** **If**    *' Remove the wall from the list to avoid reprocessing it*  primsWalls.RemoveAt(randomIndex)  **End** **While**  **End** **If**  **End** **Sub**    **Public** **Sub** kruskals()  *' Clear the kurskWalls list to prepare for the new maze generation*  kurskWalls.Clear()    *' Create a list of all possible walls in the maze*  **Dim** k **As** **New** List(**Of** Tuple(**Of** Point, Integer))  **For** x **As** Integer = 0 **To** width - 1  **For** y **As** Integer = 0 **To** height - 1  **For** i **As** Integer = 0 **To** 1  **If** maze(x, y).mazeWallBool = **False** **Then**  kurskWalls.Add(Tuple.Create(**New** Point(x, y), i))  **End** **If**  **Next**  **Next**  **Next**    *' Shuffle the list of walls randomly*  **For** i **As** Integer = kurskWalls.Count - 1 **To** 1 **Step** -1  **Dim** j **As** Integer = random.**Next**(i + 1)  **Dim** temp **As** Tuple(**Of** Point, Integer) = kurskWalls(i)  kurskWalls(i) = kurskWalls(j)  kurskWalls(j) = temp  **Next**     |  | | --- | | Objective: G |   *' Initialize the sets for each cell*  **Dim** setId **As** Integer = 0  **For** x **As** Integer = 0 **To** width - 1  **For** y **As** Integer = 0 **To** height - 1  kursSets(**New** Point(x, y)) = setId  setId += 1  **Next**  **Next**    *' Check if they want animations*  **If** instantAnimationBtn.Checked = **False** **Then** *' Want Animations*  animationLock(**True**)  generationPointTimer.Enabled = **True**  **Else**  *' Iterate through the shuffled list of walls*  **For** **Each** wall **As** Tuple(**Of** Point, Integer) **In** kurskWalls  **Dim** cellCoords **As** Point = wall.Item1  **Dim** direction **As** Integer = wall.Item2  **Dim** cell **As** Cell = maze(cellCoords.X, cellCoords.Y)  *' Calculate the neighboring cell coordinates based on the direction*  **If** direction = 0 **Then**  kursNeighbourCoords = **New** Point(cellCoords.X, cellCoords.Y - 1)  **Else**  kursNeighbourCoords = **New** Point(cellCoords.X + 1, cellCoords.Y)  **End** **If**    *' Check if the neighboring cell is within the maze bounds*  **If** kursNeighbourCoords.X >= 0 **AndAlso** kursNeighbourCoords.X < width **AndAlso** kursNeighbourCoords.Y >= 0 **AndAlso** kursNeighbourCoords.Y < height **Then**  *' Check if the cells connected by the wall are not in the same set*  **If** kursSets(cellCoords) <> kursSets(kursNeighbourCoords) **Then**  *' Remove the wall to connect the cells*  cell.breakWall(direction) *' Merge the sets of the two cells*  **Dim** setIdToReplace **As** Integer = kursSets(kursNeighbourCoords)  **Dim** setIdToKeep **As** Integer = kursSets(cellCoords)  **For** **Each** key **As** Point **In** kursSets.Keys.ToList()  **If** kursSets(key) = setIdToReplace **Then**  kursSets(key) = setIdToKeep  **End** **If**  **Next**  **End** **If**  **End** **If**  **Next**  kursCurrentWallIndex = 0  **End** **If**  **End** **Sub**    **Public** **Sub** aldousBroder(startX **As** Integer, startY **As** Integer)  abTotalCells = (width - 2) \* (height - 2)  currentX = startX  currentY = startY    **If** abHasBeenAnimated = **False** **Then**  abVisitedCells = 0  **End** **If**      *' Mark the starting cell as visited*  maze(currentX, currentY).visited = **True**  abVisitedCells += 1    *' Check if they want animations*  **If** instantAnimationBtn.Checked = **False** **Then** *' Want Animations*  animationLock(**True**)  generationPointTimer.Enabled = **True**  **Else**  *' Continue until all cells have been visited*  **While** abVisitedCells < abTotalCells  *' Move to a random neighboring cell*  **Dim** randomDirectionIndex **As** Integer = rnd.**Next**(directions.Length)  **Dim** newX **As** Integer = currentX + directions(randomDirectionIndex).X  **Dim** newY **As** Integer = currentY + directions(randomDirectionIndex).Y    *' Check if the new position is within the maze bounds, excluding the border cells*  **If** newX > 0 **AndAlso** newX < width - 1 **AndAlso** newY > 0 **AndAlso** newY < height - 1 **Then**  *' If the neighboring cell has not been visited yet, remove the wall between the current cell and the neighboring cell*  **If** **Not** maze(newX, newY).visited **Then**  maze(currentX, currentY).breakWall(randomDirectionIndex)  maze(newX, newY).visited = **True**   |  | | --- | | Objective: G |   abVisitedCells += 1  **End** **If**    *' Set the current position to the new position*  currentX = newX  currentY = newY  **End** **If**  **End** **While**  abHasBeenAnimated = **False**  **End** **If**  **End** **Sub**     |  | | --- | | Objective: F |   **Private** **Sub** generationPointTimer\_Tick(sender **As** Object, e **As** EventArgs)  *' Cancel animation if needed*  **If** cancelAnimation = **True** **Then**  generationPointTimer.Enabled = **False**  resetType = "G"  resetMaze()  animationLock(**False**)  cancelAnimation = **False**  **Exit** **Sub**  **End** **If**    **If** generationAlgorithm = "DFS Backtracker" **Then**  **If** generationStack.Count > 0 **Then**  **Dim** currentCell = generationStack.Peek()  **Dim** cell = maze(currentCell.X, currentCell.Y)    *' Highlight the top of the stack*  **If** currentCell <> mazeEntry **And** currentCell <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.Yellow), currentCell.X \* M, currentCell.Y \* M, M, M)  maze(currentCell.X, currentCell.Y).drawWalls()  **End** **If**      *' Mark current cell as visited*  cell.visited = **True**    *' Get a list of unvisited neighbors*  **Dim** unvisitedNeighbors = cell.checkUnvistedNeighbours()    **If** unvisitedNeighbors.All(**Function**(p) p.Equals(Point.Empty)) = **True** **Then**  generationStack.Pop()  **Else**  *' Make a new list that only contains the non empty values from neighbour*  **Dim** validNeigbours **As** **New** List(**Of** Point)  **For** **Each** point **In** unvisitedNeighbors  **If** point <> Point.Empty **Then**  validNeigbours.Add(point)  **End** **If**  **Next**    *' Randomly pick a valid neighbour. Find the index of that point within the orginal neighbour list and set that to direction*  **Dim** direction = unvisitedNeighbors.IndexOf(validNeigbours(rnd.**Next**(0, validNeigbours.Count())))    *' Break the wall between the current cell and the chosen neighbor*  **Dim** randomNeighbor = cell.breakWall(direction)    *' Add the neighbor to the stack*  generationStack.Push(randomNeighbor)  **End** **If**    *' Update the maze and the maze display*  mazeBox.Image = mazeImage   |  | | --- | | Objective: F |   mazeBox.Update()    *' Resrt the top of the stack*  **If** currentCell <> mazeEntry **And** currentCell <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.White), currentCell.X \* M, currentCell.Y \* M, M, M)  maze(currentCell.X, currentCell.Y).drawWalls()  **End** **If**  **Else**  drawMaze()  *' Update the maze and the maze display*  mazeBox.Image = mazeImage  mazeBox.Update()    animationLock(**False**)  *' Stop the timer when the maze is complete*  generationPointTimer.Enabled = **False**  **End** **If**  **ElseIf** generationAlgorithm = "Randomised Prims" **Then**  **If** visitedCells.Count < totalCells **AndAlso** primsWalls.Count > 0 **Then**  **Dim** randomIndex **As** Integer = rnd.**Next**(primsWalls.Count)  **Dim** randomWall **As** Tuple(**Of** Point, Integer) = primsWalls(randomIndex)  **Dim** cellCoords **As** Point = randomWall.Item1  **Dim** direction **As** Integer = randomWall.Item2  **Dim** cell **As** Cell = maze(cellCoords.X, cellCoords.Y)    *' Highlight all walls that could be collapsed*  **For** **Each** wall **As** Tuple(**Of** Point, Integer) **In** primsWalls  **If** wall.Item1 <> mazeEntry **And** wall.Item1 <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.Yellow), wall.Item1.X \* M, wall.Item1.Y \* M, M, M)  maze(wall.Item1.X, wall.Item1.Y).drawWalls()  **End** **If**  **Next**    mazeBox.Image = mazeImage  mazeBox.Update()  *' Rest all walls that could be collapsed*  **For** **Each** wall **As** Tuple(**Of** Point, Integer) **In** primsWalls  **If** wall.Item1 <> mazeEntry **And** wall.Item1 <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(bgColour), wall.Item1.X \* M, wall.Item1.Y \* M, M, M)  maze(wall.Item1.X, wall.Item1.Y).drawWalls()  **End** **If**  **Next**    **Dim** neighbour **As** Point = cell.checkUnvistedNeighbours()(direction)  **If** neighbour <> Point.Empty **AndAlso** **Not** maze(neighbour.X, neighbour.Y).visited **Then**  *' Remove the wall and mark the unvisited cell as visited*  **Dim** conCell **As** Point = cell.breakWall(direction) *' Merge the sets of the two cells*  **If** neighbour <> mazeEntry **And** neighbour <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(bgColour), neighbour.X \* M, neighbour.Y \* M, M, M)  maze(neighbour.X, neighbour.Y).drawWalls()  **End** **If**      *' Draw the cell background and fill*  **If** **New** Point(cell.x, cell.y) <> mazeEntry **And** **New** Point(cell.x, cell.y) <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(bgColour), cell.wallPos(0, 0).X, cell.wallPos(1, 0).Y, M, M)  cell.drawWalls()  **End** **If**  *' Update the maze and the maze display*  maze(neighbour.X, neighbour.Y).visited = **True**  visitedCells.Add(maze(neighbour.X, neighbour.Y))    *' Add the neighboring walls of the cell to the walls list*  **For** i **As** Integer = 0 **To** 3  **If** **Not** maze(neighbour.X, neighbour.Y).walls(i) **Then** **Continue** **For**  primsWalls.Add(Tuple.Create(**New** Point(neighbour.X, neighbour.Y), i))  **Next**  **End** **If**  *' Remove the wall from the list to avoid reprocessing it*  primsWalls.RemoveAt(randomIndex)  **Else**   |  | | --- | | Objective: F |   drawMaze()  *' Update the maze and the maze display*  mazeBox.Image = mazeImage  mazeBox.Update()    animationLock(**False**)  *' Stop the timer when the maze is complete*  generationPointTimer.Enabled = **False**  **End** **If**  **ElseIf** generationAlgorithm = "Kruskal 's" **Then**  **If** kursCurrentWallIndex < kurskWalls.Count **Then**  **Dim** wall **As** Tuple(**Of** Point, Integer) = kurskWalls(kursCurrentWallIndex)  **Dim** cellCoords **As** Point = wall.Item1  **Dim** direction **As** Integer = wall.Item2  **Dim** cell **As** Cell = maze(cellCoords.X, cellCoords.Y)  *' Calculate the neighboring cell coordinates based on the direction*  **If** direction = 0 **Then**  kursNeighbourCoords = **New** Point(cellCoords.X, cellCoords.Y - 1)  **Else**  kursNeighbourCoords = **New** Point(cellCoords.X + 1, cellCoords.Y)  **End** **If**    *' Check if the neighboring cell is within the maze bounds*  **If** kursNeighbourCoords.X >= 0 **AndAlso** kursNeighbourCoords.X < width **AndAlso** kursNeighbourCoords.Y >= 0 **AndAlso** kursNeighbourCoords.Y < height **Then**  *' Check if the cells connected by the wall are not in the same set*  **If** kursSets(cellCoords) <> kursSets(kursNeighbourCoords) **Then**  *' Remove the wall to connect the cells*  cell.breakWall(direction) *' Merge the sets of the two cells*  *' Remove the wall to connect the cells*  **Dim** conCell **As** Point = cell.breakWall(direction) *' Merge the sets of the two cells*  *' Draw the cell background and fill*  **If** **New** Point(cell.x, cell.y) <> mazeEntry **And** **New** Point(cell.x, cell.y) <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(bgColour), cell.wallPos(0, 0).X, cell.wallPos(1, 0).Y, M, M)  cell.drawWalls()  **End** **If**    **If** maze(conCell.X, conCell.Y).mazeWallBool = **False** **And** conCell <> mazeEntry **And** conCell <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(bgColour), maze(conCell.X, conCell.Y).wallPos(0, 0).X, maze(conCell.X, conCell.Y).wallPos(1, 0).Y, M, M)  maze(conCell.X, conCell.Y).drawWalls()  **End** **If**  mazeBox.Image = mazeImage  mazeBox.Update()    generationPointTimer.Enabled = **False**  **Dim** setIdToReplace **As** Integer = kursSets(kursNeighbourCoords)  **Dim** setIdToKeep **As** Integer = kursSets(cellCoords)  **For** **Each** key **As** Point **In** kursSets.Keys.ToList()  **If** kursSets(key) = setIdToReplace **Then**  kursSets(key) = setIdToKeep  **End** **If**  **Next**  **End** **If**  **End** **If**    *' Increment the wall index*  kursCurrentWallIndex += 1  generationPointTimer.Enabled = **True**  **Else**  *' Stop the timer when the algorithm is finished*  animationLock(**False**)  generationPointTimer.Enabled = **False**    **End** **If**  **ElseIf** generationAlgorithm = "Aldous-Border" **Then**  abHasBeenAnimated = **True**  *' Continue until all cells have been visited*  **If** abVisitedCells < abTotalCells **Then**  *' Move to a random neighboring cell*  **Dim** randomDirectionIndex **As** Integer = rnd.**Next**(directions.Length)  **Dim** newX **As** Integer = currentX + directions(randomDirectionIndex).X  **Dim** newY **As** Integer = currentY + directions(randomDirectionIndex).Y    *' Check if the new position is within the maze bounds, excluding the border cells*  **If** newX > 0 **AndAlso** newX < width - 1 **AndAlso** newY > 0 **AndAlso** newY < height - 1 **Then**  *' If the neighboring cell has not been visited yet, remove the wall between the current cell and the neighboring cell*  **If** **Not** maze(newX, newY).visited **Then**  **Dim** conCell **As** Point  conCell = maze(currentX, currentX).breakWall(randomDirectionIndex)    **If** **New** Point(currentX, currentX) <> mazeEntry **And** **New** Point(currentX, currentX) <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.Yellow), maze(currentX, currentX).wallPos(0, 0).X, maze(currentX, currentX).wallPos(1, 0).Y, M, M)  maze(currentX, currentX).drawWalls()  **End** **If**    **If** maze(conCell.X, conCell.Y).mazeWallBool = **False** **And** conCell <> mazeEntry **And** conCell <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(bgColour), maze(conCell.X, conCell.Y).wallPos(0, 0).X, maze(conCell.X, conCell.Y).wallPos(1, 0).Y, M, M)  maze(conCell.X, conCell.Y).drawWalls()  **End** **If**    mazeBox.Image = mazeImage  mazeBox.Update()    maze(newX, newY).visited = **True**  abVisitedCells += 1  **End** **If**  **If** **New** Point(currentX, currentX) <> mazeEntry **And** **New** Point(currentX, currentX) <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(bgColour), maze(currentX, currentX).wallPos(0, 0).X, maze(currentX, currentX).wallPos(1, 0).Y, M, M)  maze(currentX, currentX).drawWalls()  **End** **If**      *' Set the current position to the new position*  currentX = newX  currentY = newY  **End** **If**  **Else**  abHasBeenAnimated = **False**  *' Stop the timer when the algorithm is finished*  animationLock(**False**)  generationPointTimer.Enabled = **False**  **End** **If**  **End** **If**  **End** **Sub**    *' Solving*  **Private** **Function** distanceCalc(a **As** Point, b **As** Point) **As** Double  **Return** Math.Abs(a.X - b.X) + Math.Abs(a.Y - b.Y)  **End** **Function**  **Private** **Sub** dijkstra()  *' Clear the gWeight dictionary and solvedVisted list*   |  | | --- | | Objective:C |   gWeights.Clear()  solvedVisited.Clear()    *' Initialize the parents dictionary and the priority queue*  **Dim** parents **As** **New** Dictionary(**Of** Point, Point)  **Dim** pQueue **As** **New** PriorityQueue(**Of** Double, Point)()    *' Set the weight of entry to 0 and enqeueu to the priority queue*  gWeights(mazeEntry) = 0  pQueue.Enqueue(0, mazeEntry)    *' Continue until the proprity queue is empty*  **While** **Not** pQueue.isEmpty()  *' Dequeue the node with the lowest weight from the priority queue*  **Dim** current **As** Point = pQueue.Dequeue()    *' Add the dequeued node to the solvedVisited queue*  solvedVisited.Enqueue(current)    *' Check for exit*  **If** current = mazeExit **Then**  **Exit** **While**  **End** **If**    *' Go through each connect neighbour of the current node*  **For** **Each** neighbour **In** maze(current.X, current.Y).connectedCell  *' Calculate weight of neighbour. In this to get to a connected node holds a weight of 1*  **Dim** weight **As** Double = gWeights(current) + 1    *' If the neighbour's weight is not already in the dictionary, set it to a large value*  **If** **Not** gWeights.ContainsKey(neighbour) **Then**  gWeights(neighbour) = Double.MaxValue  **End** **If**    *' Update the neighbours weight and parent if the calculated weight is less*  **If** weight < gWeights(neighbour) **Then**  gWeights(neighbour) = weight  maxWeight = Math.Max(maxWeight, weight)  parents(neighbour) = current    *' If the neight is not in the priority queue, add it*  **If** **Not** pQueue.Contains(neighbour) **Then**  pQueue.Enqueue(weight, neighbour)  **End** **If**  **End** **If**  **Next**  **End** **While**    *' Reconstruct the path*  **Dim** currentNode **As** Point = mazeExit  **While** currentNode <> mazeEntry **AndAlso** parents.ContainsKey(currentNode)  currentNode = parents(currentNode)  **If** currentNode <> mazeEntry **Then**  path.Enqueue(currentNode)  **End** **If**  **End** **While**    *' Check If they want animations*  **If** instantAnimationBtn.Checked = **True** **Or** cancelAnimation = **True** **Then**  *' Mark the path as solved*  **For** **Each** node **In** path  maze(node.X, node.Y).mazeSolved = **True**  **Next**  path.Clear()  **ElseIf** instantAnimationBtn.Checked = **False** **Then**  *' Enable animations and lock other controls*  animationLock(**True**)  heatMapAnimationTimer.Enabled = **True**  **End** **If**  **End** **Sub**     |  | | --- | | Objective: H |   **Private** **Sub** BFS()  *' Clear solvedVisted list and branchingPoints list*  solvedVisited.Clear()  branchingPoints.Clear()    *' Initialize the queue, parent dictionary and currentNod*  **Dim** queue **As** **New** Queue(**Of** Point)  **Dim** parents **As** **New** Dictionary(**Of** Point, Point)()  **Dim** currentNode **As** Point    *' Enqueue the starting point*  queue.Enqueue(mazeEntry)    *' Continue searching until the queue is empty*  **While** queue.Count <> 0  *' Dequeue the next code in the queue*  currentNode = queue.Dequeue()    *' Check for the exit*  **If** currentNode = mazeExit **Then**  **Exit** **While**  **End** **If**   |  | | --- | | Objective: H |     *' Go through each connected cell of currentNode*  **For** **Each** point **In** maze(currentNode.X, currentNode.Y).connectedCell  **If** **Not** solvedVisited.Contains(point) **Then**  solvedVisited.Enqueue(point)  parents(point) = currentNode  queue.Enqueue(point)  **End** **If**  **Next**    **If** maze(currentNode.X, currentNode.Y).connectedCell.Count > 2 **Then**  branchingPoints.Add(currentNode)  **End** **If**  **End** **While**    *' Reconstruct the path*  currentNode = mazeExit  **While** currentNode <> mazeEntry **AndAlso** parents.ContainsKey(currentNode)  currentNode = parents(currentNode)  **If** currentNode <> mazeEntry **Then**  path.Enqueue(currentNode)  **End** **If**  **End** **While**    *' Check if they want animations*  **If** instantAnimationBtn.Checked = **True** **Or** cancelAnimation = **True** **Then**  *' Mark the path as solved*  **For** **Each** node **In** path  maze(node.X, node.Y).mazeSolved = **True**  **Next**  path.Clear()  **ElseIf** instantAnimationBtn.Checked = **False** **Then**  *' Enable animations and lock other controls*  animationLock(**True**)  heatMapAnimationTimer.Enabled = **True**  **End** **If**  **End** **Sub**  **Private** **Sub** Astar(**Optional** **ByVal** helper **As** Boolean = **False**)  *' Clear the gWeight dictionary and solvedVisted list*  gWeights.Clear()  solvedVisited.Clear()    *' Initailize the parent dictionary and priority queue for open nodes*  **Dim** parents **As** **New** Dictionary(**Of** Point, Point)  **Dim** pQueue **As** **New** PriorityQueue(**Of** Double, Point)()    *' Set the starting points gWeight to 0 and enqueue it with its heuristic value*  gWeights(mazeEntry) = 0  pQueue.Enqueue(distanceCalc(mazeEntry, mazeExit), mazeEntry)    *' Continue searching until the priority queue is empty*  **While** **Not** pQueue.isEmpty()  *' Dequeue the node with the lowest*  **Dim** current **As** Point = pQueue.Dequeue()    *' Add the dequeued node to the visited nodes queue*  solvedVisited.Enqueue(current)    *' Check for exit node*  **If** current = mazeExit **Then**  **Exit** **While**  **End** **If**    *' Go through each connected node of the current node*  **For** **Each** neighbour **In** maze(current.X, current.Y).connectedCell  *' Calculate heuristic weight*  **Dim** heuristicWeight **As** Double = gWeights(current) + distanceCalc(current, neighbour)    *' Set the neighbour's gWeight to a large value if its not in gWeights*  **If** **Not** gWeights.ContainsKey(neighbour) **Then**  gWeights(neighbour) = Double.MaxValue  **End** **If**    *' Update the neighbours gWeight and parent if the heuristic weight is lower*  **If** heuristicWeight < gWeights(neighbour) **Then**  parents(neighbour) = current   |  | | --- | | Objective: H |   gWeights(neighbour) = heuristicWeight  maxWeight = Math.Max(maxWeight, heuristicWeight)  **Dim** fWeight **As** Double = gWeights(neighbour) + distanceCalc(neighbour, mazeExit)    *' If the neighbour is not in the priority queue, add it with the calculated fWeight*  **If** **Not** pQueue.Contains(neighbour) **Then**  pQueue.Enqueue(fWeight, neighbour)  **End** **If**  **End** **If**  **Next**  **End** **While**    *' Reconstruct the path*  **Dim** currentNode **As** Point = mazeExit  **While** currentNode <> mazeEntry **AndAlso** parents.ContainsKey(currentNode)  currentNode = parents(currentNode)  **If** currentNode <> mazeEntry **Then**  path.Enqueue(currentNode)  **End** **If**  **End** **While**    **If** helper = **True** **Then**  helperPath = **New** Queue(**Of** Point)(path)  path.Clear()  **Else**  *' Check if they want animations*  **If** instantAnimationBtn.Checked = **True** **Or** cancelAnimation = **True** **Then**  *' Mark the path as solved*  **For** **Each** node **In** path  maze(node.X, node.Y).mazeSolved = **True**  **Next**  path.Clear()  **ElseIf** instantAnimationBtn.Checked = **False** **Then**  *' Enable animations and lock other controls*  animationLock(**True**)  heatMapAnimationTimer.Enabled = **True**  **End** **If**  **End** **If**  **End** **Sub**  **Private** **Sub** wallFollower(type **As** String)  *' Initailize the current node to the entry*  **Dim** node **As** Point = mazeEntry  *' Initailize the direction queue and index for left/right-hand rule navigation*  **Dim** directionQueue **As** **New** CircularQueue(**Of** Integer)({0, 1, 2, 3})  **Dim** index **As** Integer    *' If the instant animation is unchecked, call A\* for the most effecient path*  **If** instantAnimationBtn.Checked = **False** **Then**  Astar(helper:=**True**)  **End** **If**    *' Continue until the maze exit is reached*  **While** node <> mazeExit  *' Determine next direction base on wall follower type*  **If** type = "LHR" **Then**  index = directionQueue.turnLeft  **ElseIf** type = "RHR" **Then**  index = directionQueue.turnRight  **End** **If**    *' Check if the next cell is connected in the given direction*  **If** maze(node.X, node.Y).checkConnectedCell(index) <> Point.Empty **Then**  node = maze(node.X, node.Y).checkConnectedCell(index)  **Else**  *' If the next cell is not connected, rotate until a connected cell in found*  **Do**  **If** type = "LHR" **Then**  index = directionQueue.turnRight  **ElseIf** type = "RHR" **Then**  index = directionQueue.turnLeft  **End** **If**  **Loop** Until maze(node.X, node.Y).checkConnectedCell(index) <> Point.Empty     |  | | --- | | Objective: H |   *' Move to the connected cell*  node = maze(node.X, node.Y).checkConnectedCell(index)  **End** **If**    *' Dont add entry or exit to the path*  **If** node <> mazeEntry **And** node <> mazeExit **Then**  path.Enqueue(node)  **End** **If**  **End** **While**    *' Check if they want animations*  **If** instantAnimationBtn.Checked = **True** **Or** cancelAnimation = **True** **Then**  *' Mark the path as solved*  **For** **Each** node **In** path  maze(node.X, node.Y).mazeSolved = **True**  **Next**  path.Clear()  **ElseIf** instantAnimationBtn.Checked = **False** **Then**  *' Enable animations and lock other controls*  animationLock(**True**)  heatMapAnimationTimer.Enabled = **True**  **End** **If**  **End** **Sub**       |  | | --- | | Objective: F |   *' Animation*  **Private** **Sub** heatMapAnimationTimer\_Tick(sender **As** Object, e **As** EventArgs)  *' Cancel animation if needed*  **If** cancelAnimation = **True** **Then**  heatMapAnimationTimer.Enabled = **False**  solvedPathAnimationTimer.Enabled = **False**  resetType = "S"  resetMaze()  animationLock(**False**)  cancelAnimation = **False**  **Exit** **Sub**  **End** **If**    *' Creating the heat map*  **If** solveAlgorithm = "A\*" **Or** solveAlgorithm = "Dijkstra's" **Then**  *' If there are nodes in the solvedVisited*  **If** solvedVisited.Count > 0 **Then**  **Dim** node **As** Point = solvedVisited.Dequeue  *' If the node is not the entry or exit*  **If** node <> mazeEntry **And** node <> mazeExit **Then**  *' Calculate the normalized weight and draw heat map*  **Dim** normalisedWeight **As** Double = gWeights(node) / maxWeight  mazeImageGraphics.FillRectangle(**New** SolidBrush(interpolateColour(pinkColor, purpleColor, normalisedWeight)), node.X \* M, node.Y \* M, M, M)    *' Draws walls and update the maze box*  maze(node.X, node.Y).drawWalls()  mazeBox.Image = mazeImage  mazeBox.Update()  **End** **If**  **Else**  *' Clear the solvedVisited queue, draw the maze, and enable the solvedPathAnimationTimer*  solvedVisited.Clear()  drawMaze()  heatMapAnimationTimer.Enabled = **False**  solvedPathAnimationTimer.Enabled = **True**  **End** **If**  **ElseIf** solveAlgorithm = "Wall Follower LHR" **Or** solveAlgorithm = "Wall Follower RHR" **Then**  *' If there are nodes in path*  **If** path.Count > 0 **Then**  **Dim** node **As** Point = path.Dequeue    *' Draw the previous cell in the solved path colour or aqua colour*  **If** passedPath.Count > 0 **Then**  **Dim** previousPath **As** Point = passedPath.Last()  **If** helperPath.Contains(previousPath) **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(solveColour), previousPath.X \* M, previousPath.Y \* M, M, M)  **Else**  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.Aqua), previousPath.X \* M, previousPath.Y \* M, M, M)  **End** **If**  maze(previousPath.X, previousPath.Y).mazeSolved = **True**   |  | | --- | | Objective: F |   maze(previousPath.X, previousPath.Y).drawWalls()  **End** **If**    *' Draw the current cell in the yellow color, draw the walls, and update the maze display*  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.Yellow), node.X \* M, node.Y \* M, M, M)  maze(node.X, node.Y).drawWalls()  passedPath.Add(node)  *' Updates Maze box*  mazeBox.Image = mazeImage  mazeBox.Update()  **Else**  *' Draw the last cell in the solved path color,draw the walls, and update the maze display*  **If** passedPath.Count > 0 **Then**  **Dim** lastPath **As** Point = passedPath.Last()  mazeImageGraphics.FillRectangle(**New** SolidBrush(solveColour), lastPath.X \* M, lastPath.Y \* M, M, M)  maze(lastPath.X, lastPath.Y).drawWalls()  maze(lastPath.X, lastPath.Y).mazeSolved = **True**  drawMaze()  mazeBox.Image = mazeImage  mazeBox.Update()  **End** **If**  *' Clear the helperPath, path, and passedPath queues, disable the timers, and unlock the animation*  helperPath.Clear()  path.Clear()  passedPath.Clear()  animationLock(**False**)  heatMapAnimationTimer.Enabled = **False**  solvedPathAnimationTimer.Enabled = **False**  **End** **If**  **ElseIf** solveAlgorithm = "Breath Frist Search" **Then**  *' If there are nodes in the solvedVisited queue*  **If** solvedVisited.Count > 0 **Then**  **Dim** node **As** Point = solvedVisited.Dequeue  *' If the node is not entry or exit*  **If** node <> mazeEntry **And** node <> mazeExit **Then**  *' Draw the node in yellow if it's a branching point or the solveColour otherwise*  **If** branchingPoints.Contains(node) **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.Yellow), node.X \* M, node.Y \* M, M, M)  **Else**  mazeImageGraphics.FillRectangle(**New** SolidBrush(solveColour), node.X \* M, node.Y \* M, M, M)  **End** **If**  *'Draw the walls And update the maze display*  maze(node.X, node.Y).drawWalls()  mazeBox.Image = mazeImage  mazeBox.Update()  **End** **If**  **Else**  *' Clear the solvedVisited queue, draw the maze, and enable the solvedPathAnimationTimer*  solvedVisited.Clear()  drawMaze()  solvedPathAnimationTimer.Enabled = **True**  heatMapAnimationTimer.Enabled = **False**  **End** **If**  **End** **If**  **End** **Sub**  **Private** **Sub** solvedPathAnimationTimer\_Tick(sender **As** Object, e **As** EventArgs)  *' Cancel animation if needed*  **If** cancelAnimation = **True** **Then**  heatMapAnimationTimer.Enabled = **False**  solvedPathAnimationTimer.Enabled = **False**  resetType = "S"  resetMaze()  animationLock(**False**)  cancelAnimation = **False**  **Exit** **Sub**  **End** **If**  *' Creating the Solved Path*  **If** path.Count > 0 **Then**  **Dim** currentPath **As** Point = path.Dequeue     |  | | --- | | Objective: F |   *' Draw the previous cell in the solved path color*  **If** passedPath.Count > 0 **Then**  **Dim** previousPath **As** Point = passedPath.Last()  mazeImageGraphics.FillRectangle(**New** SolidBrush(solveColour), previousPath.X \* M, previousPath.Y \* M, M, M)  maze(previousPath.X, previousPath.Y).drawWalls()  **End** **If**    *' Draw the current cell in the yellow color*  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.Yellow), currentPath.X \* M, currentPath.Y \* M, M, M)  maze(currentPath.X, currentPath.Y).drawWalls()    *' Add the current cell to the passedPath list and mark it as solved*  passedPath.Add(currentPath)  maze(currentPath.X, currentPath.Y).mazeSolved = **True**    *' Updates maze box*  mazeBox.Image = mazeImage  mazeBox.Update()  **Else**  *' If there are no more cells in the path, draw the last cell in the solved path*  **If** passedPath.Count > 0 **Then**  **Dim** lastPath **As** Point = passedPath.Last()  mazeImageGraphics.FillRectangle(**New** SolidBrush(solveColour), lastPath.X \* M, lastPath.Y \* M, M, M)  maze(lastPath.X, lastPath.Y).drawWalls()    *' Updates maze box*  mazeBox.Image = mazeImage  mazeBox.Update()  **End** **If**    *' Disable the solvedPathAnimationTimer, unlock the animation, and clear the path and passedPath list*  solvedPathAnimationTimer.Enabled = **False**  animationLock(**False**)  path.Clear()  passedPath.Clear()  **End** **If**  **End** **Sub**     |  | | --- | | Objective: J |   **Private** **Sub** deadEndRemover()  **Dim** numToBeRemoved **As** Integer  **Dim** deadEnd **As** Point  **Dim** node **As** Point  **Dim** direction **As** Integer  *' Find the deadends*  **For** **Each** cell **In** maze  cell.deadEndFinder()  **Next**  *' Calculate the amount of dead end to remove*  numToBeRemoved = Math.Round(deadEndPos.Count() \* deadEndPercent)  **Dim** removed **As** Integer = 0  **While** removed <> numToBeRemoved  *' Pick a random cell*  deadEnd = deadEndPos(rnd.**Next**(0, deadEndPos.Count))  *' Finds valid indexs*  **Dim** validIdexs **As** **New** List(**Of** Integer)  **For** i **As** Integer = 0 **To** 3  **If** maze(deadEnd.X, deadEnd.Y).walls(i) **Then**  validIdexs.Add(i)  **End** **If**  **Next**  *' Pick a valid wall to break*  **Do**  direction = validIdexs(rnd.**Next**(0, validIdexs.Count))  node = maze(deadEnd.X, deadEnd.Y).breakWall(direction)  **Loop** **While** node.IsEmpty  *' Removes from dead end list as changes the number of deadends removed.*  deadEndPos.Remove(deadEnd)  **If** deadEndPos.Contains(node) **Then**  deadEndPos.Remove(node)  removed += 1  **End** **If**  removed += 1  **End** **While**  deadEndPos.Clear()  animationLock(**False**)  **End** **Sub**  *' USER INPUT START*  **Private** **Sub** generateBtn\_Click(sender **As** Object, e **As** EventArgs)  *' Saves Maze Properties inputted by the user*  *' Checking that the values inputed for width and height are valid*  **If** Integer.TryParse(widthTxtBox.Text, width) **And** width > 2 **And** Integer.TryParse(heightTxtBox.Text, height) **And** height > 2 **Then**  width -= 1  height -= 1  **Else**  MsgBox("Make sure width and height are integers greater than 3", MsgBoxStyle.OkOnly, "Invalid Input")  **Exit** **Sub**  **End** **If**    mazeEntryType = mazeEntryCombo.Text  generationAlgorithm = generationCombo.Text      *' Changes multiplier value depending on the maze size*  **If** Math.Floor(Math.Min(1220 / Int(widthTxtBox.Text), 690 / Int(heightTxtBox.Text))) < 3 **Then**  downlaodGenerated = MsgBox("Maze is too big to display!" & vbCrLf & "Want to download?" & vbCrLf & "WARNING! DEPENDING ON HARDWARE THIS MAY TAKE A LONG TIME", MsgBoxStyle.OkCancel, "Maze too big!")  *' If they want to download the maze will generate, statistics will be show and the maze will be downloaded*  **If** downlaodGenerated = DialogResult.OK **Then**  instantAnimationBtn.Checked = **True**  M = 3  **Else**  *' If they dont wan to download it will reset statistics and exit the sub*  genTimeLbl.Text = "Generation Time: "  solveTimeLbl.Text = "Sove Time: "  drawTimeLbl.Text = "Draw Time: "  deadEndCountLbl.Text = "Dead End Count: "  deadEndTimeLbl.Text = "Dead End Time: "  totalTimeLbl.Text = "Total Time "  **Exit** **Sub**  **End** **If**  **Else**  *' If the multipier is below 3 that means it can be displayed on the form*  M = Math.Floor(Math.Min(1220 / Int(widthTxtBox.Text), 690 / Int(heightTxtBox.Text)))  **End** **If**    *' Intializes the maze*  initializeMaze()  *' Allows program to know whether or not a maze has been generated*  mazeGenerated = **True**    *' Resets old timer, Starts new timer, Upates Status*  statusLbl.Text = "Status: Generating"  statusLbl.Update()  generationTimer.Reset()  generationTimer.Start()    *' Checks what generation algorithm user has chosen*  **If** generationAlgorithm = "DFS Backtracker" **Then**  **If** imageInputted **Then**   |  | | --- | | Objective: B |   **For** **Each** component **In** imgComponents  **Dim** point **As** Point = component(rnd.**Next**(0, component.Count))  DFSbacktracker(point.X, point.Y)  **Next**  **Else**  DFSbacktracker(rnd.**Next**(1, width), rnd.**Next**(1, height))  **End** **If**  **ElseIf** generationAlgorithm = "Randomised Prims" **Then**  **If** imageInputted **Then**  **For** **Each** component **In** imgComponents  **Dim** point **As** Point = component(rnd.**Next**(0, component.Count))  randomisedPrims(point.X, point.Y)  **Next**  **Else**  randomisedPrims(rnd.**Next**(1, width), rnd.**Next**(1, height))  **End** **If**  **ElseIf** generationAlgorithm = "Kruskal 's" **Then**  **If** imageInputted **Then**  MsgBox("You can't use this algorithm with mazes." & vbCrLf & "Try another one!", MsgBoxStyle.OkOnly, "Invalid Input")  **Exit** **Sub**  **Else**  kruskals()  **End** **If**  **ElseIf** generationAlgorithm = "Aldous-Border" **Then**  **If** imageInputted **Then**  MsgBox("You can't use this algorithm with mazes." & vbCrLf & "Try another one!", MsgBoxStyle.OkOnly, "Invalid Input")  **Exit** **Sub**  **Else**  width += 1  height += 1  aldousBroder(rnd.**Next**(1, width), rnd.**Next**(1, height))  width -= 1  height -= 1  **End** **If**  **End** **If**    generationTimer.**Stop**()  *' Draws generate maze*  drawMaze()    *' Checks if it should download the maze*  **If** downlaodGenerated = DialogResult.OK **Then**  downloadMaze()  **Else**  *' Upadtes Maze box*  mazeBox.Image = mazeImage  **End** **If**    *' Updates Statistics*  *' Find the dead end count*  **For** **Each** cell **In** maze  cell.deadEndFinder()  **Next**  deadEndToShow = deadEndPos.Count()  deadEndPos.Clear()   |  | | --- | | Objective: L |   *' Displays Statistics*  genTimeLbl.Text = "Generation Time: " & Str(generationTimer.ElapsedMilliseconds() / 1000) & "s"  solveTimeLbl.Text = "Sove Time: "  drawTimeLbl.Text = "Draw Time: " & Str(drawTimer.ElapsedMilliseconds() / 1000) & "s"  deadEndCountLbl.Text = "Dead End Count: " & Str(deadEndToShow)  deadEndTimeLbl.Text = "Dead End Time: "  totalTimeLbl.Text = "Total Time " & Str((generationTimer.ElapsedMilliseconds() + solveTimer.ElapsedMilliseconds() + drawTimer.ElapsedMilliseconds() + deadEndTimer.ElapsedMilliseconds()) / 1000) & "s"  *' Resets Status, ' Resets Dialog Result*  statusLbl.Text = "Status: Doing Nothing"  downlaodGenerated = DialogResult.Cancel  **End** **Sub**  **Private** **Sub** solveBtn\_Click(sender **As** Object, e **As** EventArgs)  *' Makes sure a maze has been generated*  **If** mazeGenerated = **False** **Then**  MsgBox("No maze generated!" & vbCrLf & "Please press the generate button", MsgBoxStyle.OkOnly, "No maze generated")  **Exit** **Sub**  **End** **If**  *' Sets solving algorithim to what the user has selected*  solveAlgorithm = solveCombo.Text    *' Reset all cells that have .mazeSolved = True*  **For** **Each** cell **In** maze  cell.mazeSolved = **False**  **Next**    *' Checks if the maze can be displayed*  **If** Math.Floor(Math.Min(1220 / Int(widthTxtBox.Text), 690 / Int(heightTxtBox.Text))) < 3 **Then**  *' If it cant be displayed ask if they want to download*  downlaodSolved = MsgBox("Maze is too big to display!" & vbCrLf & "Want to download?" & vbCrLf & "WARNING! DEPENDING ON HARDWARE THIS MAY TAKE A LONG TIME", MsgBoxStyle.OkCancel, "Maze too big!")  *' If they don't want to download exit sub*  **If** downlaodSolved = DialogResult.Cancel **Then**  **Exit** **Sub**  **End** **If**  **End** **If**   |  | | --- | | This code will find the value of the maze multiplier (m), if m is less than 3, they will be prompted to download the maze. This done to keep the mazes displayed on the form looking good |   *' Resets old timer, Starts new timer, Upates Status*  solveTimer.Reset()  solveTimer.Start()  statusLbl.Text = "Status: Solving"  statusLbl.Update()  *' Checks what solving algorithm user has chosen*  **If** solveAlgorithm = "Dijkstra's" **Then**  dijkstra()  **ElseIf** solveAlgorithm = "Breath Frist Search" **Then**  BFS()  **ElseIf** solveAlgorithm = "A\*" **Then**  Astar()  **ElseIf** solveAlgorithm = "Wall Follower LHR" **Then**  wallFollower("LHR")  **ElseIf** solveAlgorithm = "Wall Follower RHR" **Then**  wallFollower("RHR")  **End** **If**  solveTimer.**Stop**()  *' Upadtes Maze box*  drawMaze()    *' They want to download the solved image*  **If** downlaodSolved = DialogResult.OK **Then**  downloadMaze()  **Else**  *' They dont want to download the solved image*  *' Check if it's drawable*  **If** Math.Floor(Math.Min(1220 / Int(widthTxtBox.Text), 690 / Int(heightTxtBox.Text))) >= 3 **Then**  mazeBox.Image = mazeImage  **End** **If**  **End** **If**   |  | | --- | | Objective: L |     *' Displays Statistics*  solveTimeLbl.Text = "Sove Time: " & Str(solveTimer.ElapsedMilliseconds() / 1000) & "s"  totalTimeLbl.Text = "Total Time " & Str((generationTimer.ElapsedMilliseconds() + solveTimer.ElapsedMilliseconds() + drawTimer.ElapsedMilliseconds() + deadEndTimer.ElapsedMilliseconds()) / 1000) & "s"  *' Resets Status*  statusLbl.Text = "Status: Doing Nothing"  **End** **Sub**  **Private** **Sub** deadEndRemoverBtn\_Click(sender **As** Object, e **As** EventArgs)  *' Makes sure a maze has been generated*  **If** mazeGenerated = **False** **Then**  MsgBox("No maze generated!" & vbCrLf & "Please press the generate button", MsgBoxStyle.OkOnly, "No maze generated")  **Exit** **Sub**  **End** **If**    *' Saves the inputted percentage*  *' Checking that the values inputted for dead end remover is valid*  **If** Double.TryParse(deadEndRemoverTxtBox.Text, deadEndPercent) **And** deadEndPercent <= 1.0 **Then**  deadEndPercent = deadEndRemoverTxtBox.Text  **Else**  MsgBox("Make sure dead end remover is a decimal number or 1", MsgBoxStyle.OkOnly, "Invalid Input")  **Exit** **Sub**  **End** **If**    *' Resets old timer, Starts new timer, Upates Status*  deadEndTimer.Reset()  deadEndTimer.Start()  statusLbl.Text = "Status: Removing Dead Ends"  statusLbl.Update()    *' Removes dead ends*  deadEndRemover()  deadEndTimer.**Stop**()  *' Removes the solved value for each cell*  **For** **Each** cell **In** maze  cell.mazeSolved = **False**  **Next**    *' Upadtes Maze box*  drawMaze()  mazeBox.Image = mazeImage    *' Find the dead end count*  **For** **Each** cell **In** maze  cell.deadEndFinder()  **Next**  deadEndToShow = deadEndPos.Count()  deadEndPos.Clear()   |  | | --- | | Objective: L |   *' Displays Statistics*  solveTimeLbl.Text = "Sove Time: "  drawTimeLbl.Text = "Draw Time: " & Str(drawTimer.ElapsedMilliseconds() / 1000) & "s"  deadEndCountLbl.Text = "Dead End Count: " & Str(deadEndToShow)  deadEndTimeLbl.Text = "Dead End Time: " & Str(deadEndTimer.ElapsedMilliseconds() / 1000) & "s"  totalTimeLbl.Text = "Total Time " & Str((generationTimer.ElapsedMilliseconds() + solveTimer.ElapsedMilliseconds() + drawTimer.ElapsedMilliseconds() + deadEndTimer.ElapsedMilliseconds()) / 1000) & "s"  *' Resets Status*  statusLbl.Text = "Status: Doing Nothing"    **End** **Sub**   |  | | --- | | Objective: D |   *' Setting Colour Customisation*  **Private** **Function** selectColour() **As** Color *' Opens a colour picker and returns the selected colour*  colorDialog.ShowDialog() *' Opens colour picker*  **Return** colorDialog.Color *' Returns picked colour*  **End** **Function**   |  | | --- | | Objective: E |   **Private** **Sub** downloadMaze()  **If** mazeGenerated = **True** **Then**  **Dim** openFile **As** **New** SaveFileDialog  openFile.FileName = **Nothing**  openFile.Filter = "JPG File's |\*.jpg"  openFile.ShowDialog()  **Try**  mazeImage.Save(openFile.FileName)  **Catch** ex **As** Exception  *' They didn't select a file location*  **End** **Try**  **Else**  MsgBox("No maze generated!" & vbCrLf & "Please press the generate button", MsgBoxStyle.OkOnly, "No maze generated")  **End** **If**  **End** **Sub**  **Private** **Sub** bgColourBtn\_Click(sender **As** Object, e **As** EventArgs)  bgColour = selectColour() *' Selects background colour*  bgColourBtn.Text = bgColour.ToString  **End** **Sub**  **Private** **Sub** mazeColourBtn\_Click(sender **As** Object, e **As** EventArgs)  mazeColour = selectColour() *' Selects maze colour*  mazeColourBtn.Text = mazeColour.ToString  **End** **Sub**  **Private** **Sub** solveColourBtn\_Click(sender **As** Object, e **As** EventArgs)  solveColour = selectColour() *' Selects solve colour*  solveColourBtn.Text = solveColour.ToString  **End** **Sub**  **Private** **Sub** downloadBtn\_Click(sender **As** Object, e **As** EventArgs)  downloadMaze()  **End** **Sub**  **Private** **Function** componentAnalysis(**ByVal** image **As** Bitmap) **As** List(**Of** List(**Of** Point))  largestComponent.Clear()  *' Create a list to store the components*  **Dim** components **As** **New** List(**Of** List(**Of** Point))()  *' Create a 2D array to track which pixels have been visited*  **Dim** visited(image.Width - 1, image.Height - 1) **As** Boolean  *' Loop through each pixel in the image*  **For** y **As** Integer = 0 **To** image.Height - 1  **For** x **As** Integer = 0 **To** image.Width - 1  *' If this pixel has not been visted and its white(255,255,255)*  **If** **Not** visited(x, y) **And** image.GetPixel(x, y) = Color.FromArgb(255, 255, 255) **Then**  *' Create a list to store the pixels in the component*  **Dim** component **As** **New** List(**Of** Point)()  *' Create a generationStack to store the pixels that need to be checked*  **Dim** generationStack **As** **New** Stack(**Of** Point)()  generationStack.Push(**New** Point(x, y))  *' Until all pixels have been checked*  **While** generationStack.Count > 0  *' Store the current pixel*  **Dim** pixel **As** Point = generationStack.Pop()  *' If this pixel has not been visted and is white*  **If** **Not** visited(pixel.X, pixel.Y) **And** image.GetPixel(pixel.X, pixel.Y) = Color.FromArgb(255, 255, 255) **Then**  visited(pixel.X, pixel.Y) = **True**  *' Add to the componet*  component.Add(pixel)  *' Add neighbours to the generationStack*  **If** pixel.X > 0 **Then**  generationStack.Push(**New** Point(pixel.X - 1, pixel.Y))  **End** **If**  **If** pixel.X < image.Width - 1 **Then**  generationStack.Push(**New** Point(pixel.X + 1, pixel.Y))  **End** **If**  **If** pixel.Y > 0 **Then**  generationStack.Push(**New** Point(pixel.X, pixel.Y - 1))  **End** **If**  **If** pixel.Y < image.Height - 1 **Then**  generationStack.Push(**New** Point(pixel.X, pixel.Y + 1))  **End** **If**  **End** **If**  **End** **While**  components.Add(component)   |  | | --- | | This while loop will put the correct cells into the correct components list. Using the generationStack it will check the neighbours. Those neighbours will then be put into consideration to see what component list they should be in. |   **End** **If**  **Next**  **Next**  *' Find the largest component*  **For** **Each** component **As** List(**Of** Point) **In** components  **If** component.Count > largestComponent.Count **Then**  largestComponent = component  **End** **If**  **Next**  **Return** components  **End** **Function**    **Private** **Sub** imageInputBtn\_Click(sender **As** Object, e **As** EventArgs)  *' Rest variable*  **If** imageInputted = **True** **Then**  inputImage.Dispose()  mazeWallList.Clear()  imgComponents.Clear()  mazeBox.Image = **Nothing**  widthTxtBox.Text = 0  heightTxtBox.Text = 0  imageInputted = **False**  imageInputBtn.Text = "Input Image"  **Exit** **Sub**  **End** **If**   |  | | --- | | Objective: M |   *' Requests and store image in memory*  openFileDialog1.FileName = ""  openFileDialog1.Filter = "JPG Files(\*.jpg)|\*.jpg"  **If** openFileDialog1.ShowDialog = Windows.Forms.DialogResult.OK **Then**  inputImage = **New** Bitmap(Image.FromFile(openFileDialog1.FileName))  *' Calculate the new image dimensions*  **Dim** newWidth **As** Integer = inputImage.Width + 2  **Dim** newHeight **As** Integer = inputImage.Height + 2  *' Create the new bitmap with the larger dimensions*  **Dim** newImage **As** **New** Bitmap(newWidth, newHeight)  *' Draw the original image onto the new bitmap*  **Using** g **As** Graphics = Graphics.FromImage(newImage)  g.DrawImage(inputImage, **New** Point(1, 1))  **End** **Using**  *' Draw a border around the image*  **Using** g **As** Graphics = Graphics.FromImage(newImage)  **Using** p **As** **New** Pen(Color.Black)  g.DrawRectangle(p, **New** Rectangle(0, 0, newWidth - 1, newHeight - 1))  **End** **Using**  **End** **Using**  inputImage = newImage  *' Sets width and height text boxs*  widthTxtBox.Text = inputImage.Width  heightTxtBox.Text = inputImage.Height  **Else**  **Exit** **Sub**  **End** **If**    **Dim** currentPixel **As** Color  *' Turns to grayscale*  **For** x **As** Integer = 0 **To** inputImage.Width - 1  **For** y **As** Integer = 0 **To** inputImage.Height - 1  currentPixel = inputImage.GetPixel(x, y)  *' Finds lumiosity*  luminosity = (currentPixel.R \* R) ^ GAMMA + (currentPixel.B \* B) ^ GAMMA + (currentPixel.G \* G) ^ GAMMA  *' Altrting the gradient thershold*  **If** luminosity <= 125 **Then**  inputImage.SetPixel(x, y, Color.FromArgb(0, 0, 0))  mazeWallList.Add(**New** Point(x, y))  **Else**  inputImage.SetPixel(x, y, Color.FromArgb(255, 255, 255))  **End** **If**   |  | | --- | | This code calculates the luminosity value of a given pixel. The if statement controls the threshold altering and the masking. All cells that need to masked will be stored in mazeWallList. |   **Next**  **Next**  imgComponents = componentAnalysis(inputImage)    imageInputBtn.Text = "Cancel Image"  imageInputted = **True**  **End** **Sub**    **Private** **Sub** helperBtn\_Click(sender **As** Object, e **As** EventArgs)  **Me**.Hide()  Form2.Show()  **End** **Sub**    **Private** **Sub** cancelAnimationBtn\_Click(sender **As** Object, e **As** EventArgs)  cancelAnimation = **True**  **End** **Sub**  *' USER INPUT END*  End Class |

## Helper Form {NOT COMPLETED}

|  |
| --- |
| **Public** Class Form2  Private currentIndex As Integer = 0  Const MAX\_NUMBER\_OF\_IMAGES = 10  Public Sub New()  ' This call is required by the designer.  InitializeComponent()  ' Add any initialization after the InitializeComponent() call.  End Sub    Private Sub backBtn\_Click(sender As Object, e As EventArgs) Handles backBtn.Click  Me.Hide()  Form1.Show()  End Sub    Private Sub Form2\_Load(sender As Object, e As EventArgs) Handles MyBase.Load  images.Image = indexImages(currentIndex)    End Sub    Private Sub Form2\_FormClosed(sender As Object, e As FormClosedEventArgs)  Me.Hide()  Form1.Show()  End Sub    Private Sub goLeftbtn\_Click(sender As Object, e As EventArgs) Handles goLeftbtn.Click  currentIndex -= 1  If currentIndex < 0 Then  currentIndex = MAX\_NUMBER\_OF\_IMAGES  End If    images.Image = indexImages(currentIndex)  End Sub    Private Sub goRightBtn\_Click(sender As Object, e As EventArgs) Handles goRightBtn.Click  currentIndex += 1  If currentIndex > MAX\_NUMBER\_OF\_IMAGES Then  currentIndex = 0  End If    images.Image = indexImages(currentIndex)  End Sub    Private Function indexImages(ByVal index As Integer) As Image  Select Case index  Case 0 ' Width/Height  tutorialTxt.Text = "You can change the size of your maze using the width and height textboxs. Values between 3-1000."  Return My.Resources.width\_height  Case 1 ' Maze Entry  tutorialTxt.Text = "You can change where the maze entry and exit cells are."  Return My.Resources.maze\_entry  Case 2 ' Dead End Remover  tutorialTxt.Text = "You can remove a percentage of dead ends using dead end remover textbox. Values between 0-1."  Return My.Resources.dead\_end\_remover  Case 3 ' Colours  tutorialTxt.Text = "You can customise the background, maze and solve colours."  Return My.Resources.colours  '|NEED DOING|  Case 4 ' Generation Algorithim  tutorialTxt.Text = "You can change which algorithim generates the maze."  '|NEED DOING|  Case 5 ' Solving Algorithim  tutorialTxt.Text = "You can change which algorithim solves the maze"  '|NEED DOING|  Case 6 ' Instant Animation  tutorialTxt.Text = "You can see how each algorithim works (includes: Generation, Solving and Dead End Removal algoritim)."  Case 7 ' Download Button  tutorialTxt.Text = "You can download any maze that is currently being displayed."  Return My.Resources.download  '|NEED DOING|  Case 8 ' Image Input  tutorialTxt.Text = "You can input a .JPG image to turn it into a maze."  '|NEED DOING|  Case 9 ' Stats  tutorialTxt.Text = "You can see the statisitics of each maze being displayed."  '|NEED DOING|  Case 10 ' Status  tutorialTxt.Text = "You can see how far along the maze generation is."    Case Else  Return Nothing  End Select  End Function  **End** **Class** |

# Testing

## Testing URL

<https://youtube.com/playlist?list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN>

###### Quick Note

In some of the videos, especially when generating larger mazes, there will be a long period of silence where I don’t talk, and the application doesn’t change. This is due to my computer trying to process the mazes.

After test number 36 there is an error with the numbering. I’ve kept it as it is easier when keeping up with the videos.

## Testing maze generation and initialisation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Video Test Number** | **Test Number** | **Description of test** | **Input data and it’s type** | **Expected Result** |
| 1.0 | 1 | Generate a 4,4 maze with any generation algorithm. | Boundary | A maze should be generated where the centre cell is the maze Entry and Exit. |
| 1.0 | 2 | Generate a 122,69 maze with any generation algorithm. | Normal | A maze should be generated and be displayed on the maze box. |
| 1.0 | 3 | Generate a 1000,1000 maze with any generation algorithm. | Boundary | The user should be prompted to download a maze. The maze downloaded should 1000,1000 |
| 1.1 | 4 | Generate a 2,3 maze with any generation algorithm. | Erroneous | The user should be prompted with an error message telling then the maze is not the allowed size. |
| 1.1 | 5 | Generate a 1000,1001 maze with any generation algorithm. | Erroneous | The user should be prompted with an error message telling then the maze is not the allowed size. |
| 1.2 | 6 | Generate a 122,69 maze with ‘DFS Backtracking’ | Normal | A maze should be generated and displayed on the maze box |
| 1.2 | 7 | Generate a 122,69 maze with ‘Randomised Prims’ | Normal | A maze should be generated and displayed on the maze box |
| 1.2 | 8 | Generate a 122,69 maze with ‘Kurskal’s’ | Normal | A maze should be generated and displayed on the maze box |
| 1.2 | 9 | Generate a 122,69 maze with ‘Aldous-border’ | Normal | A maze should be generated and displayed on the maze box |

## Testing maze solving

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Video Test Number** | **Test Number** | **Description of test** | **Input data and it’s type** | **Expected Result** |
| 2.0 | 11 | Generate a 4,4 maze with any generation algorithm.  Then press solve with any solving algorithm. | Boundary | A maze should be shown where the centre cell is the maze Entry and Exit. |
| 2.0 | 12 | Generate a 122,69 maze with any generation algorithm.  Then press solve with any solving algorithm. | Normal | A maze should be solved and be displayed on the maze box. |
| 2.0 | 13 | Generate a 1000,1000 maze with any generation algorithm.  Then press solve with any solving algorithm. | Boundary | The user should be prompted to download a maze. The maze downloaded should 1000,1000 and solved. |
| 2.1 | 14 | Generate a 122,69 maze with any generation algorithm.  Solve the maze with ‘Dijkstra’s’ | Normal | A maze should be solved and displayed on the maze box |
| 2.1 | 15 | Generate a 122,69 maze with any generation algorithm.  Solve the maze with ‘Breadth First Search’ | Normal | A maze should be solved and displayed on the maze box |
| 2.1 | 16 | Generate a 122,69 maze with any generation algorithm.  Solve the maze with ‘A\*’ | Normal | A maze should be solved and displayed on the maze box |
| 2.1 | 17 | Generate a 122,69 maze with any generation algorithm.  Solve the maze with ‘Wall Follower LHR’ | Normal | A maze should be solved and displayed on the maze box |
| 2.1 | 18 | Generate a 122,69 maze with any generation algorithm.  Solve the maze with ‘Wall Follower RHR’ | Normal | A maze should be solved and displayed on the maze box |

## Testing dead-end removal

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Video Test Number** | **Test Number** | **Description of test** | **Input data and it’s type** | **Expected Result** |
| 3.0 | 19 | Generate a 4,4 maze with any generation algorithm.  Input 1 in dead-end remover.  Then press remove dead ends. | Boundary | A maze should be generated where the centre cell is the maze Entry and Exit. |
| 3.0 | 20 | Generate a 122,69 maze with any generation algorithm.  Input 1 in dead-end remover.  Then press remove dead ends. | Normal | A maze should be generated with 0 dead-ends and be displayed on the maze box. |
| 3.0 | 21 | Generate a 1000,1000 maze with any generation algorithm.  Input 1 in dead-end remover.  Then press remove dead ends. | Boundary | The user should be prompted to download a maze. The maze downloaded should 1000,1000. After removing dead ends are pressed, they will be prompted to download a maze. The maze downloaded should be 1000,1000 and have 0 dead ends. |
| 3.1 | 22 | Generate a 122,69 maze with any generation algorithm.  Input 0.5 in dead-end remover.  Then press remove dead ends. | Normal | A maze should be generated with half of the initial dead-ends and be displayed on the maze box. |
| 3.1 | 23 | Generate a 122,69 maze with any generation algorithm.  Input -0.5 in dead-end remover.  Then press remove dead ends. | Erroneous | There should be a prompt explaining this isn’t a valid input. |
| 3.1 | 24 | Generate a 122,69 maze with any generation algorithm.  Input 1.1 in dead-end remover.  Then press remove dead ends. | Erroneous | There should be a prompt explaining this isn’t a valid input. |

## Testing maze entry-exit

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Video Test Number** | **Test Number** | **Description of test** | **Input data and it’s type** | **Expected Result** |
| 4.0 | 25 | Input any maze entry option.  Generate a 3,3 maze with any generation algorithm. | Boundary | A maze should be generated where the centre cell is the maze Entry and Exit. |
| 4.0 | 26 | Input ‘Random’ maze entry option.  Generate a 122,69 maze with any generation algorithm. | Normal | A maze should be generated with the entry and exit cells changing each time and be displayed on the maze box. |
| 4.0 | 27 | Input ‘Top-Bottom’ maze entry option.  Generate a 122,69 maze with any generation algorithm. | Normal | A maze should be generated with the entry and exit cells being randomly on top or bottom and be displayed on the maze box. |
| 4.0 | 28 | Input ‘Right-Left’ maze entry option.  Generate a 122,69 maze with any generation algorithm. | Normal | A maze should be generated with the entry and exit cells being randomly on right or left and be displayed on the maze box. |
| 4.0 | 29 | Input ‘Diagonal’ maze entry option.  Generate a 122,69 maze with any generation algorithm. | Normal | A maze should be generated with the entry and exit cells being randomly on the top left or on the bottom right and be displayed on the maze box. |

## Testing maze colours

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Video Test Number** | **Test Number** | **Description of test** | **Input data and it’s type** | **Expected Result** |
| 5.0 | 30 | Input any colour for background colour.  Generate a 122,69 maze with any generation algorithm. | Normal | A maze should be generated with the selected colour as the background colour and be displayed on the maze box. |
| 5.0 | 31 | Input any colour for maze colour.  Generate a 122,69 maze with any generation algorithm. | Normal | A maze should be generated with the selected colour as the maze colour and be displayed on the maze box. |
| 5.0 | 32 | Input any colour for solve colour.  Generate a 122,69 maze with any generation algorithm.  Then press solve with any solving algorithm. | Normal | A maze should be generated with the selected colour as the solve path colour and be displayed on the maze box. |

## Testing maze animations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Video Test Number** | **Test Number** | **Description of test** | **Input data and it’s type** | **Expected Result** |
| 6.0 | 33 | Uncheck instant animations.  Generate a 7,7 maze with ‘DFS Backtracking’ | Normal | A maze should be generated with an animation of the appropriate algorithm and displayed on the maze box |
| 6.0 | 34 | Uncheck instant animations.  Generate a 7,7 maze with ‘Randomised Prims’ | Normal | A maze should be generated with an animation of the appropriate algorithm and displayed on the maze box |
| 6.0 | 35 | Uncheck instant animations.  Generate a 7,7 maze with ‘Kurskal’s’ | Normal | A maze should be generated with an animation of the appropriate algorithm and displayed on the maze box |
| 6.0 | 36 | Uncheck instant animations.  Generate a 7,7 maze with ‘Aldous-border’ | Normal | A maze should be generated with an animation of the appropriate algorithm and displayed on the maze box |
| 6.1 | 38 | Uncheck instant animations.  Generate a 20,20 maze with any generation algorithm.  Solve the maze with ‘Dijkstra’s’ | Normal | A maze should be solved with an animation of the appropriate algorithm and displayed on the maze box |
| 6.1 | 39 | Uncheck instant animations.  Generate a 20,20 maze with any generation algorithm.  Solve the maze with ‘Breadth First Search’ | Normal | A maze should be solved with an animation of the appropriate algorithm and displayed on the maze box |
| 6.1 | 40 | Uncheck instant animations.  Generate a 20,20 maze with any generation algorithm.  Solve the maze with ‘A\*’ | Normal | A maze should be solved with an animation of the appropriate algorithm and displayed on the maze box |
| 6.1 | 41 | Uncheck instant animations.  Generate a 20,20 maze with any generation algorithm.  Solve the maze with ‘Wall Follower LHR’ | Normal | A maze should be solved with an animation of the appropriate algorithm and displayed on the maze box |
| 6.1 | 42 | Uncheck instant animations.  Generate a 20,20 maze with any generation algorithm.  Solve the maze with ‘Wall Follower RHR’ | Normal | A maze should be solved with an animation of the appropriate algorithm and displayed on the maze box |

## Image to maze testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Video Test Number** | **Test Number** | **Description of test** | **Input data and it’s type** | **Expected Result** |
| 7.0 | 43 | Trying to input an image that is not .JPG. | Erroneous | Only images that are .JPG should show in the file explorer. |
| 7.0 | 44 | Trying to input an image that is .JPG. | Normal | The status should change to ‘Image Uploaded’. |
| 7.0 | 45 | After uploading a valid image, cancel the image. | Normal | The status should change to ‘Doing Nothing’ and a normal square maze should be able to be generated. |
| 7.1 | 46 | Upload an image that is 122,69 in size and press generate. | Normal | There should no prompt to download the image and the maze should show in the maze box. |
| 7.1 | 47 | Upload an image that is 122,69 in size and spam generate. | Normal | All components should have a maze generated. |
| 7.1 | 48 | Upload an image that is 407,230 in size and press generate. Then press download. | Normal | There should prompt to download the image and the maze should not show in the maze box. The downloaded maze should be fully generated. |
| 7.2 | 49 | Upload an image that is 1001,1000 in size and press generate. | Erroneous | There should be a prompt explaining that the image is too large. |
| 7.3 | 50 | Upload an image that is 122,69 in size and press generate.  Then press solve with any solving algorithm. | Normal | The biggest component of the maze should have a solved path. |
| 7.3 | 51 | Upload an image that is 122,69 in size and press generate.  Then press solve with any solving algorithm. Then press download. | Normal | There should prompt to download the image and the maze should not show in the maze box. The downloaded maze should have the biggest component solved. |

## General Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Video Test Number** | **Test Number** | **Description of test** | **Input data and it’s type** | **Expected Result** |
| 8.0 | 52 | Generate a 122, 69 maze, with any generation algorithm.  Spam the generate button. | Normal | A new maze should be generated every time the button is pressed. |
| 8.0 | 53 | Generate a 122,69 maze, with any generation algorithm.  Pick any generation algorithm and spam the solve button. | Normal | On the first press the solved path should appear. Every time after, the output should be the same. |
| 8.0 | 54 | Generate a 122,69 maze, with any generation algorithm.  Input 1 in dead-end remover.  Spam the remove dead-ends button. | Normal | On the first press 100% of the dead-ends should be gone. Every time after, the output should be the same. |
| 8.0 | 55 | Generate a 122,69 maze, with any generation algorithm.  Input 0.5 in dead-end remover.  Spam the remove dead-ends button. | Normal | On the first press 50% of the dead-ends should be gone. Every time after, the 50% of the current dead ends should be gone. At some point there should only be one dead end left in the maze. |
| 8.0 | 56 | Spam between the helper form and the maze form | Normal | There should be a short load between each click. The program shouldn’t crash. |
| 8.0 | 57 | In the helper form, spam through the GIF . | Normal | Each GIF should display properly and it’s respective description. |

# 

# Evaluation

## Personal Feedback

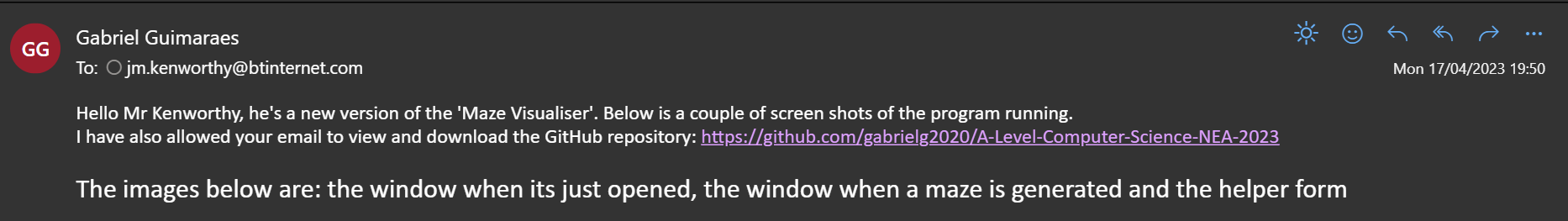
In my view, the maze software project turned out really well. By leveraging research and end-user feedback, I successfully developed a program that serves its intended purpose. I even found myself enjoying the program outside of testing and programming, experimenting with different images and observing how various generation and solving algorithms work with their corresponding animations. I think I properly achieve the objects I set for myself in the beginning of this project, primary and secondary. Overall, I would consider this project a success.

Nevertheless, there is still room for improvement. I believe certain algorithms could be faster when handling larger mazes, and incorporating recursion in some of the generation algorithms would be a personal aim of mine. Additionally, adding a print button would be a convenient feature for when you would like to complete a maze yourself.

## Objective Evaluation

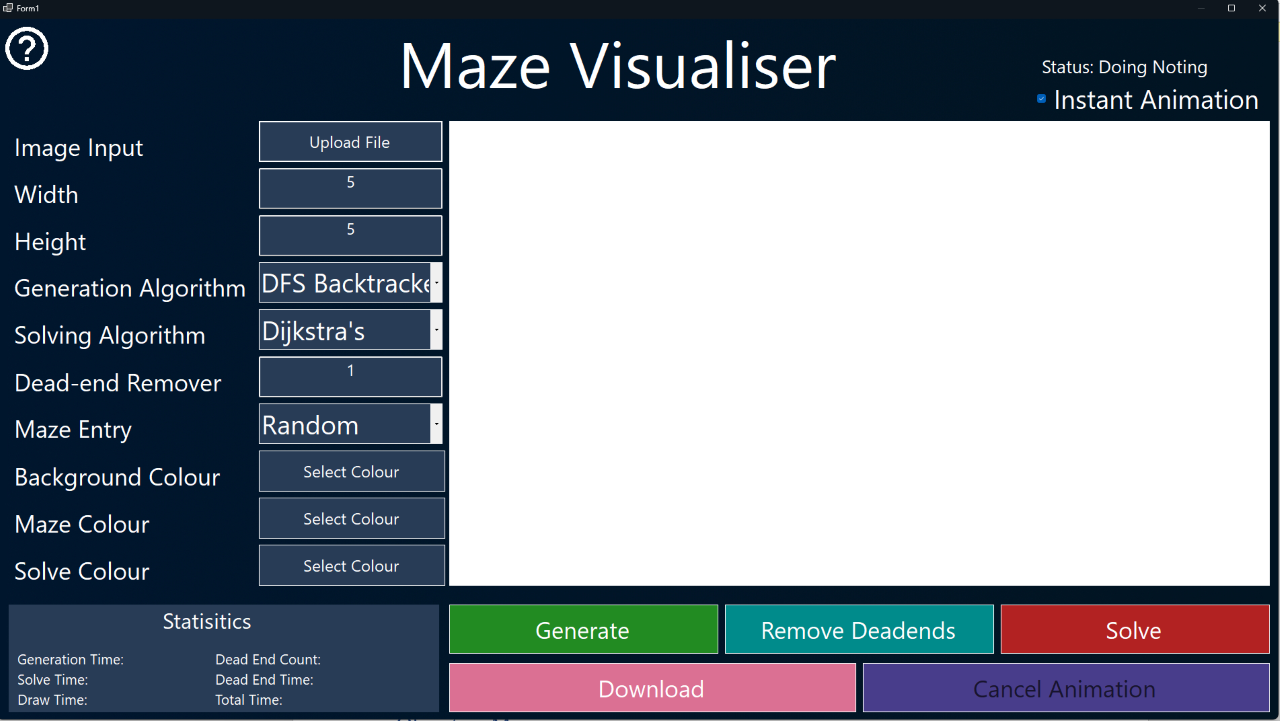
|  |  |
| --- | --- |
| **Objective** | **Evauation** |
| **A** | Objective A was to ‘Create a maze grid’. This objective has been achieved completely, Users can input a width and height value within the limits and the correct number of cell objects are initialised. |
| **B** | Objective B was to ‘Generate a maze’. This objective has been achieved completely and even been expanded upon in Objective G. Users can now use the Randomised DFS algorithim to generate a maze and the correct maze is displayed on the window. |
| **C** | Objective C was to ‘Solve a maze’. This objective has been achieved completely and even expanded upon in Objective H. Users can now use Dijkstra’s algorithim to solve a generated maze and will always generate the shortest path. Also, the correct path is displayed on the window. |
| **D** | Objective D was to ‘Customise the maze colours. This objective has been achieved completely. When the change colour buttons are pressed, a colour picker dialog shows where users can customise the colour of the background, maze and solved path. The correct colours are displayed on the window. |
| **E** | Objective E was to ‘Download the maze’. This objective has been achieved completely. The program checks if a maze has been generated before downloading. A file explorer dialog shows which allows the user to navigate to the correct file destination. It also allows renaming of the image and forcing the .JPEG format. |
| **F** | Objective F was to ‘Animate the maze’. This objective has been achieved completely. The user can choose if they want to see the algorithim animation; they can also cancel the animation at any point and the maze will complete solving or generating. Each algorithim has its own animation. |
| **G** | Objective G was to ‘Implement more generation algorithms. This objective has been partially completed. Randomised Prims, Kurskals and Aldous-Border has been implemented correctly. However, during coding I realised that Wave Function Collapse was not appropriate for this type of procedural generation - this is further explored in the Implentation section. Nevertheless, the users have a drop-down menu to select out of all the generation algorithms implemented and they all have their own animations. |
| **H** | Objective H was to ‘Implement more solving algorithms. This objective has been achieved completely. Breadth-First Search, A\*, Wall Follower (Right-Hand & Left-Hand Rule) have all been implemented correctly. The users have a drop-down menu to select out of all the solving algorithms implemented and they have their own animations. |
| **I** | Objective I was to ‘Change the maze entry and exit’. This objective has been achieved completely. The user has a drop-down menu to choose out of 4 entry-exit options. Each time they change the entry-exit option a press generates the green (entry) and red (exit) points change and are corrected shown on the maze display. |
| **J** | Objective J was to ‘Remove dead ends. This objective has been achieved completely. The user can input a percentage (as a decimal, which is the percentage of dead ends they want to remove) and when they press the Remove Dead Ends button the correct number of dead ends are removed randomly. |
| **K** | Objective K was to ‘Make an aesthetic and efficient GUI’. This objective has been achieved poorly. I did not implement visual feedback of a 20% darkness on hovered button and I did not stick to the colour pallette or the 70-20-10 rule. Font sizes were not consistent across the program. On the other hand, I did use the same font on all text, and all images used were .JPEG compressed, inputs indicated what they required and all elements were made in the designer. I also created a new ‘Helper’ window which, when accessed, had a carasol of text explaining what each customisation option does, alongside GIFS showing what each customisation option does. |
| **L** | Objective L was to ‘Implement Statistics’. This objective was achieved completely. All generation, solving, drawing and removal of dead ends are timed accurately. Using these times, the total time is then calculated. The number of dead ends is counted and displayed. All of this information is displayed neatly on the window. |
| **M** | Objective M was to ‘Be able to input an image to turn into a maze’. This objective was achieved completely. I used the file directory dialog to allow users to have a file explorer when finding the image, they want to upload, this also allowed me to filter for .JPEG only. However, instead of storing the image in secondary storage which was inefficient, I kept it in primary storage (this is explained further in the Implentation section). I used the RGB value of each pixel in the inputted image to produce a luminosity value. I then used this luminosity value and an altered grayscale threshold to determine if the said pixel should be black or white. I then used an array to mask the image to the maze. I needed to use a component analysis algorithim to find which area of the masked maze required generating, and where to place the entry and exit cells. |

## End User Feedback

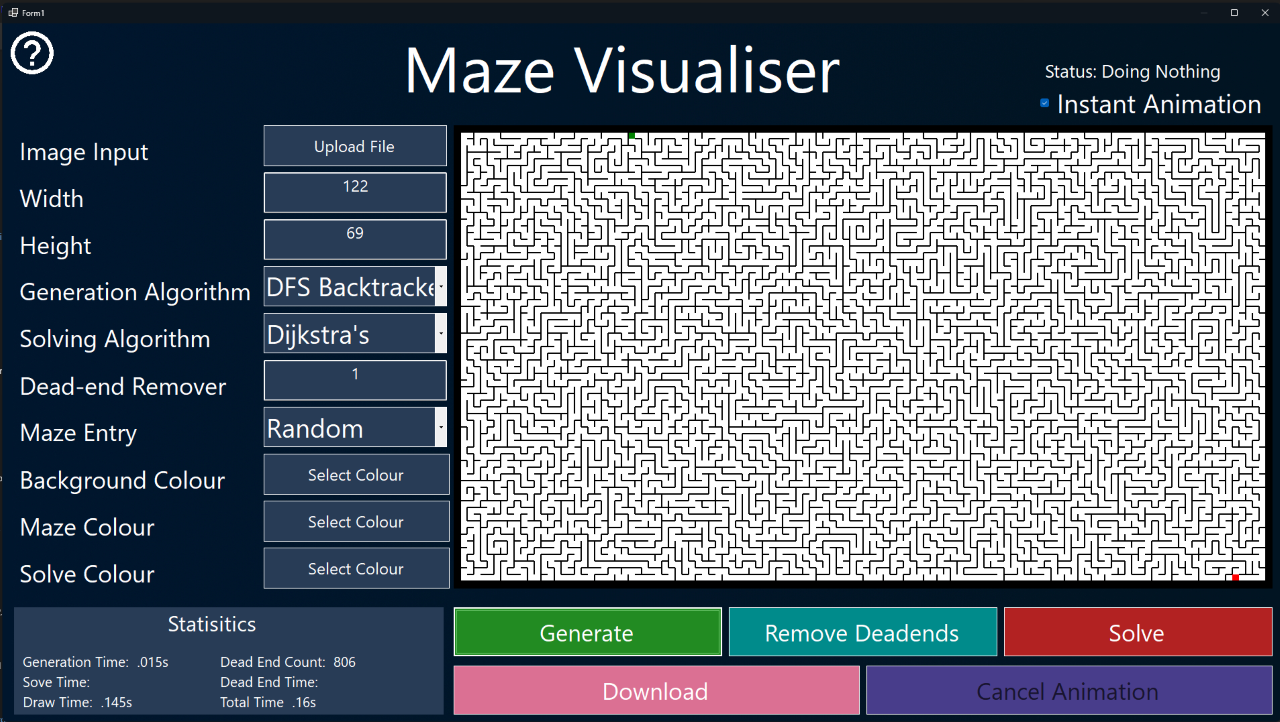


##### Images attached

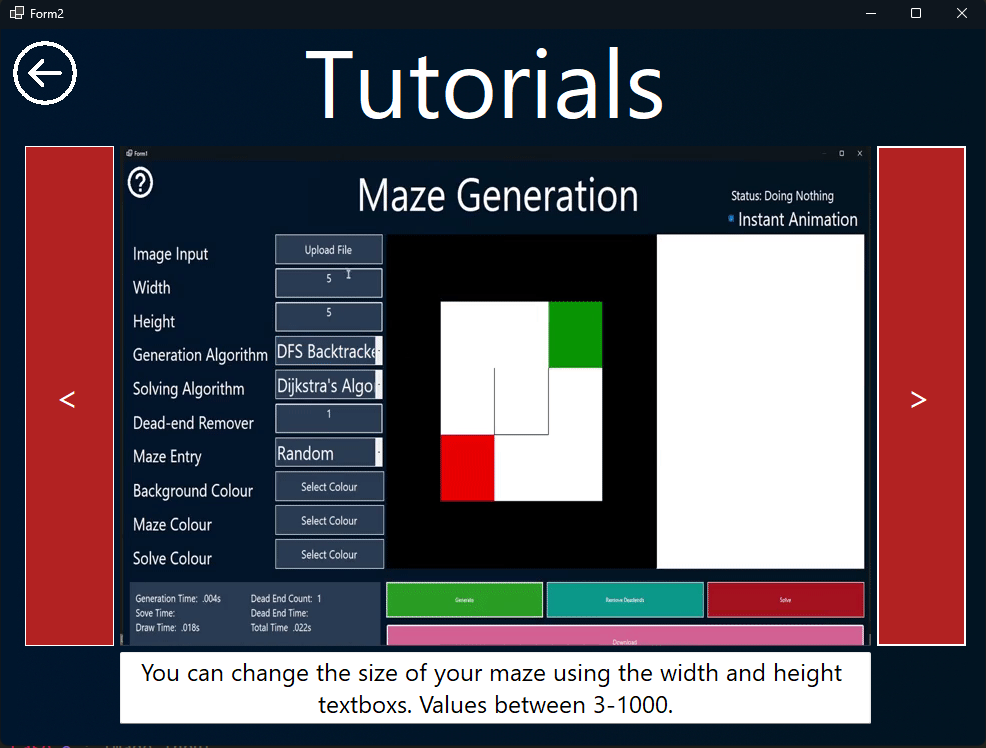
###### The window when it’s just opened

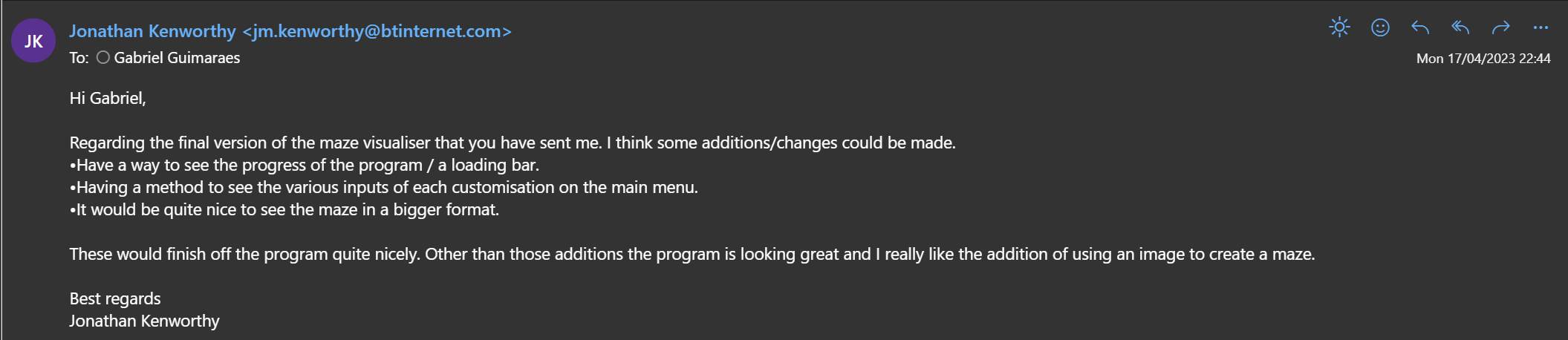


###### The window when a maze is generated



###### The helper form





## Reflections on end-user feedback

### Were the previous feedback points met?

|  |  |
| --- | --- |
| **Feedback Point** | **Was the feedback point meet and how?** |
| Add a way for the user to see what each customisation option does. | Yes, however I had a different idea than what I thought of initially. Instead of small (i)s that when hovered will bring up a box that explains an input’s purpose, I made a whole new window (called the helper window) that when clicked, will have a carousel of GIFs showing, instead of telling, what each input does. |
| Create equal spacing between each input box. | Yes, this was met even after different GUI changes. |
| Add what type the input should be. | Yes, but again the implantation is different than what I initially thought of. Each box has example data that can be used straight when the user opens the application. The type of input can then be implied by the user. If they still don’t understand they can click the helper window and there will be an explanation, alongside examples. |
| Add statistics section below download button. | Yes, the position of the statistics area was moved beneath the inputs. I added more statistics and made it larger for better readability. |

### New requirements from end-user

|  |  |
| --- | --- |
| **Feedback Point** | **What I could’ve done to improve.** |
| Have a way to see the progress of the program. | Have a progress bar to visually see the progress at any given stage (i.e., generation/downloading). |
| Have a way to see the type of input each customisation option does on the main form. | Should’ve still implemented some way to see the type of input required on the main form. |
| Have a way to see bigger mazes on the main form. | Increase the size of the maze display to fit in larger mazes and images. |

## Further improvements

### Responsive application

Implementing a responsive UI would’ve significantly enhanced the user experience, as it ensures it’s compatible across various screen sizes. Also, it contributes to making the software seem more professional.

### Different maze types

Incorporating various maze designs can significantly expand the range of customization options available to users. They add visual appeal but also create a more engaging and dynamic experience.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Circular Maze | Hexagonal Maze | Triagular Maze |
|  | Mazes |  |
| Weave Maze | Sherical Maze | Mobius-Strip Maze |

### Making this a web or mobile application

Integrating a web or mobile app allows for this software to be portable, whether you have an internet connection or not.

# Objective Mapping

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objective** | **Completion Date** | **Analysis** | **Design** | **Technical Solution** |
| **A** | 30.01.23 | N/A | 37, 51 | 72 |
| **B** | 1.02.23 | 11-12, 16 | 38, 51 | 76,89-90 |
| **C** | 5.02.23 | 12-13, 16 | 38, 51 | 82-83 |
| **D** | 30.01.23 | N/A | 38, 46, 51 | 92 |
| **E** | 18.02.23 | 18, 24 | 38, 49, 51 | 92 |
| **F** | 28.03.23 | N/A | 51 | 79-82, 86-88 |
| **G** | 29.03.23 | 16 | 51-52 | 76-78 |
| **H** | 28.03.23 | 16 | 52-54 | 83-86 |
| **I** | 3.02.23 | 15 | 37, 42-45, 51 | 72-73 |
| **J** | 14.03.23 | 13-14 | 38, 48, 51 | 78 |
| **K** | N/A | 6-7, 16, 18 | N/A | N/A |
| **L** | 28.2.23 | 18 | 38, 47, 50, 51 | 90-92 |
| **M** | 6.03.23 | 8-11 | 37, 39-41, 51 | 93-94 |

# Appendix 1

## Unhighlighted code

### Maze Form

|  |
| --- |
| **Public** **Class** Form1  *' Class instanse used to randomize numbers*  **Private** rnd **As** **New** Random  *' Drawing Variables*  **Private** **Const** PEN\_SIZE **As** Integer = 2  **Private** M **As** Integer = 3  *' Maze properties*  **Private** maze **As** Cell(,)  **Private** width **As** Integer  **Private** height **As** Integer  **Private** deadEndPercent **As** Double  **Private** mazeEntryType **As** String  **Private** mazeEntry **As** Point  **Private** mazeExit **As** Point  **Private** deadEndPos **As** **New** List(**Of** Point)  *' Maze Colour Customisation*  **Private** bgColour **As** Color = Color.White  **Private** mazeColour **As** Color = Color.Black  **Private** solveColour **As** Color = Color.Silver  **Private** pinkColor **As** Color = Color.FromArgb(255, 0, 220) *' Pink color*  **Private** purpleColor **As** Color = Color.FromArgb(0, 0, 124) *' Purple color*  *' Animation Variables*  **Private** **Const** T **As** Integer = 50  **Private** passedPath **As** **New** List(**Of** Point)  **Private** solvedVisited **As** **New** Queue(**Of** Point)  **Private** maxWeight **As** Integer  **Public** cancelAnimation **As** Boolean = **False**  **Public** resetType **As** String  *' Maze Generation/Solving Inputs*  **Private** generationAlgorithm **As** String  *' Used in DFSbacktracker*  **Dim** generationStack **As** **New** Stack(**Of** Point)  *' Used in randomizedPrims*  **Private** visitedCells **As** **New** List(**Of** Cell)  **Private** primsWalls **As** **New** List(**Of** Tuple(**Of** Point, Integer))  *' Used in kurskals*  **Private** kurskWalls **As** **New** List(**Of** Tuple(**Of** Point, Integer))  **Private** kursSets **As** **New** Dictionary(**Of** Point, Integer)  **Private** kursNeighbourCoords **As** Point  **Private** kursCurrentWallIndex **As** Integer = 0  *' Used in aldousBroder*  **Private** abTotalCells **As** Integer  **Private** abVisitedCells **As** Integer = 0  **Private** random **As** **New** Random()  **Private** currentX **As** Integer  **Private** currentY **As** Integer  **Private** directions() **As** Point = {**New** Point(0, -1), **New** Point(1, 0), **New** Point(0, 1), **New** Point(-1, 0)}  **Private** abHasBeenAnimated **As** Boolean = **False**  *' Used in Astar()*  **Public** gWeights **As** **New** Dictionary(**Of** Point, Double)  *' Used in BFS()*  **Public** branchingPoints **As** **New** List(**Of** Point)  **Private** solveAlgorithm **As** String  **Private** mazeWallCount **As** Integer = 0  **Private** totalCells **As** Integer = 0  **Public** path **As** **New** Queue(**Of** Point)  **Public** helperPath **As** **New** Queue(**Of** Point)  *' Controls when the from draws*  **Private** mazeImage **As** Bitmap  **Private** mazeImageGraphics **As** Graphics  **Private** downlaodGenerated **As** DialogResult  **Private** downlaodSolved **As** DialogResult  *' Stats Variables*  **Private** deadEndToShow **As** Integer  **Private** solveTimer **As** **New** Stopwatch  **Private** generationTimer **As** **New** Stopwatch  **Private** drawTimer **As** **New** Stopwatch  **Private** deadEndTimer **As** **New** Stopwatch  *' Babyproofing Variables*  **Private** mazeGenerated **As** Boolean = **False**  *' Image to maze Variables*  **Private** imageInputted **As** Boolean = **False**  **Private** inputImage **As** Bitmap  **Private** mazeWallList **As** **New** List(**Of** Point)  **Private** luminosity **As** Double  **Private** **Const** GAMMA **As** Double = 1.0  **Private** **Const** R **As** Double = 0.2126  **Private** **Const** G **As** Double = 0.7152  **Private** **Const** B **As** Double = 0.0722  **Private** imgComponents **As** **New** List(**Of** List(**Of** Point))  **Private** largestComponent **As** **New** List(**Of** Point)  *' Circular Queue Class*  **Class** CircularQueue(**Of** T)  **Private** **ReadOnly** items **As** List(**Of** T)  **Private** currentIndex **As** Integer    *' Constructor*  **Public** **Sub** **New**(i **As** IEnumerable(**Of** T))  *' This assigns the items in the queue*  items = **New** List(**Of** T)(i)  currentIndex = 0  **End** **Sub**    *' This function return value will be same type and the input*  **Public** **Function** turnRight() **As** T  **If** items.Count = 0 **Then**  *' This helps when debugging*  **Throw** **New** InvalidOperationException("The queue is empty")  **End** **If**      currentIndex = (currentIndex + 1) **Mod** items.Count  **Dim** i **As** T = items(currentIndex)  **Return** i  **End** **Function**    *' This function return value will be same type and the input*  **Public** **Function** turnLeft() **As** T  **If** items.Count = 0 **Then**  *' This helps when debugging*  **Throw** **New** InvalidOperationException("The queue is empty")  **End** **If**    currentIndex = (currentIndex - 1 + items.Count) **Mod** items.Count  **Dim** i **As** T = items(currentIndex)    **Return** i  **End** **Function**  **End** **Class**  *' Priority Queue Class*  **Public** **Class** PriorityQueue(**Of** priority **As** IComparable, value)  **Private** **ReadOnly** dictionary **As** SortedDictionary(**Of** priority, Queue(**Of** value))    *' Constructor*  **Public** **Sub** **New**()  *' This assigns the items in the dictionary*  dictionary = **New** SortedDictionary(**Of** priority, Queue(**Of** value))()  **End** **Sub**    *' Adds values to the queue*  **Public** **Sub** Enqueue(priority **As** priority, value **As** value)  *' If we have a new priority we create a new queue*  **If** **Not** dictionary.ContainsKey(priority) **Then**  dictionary(priority) = **New** Queue(**Of** value)()  **End** **If**  *' Add value to queue*  dictionary(priority).Enqueue(value)  **End** **Sub**    *' Removes values to the queue*  **Public** **Function** Dequeue() **As** value  *' This helps when debugging*  **If** dictionary.Count = 0 **Then**  **Throw** **New** InvalidOperationException("The priority queue is empty.")  **End** **If**    **Dim** firstPair **As** KeyValuePair(**Of** priority, Queue(**Of** value)) = dictionary.First()  **Dim** value **As** value = firstPair.Value.Dequeue()    **If** firstPair.Value.Count = 0 **Then**  dictionary.Remove(firstPair.Key)  **End** **If**    **Return** value  **End** **Function**    *' Checks if the whole queue is empty*  **Public** **Function** isEmpty() **As** Boolean  **Return** dictionary.Count = 0  **End** **Function**    *' Returns the number of items in the queue*  **Public** **Function** Count() **As** Integer  **Dim** totalCount **As** Integer = 0  **For** **Each** q **In** dictionary.Values  totalCount += q.Count  **Next**  **Return** totalCount  **End** **Function**    *' Checks if a value is in the queue*  **Public** **Function** Contains(v **As** value) **As** Boolean  **For** **Each** q **In** dictionary.Values  **If** q.Contains(v) **Then**  **Return** **True**  **End** **If**  **Next**  **Return** **False**  **End** **Function**  **End** **Class**  *' Cell Class*  **Public** **Class** Cell  *' Postion Properties*  **Public** x **As** Integer  **Public** y **As** Integer  *' Wall Properties*  **Public** walls **As** **New** List(**Of** Boolean)({**True**, **True**, **True**, **True**})  **Public** wallPos(3, 1) **As** Point  *' Cell Type*  **Public** mazeWallBool **As** Boolean = **False**  **Public** mazeEntryBool **As** Boolean = **False**  **Public** mazeExitBool **As** Boolean = **False**  **Public** mazeSolved **As** Boolean = **False**  *' Generate/Solve Properties*  **Public** visited **As** Boolean = **False**  **Public** connectedCell **As** **New** List(**Of** Point)    *' Method to draw walls*  **Public** **Sub** drawWalls()  **For** wall **As** Integer = 0 **To** 3  **If** walls(wall) = **True** **And** Form1.mazeColour = Color.Empty **Then**  Form1.mazeImageGraphics.DrawLine(**New** Pen(Color.Black, PEN\_SIZE), wallPos(wall, 0), wallPos(wall, 1))  Form1.mazeImageGraphics.DrawLine(**New** Pen(Color.Black, PEN\_SIZE), wallPos(wall, 0), wallPos(wall, 1))  **ElseIf** walls(wall) = **True** **Then** *' If user hasnt selected colour*  Form1.mazeImageGraphics.DrawLine(**New** Pen(Form1.mazeColour, PEN\_SIZE), wallPos(wall, 0), wallPos(wall, 1))  Form1.mazeImageGraphics.DrawLine(**New** Pen(Form1.mazeColour, PEN\_SIZE), wallPos(wall, 0), wallPos(wall, 1))  **End** **If**  **Next**  **End** **Sub**  *' Method to find dead-ends*  **Public** **Sub** deadEndFinder()  **Dim** wallCount **As** Integer = 0  *' Checks each wall*  **For** **Each** wall **In** walls  **If** wall = **True** **And** mazeWallBool = **False** **Then**  wallCount += 1  **End** **If**  **Next**  **If** wallCount = 3 **And** **Not** Form1.deadEndPos.Contains(**New** Point(x, y)) **Then**  Form1.deadEndPos.Add(**New** Point(x, y))  **End** **If**  **End** **Sub**  *' Method to break wall*  **Public** **Function** breakWall(**ByVal** d **As** Integer)  *' Makes sure the cell isn't a maze wall*  **If** mazeWallBool = **True** **Then**  **Return** Point.Empty  **Exit** **Function**  **End** **If**  *' Breaks wall depending on the d (direction)*  **Select** **Case** d  *' The wall selected must be broken but also the neighbours wall*  **Case** 0 *' Breaking the top wall*  **If** Form1.maze(x, y - 1).mazeWallBool = **True** **Then**  **Return** Point.Empty  **Exit** **Function**  **End** **If**  walls(d) = **False**  Form1.maze(x, y - 1).walls(d + 2) = **False**  connectedCell.Add(**New** Point(x, y - 1))  Form1.maze(x, y - 1).connectedCell.Add(**New** Point(x, y))  **Case** 1 *' Breaking the right wall*  **If** Form1.maze(x + 1, y).mazeWallBool = **True** **Then**  **Return** Point.Empty  **Exit** **Function**  **End** **If**  walls(d) = **False**  Form1.maze(x + 1, y).walls(d + 2) = **False**  connectedCell.Add(**New** Point(x + 1, y))  Form1.maze(x + 1, y).connectedCell.Add(**New** Point(x, y))  **Case** 2 *' Breaking the bottom wall*  **If** Form1.maze(x, y + 1).mazeWallBool = **True** **Then**  **Return** Point.Empty  **Exit** **Function**  **End** **If**  walls(d) = **False**  Form1.maze(x, y + 1).walls(d - 2) = **False**  connectedCell.Add(**New** Point(x, y + 1))  Form1.maze(x, y + 1).connectedCell.Add(**New** Point(x, y))  **Case** 3 *' Breaking the left wall*  **If** Form1.maze(x - 1, y).mazeWallBool = **True** **Then**  **Return** Point.Empty  **Exit** **Function**  **End** **If**  walls(d) = **False**  Form1.maze(x - 1, y).walls(d - 2) = **False**  connectedCell.Add(**New** Point(x - 1, y))  Form1.maze(x - 1, y).connectedCell.Add(**New** Point(x, y))  **End** **Select**  **Return** connectedCell(connectedCell.Count() - 1)  **End** **Function**    *' Method to check unvisted neighbours*  **Public** **Function** checkUnvistedNeighbours() **As** List(**Of** Point)  **Dim** neighbours **As** **New** List(**Of** Point)    **If** mazeWallBool = **True** **Then**  **Return** {Point.Empty, Point.Empty, Point.Empty, Point.Empty}.ToList  **Exit** **Function**  **End** **If**    **If** Form1.maze(x, y - 1).visited = **False** **Then**  neighbours.Add(**New** Point(x, y - 1))  **Else**  neighbours.Add(Point.Empty)  **End** **If**    **If** Form1.maze(x + 1, y).visited = **False** **Then**  neighbours.Add(**New** Point(x + 1, y))  **Else**  neighbours.Add(Point.Empty)  **End** **If**    **If** Form1.maze(x, y + 1).visited = **False** **Then**  neighbours.Add(**New** Point(x, y + 1))  **Else**  neighbours.Add(Point.Empty)  **End** **If**    **If** Form1.maze(x - 1, y).visited = **False** **Then**  neighbours.Add(**New** Point(x - 1, y))  **Else**  neighbours.Add(Point.Empty)  **End** **If**  **Return** neighbours  **End** **Function**  *' Returns a boolean if cell has a connecter neighbour*  **Public** **Function** checkConnectedCell(d **As** Integer)  **Select** **Case** d  **Case** 0 *' Check above*  **If** connectedCell.Contains(**New** Point(x, y - 1)) **Then**  **Return** **New** Point(x, y - 1)  **End** **If**  **Case** 1 *' Check Right*  **If** connectedCell.Contains(**New** Point(x + 1, y)) **Then**  **Return** **New** Point(x + 1, y)  **End** **If**  **Case** 2 *' Check Below*  **If** connectedCell.Contains(**New** Point(x, y + 1)) **Then**  **Return** **New** Point(x, y + 1)  **End** **If**  **Case** 3 *' Check Left*  **If** connectedCell.Contains(**New** Point(x - 1, y)) **Then**  **Return** **New** Point(x - 1, y)  **End** **If**  **End** **Select**  **Return** Point.Empty  **End** **Function**  **End** **Class**    **Public** **Sub** **New**()  *' This call is required by the designer.*  InitializeComponent()  *' Add any initialization after the InitializeComponent() call.*  **AddHandler** helperBtn.Click, **AddressOf** helperBtn\_Click  **AddHandler** imageInputBtn.Click, **AddressOf** imageInputBtn\_Click  **AddHandler** bgColourBtn.Click, **AddressOf** bgColourBtn\_Click  **AddHandler** mazeColourBtn.Click, **AddressOf** mazeColourBtn\_Click  **AddHandler** solveColourBtn.Click, **AddressOf** solveColourBtn\_Click  **AddHandler** generateBtn.Click, **AddressOf** generateBtn\_Click  **AddHandler** solveBtn.Click, **AddressOf** solveBtn\_Click  **AddHandler** deadEndRemoverBtn.Click, **AddressOf** deadEndRemoverBtn\_Click  **AddHandler** downloadBtn.Click, **AddressOf** downloadBtn\_Click  **AddHandler** cancelAnimationBtn.Click, **AddressOf** cancelAnimationBtn\_Click  **AddHandler** solvedPathAnimationTimer.Tick, **AddressOf** solvedPathAnimationTimer\_Tick  **AddHandler** heatMapAnimationTimer.Tick, **AddressOf** heatMapAnimationTimer\_Tick  **End** **Sub**  **Private** **Sub** Form1\_Load(sender **As** Object, e **As** EventArgs) **Handles** **MyBase**.Load      generationCombo.SelectedIndex = 0 *' Makes index 0 default displayed on the combo list(so currently shows "DFS Backtracker" initially*  solveCombo.SelectedIndex = 0 *' Default displays (Dijkstra's Algortimn)*  mazeEntryCombo.SelectedIndex = 0 *' Default displays "Random"*    cancelAnimationBtn.Enabled = **False**    solvedPathAnimationTimer.Interval = T  heatMapAnimationTimer.Interval = T  generationPointTimer.Interval = T  heatMapAnimationTimer.Enabled = **False**  solvedPathAnimationTimer.Enabled = **False**  generationPointTimer.Enabled = **False**    **End** **Sub**  **Private** **Sub** initializeMaze()  mazeWallCount = 0    *' Resets old timer, Starts new timer, Upates Status*  statusLbl.Text = "Status: Initializing Maze"  statusLbl.Update()  *' Initialize each cell with correct: Type and Wall Position*  maze = **New** Cell(width, height) {}  mazeImage = **New** Bitmap(((width + 1) \* M) + M, ((height + 1) \* M) + M)  mazeImageGraphics = Graphics.FromImage(mazeImage)  **For** i **As** Integer = 0 **To** width  **For** j **As** Integer = 0 **To** height  *' Giving each cell their index*  maze(i, j) = **New** Cell  maze(i, j).x = i  maze(i, j).y = j    *' Setting the maze wall cells with the mazeWallBool*  **If** i = 0 **Or** j = 0 **Or** i = width **Or** j = height **Or** mazeWallList.Contains(**New** Point(i, j)) **Then**  maze(i, j).mazeWallBool = **True**  maze(i, j).visited = **True**  mazeWallCount += 1  **Else**  totalCells += 1  **End** **If**    *' Giving each cell wall a start(0), end(1) and position on the screen*  **Dim** posi **As** Integer = i \* M  **Dim** posj **As** Integer = j \* M  **For** **Each** wall **In** maze(i, j).walls  *' Top Wall*  maze(i, j).wallPos(0, 0) = **New** Point(posi, posj)  maze(i, j).wallPos(0, 1) = **New** Point(posi + M, posj)  *' Right Wall*  maze(i, j).wallPos(1, 0) = **New** Point(posi + M, posj)  maze(i, j).wallPos(1, 1) = **New** Point(posi + M, posj + M)  *' Bottom Wall*  maze(i, j).wallPos(2, 0) = **New** Point(posi, posj + M)  maze(i, j).wallPos(2, 1) = **New** Point(posi + M, posj + M)  *' Left Wall*  maze(i, j).wallPos(3, 0) = **New** Point(posi, posj)  maze(i, j).wallPos(3, 1) = **New** Point(posi, posj + M)  **Next**  **Next**  **Next**    *' Setting Maze Entry and Exit*  setMazeEntryExit()  **End** **Sub**  **Private** **Sub** setMazeEntryExit()  **If** imageInputted = **True** **Then**  *' Randomly picks a side*  **Dim** side **As** Integer = rnd.**Next**(1, 2)    **If** side = 1 **Then**  mazeEntry = **New** Point(largestComponent.OrderByDescending(**Function**(p) p.X).First())  mazeExit = **New** Point(largestComponent.OrderByDescending(**Function**(p) p.X).Last())  **Else**  **Dim** orderedList **As** List(**Of** Point) = largestComponent.OrderByDescending(**Function**(p) p.Y)  mazeEntry = **New** Point(orderedList.First())  mazeExit = **New** Point(orderedList.Last())  **End** **If**  **Else**  **Select** **Case** mazeEntryType  **Case** "Random"  Randomize()  **Dim** randomType **As** Integer = rnd.**Next**(0, 4)  *' Chooses randomly what type of maze entry it will be*  **Select** **Case** randomType  **Case** 0 *' Start at a random top postion, finish at a random bottom position*  mazeEntry = **New** Point(rnd.**Next**(1, width), 1)  mazeExit = **New** Point(rnd.**Next**(1, width), height - 1)  **Case** 1 *' Start at a random bottom postion, finish at a random top position*  mazeEntry = **New** Point(rnd.**Next**(1, width), height - 1)  mazeExit = **New** Point(rnd.**Next**(1, width), 1)  **Case** 2 *' Start at a random right postion, finish at a random left positon*  mazeEntry = **New** Point(1, rnd.**Next**(1, height))  mazeExit = **New** Point(width - 1, rnd.**Next**(1, height))  **Case** 3 *' Start at a random left postion, finish at a random right positon*  mazeEntry = **New** Point(width - 1, rnd.**Next**(1, height))  mazeExit = **New** Point(1, rnd.**Next**(1, height))  **End** **Select**  **Case** "Top - Bottom"  mazeEntry = **New** Point(Math.Round(width / 2), 1)  mazeExit = **New** Point(Math.Round(width / 2), height - 1)  **Case** "Right - Left"  mazeEntry = **New** Point(1, Math.Round(height / 2))  mazeExit = **New** Point(width - 1, Math.Round(height / 2))  **Case** "Diagonal"  mazeEntry = **New** Point(1, 1)  mazeExit = **New** Point(width - 1, height - 1)  **End** **Select**  **End** **If**  *' Setting the entry cell with the mazeEntryBool*  maze(mazeEntry.X, mazeEntry.Y).mazeEntryBool = **True**  maze(mazeEntry.X, mazeEntry.Y).mazeWallBool = **False**  *' Setting the exit cell with the mazeExitBool*  maze(mazeExit.X, mazeExit.Y).mazeExitBool = **True**  maze(mazeExit.X, mazeExit.Y).mazeWallBool = **False**  **End** **Sub**  **Private** **Sub** drawMaze() *' If False is passed through then the background cells will be drawn*  *' Resets old timer, Starts new timer, Upates Status*  drawTimer.Reset()  drawTimer.Start()  statusLbl.Text = "Status: Drawing Maze"  statusLbl.Update()  *' Create brush objects for each color*  **Dim** bgBrush **As** **New** SolidBrush(bgColour)  **Dim** mazeBrush **As** **New** SolidBrush(mazeColour)  **Dim** solvedBrush **As** **New** SolidBrush(solveColour)  **Dim** entryBrush **As** **New** SolidBrush(Color.Green)  **Dim** exitBrush **As** **New** SolidBrush(Color.Red)    **For** **Each** cell **In** maze  *' Determine the fill color based on cell properties*  **Dim** fillBrush **As** Brush = bgBrush  **If** cell.mazeWallBool **Then**  fillBrush = mazeBrush  **End** **If**  **If** cell.mazeEntryBool **Then**  fillBrush = entryBrush  **End** **If**  **If** cell.mazeExitBool **Then**  fillBrush = exitBrush  **End** **If**  **If** cell.mazeSolved **Then**  fillBrush = solvedBrush  **End** **If**    *' Draw the cell background and fill*  mazeImageGraphics.FillRectangle(fillBrush, cell.wallPos(0, 0).X, cell.wallPos(1, 0).Y, M, M)    *' Draw the walls*  cell.drawWalls()  **Next**    *' Dispose of the brush objects*  bgBrush.Dispose()  mazeBrush.Dispose()  entryBrush.Dispose()  exitBrush.Dispose()  solvedBrush.Dispose()  drawTimer.**Stop**()  **End** **Sub**  **Private** **Sub** resetMaze()  instantAnimationBtn.Checked = **True**  *' Clear the collections*  solvedVisited.Clear()  helperPath.Clear()  path.Clear()  passedPath.Clear()    *' Reset the mazeSolved property for each cell*  **For** x **As** Integer = 0 **To** width - 1  **For** y **As** Integer = 0 **To** height - 1  maze(x, y).mazeSolved = **False**  **Next**  **Next**    **If** resetType = "G" **Then**  **If** generationAlgorithm = "DFS Backtracker" **Then**  **If** imageInputted **Then**  **For** **Each** component **In** imgComponents  **Dim** point **As** Point = component(rnd.**Next**(0, component.Count))  DFSbacktracker(point.X, point.Y)  **Next**  **Else**  DFSbacktracker(rnd.**Next**(1, width), rnd.**Next**(1, height))  **End** **If**  **ElseIf** generationAlgorithm = "Randomised Prims" **Then**  **If** imageInputted **Then**  **For** **Each** component **In** imgComponents  **Dim** point **As** Point = component(rnd.**Next**(0, component.Count))  randomisedPrims(point.X, point.Y)  **Next**  **Else**  randomisedPrims(rnd.**Next**(1, width), rnd.**Next**(1, height))  **End** **If**  **ElseIf** generationAlgorithm = "Kruskal 's" **Then**  **If** imageInputted **Then**  MsgBox("You can't use this algorithm with mazes." & vbCrLf & "Try another one!", MsgBoxStyle.OkOnly, "Invalid Input")  **Exit** **Sub**  **Else**  kruskals()  **End** **If**  **ElseIf** generationAlgorithm = "Aldous-Border" **Then**  **If** imageInputted **Then**  MsgBox("You can't use this algorithm with mazes." & vbCrLf & "Try another one!", MsgBoxStyle.OkOnly, "Invalid Input")  **Exit** **Sub**  **Else**  width += 1  height += 1  aldousBroder(rnd.**Next**(1, width), rnd.**Next**(1, height))  width -= 1  height -= 1  **End** **If**  **End** **If**  **ElseIf** resetType = "S" **Then**  *' Checks what solving algorithm user has chosen*  **If** solveAlgorithm = "Dijkstra's" **Then**  dijkstra()  **ElseIf** solveAlgorithm = "Breath Frist Search" **Then**  BFS()  **ElseIf** solveAlgorithm = "A\*" **Then**  Astar()  **ElseIf** solveAlgorithm = "Wall Follower LHR" **Then**  wallFollower("LHR")  **ElseIf** solveAlgorithm = "Wall Follower RHR" **Then**  wallFollower("RHR")  **End** **If**  **End** **If**        *' Redraw the maze*  drawMaze()  mazeBox.Image = mazeImage  mazeBox.Update()    instantAnimationBtn.Checked = **False**  **End** **Sub**  **Private** **Sub** animationLock(Lock **As** Boolean) *' Locks all inputs to prevent backlogging and crashes*  **If** Lock = **True** **Then**  *' Generate Button*  generateBtn.BackColor = Color.FromArgb(18, 73, 18)  generateBtn.Enabled = **False**  *' Solve Button*  solveBtn.BackColor = Color.FromArgb(112, 22, 22)  solveBtn.Enabled = **False**  *' Dead End Remover Button*  deadEndRemoverBtn.BackColor = Color.FromArgb(0, 73, 73)  deadEndRemoverBtn.Enabled = **False**  *' Download Button*  downloadBtn.BackColor = Color.FromArgb(19, 28, 40)  downloadBtn.Enabled = **False**  *' Rest Button*  imageInputBtn.BackColor = Color.FromArgb(19, 28, 40)  imageInputBtn.Enabled = **False**  bgColourBtn.BackColor = Color.FromArgb(19, 28, 40)  bgColourBtn.Enabled = **False**  mazeColourBtn.BackColor = Color.FromArgb(19, 28, 40)  mazeColourBtn.Enabled = **False**  solveColourBtn.BackColor = Color.FromArgb(19, 28, 40)  solveColourBtn.Enabled = **False**  cancelAnimationBtn.BackColor = Color.MediumSlateBlue  cancelAnimationBtn.Enabled = **True**  *' Rest TextBoxs*  widthTxtBox.BackColor = Color.FromArgb(19, 28, 40)  widthTxtBox.Enabled = **False**  heightTxtBox.BackColor = Color.FromArgb(19, 28, 40)  heightTxtBox.Enabled = **False**  deadEndRemoverTxtBox.BackColor = Color.FromArgb(19, 28, 40)  deadEndRemoverTxtBox.Enabled = **False**  *' Rest ComboBoxs*  generationCombo.BackColor = Color.FromArgb(19, 28, 40)  generationCombo.Update()  solveCombo.BackColor = Color.FromArgb(19, 28, 40)  solveCombo.Update()  mazeEntryCombo.BackColor = Color.FromArgb(19, 28, 40)  mazeEntryCombo.Update()  **Else**  *' Generate Button*  generateBtn.BackColor = Color.ForestGreen  generateBtn.Enabled = **True**  *' Solve Button*  solveBtn.BackColor = Color.Firebrick  solveBtn.Enabled = **True**  *' Dead End Remover Button*  deadEndRemoverBtn.BackColor = Color.DarkCyan  deadEndRemoverBtn.Enabled = **True**  *' Download Button*  downloadBtn.BackColor = Color.PaleVioletRed  downloadBtn.Enabled = **True**  *' Rest Button*  imageInputBtn.BackColor = Color.FromArgb(40, 60, 86)  imageInputBtn.Enabled = **True**  bgColourBtn.BackColor = Color.FromArgb(40, 60, 86)  bgColourBtn.Enabled = **True**  mazeColourBtn.BackColor = Color.FromArgb(40, 60, 86)  mazeColourBtn.Enabled = **True**  solveColourBtn.BackColor = Color.FromArgb(40, 60, 86)  solveColourBtn.Enabled = **True**  cancelAnimationBtn.BackColor = Color.DarkSlateBlue  cancelAnimationBtn.Enabled = **False**  *' Rest TextBoxs*  widthTxtBox.BackColor = Color.FromArgb(40, 60, 86)  widthTxtBox.Enabled = **True**  heightTxtBox.BackColor = Color.FromArgb(40, 60, 86)  heightTxtBox.Enabled = **True**  deadEndRemoverTxtBox.BackColor = Color.FromArgb(40, 60, 86)  deadEndRemoverTxtBox.Enabled = **True**  *' Rest ComboBoxs*  generationCombo.BackColor = Color.FromArgb(40, 60, 86)  generationCombo.Enabled = **True**  solveCombo.BackColor = Color.FromArgb(40, 60, 86)  solveCombo.Enabled = **True**  mazeEntryCombo.BackColor = Color.FromArgb(40, 60, 86)  mazeEntryCombo.Enabled = **True**  **End** **If**  **End** **Sub**  *' Interpolate between two colors based on a ratio (0.0 to 1.0)*  **Function** interpolateColour(color1 **As** Color, color2 **As** Color, ratio **As** Double) **As** Color  **Dim** r **As** Double = Int(color1.R) + (Int(color2.R) - Int(color1.R)) \* ratio  **Dim** g **As** Double = Int(color1.G) + (Int(color2.G) - Int(color1.G)) \* ratio  **Dim** b **As** Double = Int(color1.B) + (Int(color2.B) - Int(color1.B)) \* ratio  **Return** Color.FromArgb((r), (g), (b))  **End** **Function**  *' Generating*  **Private** **Sub** DFSbacktracker(**ByVal** x **As** Integer, **ByVal** y **As** Integer)  generationStack.Push(**New** Point(x, y))  *' Check if they want animations*  **If** instantAnimationBtn.Checked = **False** **Then** *' Want Animations*  animationLock(**True**)  generationPointTimer.Enabled = **True**  **Else**  **Dim** direction **As** Integer    **While** generationStack.Count > 0  **Dim** currentCell = generationStack.Peek()  **Dim** cell = maze(currentCell.X, currentCell.Y)    *' Mark current cell as visited*  cell.visited = **True**    *' Get a list of unvisited neighbors*  **Dim** unvisitedNeighbors = cell.checkUnvistedNeighbours()    **If** unvisitedNeighbors.All(**Function**(p) p.Equals(Point.Empty)) = **True** **Then**  generationStack.Pop()  **Continue** **While**  **End** **If**    *' Make a new list that only contains the non empty values from neighbour*  **Dim** validNeigbours **As** **New** List(**Of** Point)  **For** **Each** point **In** unvisitedNeighbors  **If** point <> Point.Empty **Then**  validNeigbours.Add(point)  **End** **If**  **Next**    *' Randomly pick a valid neighbour. Find the index of that point within the orginal neighbour list and set that to direction*  Randomize()  direction = unvisitedNeighbors.IndexOf(validNeigbours(rnd.**Next**(0, validNeigbours.Count())))    *' Break the wall between the current cell and the chosen neighbor*  **Dim** randomNeighbor = cell.breakWall(direction)    *' Add the neighbor to the generationStack*  generationStack.Push(randomNeighbor)  **End** **While**  **End** **If**  **End** **Sub**    **Public** **Sub** randomisedPrims(**ByVal** x **As** Integer, **ByVal** y **As** Integer)  *' Clear visitedCells and primsWalls to prepare for the new maze generation*  visitedCells.Clear()  primsWalls.Clear()    *' Start with an arbitrary cell*  **Dim** currentCell **As** Cell = maze(x, y)  currentCell.visited = **True**  visitedCells.Add(currentCell)    *' Add the walls of the initial cell to the list*  **For** i **As** Integer = 0 **To** 3  **If** **Not** currentCell.walls(i) **Then** **Continue** **For**  primsWalls.Add(Tuple.Create(**New** Point(x, y), i))  **Next**    *' Check if they want animations*  **If** instantAnimationBtn.Checked = **False** **Then** *' Want Animations*  animationLock(**True**)  generationPointTimer.Enabled = **True**  **Else**  *' Continue until all cells are visited and there are no more walls to process*  **While** visitedCells.Count < totalCells **AndAlso** primsWalls.Count > 0  *' Choose a wall connected to the visited cells uniformly at random*  **Dim** randomIndex **As** Integer = rnd.**Next**(primsWalls.Count)  **Dim** randomWall **As** Tuple(**Of** Point, Integer) = primsWalls(randomIndex)  **Dim** cellCoords **As** Point = randomWall.Item1  **Dim** direction **As** Integer = randomWall.Item2  **Dim** cell **As** Cell = maze(cellCoords.X, cellCoords.Y)  *' If the wall separates a visited cell from an unvisited cell*  **Dim** neighbour **As** Point = cell.checkUnvistedNeighbours()(direction)  **If** neighbour <> Point.Empty **AndAlso** **Not** maze(neighbour.X, neighbour.Y).visited **Then**  *' Remove the wall and mark the unvisited cell as visited*  cell.breakWall(direction)  maze(neighbour.X, neighbour.Y).visited = **True**  visitedCells.Add(maze(neighbour.X, neighbour.Y))  *' Add the neighboring walls of the cell to the walls list*  **For** i **As** Integer = 0 **To** 3  **If** **Not** maze(neighbour.X, neighbour.Y).walls(i) **Then** **Continue** **For**  primsWalls.Add(Tuple.Create(**New** Point(neighbour.X, neighbour.Y), i))  **Next**  **End** **If**    *' Remove the wall from the list to avoid reprocessing it*  primsWalls.RemoveAt(randomIndex)  **End** **While**  **End** **If**  **End** **Sub**    **Public** **Sub** kruskals()  *' Clear the kurskWalls list to prepare for the new maze generation*  kurskWalls.Clear()    *' Create a list of all possible walls in the maze*  **Dim** k **As** **New** List(**Of** Tuple(**Of** Point, Integer))  **For** x **As** Integer = 0 **To** width - 1  **For** y **As** Integer = 0 **To** height - 1  **For** i **As** Integer = 0 **To** 1  **If** maze(x, y).mazeWallBool = **False** **Then**  kurskWalls.Add(Tuple.Create(**New** Point(x, y), i))  **End** **If**  **Next**  **Next**  **Next**    *' Shuffle the list of walls randomly*  **For** i **As** Integer = kurskWalls.Count - 1 **To** 1 **Step** -1  **Dim** j **As** Integer = random.**Next**(i + 1)  **Dim** temp **As** Tuple(**Of** Point, Integer) = kurskWalls(i)  kurskWalls(i) = kurskWalls(j)  kurskWalls(j) = temp  **Next**    *' Initialize the sets for each cell*  **Dim** setId **As** Integer = 0  **For** x **As** Integer = 0 **To** width - 1  **For** y **As** Integer = 0 **To** height - 1  kursSets(**New** Point(x, y)) = setId  setId += 1  **Next**  **Next**    *' Check if they want animations*  **If** instantAnimationBtn.Checked = **False** **Then** *' Want Animations*  animationLock(**True**)  generationPointTimer.Enabled = **True**  **Else**  *' Iterate through the shuffled list of walls*  **For** **Each** wall **As** Tuple(**Of** Point, Integer) **In** kurskWalls  **Dim** cellCoords **As** Point = wall.Item1  **Dim** direction **As** Integer = wall.Item2  **Dim** cell **As** Cell = maze(cellCoords.X, cellCoords.Y)  *' Calculate the neighboring cell coordinates based on the direction*  **If** direction = 0 **Then**  kursNeighbourCoords = **New** Point(cellCoords.X, cellCoords.Y - 1)  **Else**  kursNeighbourCoords = **New** Point(cellCoords.X + 1, cellCoords.Y)  **End** **If**    *' Check if the neighboring cell is within the maze bounds*  **If** kursNeighbourCoords.X >= 0 **AndAlso** kursNeighbourCoords.X < width **AndAlso** kursNeighbourCoords.Y >= 0 **AndAlso** kursNeighbourCoords.Y < height **Then**  *' Check if the cells connected by the wall are not in the same set*  **If** kursSets(cellCoords) <> kursSets(kursNeighbourCoords) **Then**  *' Remove the wall to connect the cells*  cell.breakWall(direction) *' Merge the sets of the two cells*  **Dim** setIdToReplace **As** Integer = kursSets(kursNeighbourCoords)  **Dim** setIdToKeep **As** Integer = kursSets(cellCoords)  **For** **Each** key **As** Point **In** kursSets.Keys.ToList()  **If** kursSets(key) = setIdToReplace **Then**  kursSets(key) = setIdToKeep  **End** **If**  **Next**  **End** **If**  **End** **If**  **Next**  kursCurrentWallIndex = 0  **End** **If**  **End** **Sub**    **Public** **Sub** aldousBroder(startX **As** Integer, startY **As** Integer)  abTotalCells = (width - 2) \* (height - 2)  currentX = startX  currentY = startY    **If** abHasBeenAnimated = **False** **Then**  abVisitedCells = 0  **End** **If**      *' Mark the starting cell as visited*  maze(currentX, currentY).visited = **True**  abVisitedCells += 1    *' Check if they want animations*  **If** instantAnimationBtn.Checked = **False** **Then** *' Want Animations*  animationLock(**True**)  generationPointTimer.Enabled = **True**  **Else**  *' Continue until all cells have been visited*  **While** abVisitedCells < abTotalCells  *' Move to a random neighboring cell*  **Dim** randomDirectionIndex **As** Integer = rnd.**Next**(directions.Length)  **Dim** newX **As** Integer = currentX + directions(randomDirectionIndex).X  **Dim** newY **As** Integer = currentY + directions(randomDirectionIndex).Y    *' Check if the new position is within the maze bounds, excluding the border cells*  **If** newX > 0 **AndAlso** newX < width - 1 **AndAlso** newY > 0 **AndAlso** newY < height - 1 **Then**  *' If the neighboring cell has not been visited yet, remove the wall between the current cell and the neighboring cell*  **If** **Not** maze(newX, newY).visited **Then**  maze(currentX, currentY).breakWall(randomDirectionIndex)  maze(newX, newY).visited = **True**  abVisitedCells += 1  **End** **If**    *' Set the current position to the new position*  currentX = newX  currentY = newY  **End** **If**  **End** **While**  abHasBeenAnimated = **False**  **End** **If**  **End** **Sub**    **Private** **Sub** generationPointTimer\_Tick(sender **As** Object, e **As** EventArgs)  *' Cancel animation if needed*  **If** cancelAnimation = **True** **Then**  generationPointTimer.Enabled = **False**  resetType = "G"  resetMaze()  animationLock(**False**)  cancelAnimation = **False**  **Exit** **Sub**  **End** **If**    **If** generationAlgorithm = "DFS Backtracker" **Then**  **If** generationStack.Count > 0 **Then**  **Dim** currentCell = generationStack.Peek()  **Dim** cell = maze(currentCell.X, currentCell.Y)    *' Highlight the top of the stack*  **If** currentCell <> mazeEntry **And** currentCell <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.Yellow), currentCell.X \* M, currentCell.Y \* M, M, M)  maze(currentCell.X, currentCell.Y).drawWalls()  **End** **If**      *' Mark current cell as visited*  cell.visited = **True**    *' Get a list of unvisited neighbors*  **Dim** unvisitedNeighbors = cell.checkUnvistedNeighbours()    **If** unvisitedNeighbors.All(**Function**(p) p.Equals(Point.Empty)) = **True** **Then**  generationStack.Pop()  **Else**  *' Make a new list that only contains the non empty values from neighbour*  **Dim** validNeigbours **As** **New** List(**Of** Point)  **For** **Each** point **In** unvisitedNeighbors  **If** point <> Point.Empty **Then**  validNeigbours.Add(point)  **End** **If**  **Next**    *' Randomly pick a valid neighbour. Find the index of that point within the orginal neighbour list and set that to direction*  **Dim** direction = unvisitedNeighbors.IndexOf(validNeigbours(rnd.**Next**(0, validNeigbours.Count())))    *' Break the wall between the current cell and the chosen neighbor*  **Dim** randomNeighbor = cell.breakWall(direction)    *' Add the neighbor to the stack*  generationStack.Push(randomNeighbor)  **End** **If**    *' Update the maze and the maze display*  mazeBox.Image = mazeImage  mazeBox.Update()    *' Resrt the top of the stack*  **If** currentCell <> mazeEntry **And** currentCell <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.White), currentCell.X \* M, currentCell.Y \* M, M, M)  maze(currentCell.X, currentCell.Y).drawWalls()  **End** **If**  **Else**  drawMaze()  *' Update the maze and the maze display*  mazeBox.Image = mazeImage  mazeBox.Update()    animationLock(**False**)  *' Stop the timer when the maze is complete*  generationPointTimer.Enabled = **False**  **End** **If**  **ElseIf** generationAlgorithm = "Randomised Prims" **Then**  **If** visitedCells.Count < totalCells **AndAlso** primsWalls.Count > 0 **Then**  **Dim** randomIndex **As** Integer = rnd.**Next**(primsWalls.Count)  **Dim** randomWall **As** Tuple(**Of** Point, Integer) = primsWalls(randomIndex)  **Dim** cellCoords **As** Point = randomWall.Item1  **Dim** direction **As** Integer = randomWall.Item2  **Dim** cell **As** Cell = maze(cellCoords.X, cellCoords.Y)    *' Highlight all walls that could be collapsed*  **For** **Each** wall **As** Tuple(**Of** Point, Integer) **In** primsWalls  **If** wall.Item1 <> mazeEntry **And** wall.Item1 <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.Yellow), wall.Item1.X \* M, wall.Item1.Y \* M, M, M)  maze(wall.Item1.X, wall.Item1.Y).drawWalls()  **End** **If**  **Next**    mazeBox.Image = mazeImage  mazeBox.Update()  *' Rest all walls that could be collapsed*  **For** **Each** wall **As** Tuple(**Of** Point, Integer) **In** primsWalls  **If** wall.Item1 <> mazeEntry **And** wall.Item1 <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(bgColour), wall.Item1.X \* M, wall.Item1.Y \* M, M, M)  maze(wall.Item1.X, wall.Item1.Y).drawWalls()  **End** **If**  **Next**    **Dim** neighbour **As** Point = cell.checkUnvistedNeighbours()(direction)  **If** neighbour <> Point.Empty **AndAlso** **Not** maze(neighbour.X, neighbour.Y).visited **Then**  *' Remove the wall and mark the unvisited cell as visited*  **Dim** conCell **As** Point = cell.breakWall(direction) *' Merge the sets of the two cells*  **If** neighbour <> mazeEntry **And** neighbour <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(bgColour), neighbour.X \* M, neighbour.Y \* M, M, M)  maze(neighbour.X, neighbour.Y).drawWalls()  **End** **If**      *' Draw the cell background and fill*  **If** **New** Point(cell.x, cell.y) <> mazeEntry **And** **New** Point(cell.x, cell.y) <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(bgColour), cell.wallPos(0, 0).X, cell.wallPos(1, 0).Y, M, M)  cell.drawWalls()  **End** **If**  *' Update the maze and the maze display*  maze(neighbour.X, neighbour.Y).visited = **True**  visitedCells.Add(maze(neighbour.X, neighbour.Y))    *' Add the neighboring walls of the cell to the walls list*  **For** i **As** Integer = 0 **To** 3  **If** **Not** maze(neighbour.X, neighbour.Y).walls(i) **Then** **Continue** **For**  primsWalls.Add(Tuple.Create(**New** Point(neighbour.X, neighbour.Y), i))  **Next**  **End** **If**  *' Remove the wall from the list to avoid reprocessing it*  primsWalls.RemoveAt(randomIndex)  **Else**  drawMaze()  *' Update the maze and the maze display*  mazeBox.Image = mazeImage  mazeBox.Update()    animationLock(**False**)  *' Stop the timer when the maze is complete*  generationPointTimer.Enabled = **False**  **End** **If**  **ElseIf** generationAlgorithm = "Kruskal 's" **Then**  **If** kursCurrentWallIndex < kurskWalls.Count **Then**  **Dim** wall **As** Tuple(**Of** Point, Integer) = kurskWalls(kursCurrentWallIndex)  **Dim** cellCoords **As** Point = wall.Item1  **Dim** direction **As** Integer = wall.Item2  **Dim** cell **As** Cell = maze(cellCoords.X, cellCoords.Y)  *' Calculate the neighboring cell coordinates based on the direction*  **If** direction = 0 **Then**  kursNeighbourCoords = **New** Point(cellCoords.X, cellCoords.Y - 1)  **Else**  kursNeighbourCoords = **New** Point(cellCoords.X + 1, cellCoords.Y)  **End** **If**    *' Check if the neighboring cell is within the maze bounds*  **If** kursNeighbourCoords.X >= 0 **AndAlso** kursNeighbourCoords.X < width **AndAlso** kursNeighbourCoords.Y >= 0 **AndAlso** kursNeighbourCoords.Y < height **Then**  *' Check if the cells connected by the wall are not in the same set*  **If** kursSets(cellCoords) <> kursSets(kursNeighbourCoords) **Then**  *' Remove the wall to connect the cells*  cell.breakWall(direction) *' Merge the sets of the two cells*  *' Remove the wall to connect the cells*  **Dim** conCell **As** Point = cell.breakWall(direction) *' Merge the sets of the two cells*  *' Draw the cell background and fill*  **If** **New** Point(cell.x, cell.y) <> mazeEntry **And** **New** Point(cell.x, cell.y) <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(bgColour), cell.wallPos(0, 0).X, cell.wallPos(1, 0).Y, M, M)  cell.drawWalls()  **End** **If**    **If** maze(conCell.X, conCell.Y).mazeWallBool = **False** **And** conCell <> mazeEntry **And** conCell <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(bgColour), maze(conCell.X, conCell.Y).wallPos(0, 0).X, maze(conCell.X, conCell.Y).wallPos(1, 0).Y, M, M)  maze(conCell.X, conCell.Y).drawWalls()  **End** **If**  mazeBox.Image = mazeImage  mazeBox.Update()    generationPointTimer.Enabled = **False**  **Dim** setIdToReplace **As** Integer = kursSets(kursNeighbourCoords)  **Dim** setIdToKeep **As** Integer = kursSets(cellCoords)  **For** **Each** key **As** Point **In** kursSets.Keys.ToList()  **If** kursSets(key) = setIdToReplace **Then**  kursSets(key) = setIdToKeep  **End** **If**  **Next**  **End** **If**  **End** **If**    *' Increment the wall index*  kursCurrentWallIndex += 1  generationPointTimer.Enabled = **True**  **Else**  *' Stop the timer when the algorithm is finished*  animationLock(**False**)  generationPointTimer.Enabled = **False**    **End** **If**  **ElseIf** generationAlgorithm = "Aldous-Border" **Then**  abHasBeenAnimated = **True**  *' Continue until all cells have been visited*  **If** abVisitedCells < abTotalCells **Then**  *' Move to a random neighboring cell*  **Dim** randomDirectionIndex **As** Integer = rnd.**Next**(directions.Length)  **Dim** newX **As** Integer = currentX + directions(randomDirectionIndex).X  **Dim** newY **As** Integer = currentY + directions(randomDirectionIndex).Y    *' Check if the new position is within the maze bounds, excluding the border cells*  **If** newX > 0 **AndAlso** newX < width - 1 **AndAlso** newY > 0 **AndAlso** newY < height - 1 **Then**  *' If the neighboring cell has not been visited yet, remove the wall between the current cell and the neighboring cell*  **If** **Not** maze(newX, newY).visited **Then**  **Dim** conCell **As** Point  conCell = maze(currentX, currentX).breakWall(randomDirectionIndex)    **If** **New** Point(currentX, currentX) <> mazeEntry **And** **New** Point(currentX, currentX) <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.Yellow), maze(currentX, currentX).wallPos(0, 0).X, maze(currentX, currentX).wallPos(1, 0).Y, M, M)  maze(currentX, currentX).drawWalls()  **End** **If**    **If** maze(conCell.X, conCell.Y).mazeWallBool = **False** **And** conCell <> mazeEntry **And** conCell <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(bgColour), maze(conCell.X, conCell.Y).wallPos(0, 0).X, maze(conCell.X, conCell.Y).wallPos(1, 0).Y, M, M)  maze(conCell.X, conCell.Y).drawWalls()  **End** **If**    mazeBox.Image = mazeImage  mazeBox.Update()    maze(newX, newY).visited = **True**  abVisitedCells += 1  **End** **If**  **If** **New** Point(currentX, currentX) <> mazeEntry **And** **New** Point(currentX, currentX) <> mazeExit **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(bgColour), maze(currentX, currentX).wallPos(0, 0).X, maze(currentX, currentX).wallPos(1, 0).Y, M, M)  maze(currentX, currentX).drawWalls()  **End** **If**      *' Set the current position to the new position*  currentX = newX  currentY = newY  **End** **If**  **Else**  abHasBeenAnimated = **False**  *' Stop the timer when the algorithm is finished*  animationLock(**False**)  generationPointTimer.Enabled = **False**  **End** **If**  **End** **If**  **End** **Sub**    *' Solving*  **Private** **Function** distanceCalc(a **As** Point, b **As** Point) **As** Double  **Return** Math.Abs(a.X - b.X) + Math.Abs(a.Y - b.Y)  **End** **Function**  **Private** **Sub** dijkstra()  *' Clear the gWeight dictionary and solvedVisted list*  gWeights.Clear()  solvedVisited.Clear()    *' Initialize the parents dictionary and the priority queue*  **Dim** parents **As** **New** Dictionary(**Of** Point, Point)  **Dim** pQueue **As** **New** PriorityQueue(**Of** Double, Point)()    *' Set the weight of entry to 0 and enqeueu to the priority queue*  gWeights(mazeEntry) = 0  pQueue.Enqueue(0, mazeEntry)    *' Continue until the proprity queue is empty*  **While** **Not** pQueue.isEmpty()  *' Dequeue the node with the lowest weight from the priority queue*  **Dim** current **As** Point = pQueue.Dequeue()    *' Add the dequeued node to the solvedVisited queue*  solvedVisited.Enqueue(current)    *' Check for exit*  **If** current = mazeExit **Then**  **Exit** **While**  **End** **If**    *' Go through each connect neighbour of the current node*  **For** **Each** neighbour **In** maze(current.X, current.Y).connectedCell  *' Calculate weight of neighbour. In this to get to a connected node holds a weight of 1*  **Dim** weight **As** Double = gWeights(current) + 1    *' If the neighbour's weight is not already in the dictionary, set it to a large value*  **If** **Not** gWeights.ContainsKey(neighbour) **Then**  gWeights(neighbour) = Double.MaxValue  **End** **If**    *' Update the neighbours weight and parent if the calculated weight is less*  **If** weight < gWeights(neighbour) **Then**  gWeights(neighbour) = weight  maxWeight = Math.Max(maxWeight, weight)  parents(neighbour) = current    *' If the neight is not in the priority queue, add it*  **If** **Not** pQueue.Contains(neighbour) **Then**  pQueue.Enqueue(weight, neighbour)  **End** **If**  **End** **If**  **Next**  **End** **While**    *' Reconstruct the path*  **Dim** currentNode **As** Point = mazeExit  **While** currentNode <> mazeEntry **AndAlso** parents.ContainsKey(currentNode)  currentNode = parents(currentNode)  **If** currentNode <> mazeEntry **Then**  path.Enqueue(currentNode)  **End** **If**  **End** **While**    *' Check If they want animations*  **If** instantAnimationBtn.Checked = **True** **Or** cancelAnimation = **True** **Then**  *' Mark the path as solved*  **For** **Each** node **In** path  maze(node.X, node.Y).mazeSolved = **True**  **Next**  path.Clear()  **ElseIf** instantAnimationBtn.Checked = **False** **Then**  *' Enable animations and lock other controls*  animationLock(**True**)  heatMapAnimationTimer.Enabled = **True**  **End** **If**  **End** **Sub**  **Private** **Sub** BFS()  *' Clear solvedVisted list and branchingPoints list*  solvedVisited.Clear()  branchingPoints.Clear()    *' Initialize the queue, parent dictionary and currentNod*  **Dim** queue **As** **New** Queue(**Of** Point)  **Dim** parents **As** **New** Dictionary(**Of** Point, Point)()  **Dim** currentNode **As** Point    *' Enqueue the starting point*  queue.Enqueue(mazeEntry)    *' Continue searching until the queue is empty*  **While** queue.Count <> 0  *' Dequeue the next code in the queue*  currentNode = queue.Dequeue()    *' Check for the exit*  **If** currentNode = mazeExit **Then**  **Exit** **While**  **End** **If**    *' Go through each connected cell of currentNode*  **For** **Each** point **In** maze(currentNode.X, currentNode.Y).connectedCell  **If** **Not** solvedVisited.Contains(point) **Then**  solvedVisited.Enqueue(point)  parents(point) = currentNode  queue.Enqueue(point)  **End** **If**  **Next**    **If** maze(currentNode.X, currentNode.Y).connectedCell.Count > 2 **Then**  branchingPoints.Add(currentNode)  **End** **If**  **End** **While**    *' Reconstruct the path*  currentNode = mazeExit  **While** currentNode <> mazeEntry **AndAlso** parents.ContainsKey(currentNode)  currentNode = parents(currentNode)  **If** currentNode <> mazeEntry **Then**  path.Enqueue(currentNode)  **End** **If**  **End** **While**    *' Check if they want animations*  **If** instantAnimationBtn.Checked = **True** **Or** cancelAnimation = **True** **Then**  *' Mark the path as solved*  **For** **Each** node **In** path  maze(node.X, node.Y).mazeSolved = **True**  **Next**  path.Clear()  **ElseIf** instantAnimationBtn.Checked = **False** **Then**  *' Enable animations and lock other controls*  animationLock(**True**)  heatMapAnimationTimer.Enabled = **True**  **End** **If**  **End** **Sub**  **Private** **Sub** Astar(**Optional** **ByVal** helper **As** Boolean = **False**)  *' Clear the gWeight dictionary and solvedVisted list*  gWeights.Clear()  solvedVisited.Clear()    *' Initailize the parent dictionary and priority queue for open nodes*  **Dim** parents **As** **New** Dictionary(**Of** Point, Point)  **Dim** pQueue **As** **New** PriorityQueue(**Of** Double, Point)()    *' Set the starting points gWeight to 0 and enqueue it with its heuristic value*  gWeights(mazeEntry) = 0  pQueue.Enqueue(distanceCalc(mazeEntry, mazeExit), mazeEntry)    *' Continue searching until the priority queue is empty*  **While** **Not** pQueue.isEmpty()  *' Dequeue the node with the lowest*  **Dim** current **As** Point = pQueue.Dequeue()    *' Add the dequeued node to the visited nodes queue*  solvedVisited.Enqueue(current)    *' Check for exit node*  **If** current = mazeExit **Then**  **Exit** **While**  **End** **If**    *' Go through each connected node of the current node*  **For** **Each** neighbour **In** maze(current.X, current.Y).connectedCell  *' Calculate heuristic weight*  **Dim** heuristicWeight **As** Double = gWeights(current) + distanceCalc(current, neighbour)    *' Set the neighbour's gWeight to a large value if its not in gWeights*  **If** **Not** gWeights.ContainsKey(neighbour) **Then**  gWeights(neighbour) = Double.MaxValue  **End** **If**    *' Update the neighbours gWeight and parent if the heuristic weight is lower*  **If** heuristicWeight < gWeights(neighbour) **Then**  parents(neighbour) = current  gWeights(neighbour) = heuristicWeight  maxWeight = Math.Max(maxWeight, heuristicWeight)  **Dim** fWeight **As** Double = gWeights(neighbour) + distanceCalc(neighbour, mazeExit)    *' If the neighbour is not in the priority queue, add it with the calculated fWeight*  **If** **Not** pQueue.Contains(neighbour) **Then**  pQueue.Enqueue(fWeight, neighbour)  **End** **If**  **End** **If**  **Next**  **End** **While**    *' Reconstruct the path*  **Dim** currentNode **As** Point = mazeExit  **While** currentNode <> mazeEntry **AndAlso** parents.ContainsKey(currentNode)  currentNode = parents(currentNode)  **If** currentNode <> mazeEntry **Then**  path.Enqueue(currentNode)  **End** **If**  **End** **While**    **If** helper = **True** **Then**  helperPath = **New** Queue(**Of** Point)(path)  path.Clear()  **Else**  *' Check if they want animations*  **If** instantAnimationBtn.Checked = **True** **Or** cancelAnimation = **True** **Then**  *' Mark the path as solved*  **For** **Each** node **In** path  maze(node.X, node.Y).mazeSolved = **True**  **Next**  path.Clear()  **ElseIf** instantAnimationBtn.Checked = **False** **Then**  *' Enable animations and lock other controls*  animationLock(**True**)  heatMapAnimationTimer.Enabled = **True**  **End** **If**  **End** **If**  **End** **Sub**  **Private** **Sub** wallFollower(type **As** String)  *' Initailize the current node to the entry*  **Dim** node **As** Point = mazeEntry  *' Initailize the direction queue and index for left/right-hand rule navigation*  **Dim** directionQueue **As** **New** CircularQueue(**Of** Integer)({0, 1, 2, 3})  **Dim** index **As** Integer    *' If the instant animation is unchecked, call A\* for the most effecient path*  **If** instantAnimationBtn.Checked = **False** **Then**  Astar(helper:=**True**)  **End** **If**    *' Continue until the maze exit is reached*  **While** node <> mazeExit  *' Determine next direction base on wall follower type*  **If** type = "LHR" **Then**  index = directionQueue.turnLeft  **ElseIf** type = "RHR" **Then**  index = directionQueue.turnRight  **End** **If**    *' Check if the next cell is connected in the given direction*  **If** maze(node.X, node.Y).checkConnectedCell(index) <> Point.Empty **Then**  node = maze(node.X, node.Y).checkConnectedCell(index)  **Else**  *' If the next cell is not connected, rotate until a connected cell in found*  **Do**  **If** type = "LHR" **Then**  index = directionQueue.turnRight  **ElseIf** type = "RHR" **Then**  index = directionQueue.turnLeft  **End** **If**  **Loop** Until maze(node.X, node.Y).checkConnectedCell(index) <> Point.Empty    *' Move to the connected cell*  node = maze(node.X, node.Y).checkConnectedCell(index)  **End** **If**    *' Dont add entry or exit to the path*  **If** node <> mazeEntry **And** node <> mazeExit **Then**  path.Enqueue(node)  **End** **If**  **End** **While**    *' Check if they want animations*  **If** instantAnimationBtn.Checked = **True** **Or** cancelAnimation = **True** **Then**  *' Mark the path as solved*  **For** **Each** node **In** path  maze(node.X, node.Y).mazeSolved = **True**  **Next**  path.Clear()  **ElseIf** instantAnimationBtn.Checked = **False** **Then**  *' Enable animations and lock other controls*  animationLock(**True**)  heatMapAnimationTimer.Enabled = **True**  **End** **If**  **End** **Sub**  *' Animation*  **Private** **Sub** heatMapAnimationTimer\_Tick(sender **As** Object, e **As** EventArgs)  *' Cancel animation if needed*  **If** cancelAnimation = **True** **Then**  heatMapAnimationTimer.Enabled = **False**  solvedPathAnimationTimer.Enabled = **False**  resetType = "S"  resetMaze()  animationLock(**False**)  cancelAnimation = **False**  **Exit** **Sub**  **End** **If**    *' Creating the heat map*  **If** solveAlgorithm = "A\*" **Or** solveAlgorithm = "Dijkstra's" **Then**  *' If there are nodes in the solvedVisited*  **If** solvedVisited.Count > 0 **Then**  **Dim** node **As** Point = solvedVisited.Dequeue  *' If the node is not the entry or exit*  **If** node <> mazeEntry **And** node <> mazeExit **Then**  *' Calculate the normalized weight and draw heat map*  **Dim** normalisedWeight **As** Double = gWeights(node) / maxWeight  mazeImageGraphics.FillRectangle(**New** SolidBrush(interpolateColour(pinkColor, purpleColor, normalisedWeight)), node.X \* M, node.Y \* M, M, M)    *' Draws walls and update the maze box*  maze(node.X, node.Y).drawWalls()  mazeBox.Image = mazeImage  mazeBox.Update()  **End** **If**  **Else**  *' Clear the solvedVisited queue, draw the maze, and enable the solvedPathAnimationTimer*  solvedVisited.Clear()  drawMaze()  heatMapAnimationTimer.Enabled = **False**  solvedPathAnimationTimer.Enabled = **True**  **End** **If**  **ElseIf** solveAlgorithm = "Wall Follower LHR" **Or** solveAlgorithm = "Wall Follower RHR" **Then**  *' If there are nodes in path*  **If** path.Count > 0 **Then**  **Dim** node **As** Point = path.Dequeue    *' Draw the previous cell in the solved path colour or aqua colour*  **If** passedPath.Count > 0 **Then**  **Dim** previousPath **As** Point = passedPath.Last()  **If** helperPath.Contains(previousPath) **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(solveColour), previousPath.X \* M, previousPath.Y \* M, M, M)  **Else**  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.Aqua), previousPath.X \* M, previousPath.Y \* M, M, M)  **End** **If**  maze(previousPath.X, previousPath.Y).mazeSolved = **True**  maze(previousPath.X, previousPath.Y).drawWalls()  **End** **If**    *' Draw the current cell in the yellow color, draw the walls, and update the maze display*  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.Yellow), node.X \* M, node.Y \* M, M, M)  maze(node.X, node.Y).drawWalls()  passedPath.Add(node)  *' Updates Maze box*  mazeBox.Image = mazeImage  mazeBox.Update()  **Else**  *' Draw the last cell in the solved path color,draw the walls, and update the maze display*  **If** passedPath.Count > 0 **Then**  **Dim** lastPath **As** Point = passedPath.Last()  mazeImageGraphics.FillRectangle(**New** SolidBrush(solveColour), lastPath.X \* M, lastPath.Y \* M, M, M)  maze(lastPath.X, lastPath.Y).drawWalls()  maze(lastPath.X, lastPath.Y).mazeSolved = **True**  drawMaze()  mazeBox.Image = mazeImage  mazeBox.Update()  **End** **If**  *' Clear the helperPath, path, and passedPath queues, disable the timers, and unlock the animation*  helperPath.Clear()  path.Clear()  passedPath.Clear()  animationLock(**False**)  heatMapAnimationTimer.Enabled = **False**  solvedPathAnimationTimer.Enabled = **False**  **End** **If**  **ElseIf** solveAlgorithm = "Breath Frist Search" **Then**  *' If there are nodes in the solvedVisited queue*  **If** solvedVisited.Count > 0 **Then**  **Dim** node **As** Point = solvedVisited.Dequeue  *' If the node is not entry or exit*  **If** node <> mazeEntry **And** node <> mazeExit **Then**  *' Draw the node in yellow if it's a branching point or the solveColour otherwise*  **If** branchingPoints.Contains(node) **Then**  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.Yellow), node.X \* M, node.Y \* M, M, M)  **Else**  mazeImageGraphics.FillRectangle(**New** SolidBrush(solveColour), node.X \* M, node.Y \* M, M, M)  **End** **If**  *'Draw the walls And update the maze display*  maze(node.X, node.Y).drawWalls()  mazeBox.Image = mazeImage  mazeBox.Update()  **End** **If**  **Else**  *' Clear the solvedVisited queue, draw the maze, and enable the solvedPathAnimationTimer*  solvedVisited.Clear()  drawMaze()  solvedPathAnimationTimer.Enabled = **True**  heatMapAnimationTimer.Enabled = **False**  **End** **If**  **End** **If**  **End** **Sub**  **Private** **Sub** solvedPathAnimationTimer\_Tick(sender **As** Object, e **As** EventArgs)  *' Cancel animation if needed*  **If** cancelAnimation = **True** **Then**  heatMapAnimationTimer.Enabled = **False**  solvedPathAnimationTimer.Enabled = **False**  resetType = "S"  resetMaze()  animationLock(**False**)  cancelAnimation = **False**  **Exit** **Sub**  **End** **If**  *' Creating the Solved Path*  **If** path.Count > 0 **Then**  **Dim** currentPath **As** Point = path.Dequeue    *' Draw the previous cell in the solved path color*  **If** passedPath.Count > 0 **Then**  **Dim** previousPath **As** Point = passedPath.Last()  mazeImageGraphics.FillRectangle(**New** SolidBrush(solveColour), previousPath.X \* M, previousPath.Y \* M, M, M)  maze(previousPath.X, previousPath.Y).drawWalls()  **End** **If**    *' Draw the current cell in the yellow color*  mazeImageGraphics.FillRectangle(**New** SolidBrush(Color.Yellow), currentPath.X \* M, currentPath.Y \* M, M, M)  maze(currentPath.X, currentPath.Y).drawWalls()    *' Add the current cell to the passedPath list and mark it as solved*  passedPath.Add(currentPath)  maze(currentPath.X, currentPath.Y).mazeSolved = **True**    *' Updates maze box*  mazeBox.Image = mazeImage  mazeBox.Update()  **Else**  *' If there are no more cells in the path, draw the last cell in the solved path*  **If** passedPath.Count > 0 **Then**  **Dim** lastPath **As** Point = passedPath.Last()  mazeImageGraphics.FillRectangle(**New** SolidBrush(solveColour), lastPath.X \* M, lastPath.Y \* M, M, M)  maze(lastPath.X, lastPath.Y).drawWalls()    *' Updates maze box*  mazeBox.Image = mazeImage  mazeBox.Update()  **End** **If**    *' Disable the solvedPathAnimationTimer, unlock the animation, and clear the path and passedPath list*  solvedPathAnimationTimer.Enabled = **False**  animationLock(**False**)  path.Clear()  passedPath.Clear()  **End** **If**  **End** **Sub**    **Private** **Sub** deadEndRemover()  **Dim** numToBeRemoved **As** Integer  **Dim** deadEnd **As** Point  **Dim** node **As** Point  **Dim** direction **As** Integer  *' Find the deadends*  **For** **Each** cell **In** maze  cell.deadEndFinder()  **Next**  *' Calculate the amount of dead end to remove*  numToBeRemoved = Math.Round(deadEndPos.Count() \* deadEndPercent)  **Dim** removed **As** Integer = 0  **While** removed <> numToBeRemoved  *' Pick a random cell*  deadEnd = deadEndPos(rnd.**Next**(0, deadEndPos.Count))  *' Finds valid indexs*  **Dim** validIdexs **As** **New** List(**Of** Integer)  **For** i **As** Integer = 0 **To** 3  **If** maze(deadEnd.X, deadEnd.Y).walls(i) **Then**  validIdexs.Add(i)  **End** **If**  **Next**  *' Pick a valid wall to break*  **Do**  direction = validIdexs(rnd.**Next**(0, validIdexs.Count))  node = maze(deadEnd.X, deadEnd.Y).breakWall(direction)  **Loop** **While** node.IsEmpty  *' Removes from dead end list as changes the number of deadends removed.*  deadEndPos.Remove(deadEnd)  **If** deadEndPos.Contains(node) **Then**  deadEndPos.Remove(node)  removed += 1  **End** **If**  removed += 1  **End** **While**  deadEndPos.Clear()  animationLock(**False**)  **End** **Sub**  *' USER INPUT START*  **Private** **Sub** generateBtn\_Click(sender **As** Object, e **As** EventArgs)  *' Saves Maze Properties inputted by the user*  *' Checking that the values inputed for width and height are valid*  **If** Integer.TryParse(widthTxtBox.Text, width) **And** width > 2 **And** Integer.TryParse(heightTxtBox.Text, height) **And** height > 2 **Then**  width -= 1  height -= 1  **Else**  MsgBox("Make sure width and height are integers greater than 3", MsgBoxStyle.OkOnly, "Invalid Input")  **Exit** **Sub**  **End** **If**    mazeEntryType = mazeEntryCombo.Text  generationAlgorithm = generationCombo.Text      *' Changes multiplier value depending on the maze size*  **If** Math.Floor(Math.Min(1220 / Int(widthTxtBox.Text), 690 / Int(heightTxtBox.Text))) < 3 **Then**  downlaodGenerated = MsgBox("Maze is too big to display!" & vbCrLf & "Want to download?" & vbCrLf & "WARNING! DEPENDING ON HARDWARE THIS MAY TAKE A LONG TIME", MsgBoxStyle.OkCancel, "Maze too big!")  *' If they want to download the maze will generate, statistics will be show and the maze will be downloaded*  **If** downlaodGenerated = DialogResult.OK **Then**  instantAnimationBtn.Checked = **True**  M = 3  **Else**  *' If they dont wan to download it will reset statistics and exit the sub*  genTimeLbl.Text = "Generation Time: "  solveTimeLbl.Text = "Sove Time: "  drawTimeLbl.Text = "Draw Time: "  deadEndCountLbl.Text = "Dead End Count: "  deadEndTimeLbl.Text = "Dead End Time: "  totalTimeLbl.Text = "Total Time "  **Exit** **Sub**  **End** **If**  **Else**  *' If the multipier is below 3 that means it can be displayed on the form*  M = Math.Floor(Math.Min(1220 / Int(widthTxtBox.Text), 690 / Int(heightTxtBox.Text)))  **End** **If**    *' Intializes the maze*  initializeMaze()  *' Allows program to know whether or not a maze has been generated*  mazeGenerated = **True**    *' Resets old timer, Starts new timer, Upates Status*  statusLbl.Text = "Status: Generating"  statusLbl.Update()  generationTimer.Reset()  generationTimer.Start()    *' Checks what generation algorithm user has chosen*  **If** generationAlgorithm = "DFS Backtracker" **Then**  **If** imageInputted **Then**  **For** **Each** component **In** imgComponents  **Dim** point **As** Point = component(rnd.**Next**(0, component.Count))  DFSbacktracker(point.X, point.Y)  **Next**  **Else**  DFSbacktracker(rnd.**Next**(1, width), rnd.**Next**(1, height))  **End** **If**  **ElseIf** generationAlgorithm = "Randomised Prims" **Then**  **If** imageInputted **Then**  **For** **Each** component **In** imgComponents  **Dim** point **As** Point = component(rnd.**Next**(0, component.Count))  randomisedPrims(point.X, point.Y)  **Next**  **Else**  randomisedPrims(rnd.**Next**(1, width), rnd.**Next**(1, height))  **End** **If**  **ElseIf** generationAlgorithm = "Kruskal 's" **Then**  **If** imageInputted **Then**  MsgBox("You can't use this algorithm with mazes." & vbCrLf & "Try another one!", MsgBoxStyle.OkOnly, "Invalid Input")  **Exit** **Sub**  **Else**  kruskals()  **End** **If**  **ElseIf** generationAlgorithm = "Aldous-Border" **Then**  **If** imageInputted **Then**  MsgBox("You can't use this algorithm with mazes." & vbCrLf & "Try another one!", MsgBoxStyle.OkOnly, "Invalid Input")  **Exit** **Sub**  **Else**  width += 1  height += 1  aldousBroder(rnd.**Next**(1, width), rnd.**Next**(1, height))  width -= 1  height -= 1  **End** **If**  **End** **If**    generationTimer.**Stop**()  *' Draws generate maze*  drawMaze()    *' Checks if it should download the maze*  **If** downlaodGenerated = DialogResult.OK **Then**  downloadMaze()  **Else**  *' Upadtes Maze box*  mazeBox.Image = mazeImage  **End** **If**    *' Updates Statistics*  *' Find the dead end count*  **For** **Each** cell **In** maze  cell.deadEndFinder()  **Next**  deadEndToShow = deadEndPos.Count()  deadEndPos.Clear()  *' Displays Statistics*  genTimeLbl.Text = "Generation Time: " & Str(generationTimer.ElapsedMilliseconds() / 1000) & "s"  solveTimeLbl.Text = "Sove Time: "  drawTimeLbl.Text = "Draw Time: " & Str(drawTimer.ElapsedMilliseconds() / 1000) & "s"  deadEndCountLbl.Text = "Dead End Count: " & Str(deadEndToShow)  deadEndTimeLbl.Text = "Dead End Time: "  totalTimeLbl.Text = "Total Time " & Str((generationTimer.ElapsedMilliseconds() + solveTimer.ElapsedMilliseconds() + drawTimer.ElapsedMilliseconds() + deadEndTimer.ElapsedMilliseconds()) / 1000) & "s"  *' Resets Status, ' Resets Dialog Result*  statusLbl.Text = "Status: Doing Nothing"  downlaodGenerated = DialogResult.Cancel  **End** **Sub**  **Private** **Sub** solveBtn\_Click(sender **As** Object, e **As** EventArgs)  *' Makes sure a maze has been generated*  **If** mazeGenerated = **False** **Then**  MsgBox("No maze generated!" & vbCrLf & "Please press the generate button", MsgBoxStyle.OkOnly, "No maze generated")  **Exit** **Sub**  **End** **If**  *' Sets solving algorithim to what the user has selected*  solveAlgorithm = solveCombo.Text    *' Reset all cells that have .mazeSolved = True*  **For** **Each** cell **In** maze  cell.mazeSolved = **False**  **Next**    *' Checks if the maze can be displayed*  **If** Math.Floor(Math.Min(1220 / Int(widthTxtBox.Text), 690 / Int(heightTxtBox.Text))) < 3 **Then**  *' If it cant be displayed ask if they want to download*  downlaodSolved = MsgBox("Maze is too big to display!" & vbCrLf & "Want to download?" & vbCrLf & "WARNING! DEPENDING ON HARDWARE THIS MAY TAKE A LONG TIME", MsgBoxStyle.OkCancel, "Maze too big!")  *' If they don't want to download exit sub*  **If** downlaodSolved = DialogResult.Cancel **Then**  **Exit** **Sub**  **End** **If**  **End** **If**  *' Resets old timer, Starts new timer, Upates Status*  solveTimer.Reset()  solveTimer.Start()  statusLbl.Text = "Status: Solving"  statusLbl.Update()  *' Checks what solving algorithm user has chosen*  **If** solveAlgorithm = "Dijkstra's" **Then**  dijkstra()  **ElseIf** solveAlgorithm = "Breath Frist Search" **Then**  BFS()  **ElseIf** solveAlgorithm = "A\*" **Then**  Astar()  **ElseIf** solveAlgorithm = "Wall Follower LHR" **Then**  wallFollower("LHR")  **ElseIf** solveAlgorithm = "Wall Follower RHR" **Then**  wallFollower("RHR")  **End** **If**  solveTimer.**Stop**()  *' Upadtes Maze box*  drawMaze()    *' They want to download the solved image*  **If** downlaodSolved = DialogResult.OK **Then**  downloadMaze()  **Else**  *' They dont want to download the solved image*  *' Check if it's drawable*  **If** Math.Floor(Math.Min(1220 / Int(widthTxtBox.Text), 690 / Int(heightTxtBox.Text))) >= 3 **Then**  mazeBox.Image = mazeImage  **End** **If**  **End** **If**    *' Displays Statistics*  solveTimeLbl.Text = "Sove Time: " & Str(solveTimer.ElapsedMilliseconds() / 1000) & "s"  totalTimeLbl.Text = "Total Time " & Str((generationTimer.ElapsedMilliseconds() + solveTimer.ElapsedMilliseconds() + drawTimer.ElapsedMilliseconds() + deadEndTimer.ElapsedMilliseconds()) / 1000) & "s"  *' Resets Status*  statusLbl.Text = "Status: Doing Nothing"  **End** **Sub**  **Private** **Sub** deadEndRemoverBtn\_Click(sender **As** Object, e **As** EventArgs)  *' Makes sure a maze has been generated*  **If** mazeGenerated = **False** **Then**  MsgBox("No maze generated!" & vbCrLf & "Please press the generate button", MsgBoxStyle.OkOnly, "No maze generated")  **Exit** **Sub**  **End** **If**    *' Saves the inputted percentage*  *' Checking that the values inputted for dead end remover is valid*  **If** Double.TryParse(deadEndRemoverTxtBox.Text, deadEndPercent) **And** deadEndPercent <= 1.0 **Then**  deadEndPercent = deadEndRemoverTxtBox.Text  **Else**  MsgBox("Make sure dead end remover is a decimal number or 1", MsgBoxStyle.OkOnly, "Invalid Input")  **Exit** **Sub**  **End** **If**    *' Resets old timer, Starts new timer, Upates Status*  deadEndTimer.Reset()  deadEndTimer.Start()  statusLbl.Text = "Status: Removing Dead Ends"  statusLbl.Update()    *' Removes dead ends*  deadEndRemover()  deadEndTimer.**Stop**()  *' Removes the solved value for each cell*  **For** **Each** cell **In** maze  cell.mazeSolved = **False**  **Next**    *' Upadtes Maze box*  drawMaze()  mazeBox.Image = mazeImage    *' Find the dead end count*  **For** **Each** cell **In** maze  cell.deadEndFinder()  **Next**  deadEndToShow = deadEndPos.Count()  deadEndPos.Clear()  *' Displays Statistics*  solveTimeLbl.Text = "Sove Time: "  drawTimeLbl.Text = "Draw Time: " & Str(drawTimer.ElapsedMilliseconds() / 1000) & "s"  deadEndCountLbl.Text = "Dead End Count: " & Str(deadEndToShow)  deadEndTimeLbl.Text = "Dead End Time: " & Str(deadEndTimer.ElapsedMilliseconds() / 1000) & "s"  totalTimeLbl.Text = "Total Time " & Str((generationTimer.ElapsedMilliseconds() + solveTimer.ElapsedMilliseconds() + drawTimer.ElapsedMilliseconds() + deadEndTimer.ElapsedMilliseconds()) / 1000) & "s"  *' Resets Status*  statusLbl.Text = "Status: Doing Nothing"    **End** **Sub**  *' Setting Colour Customisation*  **Private** **Function** selectColour() **As** Color *' Opens a colour picker and returns the selected colour*  colorDialog.ShowDialog() *' Opens colour picker*  **Return** colorDialog.Color *' Returns picked colour*  **End** **Function**  **Private** **Sub** downloadMaze()  **If** mazeGenerated = **True** **Then**  **Dim** openFile **As** **New** SaveFileDialog  openFile.FileName = **Nothing**  openFile.Filter = "JPG File's |\*.jpg"  openFile.ShowDialog()  **Try**  mazeImage.Save(openFile.FileName)  **Catch** ex **As** Exception  *' They didn't select a file location*  **End** **Try**  **Else**  MsgBox("No maze generated!" & vbCrLf & "Please press the generate button", MsgBoxStyle.OkOnly, "No maze generated")  **End** **If**  **End** **Sub**  **Private** **Sub** bgColourBtn\_Click(sender **As** Object, e **As** EventArgs)  bgColour = selectColour() *' Selects background colour*  bgColourBtn.Text = bgColour.ToString  **End** **Sub**  **Private** **Sub** mazeColourBtn\_Click(sender **As** Object, e **As** EventArgs)  mazeColour = selectColour() *' Selects maze colour*  mazeColourBtn.Text = mazeColour.ToString  **End** **Sub**  **Private** **Sub** solveColourBtn\_Click(sender **As** Object, e **As** EventArgs)  solveColour = selectColour() *' Selects solve colour*  solveColourBtn.Text = solveColour.ToString  **End** **Sub**  **Private** **Sub** downloadBtn\_Click(sender **As** Object, e **As** EventArgs)  downloadMaze()  **End** **Sub**  **Private** **Function** componentAnalysis(**ByVal** image **As** Bitmap) **As** List(**Of** List(**Of** Point))  largestComponent.Clear()  *' Create a list to store the components*  **Dim** components **As** **New** List(**Of** List(**Of** Point))()  *' Create a 2D array to track which pixels have been visited*  **Dim** visited(image.Width - 1, image.Height - 1) **As** Boolean  *' Loop through each pixel in the image*  **For** y **As** Integer = 0 **To** image.Height - 1  **For** x **As** Integer = 0 **To** image.Width - 1  *' If this pixel has not been visted and its white(255,255,255)*  **If** **Not** visited(x, y) **And** image.GetPixel(x, y) = Color.FromArgb(255, 255, 255) **Then**  *' Create a list to store the pixels in the component*  **Dim** component **As** **New** List(**Of** Point)()  *' Create a generationStack to store the pixels that need to be checked*  **Dim** generationStack **As** **New** Stack(**Of** Point)()  generationStack.Push(**New** Point(x, y))  *' Until all pixels have been checked*  **While** generationStack.Count > 0  *' Store the current pixel*  **Dim** pixel **As** Point = generationStack.Pop()  *' If this pixel has not been visted and is white*  **If** **Not** visited(pixel.X, pixel.Y) **And** image.GetPixel(pixel.X, pixel.Y) = Color.FromArgb(255, 255, 255) **Then**  visited(pixel.X, pixel.Y) = **True**  *' Add to the componet*  component.Add(pixel)  *' Add neighbours to the generationStack*  **If** pixel.X > 0 **Then**  generationStack.Push(**New** Point(pixel.X - 1, pixel.Y))  **End** **If**  **If** pixel.X < image.Width - 1 **Then**  generationStack.Push(**New** Point(pixel.X + 1, pixel.Y))  **End** **If**  **If** pixel.Y > 0 **Then**  generationStack.Push(**New** Point(pixel.X, pixel.Y - 1))  **End** **If**  **If** pixel.Y < image.Height - 1 **Then**  generationStack.Push(**New** Point(pixel.X, pixel.Y + 1))  **End** **If**  **End** **If**  **End** **While**  components.Add(component)  **End** **If**  **Next**  **Next**  *' Find the largest component*  **For** **Each** component **As** List(**Of** Point) **In** components  **If** component.Count > largestComponent.Count **Then**  largestComponent = component  **End** **If**  **Next**  **Return** components  **End** **Function**    **Private** **Sub** imageInputBtn\_Click(sender **As** Object, e **As** EventArgs)  *' Rest variable*  **If** imageInputted = **True** **Then**  inputImage.Dispose()  mazeWallList.Clear()  imgComponents.Clear()  mazeBox.Image = **Nothing**  widthTxtBox.Text = 0  heightTxtBox.Text = 0  imageInputted = **False**  imageInputBtn.Text = "Input Image"  **Exit** **Sub**  **End** **If**  *' Requests and store image in memory*  openFileDialog1.FileName = ""  openFileDialog1.Filter = "JPG Files(\*.jpg)|\*.jpg"  **If** openFileDialog1.ShowDialog = Windows.Forms.DialogResult.OK **Then**  inputImage = **New** Bitmap(Image.FromFile(openFileDialog1.FileName))  *' Calculate the new image dimensions*  **Dim** newWidth **As** Integer = inputImage.Width + 2  **Dim** newHeight **As** Integer = inputImage.Height + 2  *' Create the new bitmap with the larger dimensions*  **Dim** newImage **As** **New** Bitmap(newWidth, newHeight)  *' Draw the original image onto the new bitmap*  **Using** g **As** Graphics = Graphics.FromImage(newImage)  g.DrawImage(inputImage, **New** Point(1, 1))  **End** **Using**  *' Draw a border around the image*  **Using** g **As** Graphics = Graphics.FromImage(newImage)  **Using** p **As** **New** Pen(Color.Black)  g.DrawRectangle(p, **New** Rectangle(0, 0, newWidth - 1, newHeight - 1))  **End** **Using**  **End** **Using**  inputImage = newImage  *' Sets width and height text boxs*  widthTxtBox.Text = inputImage.Width  heightTxtBox.Text = inputImage.Height  **Else**  **Exit** **Sub**  **End** **If**    **Dim** currentPixel **As** Color  *' Turns to grayscale*  **For** x **As** Integer = 0 **To** inputImage.Width - 1  **For** y **As** Integer = 0 **To** inputImage.Height - 1  currentPixel = inputImage.GetPixel(x, y)  *' Finds lumiosity*  luminosity = (currentPixel.R \* R) ^ GAMMA + (currentPixel.B \* B) ^ GAMMA + (currentPixel.G \* G) ^ GAMMA  *' Altrting the gradient thershold*  **If** luminosity <= 125 **Then**  inputImage.SetPixel(x, y, Color.FromArgb(0, 0, 0))  mazeWallList.Add(**New** Point(x, y))  **Else**  inputImage.SetPixel(x, y, Color.FromArgb(255, 255, 255))  **End** **If**  **Next**  **Next**  imgComponents = componentAnalysis(inputImage)    imageInputBtn.Text = "Cancel Image"  imageInputted = **True**  **End** **Sub**    **Private** **Sub** helperBtn\_Click(sender **As** Object, e **As** EventArgs)  **Me**.Hide()  Form2.Show()  **End** **Sub**    **Private** **Sub** cancelAnimationBtn\_Click(sender **As** Object, e **As** EventArgs)  cancelAnimation = **True**  **End** **Sub**  *' USER INPUT END*  End Class |

### Helper Form

# Appendix 2

## Youtube Testing Links

### Playlist

<https://youtube.com/playlist?list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN>

### Feature showing and bug fixing

#### Inputting an image, turning image into grayscale, altering grayscale threshold

<https://www.youtube.com/watch?v=qi2imsWfvgo&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=1>

#### How the first version of component analysis didn't work

<https://www.youtube.com/watch?v=zo1-HKuc3lk&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=2>

#### How the second version of component analysis has improved

<https://www.youtube.com/watch?v=g-jM6Ki-7ro&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=3>

#### How the second version of the solving gradient has improved

<https://www.youtube.com/watch?v=3Sb1jeipbs4&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=4>

### 1.0

#### Video 1.0

<https://www.youtube.com/watch?v=ZvqqkZPnqUI&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=6>

#### Video 1.1

<https://www.youtube.com/watch?v=pYjzGHMlafA&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=7>

#### Video 1.2

<https://www.youtube.com/watch?v=828mFCwgLsI&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=8>

### 2.0

#### Video 2.0

<https://www.youtube.com/watch?v=O3_PD3r7KGA&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=9>

#### Video 2.1

<https://www.youtube.com/watch?v=o3A8X7dz5BM&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=10>

### 3.0

#### Video 3.0

<https://www.youtube.com/watch?v=JNvdg2ShCoM&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=11>

#### Video 3.1

<https://www.youtube.com/watch?v=6HFVwbD9yb0&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=12>

### 4.0

#### Video 4.0

<https://www.youtube.com/watch?v=B46hi7bkQnc&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=13>

### 5.0

#### Video 5.0

<https://www.youtube.com/watch?v=YWHPaFHpV9k&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=14>

#### 6.0

#### Video 6.0

<https://www.youtube.com/watch?v=DnbAlUuPl4w&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=15>

#### Video 6.1

<https://www.youtube.com/watch?v=a9BMOcBCEwU&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=16>

#### 7.0

#### Video 7.0

<https://www.youtube.com/watch?v=DuLsULzC-ds&list=PL3YAEsPABRrxRmNFIRFjes7XDmWsjn9kN&index=17>

#### Video 7.1

#### Video 7.2

#### Video 7.3

#### 8.0

#### Video 8.0

Appendix 3