Analysis Section

Design Section

Technical Solution

Testing

Evaluation

Background:

My project is going to be an investigation into maze generation, allowing me to explore a range of maze generation algorithms. The purpose of my investigation will be determining the “efficiency” of these algorithms, and come to a conclusion about which algorithm produces the “better” mazes. My end-user is a group who construct physical garden mazes. They are looking for an application which will streamline the design of mazes prior to construction, which was previously done manually. I plan on writing a program to generate/store/solve two-dimensional mazes (with multiple generation algorithms to choose from). From this, I will analyse the mazes generated by each algorithm, and attempt to quantify their difficulty/complexity. There are already many maze generators available online which utilise a plethora of algorithms, so the challenge will come from implementing my chosen algorithms in an efficient manner for my chosen programming language.

End User Discussion:

In terms of features for my maze generator, my end-user wants a range of algorithms to be easily selectable, to allow them to quickly choose the most appropriate for their needs. In addition, they would like to be able to see the solution superimposed on the maze, which will allow them to assess the suitability of each maze. They did not directly request any specific algorithms - but were clear that they wanted mazes with a single solution - so I have chosen 4 (see research) from this category which I believe have the most flexibility, and will provide a good range of maze designs to choose from. When discussing the interface, they expressed a preference for a very basic system, with only the main parameters accessible, which would be easy for their employees to utilise with little extra training required.

Other Maze Generators:

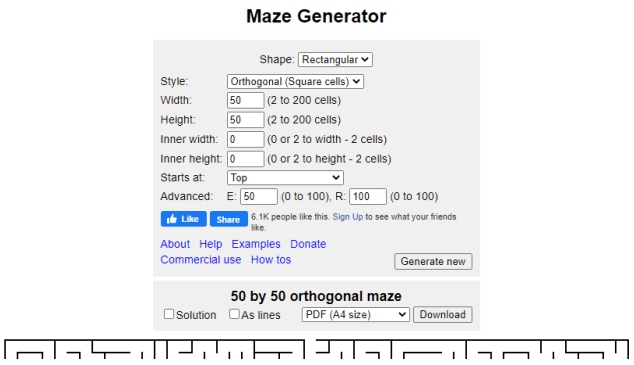


Figure 1: http://www.mazegenerator.net/

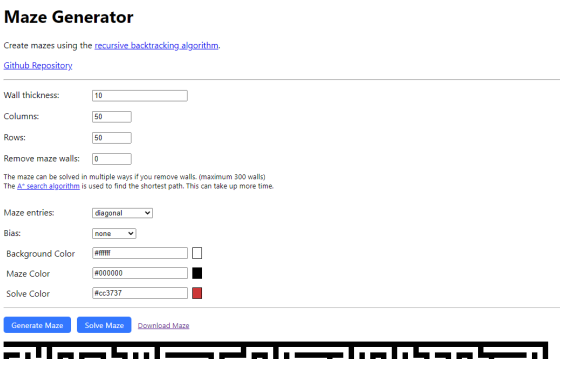


Figure 2: https://keesiemeijer.github.io/maze-generator/#generate

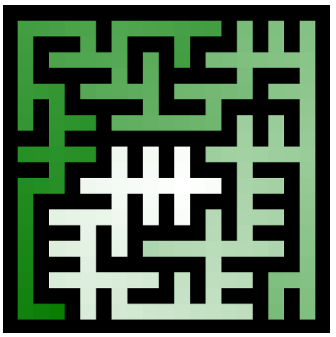
Here are two maze generators I found online (Fig 1,2). They have options for the dimensions of the maze, as well as the size of the cells in the outputted image. The second example also has a choice of colours for the cells. Where the second only has one algorithm, the first has a dropdown menu to allow selection of the desired generation algorithm. In addition, the first has a checkbox option to hide the solution after you click solve. For my application’s interface, I want to incorporate both a dropdown for the algorithms and an option for the user to show/hide the solution. The controls available to the user must be clearly laid out, and not overwhelmingly complicated. Although these current systems which I have found are functional, I believe that they could benefit from a more accessible, simplified interface.

Research:

My maze generator will produce exclusively perfect mazes, meaning they have a single solution (as requested by my end-user). This will only require one solving algorithm to be implemented, since they will all produce the same solution. Additionally, by only generating perfect mazes, it effectively "standardizes" the mazes, and allows me to better compare their complexity, by eliminating the subjective difficulty of one solution compared to many. A range of algorithms already exist for generating mazes, so I shall be implementing them, rather than designing them. The four generation algorithms I have chosen form intentionally distinct mazes, due to the different techniques used within creation. I have provided a summary of the properties of each algorithm (including the solving algorithm) below.
(Sources: info - https://www.astrolog.org/labyrnth/algrithm.htm,
diagrams - https://hurna.io/academy/algorithms/maze_generator/index.html)

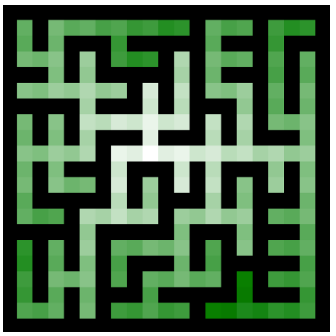
Algorithms:

Recursive Backtracker:



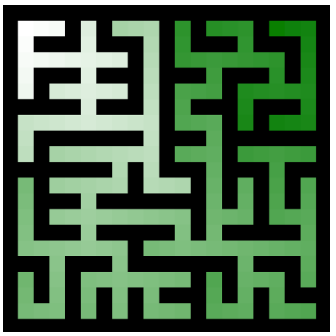
- Creates mazes with long, winding paths, and very few junctions
- This type of maze tends to have the longest solutions
- At each cell, a random neighbour is chosen, and the path is traversed (i.e. wall removed)
- A path is carved through the maze until the current cell has no neighbours, at which point the path is traversed backwards until an unvisited neighbour is found
- A stack is used to store cells in the order they are visited, allowing for the "backtracking"

Prim's (Random):



- Creates a maze with lots of junctions
- This type of maze tends to have the shortest, most direct solutions
- Any neighbour of any previously visited cell can be chosen, and a wall is then carved out between the chosen and closest visited cell
- This grows the maze out from the starting point

Kruskal's (Random):



- Solutions are similar to prims, but often a bit longer, as there are less junctions
- Paths are more windy, though not as severe as recursive backtracking
- Cells are initialised with a unique set id (sets of size 1)
- All walls are iterated over, and if the two cells the wall would connect belong to different sets, they are connected, and a union is performed on the two sets
- This ensures there are no loops in the maze

Recursive Division:

- Creates mazes with long, straight paths
- Solutions are long, but not as long as recursive backtracking

- The maze begins as an empty chamber, which is then split into four smaller chambers using two walls
- Three holes are made, such that all chambers are accessible, while no loops are made
- This process is repeated recursively on each chamber (using a stack again) until all chambers only contain a single cell

Dijkstra's (Pathfinding):

- First, the entrance cell is made active
- At each step, the current distance is added into the active cells, and then the neighbouring cells are made active
- When the exit cell contains a number, the smallest neighbouring number is followed back to the entrance, tracing a path
- This path is guaranteed to be the shortest path

<u>Objectives:</u>

1. Generate a maze using each algorithm
    a. 4 algorithms to choose from
        i. Selectable (by user) from drop-down list
    b. Selectable dimensions (by user):
        i. Up to ~200x200
    c. Time to generate:
        i. Aim < 1s for <50x50
        ii. Maximum 1 min (for 200x200)
2. Store generated maze in 2d array
    a. Should allow for simple display of data in console
3. Solve maze
    a. Using Dijkstra's pathfinding algorithm
    b. Store path in maze array
4. Display maze in console
    a. Mainly for smaller mazes, before bitmap implementation
5. Convert maze (and solution) to/from bitmap
    a. Allows mazes to be saved to a location specified by the user
    b. Allows previously mazes to be loaded into the program
    c. Save With and Without path
6. Add an interface to streamline maze generation
    a. Generated maze displayed in window
    b. Solution should be toggleable
    c. Interface should be able to handle multiple button presses
        i. Invalid button presses should not crash program

Proposed Solution:

I plan to use object-oriented programming to organise my code, as containing the maze array (and its respective algorithms) within a class will make it a lot easier to use. As I have worked with OOP very little in other languages (such as Java and C#), I think it would be wisest for me to use Python to code my solution. Another advantage of using Python is the range of modules available to fit my purposes (detailed on p8).

Modelling:

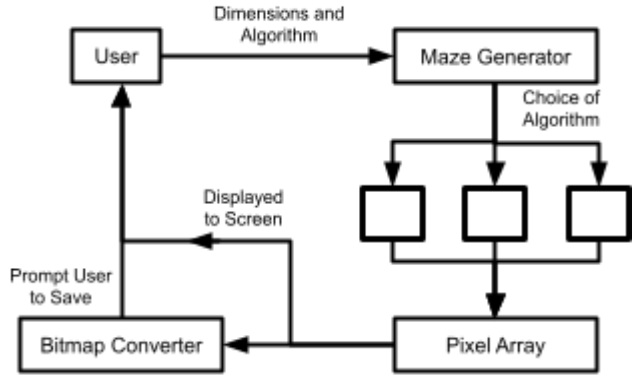
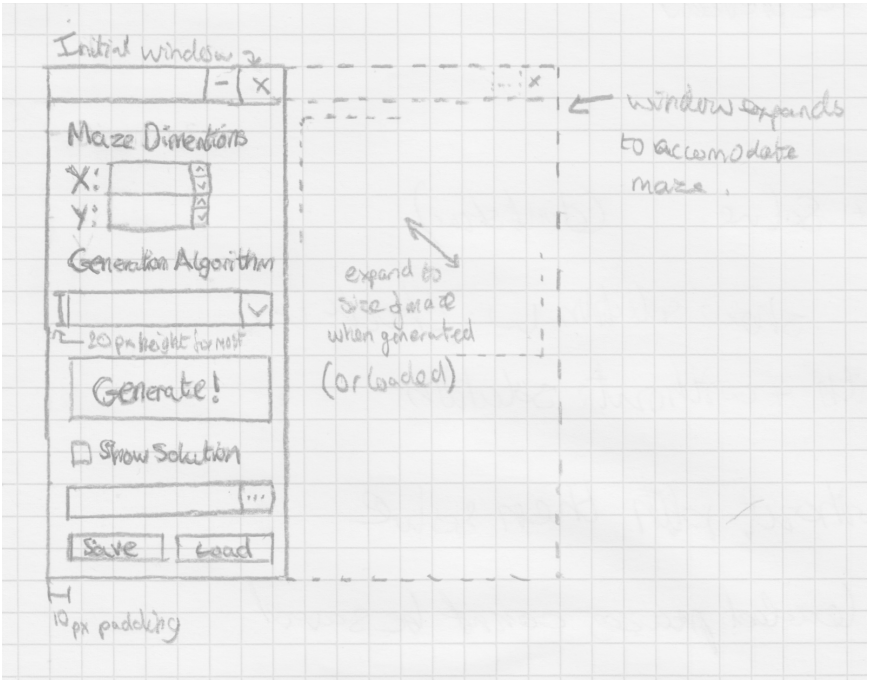


Figure 3: DFD for the maze generation

Fig. 3 shows the current system - the user inputs the maze dimensions and chosen algorithm, and the generator produces a pixel array of the maze, which can be transferred into a bitmap if requested. My system will emulate this structure, as it will make it easy to validate the user inputs (range of numbers for dimensions, and algorithms from a list), and integrate the multiple generation algorithms.

Documented Design:



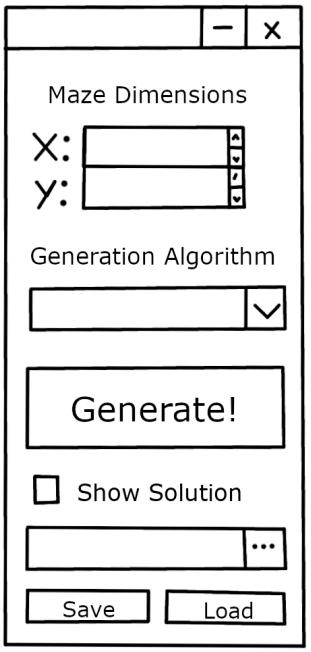
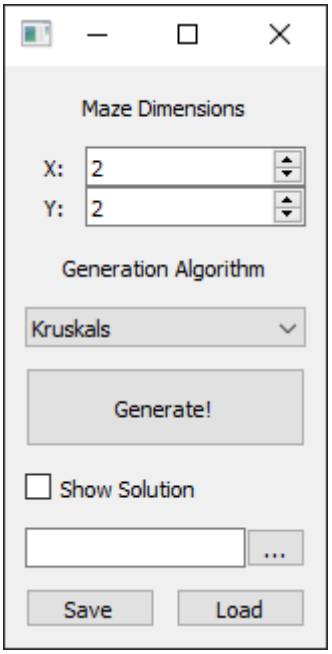


Figure 4: First Sketch

Fig. 4 is my first sketch for my UI - I have also converted the initial window to a digital design, and converted that into a prototype UI, see Fig. 5. In this design, the options for the maze generation are contained in a list format. The dimensions are determined by a number entry box and the algorithm by a drop-down selection. There is a checkbox to allow the solution to be shown/hidden. Finally, there is a place for the user to input a file path for the images to be saved in/loaded from. I have based my design mainly off the example in Fig.1 from my market research, as I think it is a lot more interactive and accessible to have a range of selectable options, which should allow the user to begin generating a range of mazes without requiring knowledge of the underlying systems/algorithms. After the user clicks generate, the window should expand to accommodate the size of the maze being displayed.



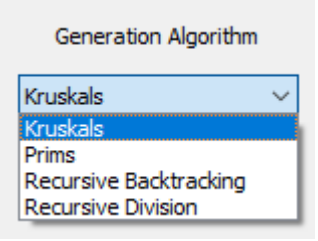
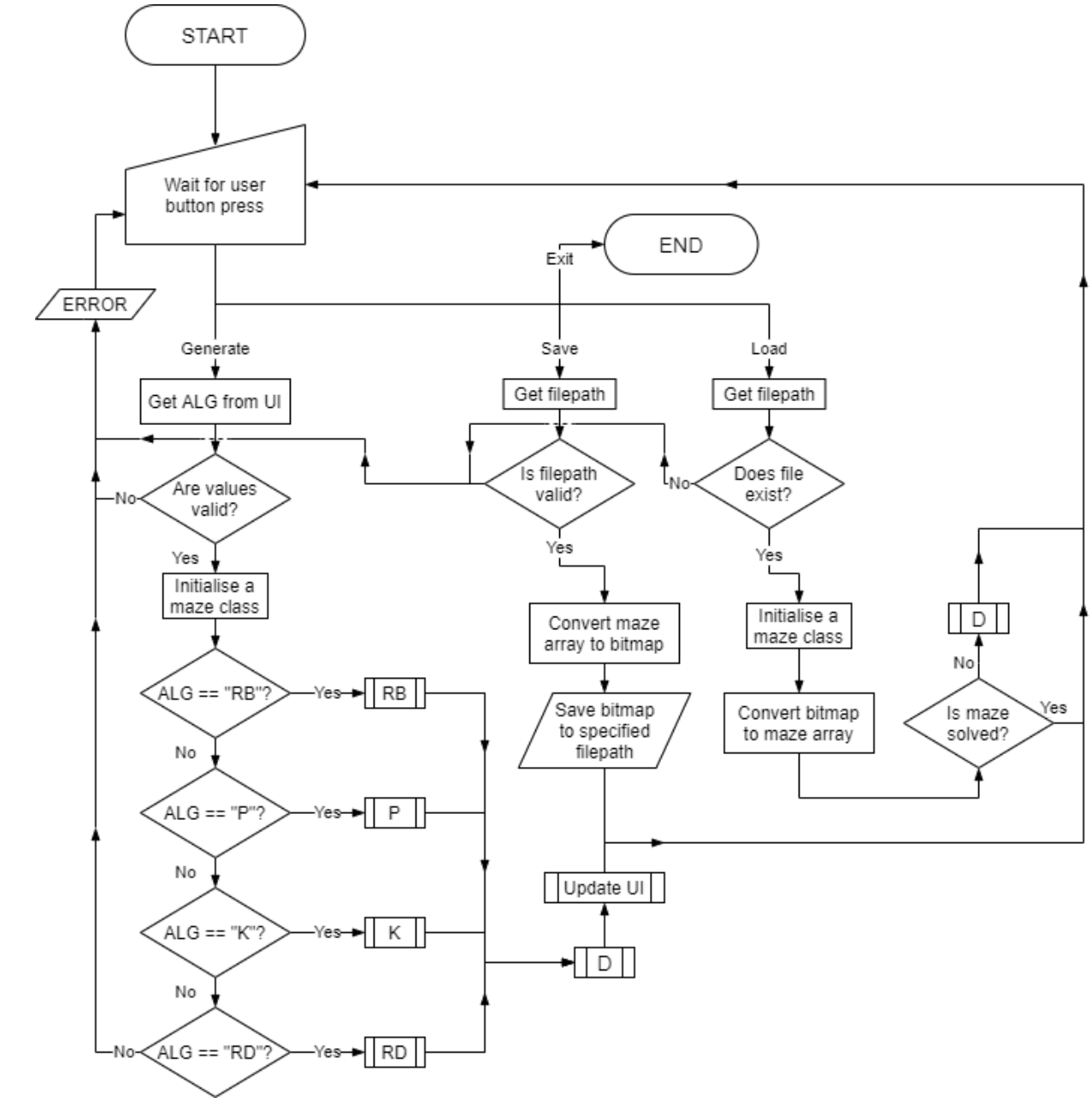


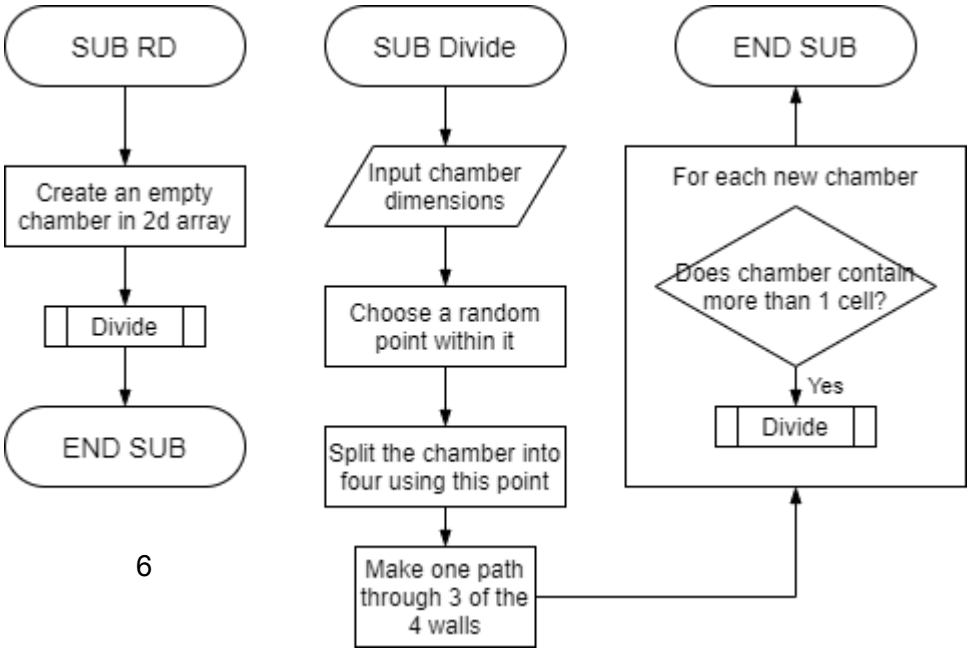
Figure 5: UI Prototype

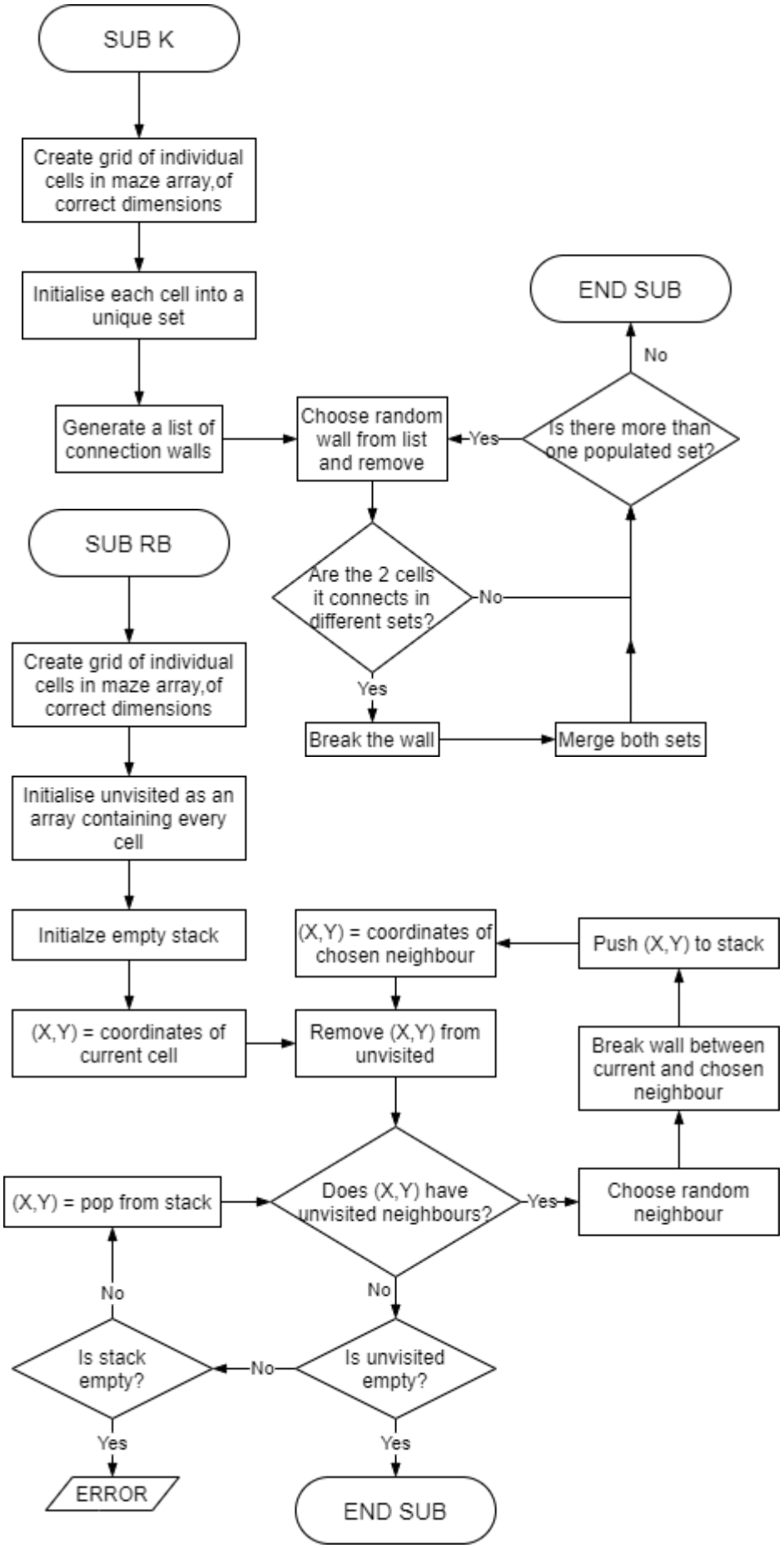
Figure 6: Console Display

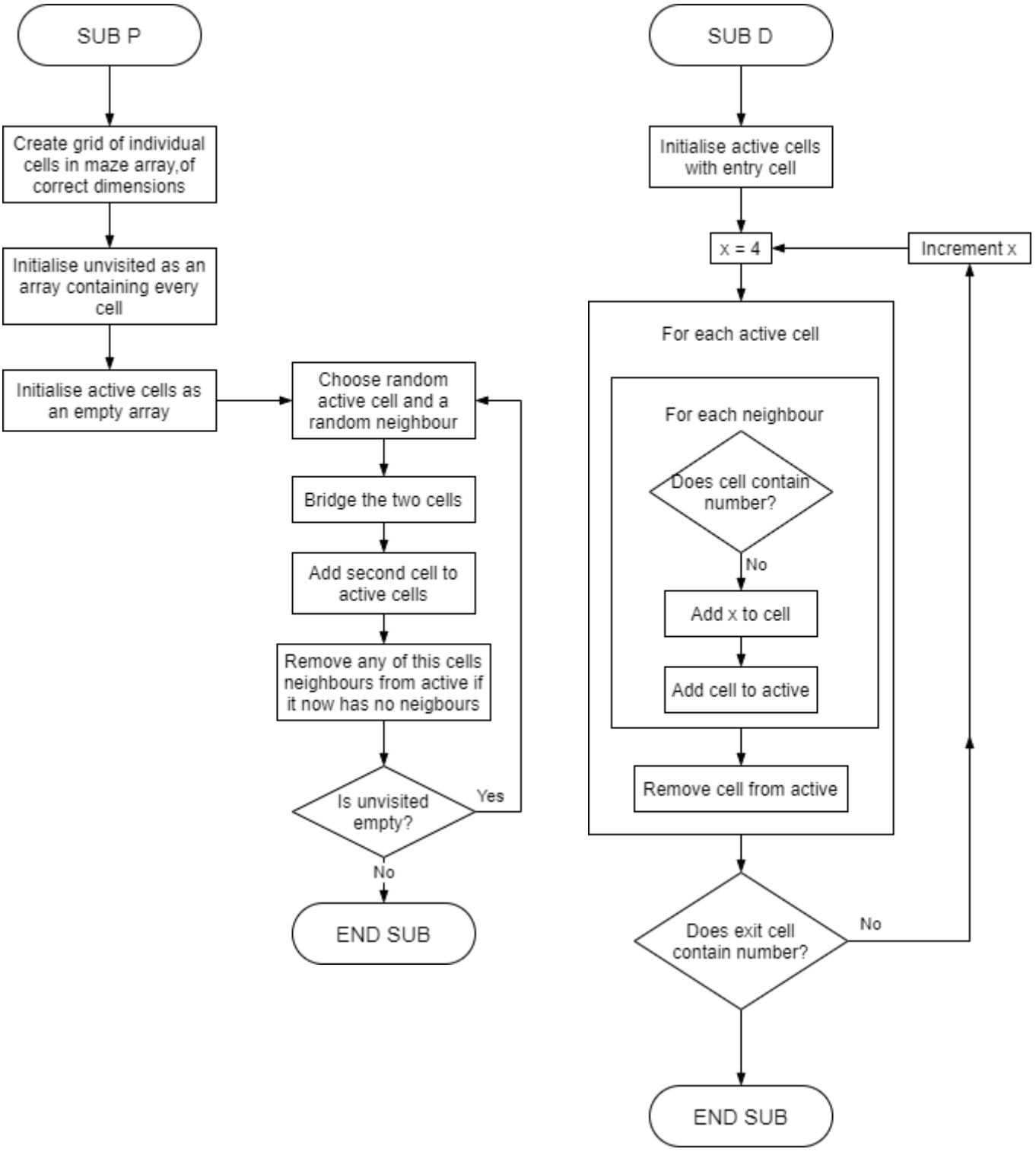
Fig. 6 shows my original display, which uses the command line. This console implementation uses some basic ascii characters to imitate the pixels of the maze. While this does work, it lacks practicality, and is not suitable for displaying the larger mazes that my program is designed to generate.

Flowchart:


START
Wait for user button press
END
Exit
ERROR
Generate
Save
Load
Get ALG from UI
Get filepath
Get filepath
Are values valid?
No
Yes
Is filepath valid?
No
Yes
Does file exist?
Yes
Initialise a maze class
Convert maze array to bitmap
Initialise a maze class
D
No
Is maze solved?
Yes
Save bitmap to specified filepath
Convert bitmap to maze array
ALG == "RB"?
Yes
RB
No
ALG == "P"?
Yes
P
No
Update UI
ALG == "K"?
Yes
K
D
No
ALG == "RD"?
No
Yes
RD


SUB RD
Create an empty chamber in 2d array
Divide
END SUB
SUB Divide
Input chamber dimensions
Choose a random point within it
Split the chamber into four using this point
Make one path through 3 of the 4 walls
END SUB
For each new chamber
Does chamber contain more than 1 cell?
Yes
Divide


SUB K
Create grid of individual cells in maze array,of correct dimensions
Initialise each cell into a unique set
Generate a list of connection walls
Choose random wall from list and remove
Yes
Is there more than one populated set?
No
END SUB
Are the 2 cells it connects in different sets?
No
Yes
Break the wall
Merge both sets
SUB RB
Create grid of individual cells in maze array,of correct dimensions
Initialise unvisited as an array containing every cell
Initialze empty stack
(X,Y) = coordinates of current cell
Remove (X,Y) from unvisited
(X,Y) = coordinates of chosen neighbour
Push (X,Y) to stack
Break wall between current and chosen neighbour
Does (X,Y) have unvisited neighbours?
Yes
Choose random neighbour
No
(X,Y) = pop from stack
No
Is stack empty?
No
Is unvisited empty?
Yes
ERROR
Yes
END SUB



This flowchart simply illustrates the workings of each of the algorithms used within maze generation/solving, as well as outlining the main system (i.e. the UI, and how it interacts with, and calls methods from, the maze class). Pseudocode for the algorithms can also be found on p11. The names of the algorithms have been shortened to K, P, RB, RD and D.

Modelling:

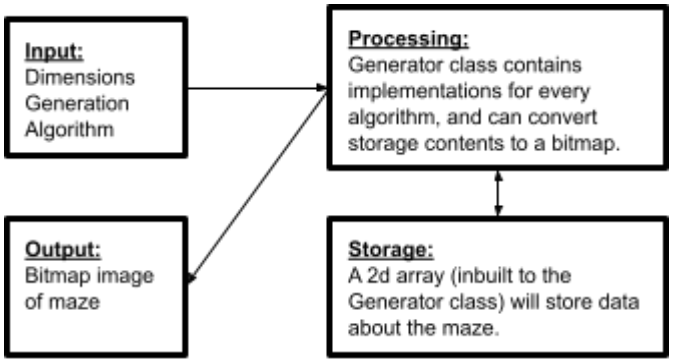


Figure 7: ISPO for the system

Fig. 7 is an overview of how the whole system will function. It shows that the generator class ("processing") will handle both the inputs and outputs, which allows it to have full control over the 2D maze array ("storage"). As a result, this array can be stored as a private variable in the generator class itself, rather than as a global variable.

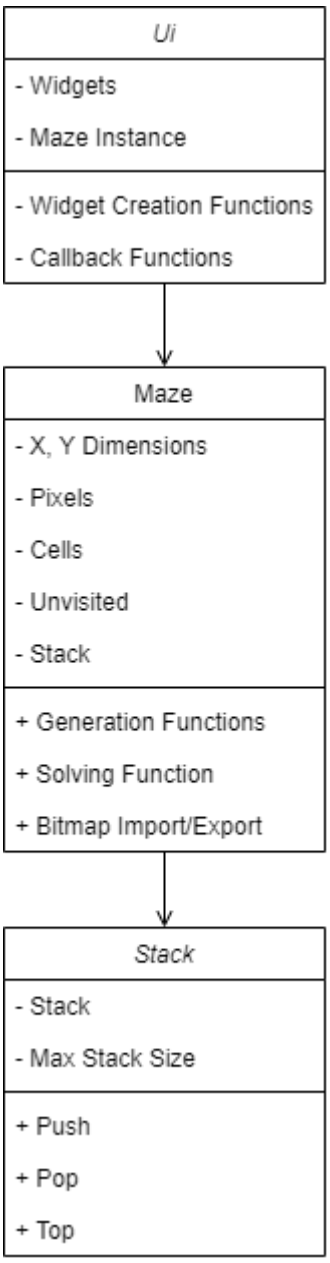


Figure 8: Class Diagram

Fig. 8 shows an overview of the classes I will be using to model my system. There is some dependency between classes; the Ui class "Callback functions" implement the Maze class (through the stored "maze instance"), and certain "Generation Functions" within the Maze class use a stack (stored within the "stack" variable). Due to the structure of my system, my classes will not require the implementation of any inheritance. Nonetheless, I believe that an object-oriented approach will be most appropriate for this project. By organizing all the relevant details about a maze in a container, along with the set of algorithms for generation/solving, you can then create a "maze" with all its accompanying data, rather than working on a single global array.
A pseudocode format of this class structure can be found on p10.

Libraries and Imports:

In each of my algorithms, the random element of the generation will be handled by python's "random" module. For my UI, I have chosen to use "PyQt5", as I think it should be easier to handle the recess than with Tkinter. As I need an image handling module for my bitmap conversions, I have chosen "PIL", since it has a version of its Image module which is compatible with Qt. I will also require the "os" module for dealing with file locations. Kruskal's algorithm requires the union of large sets, and to do this, I will use a module from Python Algorithms *(Documentation: https://python-algorithms.readthedocs.io/en/stable/readme.html)*, a library of algorithms written in python, called "unionfind". I also want to be able to time my algorithms (mainly for testing, though it might be useful in the final program), so I will be using "timeit".
As a few of these modules are external, my program may benefit from being compiled into an executable. To do this, I could use the module "pyinstaller", which allows you to create an

independent executable. This will be especially useful if I decide to keep each class in a separate file.

System Pseudocode:

```
Class Stack:
    Private:
        Array stack
    Public:
        Procedure push(x) // adds input "x" to top of stack
        Function Pop() // returns top element and removes from stack
        Function Top() // returns top element

Class Maze:
    Private:
        Int x_dimensions
        Int y_dimensions
        // dimensions of maze
        Array pixels // picture of maze in 2d array
        Array cells // coordinates of all cells
        Array unvisited // coordinates of unvisited cells
    Public:
        Procedure setupMaze() // sets up grid to be compatible with algorithms
        Procedure displayMaze() // uses pixel array to print maze to console
        Procedure resetUnvisited() // initializes "unvisited" array with all cells
        Function getNeighbours(x, y) // returns coordinates of neighbouring cells
        Procedure recursiveBacktracking()
        Procedure primRandom()
        Procedure kruskalRandom()
        Procedure recursiveDivision()
        // generation algorithms
        Procedure dijkstraPathfinding()
        // solving algorithm
        Function generateBitmap() // generates from "pixels" array
        Procedure importBitmap(filepath) // writes data into "pixels" array

Class UI:
    Private:
        Array widgets
        Procedure createLabel(text, xpos, ypos)
        Procedure createButton(text, callback, xpos, ypos)
        Procedure createSpinbox(min, max, xpos, ypos)
        Procedure createListbox(options, xpos, ypos)
        Procedure createFilepathInput(xpos, ypos)
        // creates the respective widget and stores it in the array
        // all private as would only be called in the constructor
        Procedure updateImage(img) // displays image and resizes window
```

Algorithm Pseudocode:

```
Procedure recursiveBacktracking()
    Remove current cell from unvisited
    While there are unvisited cells:
        If there are unvisited neighbouring cells:
            Choose random neighbour to current
            Remove chosen from unvisited
            Remove wall between chosen and current
            Push current cell to stack
            Set current to chosen
        Else:
            Pop previous cell from stack

Procedure primRandom()
    Remove current cell from unvisited and add to active
    While there are unvisited cells:
        Choose random neighbour to any active cell
        Remove wall between chosen and current
        Remove chosen cell from unvisited
        If chosen still has neighbours:
            Add chosen cell to active
        For each active neighbour of the chosen cell:
            If it has no unvisited neigbours:
                Remove from active

Procedure kruskalRandom()
    Assign each cell to a unique set
    Create array of all walls in random order
    For each wall:
        Get coordinates of connecting cells
        If cells belong to different sets:
            Remove wall between them
            Perform union on sets

Procedure recursiveDivision()
    Setup maze as empty chamber
    Push chamber dimensions to stack
    While stack is not empty:
        Pop chamber dimensions from stack
        Choose random point within chamber
        Divide chamber into 4 around point
        Push 4 chamber dimensions to stack
        For 3 of the dividing walls:
            Remove random wall within line
```

System Overview:

The system is programmed in Python, using Qt for the GUI. It allows the user to generate mazes using a set of 4 algorithms, and customise the dimensions. The user can also both save and load mazes generated through the program. Fig. 9 is a visual representation of the system, with its 3 main processes.

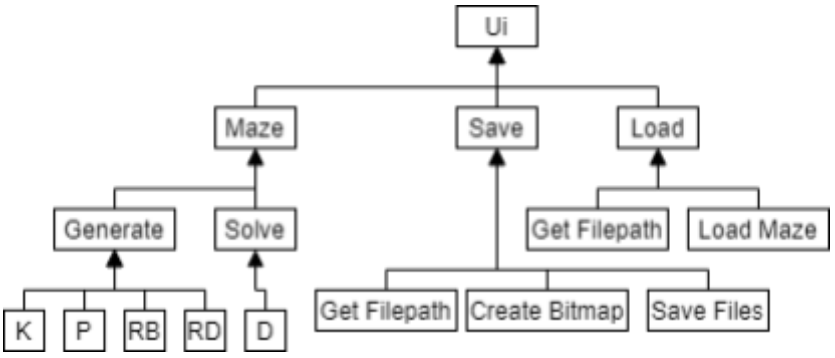


Figure 9: Top down diagram of the system

Generation Algorithms:

In my implementation of Kruskal's algorithm, the coordinates of all the walls are generated first, then one is selected at random each iteration and removed by altering the value in *self._pixels[y][x]*. The while loop is driven by the union-find structure - *self._unionFind._count* is the number of unique sets, so the loop breaks when this reaches 1. Prim's algorithm also runs using a while loop, until there are no unvisited cells remaining. It uses the function *self.getNeighbours* to get the unvisited neighbours of the selected cell. The final for loop clears up the *activeCells* array to prevent a cell with no unvisited neighbours from being selected. Both recursive algorithms use a stack - I created my own as the larger mazes would often exceed the limit of Python's built-in stack. The Recursive Backtracking algorithm uses the stack to store the current working position after each iteration *stack.push((x,y))*, allowing for the "backtracking". Recursive Division uses the stack to store the dimensions of each chamber - *pivot_* is the point where the chamber is split.

Object-Oriented Programming:

As described in the design section (Fig. 8 is repeated here), my program consists of 3 classes, with a small level of dependence - Maze uses Stack, and Ui implements Maze. The Maze class contains the generation/solving procedures, and the Ui class contains the callback functions responsible for calling algorithms from the Maze class. I decided to keep all the classes, along with the main function, within one file - it reduced the requirement for repeated imports. The use of classes to create a modular system means any extra functionality could be easily introduced.

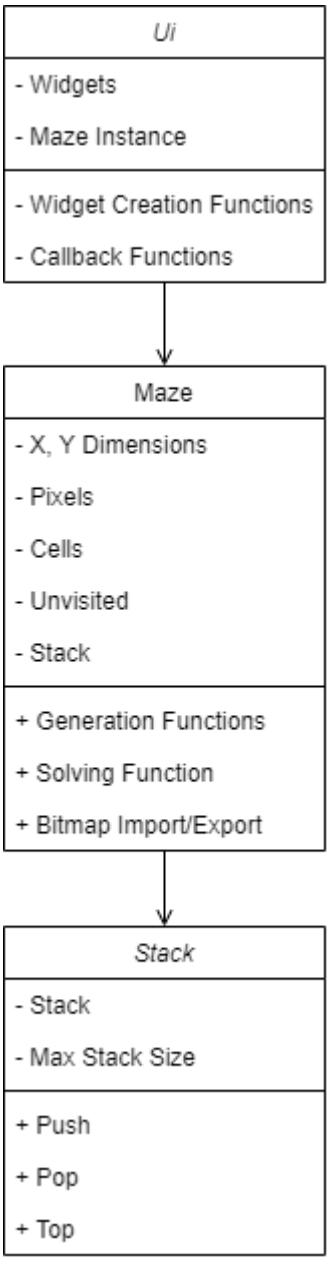


Figure 8: Class Diagram

User Interface:

Fig. 10 shows a screenshot of my GUI. The options for the generation are in a list format on the left of the window, leaving space on the right for the maze to be displayed. In this screenshot, the “show solution” box is toggled on, so the solution is being displayed. My GUI was designed to be simple to use, and still provide the user with a range of options for generation.

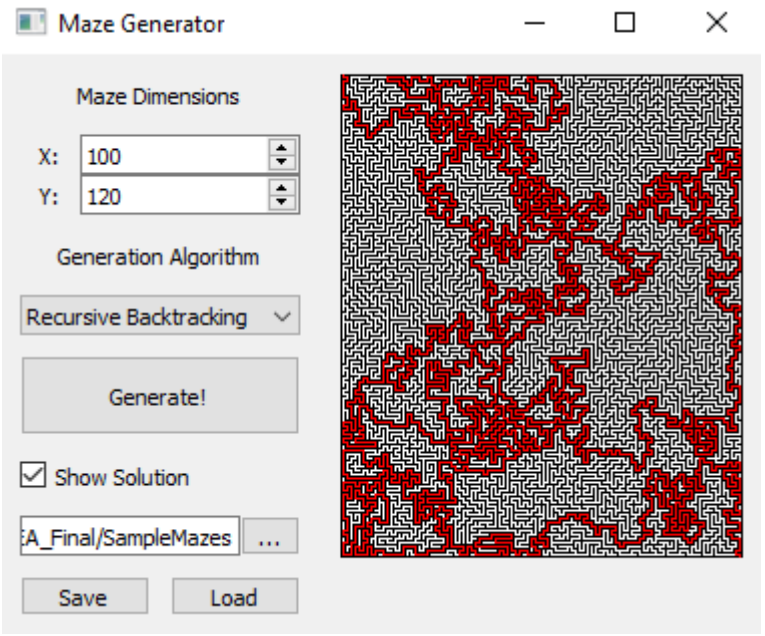


Figure 10: GUI screenshot

Annotated Code:

```
from random import *
import python_algorithms.basic.union_find as uf
from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
from PIL import Image, ImageQt
from timeit import timeit
import os
import sys

class Stack:
    def __init__(self, array=[]):
        self._maxStack = 0
        if self._maxStack != 0 and len(array) > self._maxStack:
            raise ValueError("Input too large")
        else:
            self._stack = array

    def push(self, element):
        if self._maxStack != 0:
            if len(self._stack) < self._maxStack:
                self._stack.append(element)
            else:
                raise ValueError("Stack Full")
        else:
            self._stack.append(element)

    def pop(self):
        if len(self._stack) > 0:
            return self._stack.pop()
        else:
            raise ValueError("Empty Stack")

    def top(self):
        if len(self._stack) > 0:
            x = self._stack[-1]
```

```
            return x
        else:
            raise ValueError("Empty Stack")

class Maze:
    def __init__(self, xDimensions, yDimensions):
        self._xDimensions = xDimensions
        self._yDimensions = yDimensions
        self._pixels = []
        self._cells = []
        self._unvisited = []
        self._unionFind = uf.UF(self._yDimensions * self._xDimensions)

        # fills in pixels array with 1s
        for y in range(2 * self._yDimensions + 1):
            temp = []
            for x in range(2 * self._xDimensions + 1):
                temp.append(1)
            self._pixels.append(temp)
        # fills in cells array with (x,y) tuples for all actionable coordinates
        for y in range(self._yDimensions):
            for x in range(self._xDimensions):
                self._cells.append((x, y))

    def setupMaze(self):
        # generates grid of separated cells
        for cell in self._cells:
            self._pixels[2 * cell[1] + 1][2 * cell[0] + 1] = 0
        # removes wall at start/end
        self._pixels[0][1] = 0
        self._pixels[2 * self._yDimensions][2 * self._xDimensions - 1] = 0

    def resetUnvisited(self):
        # resets the unvisited cells array
        for y in range(self._yDimensions):
            temp = []
            for x in range(self._xDimensions):
                temp.append(1)
            self._unvisited.append(temp)

    def displayMaze(self):
        # loops through every pixel in pixels array
        for y in range(2 * self._yDimensions + 1):
            temp = ""
            for x in range(2 * self._xDimensions + 1):
                if self._pixels[y][x] == 1:
                    # ■ character represents wall
                    temp = temp + "■ "
                elif self._pixels[y][x] == 2:
                    # | character represents path (if path has been found)
                    if (x, y) == (1, 0) or (x, y) == (2 * self._xDimensions - 1, 2 * self._yDimensions):
                        temp = temp + "| "
```

```
                elif self._pixels[y + 1][x] == 2 and self._pixels[y - 1][x] == 2:
                    temp = temp + " │ "
                elif self._pixels[y][x + 1] == 2 and self._pixels[y][x - 1] == 2:
                    temp = temp + "───"
                elif self._pixels[y][x + 1] == 2 and self._pixels[y + 1][x] == 2:
                    temp = temp + " ┌─"
                elif self._pixels[y][x - 1] == 2 and self._pixels[y + 1][x] == 2:
                    temp = temp + "┐ "
                elif self._pixels[y][x + 1] == 2 and self._pixels[y - 1][x] == 2:
                    temp = temp + " └─"
                elif self._pixels[y][x - 1] == 2 and self._pixels[y - 1][x] == 2:
                    temp = temp + "┘ "
                else:
                    temp = temp + "   "
            print(temp)
        print("\n\n\n")

    def getDimensions(self):
        return str(self._xDimensions)+"x"+str(self._yDimensions)

    def getNeighbours(self, x, y):
        # returns coordinates of neighbouring cells
        neighbours = {"all":[], "unvisited":[]}
        if x > 0:
            # left neighbour
            neighbours["all"].append((x - 1, y))
            if self._unvisited[y][x - 1]:
                neighbours["unvisited"].append((x - 1, y))
        if x < self._xDimensions - 1:
            # right neighbour
            neighbours["all"].append((x + 1, y))
            if self._unvisited[y][x + 1]:
                neighbours["unvisited"].append((x + 1, y))
        if y > 0:
            # top cell
            neighbours["all"].append((x, y - 1))
            if self._unvisited[y - 1][x]:
                neighbours["unvisited"].append((x, y - 1))
        if y < self._yDimensions - 1:
            # bottom cell
            neighbours["all"].append((x, y + 1))
            if self._unvisited[y + 1][x]:
                neighbours["unvisited"].append((x, y + 1))
        # returns array of neighbouring cells (if any)
        return neighbours

    def recursiveBacktracking(self):
        # initialises stack
        stack = Stack([])
        # starts at cell (0, 0)
        x, y = 0, 0
        # removes from unvisited
```

```
        self._unvisited[y][x] = 0
        # while there are still unvisited cells
        while any(1 in sub for sub in self._unvisited):
            # while there are still neighbours
            neighbours = self.getNeighbours(x, y)["unvisited"]
            while len(neighbours) > 0:
                # randomly choose cell to bridge to
                cell = choice(neighbours)
                # marks cell as visited
                self._unvisited[cell[1]][cell[0]] = 0
                # converts cell to pixel coordinates
                pixelX = 2 * x + 1
                pixelY = 2 * y + 1
                if x < cell[0]:
                    # bridge right
                    self._pixels[pixelY][pixelX + 1] = 0
                elif x > cell[0]:
                    # bridge left
                    self._pixels[pixelY][pixelX - 1] = 0
                elif y < cell[1]:
                    # bridge down
                    self._pixels[pixelY + 1][pixelX] = 0
                elif y > cell[1]:
                    # bridge up
                    self._pixels[pixelY - 1][pixelX] = 0
                # pushes current cell to stack
                stack.push((x,y))
                # sets working coordinates to next cell
                x,y = cell
                # recalculates neighbours
                neighbours = self.getNeighbours(x, y)["unvisited"]
            # when cell has no remaining neighbours, pop cell from stack
            cell = stack.pop()
            x,y = cell

    def primRandom(self):
        # array of cells from which new connections can be made
        activeCells = []
        # starts at cell (0, 0)
        x, y = 0, 0
        # removes from unvisited and adds to active
        self._unvisited[y][x] = 0
        activeCells.append((x,y))
        # while there are still unvisited cells
        while any(1 in sub for sub in self._unvisited):
            # chooses random active cell to work with
            x,y = choice(activeCells)
            # gets unvisited neighbours
            neighbours = self.getNeighbours(x, y)["unvisited"]
            # randomly choose cell to bridge to
            cell = choice(neighbours)
            # marks cell as visited
```

```
            self._unvisited[cell[1]][cell[0]] = 0
            # converts cell to pixel coordinates
            pixelX = 2 * x + 1
            pixelY = 2 * y + 1
            if x < cell[0]:
                # bridge right
                self._pixels[pixelY][pixelX + 1] = 0
            elif x > cell[0]:
                # bridge left
                self._pixels[pixelY][pixelX - 1] = 0
            elif y < cell[1]:
                # bridge down
                self._pixels[pixelY + 1][pixelX] = 0
            elif y > cell[1]:
                # bridge up
                self._pixels[pixelY - 1][pixelX] = 0
            # adds cell to active cells
            if len(self.getNeighbours(cell[0], cell[1])["unvisited"]) > 0:
                activeCells.append(cell)
            # removes cells from active with no unvisited neighbours
            for c in self.getNeighbours(cell[0], cell[1])["all"]:
                if c in activeCells and len(self.getNeighbours(c[0], c[1])["unvisited"]) == 0:
                    activeCells.remove(c)

    def kruskalRandom(self):
        # generates a list of all walls
        walls = [[2 * x, 2 * y + 1] for x in range(1,self._xDimensions) for y in range(self._yDimensions)] + [
            [2 * x + 1, 2 * y] for x in range(self._xDimensions) for y in range(1,self._yDimensions)]
        # iterate through each wall, while there are still multiple unique sets
        while self._unionFind._count > 1:
            # randomly choose a wall
            wall = choice(walls)
            # get neighbours
            neighbours = []
            if not self._pixels[wall[1]][wall[0] - 1]:
                # left cell
                neighbours.append((wall[0] - 1, wall[1]))
            if not self._pixels[wall[1]][wall[0] + 1]:
                # right cell
                neighbours.append((wall[0] + 1, wall[1]))
            if not self._pixels[wall[1] - 1][wall[0]]:
                # top cell
                neighbours.append((wall[0], wall[1] - 1))
            if not self._pixels[wall[1] + 1][wall[0]]:
                # bottom cell
                neighbours.append((wall[0], wall[1] + 1))
            cell1, cell2 = neighbours
            # checks if these cells belong to different sets, using union find
            cell1_set = self._unionFind.find(((cell1[1] - 1) // 2) * self._xDimensions + ((cell1[0] - 1) // 2))
            cell2_set = self._unionFind.find(((cell2[1] - 1) // 2) * self._xDimensions + ((cell2[0] - 1) // 2))
            if cell1_set != cell2_set:
                # remove wall
```

```
            self._pixels[wall[1]][wall[0]] = 0
            # perform union on these sets, again using union find
            self._unionFind.union(cell1_set,cell2_set)
        walls.remove(wall)

    def recursiveDivision(self):
        # sets up the initial chamber
        for y in range(1, 2 * self._yDimensions):
            for x in range(1, 2 * self._xDimensions):
                self._pixels[y][x] = 0
        self._pixels[0][1] = 0
        self._pixels[2 * self._yDimensions][2 * self._xDimensions - 1] = 0
        xRange = [0,self._xDimensions]
        yRange = [0,self._yDimensions]
        stack = Stack([[xRange, yRange]])
        while len(stack._stack) > 0:
            # pops chamber dimensions from stack
            xRange_, yRange_ = stack.pop()
            x_ = choice(range(xRange_[0],xRange_[1]-1))
            y_ = choice(range(yRange_[0],yRange_[1]-1))
            pivot_ = [2*x_+2,2*y_+2]
            genWalls = [[] for i in range(4)]
            # draws walls
            for x in range(2*xRange_[0]+1, 2*xRange_[1]+1):
                self._pixels[pivot_[1]][x] = 1
                if x % 2 == 1:
                    if x > pivot_[0]:
                        genWalls[0].append([x, pivot_[1]])
                    elif x < pivot_[0]:
                        genWalls[1].append([x, pivot_[1]])
            for y in range(2*yRange_[0]+1, 2*yRange_[1]+1):
                self._pixels[y][pivot_[0]] = 1
                if y % 2 == 1:
                    if y > pivot_[1]:
                        genWalls[2].append([pivot_[0], y])
                    elif y < pivot_[1]:
                        genWalls[3].append([pivot_[0], y])
            # creates pathways between new chambers
            r = randint(0, 3)
            w = [i for i in range(4) if i != r]
            for i in w:
                wall = choice(genWalls[i])
                self._pixels[wall[1]][wall[0]] = 0
            # pushes chambers to stack
            if x_ - xRange_[0] > 0 and y_ - yRange_[0] > 0:
                stack.push([[xRange_[0],x_+1],[yRange_[0],y_+1]])
            if xRange_[1] - 2 - x_ > 0 and y_ - yRange_[0] > 0:
                stack.push([[x_+1,xRange_[1]],[yRange_[0],y_+1]])
            if x_ - xRange_[0] > 0 and yRange_[1] - 2 - y_ > 0:
                stack.push([[xRange_[0],x_+1],[y_+1,yRange_[1]]])
            if xRange_[1] - 2 - x_ > 0 and yRange_[1] - 2 - y_ > 0:
                stack.push([[x_+1,xRange_[1]],[y_+1,yRange_[1]]])
```

```
    def dijkstraPathfinding(self):
        # defines start point and goal
        startPoint = [1,0]
        goal = [2 * self._xDimensions - 1, 2 * self._yDimensions]
        distance = 4
        # initialises active cells with start cell
        activeCells = [startPoint]
        self._pixels[startPoint[1]][startPoint[0]] = 3
        # while goal point contains no value
        while self._pixels[goal[1]][goal[0]] == 0:
            next_activeCells = []
            # iterate through each active cell
            for cell in activeCells:
                # gets all neighbours of this cell and adds to active
                neighbours = []
                if self._pixels[cell[1]][cell[0] - 1] == 0:
                    # left cell
                    neighbours.append((cell[0] - 1, cell[1]))
                    next_activeCells.append((cell[0] - 1, cell[1]))
                if self._pixels[cell[1]][cell[0] + 1] == 0:
                    # right cell
                    neighbours.append((cell[0] + 1, cell[1]))
                    next_activeCells.append((cell[0] + 1, cell[1]))
                if self._pixels[cell[1] - 1][cell[0]] == 0:
                    # top cell
                    neighbours.append((cell[0], cell[1] - 1))
                    next_activeCells.append((cell[0], cell[1] - 1))
                if self._pixels[cell[1] + 1][cell[0]] == 0:
                    # bottom cell
                    neighbours.append((cell[0], cell[1] + 1))
                    next_activeCells.append((cell[0], cell[1] + 1))
                # adds current distance to any neighbouring cells
                if len(neighbours) > 0:
                    for neighbour in neighbours:
                        self._pixels[neighbour[1]][neighbour[0]] = distance
            activeCells = next_activeCells
            distance += 1
        # backtracking
        x, y = goal
        # while current coordinate is not start
        while [x, y] != startPoint:
            nextCell = [x, y]
            # finds neighbouring cell with lowest value
            if x > 0:
                if self._pixels[y][x - 1] not in [1, 2]:
                    if self._pixels[y][x - 1] < self._pixels[nextCell[1]][nextCell[0]]:
                        # left cell
                        nextCell = [x - 1, y]
            if x < 2 * self._xDimensions:
                if self._pixels[y][x + 1] not in [1, 2]:
```

```
                if self._pixels[y][x + 1] < self._pixels[nextCell[1]][nextCell[0]]:
                    # right cell
                    nextCell = [x + 1, y]
            if y > 0:
                if self._pixels[y - 1][x] not in [1, 2]:
                    if self._pixels[y - 1][x] < self._pixels[nextCell[1]][nextCell[0]]:
                        # top cell
                        nextCell = [x, y - 1]
            if y < 2 * self._yDimensions:
                if self._pixels[y + 1][x] not in [1, 2]:
                    if self._pixels[y + 1][x] < self._pixels[nextCell[1]][nextCell[0]]:
                        # bottom cell
                        nextCell = [x, y + 1]
            # sets value of that cell to 2 (represents path)
            self._pixels[y][x] = 2
            # repeats with that cell
            x = nextCell[0]
            y = nextCell[1]
        # adds path value to start
        self._pixels[startPoint[1]][startPoint[0]] = 2
        # clears out temporary values (>2)
        for y in range(2 * self._yDimensions + 1):
            for x in range(2 * self._xDimensions + 1):
                if self._pixels[y][x] > 2:
                    self._pixels[y][x] = 0

    def genBitmap(self, withPath=True):
        # creates blank image of correct size
        img = Image.new("RGB", (2 * self._xDimensions + 1, 2 * self._yDimensions + 1))
        pix = img.load()
        # colours pixel based on value
        for i in range(img.size[0]):
            for j in range(img.size[1]):
                if self._pixels[j][i] == 0:
                    # white for blank
                    pix.__setitem__((i, j), (255, 255, 255))
                elif self._pixels[j][i] == 1:
                    # black for wall
                    pix.__setitem__((i, j), (0, 0, 0))
                elif self._pixels[j][i] == 2:
                    # red for path if selected
                    if withPath:
                        pix.__setitem__((i, j), (255, 0, 0))
                    else:
                        pix.__setitem__((i, j), (255, 255, 255))
        return img

    def importBitmap(self, img):
        # sets dimensions of the maze from the image
        self._pixels = []
        pix = img.load()
        # converts pixel colour into value for array
```

```
        for j in range(img.size[1]):
            temp = []
            for i in range(img.size[0]):
                if pix.__getitem__((i, j)) == (0, 0, 0):
                    temp.append(1)
                    # wall for black pixel
                elif pix.__getitem__((i, j)) == (255, 255, 255):
                    temp.append(0)
                    # gap for white pixel
                elif pix.__getitem__((i, j)) == (255, 0, 0):
                    temp.append(2)
                    # path for red pixel
            self._pixels.append(temp)

class Ui:
    def __init__(self, MainWindow):
        self._widgets = {}
        self._state = None
        self._mazeInstance = Maze(2, 2)
        self._MainWin = MainWindow
        # sets main window size
        self._MainWin.resize(160, 290)
        self._MainWin.setStyleSheet("background-color: light grey;")
        # populate window with widgets
        self._CreateLabel("label_mazedimensions", "Maze Dimensions", QRect(10, 10, 140, 20))
        self._CreateLabel("label_xdimensions", "X:", QRect(10, 40, 30, 20))
        self._CreateSpinBox("spinbox_xdimensions", 2, 200, QRect(40, 40, 110, 20))
        self._CreateLabel("label_ydimensions", "Y:", QRect(10, 60, 30, 20))
        self._CreateSpinBox("spinbox_ydimensions", 2, 200, QRect(40, 60, 110, 20))
        self._CreateLabel("label_generationalgorithm", "Generation Algorithm", QRect(10, 90, 140, 20))
        self._CreateComboBox("combobox_generationalgorithm",
            ["Kruskals", "Prims", "Recursive Backtracking", "Recursive Division"], QRect(10, 120, 140, 20))
        self._CreateButton("button_generate", "Generate!",
            QRect(10, 150, 140, 40), self._callback_GenerateButton)
        self._CreateCheckBox("checkbox_showsolution", "Show Solution",
            QRect(10, 200, 140, 20), self._callback_SolutionCheckBox)
        self._CreateLineEdit("lineedit_filepath", QRect(10, 230, 110, 20))
        self._CreateButton("button_browse", "...", QRect(120, 230, 30, 20),self._callback_BrowseButton)
        self._CreateButton("button_save", "Save", QRect(10, 260, 65, 20), self._callback_SaveButton)
        self._CreateButton("button_load", "Load", QRect(85, 260, 65, 20), self._callback_LoadButton)
        # show all widgets
        for w in self._widgets:
            self._widgets[w].show()

    # modular widget creation functions

    def _CreateLabel(self, label_id, text, dimensions):
        self._widgets[label_id] = QLabel(self._MainWin)
        self._widgets[label_id].setText(text)
        self._widgets[label_id].setAlignment(Qt.AlignCenter)
        self._widgets[label_id].setGeometry(dimensions)
```

```
    def _CreateSpinBox(self, spinbox_id, min, max, dimensions):
        self._widgets[spinbox_id] = QSpinBox(self._MainWin)
        self._widgets[spinbox_id].setRange(min, max)
        self._widgets[spinbox_id].setGeometry(dimensions)

    def _CreateComboBox(self, combobox_id, options, dimensions):
        self._widgets[combobox_id] = QComboBox(self._MainWin)
        self._widgets[combobox_id].addItems(options)
        self._widgets[combobox_id].setGeometry(dimensions)

    def _CreateButton(self, button_id, text, dimensions, callback):
        self._widgets[button_id] = QPushButton(self._MainWin)
        self._widgets[button_id].setText(text)
        self._widgets[button_id].setGeometry(dimensions)
        self._widgets[button_id].clicked.connect(callback)

    def _CreateCheckBox(self, checkbox_id, text, dimensions, callback):
        self._widgets[checkbox_id] = QCheckBox(self._MainWin)
        self._widgets[checkbox_id].setText(text)
        self._widgets[checkbox_id].setGeometry(dimensions)
        self._widgets[checkbox_id].clicked.connect(callback)

    def _CreateLineEdit(self, lineedit_id, dimensions):
        self._widgets[lineedit_id] = QLineEdit(self._MainWin)
        self._widgets[lineedit_id].setGeometry(dimensions)

    def _CreatePreview(self, preview_id, img):
        if preview_id in self._widgets.keys():
            self._widgets[preview_id].clear()
        dimensions = img.size
        self._MainWin.resize(180 + dimensions[0], max(290, dimensions[1] + 20))
        self._widgets[preview_id] = QLabel(self._MainWin)
        self._widgets[preview_id].setGeometry(QRect(170, 10, dimensions[0], dimensions[1]))
        self._imgQt = ImageQt.ImageQt(img)
        self._imgPixmap = QPixmap.fromImage(self._imgQt)
        self._widgets[preview_id].setPixmap(self._imgPixmap)
        self._widgets[preview_id].show()

    # callback functions

    def _generateMaze(self):
        # selects generation algorithm from input
        if self._algorithm == "Recursive Division":
            self._mazeInstance.recursiveDivision()
        else:
            self._mazeInstance.setupMaze()
            self._mazeInstance.resetUnvisited()
            if self._algorithm == "Kruskals":
                self._mazeInstance.kruskalRandom()
            elif self._algorithm == "Prims":
                self._mazeInstance.primRandom()
            elif self._algorithm == "Recursive Backtracking":
```

```
            self._mazeInstance.recursiveBacktracking()
        else:
            # fallback state if program fails to generate properly
            self._state = None

    def _callback_GenerateButton(self):
        # gets options from ui
        xDimensions = self._widgets["spinbox_xdimensions"].value()
        yDimensions = self._widgets["spinbox_ydimensions"].value()
        self._algorithm = self._widgets["combobox_generationalgorithm"].currentText()
        # initialises a instance of the maze class
        self._mazeInstance = Maze(xDimensions, yDimensions)
        # times the generation
        self._state = "generated"
        time = timeit(self._generateMaze, number=1)
        if time < 1:
            if time < 0.01:
                time = "{0:.4f}".format(time * 1000) + " ms"
            else:
                time = "{0:.2f}".format(time * 1000) + " ms"
        elif time > 60:
            time = "{0:.2f}".format(time / 60) + " min"
        else:
            if time < 10:
                time = "{0:.4f}".format(time) + " s"
            else:
                time = "{0:.3f}".format(time) + " s"
        print(str(self._mazeInstance._xDimensions)+"x"+str(self._mazeInstance._yDimensions)+
            " Generated in: " + time + " using the "+self._algorithm+" algorithm")
        # solves maze (if properly generated)
        if self._state == "generated":
            self._mazeInstance.dijkstraPathfinding()
            self._widgets["checkbox_showsolution"].setChecked(False)
            self._state = "solved"
        # generates bitmap to display (initially with no path)
        self._generatedMaze = self._mazeInstance.genBitmap(False)
        self._CreatePreview("preview_mazeimg", self._generatedMaze)
        QPixmapCache.clear()

    def _callback_SolutionCheckBox(self):
        # if maze is not solved, the checkbox will stay unchecked
        if self._state != "solved":
            self._widgets["checkbox_showsolution"].setChecked(False)
            return
        # generates bitmap with/without path (dependant on checkbox state)
        self._generatedMaze =
            self._mazeInstance.genBitmap(self._widgets["checkbox_showsolution"].isChecked())
        self._CreatePreview("preview_mazeimg", self._generatedMaze)

    def _callback_BrowseButton(self):
        if self._state != "solved":
            return
```

```
        # prompts user to select directory
        filedialog = QFileDialog()
        self._filepath = QFileDialog.getExistingDirectory(filedialog, "Select Save Directory...",
                                 self._widgets["lineedit_filepath"].text()
                                 if os.path.isdir(self._widgets["lineedit_filepath"].text()) else "",
                                 options=QFileDialog.options(filedialog))
        if self._filepath:
            # truncates filepath to valid directory
            while len(self._filepath) > 0:
                if os.path.isdir(self._filepath):
                    self._widgets["lineedit_filepath"].setText(self._filepath)
                    break
                else:
                    self._filepath = "/".join(self._filepath.split("/")[0:-1])

    def _callback_SaveButton(self):
        # if maze is not solved, it cannot be saved
        if self._state != "solved":
            return
        # gets info to create identifying filename
        self._filepath = self._widgets["lineedit_filepath"].text()
        algorithm = self._algorithm.replace(" ", "_")
        dimensions = self._mazeInstance.getDimensions()
        # prompts user to select directory if not already selected
        if not os.path.isdir(self._filepath):
            self._callback_BrowseButton()
        # generates bitmap with and without the solution and saves them
        if self._filepath:
            self._generatedMaze = self._mazeInstance.genBitmap(False)
            self._generatedMaze.save(self._filepath + "/" + algorithm + "_" + dimensions + ".png")
            self._generatedMaze = self._mazeInstance.genBitmap(True)
            self._generatedMaze.save(self._filepath + "/" + algorithm + "_" + dimensions + "_Solved.png")
            print("saved")

    def _callback_LoadButton(self):
        # prompts user to select file
        filedialog = QFileDialog()
        self._filepath, _ = QFileDialog.getOpenFileName(filedialog, "Select File to Load...",
            self._widgets["lineedit_filepath"].text(),
            "Image Files (*.png);", options=QFileDialog.options(filedialog))
        if self._filepath:
            # if valid filepath
            if os.path.exists(self._filepath):
                # opens the file and imports into the class
                img = Image.open(self._filepath)
                self._mazeInstance = Maze(int((img.size[0] - 1) / 2),int((img.size[1] - 1) / 2))
                self._mazeInstance.importBitmap(img)
                # solves maze if not solved - checks colour of entrance pixel
                if img.load().__getitem__((1, 0)) != (255, 0, 0):
                    self._mazeInstance.dijkstraPathfinding()
                self._widgets["checkbox_showsolution"].setChecked(False)
                self._state = "solved"
```

```
            # generates bitmap to display (initially with no path)
            self._generatedMaze = self._mazeInstance.genBitmap(False)
            self._CreatePreview("preview_mazeimg", self._generatedMaze)
        print("loaded")

if __name__ == "__main__":
    # initialises an instance of the Ui class, and creates the window
    app = QApplication(sys.argv)
    MasterWindow = QMainWindow()
    ui = Ui(MasterWindow)
    MasterWindow.setWindowTitle("Maze Generator")
    MasterWindow.show()
    sys.exit(app.exec_())
```

Functionality Testing:

The *Evidence* column references the video linked here:
https://drive.google.com/file/d/1gy8BV8zFY9i6hW_c8-6kYTImdBKgbs3J/view?usp=sharing

ID	Test (Objective)	Data Type	Expected Outcome	Pass / Fail	Evidence
1a	X, Y dimensions (1.b)	Normal	Any integer within the range 2 to 200 can be inputted	Pass	0:10
1b	(1.b.i)	Boundary	Any integer outside the range 2 to 200 cannot be inputted	Pass	0:39
2	Dropdown menu (1.a.i)	Normal	Any of the 4 available algorithms can be selected	Pass	0:59
3a	Generate Process	Normal	Pressing Generate button generates maze	Pass	1:18
3b	(6.a)	Normal	Window resizes and maze is displayed	Pass	1:22
3c	(6.c)	Boundary	Button pressed multiple times in a short timeframe	Pass	1:51
4a	Show Solution Checkbox (6.b/3)	Normal	Pressing checkbox toggles visibility of solution on the displayed maze	Pass	2:16
4b	(6.c.i)	Boundary	Pressing checkbox before maze has been generated has no effect	Pass	2:33
5a	Browse Button (5.a)	Normal	Pressing Browse opens file path selection dialog	Pass	2:59
5b	(5.a)	Normal	Path of selected folder is validated, then shown in the file path input	Pass	3:16
6a	Save Button (5.c)	Normal	Saves image of maze with and without solution into selected directory	Pass	3:32

6b		Boundary	If no directory has been selected, opens file path selection dialog before saving	Pass	4:09
7a	Load Button (5.b)	Normal	Opens file selection dialog	Pass	4:52
7b	(5.b)	Normal	Loads selected file into a maze class instance	Pass	5:08
7c	(3)	Normal	Solves maze if required	Pass	5:22

Performance Testing:

ID	Test (Objective)	Data Type	Expected Outcome	Pass / Fail	Evidence
8a	Minimum Size 2x2 - Kruskals	Boundary	Successful generation and solving	Pass	6:07
8b	Minimum Size 2x2 - Prims	Boundary	Successful generation and solving	Pass	6:11
8c	Minimum Size 2x2 - Recursive Backtracking	Boundary	Successful generation and solving	Pass	6:16
8d	Minimum Size 2x2 - Recursive Division	Boundary	Successful generation and solving	Pass	6:19
9a	<50x50 - Kruskals (1.c.i)	Normal	Successful generation and solving under 1s	Pass	6:35
9b	<50x50 - Prims (1.c.i)	Normal	Successful generation and solving under 1s	Pass	6:42
9c	<50x50 - Recursive Backtracking (1.c.i)	Normal	Successful generation and solving under 1s	Pass	6:48
9d	<50x50 - Recursive Division (1.c.i)	Normal	Successful generation and solving under 1s	Pass	6:53
10a	Maximum Size 200x200 - Kruskals (1.c.ii)	Boundary	Successful generation and solving under 1 min	Pass	7:14

10b	Maximum Size 200x200 - Prims (1.c.ii)	Boundary	Successful generation and solving under 1 min	Pass	8:09
10c	Maximum Size 200x200 - Recursive Backtracking (1.c.ii)	Boundary	Successful generation and solving under 1 min	Pass	8:22
10d	Maximum Size 200x200 - Recursive Division (1.c.ii)	Boundary	Successful generation and solving under 1 min	Pass	8:32

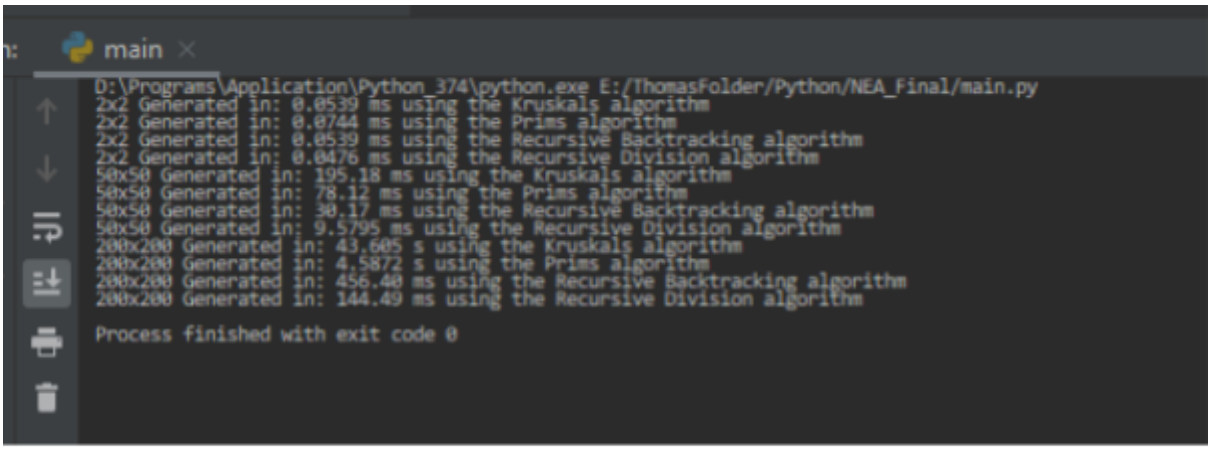


Figure 11: The time taken to generate each of the mazes during testing

Objective Summary:

Objective	Objective Description	Relevant Test(s)	Objective Met
1.a	User should be able to generate a maze using each of the algorithms, selectable using a dropdown menu	2	Yes
1.b	User should be able to select dimensions within the range 2-200 for both x and y	1a,b	Yes
1.c	The time taken to generate should be < 1s for mazes 50x50 or smaller, and < 1min for mazes 200x200 or smaller	Performance testing - (8) 9 10	Yes
2.a	The maze should be stored in an array	N/a	Yes
3.a, b	The maze should be solved using Dijkstra's algorithm and stored in the maze array	N/a	Yes
4.a	The maze should be able to display in the console	See Fig. 6* for console display	Yes
5.a, c	The user should be able to save the maze as a bitmap to a chosen file location, with and without the solution	6a(,b)	Yes
5.b	The user should be able to load previously generated maze into the program (and view the solution)	7a,b(,c)	Yes
6.a	The interface should display the generated maze, resizing if required	3b	Yes
6.b	The display of the solution should be togglable	4a	Yes
6.c	The interface should be able to handle invalid inputs	1b,3c,4b,5b,6b	Yes

This table shows that I have met my objectives set out in my analysis section. From my testing, I can conclude that my program is functional.

Investigation:

Using my program, I generated a few samples of mazes using a range of dimensions for each algorithm. In this section, I will compare each of the algorithms, and I have also chosen some notable irregular dimensions to look into.

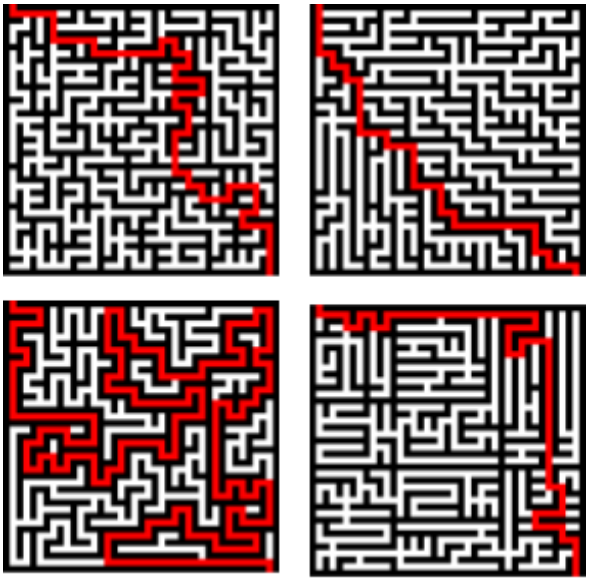


Figure 12: A 20x20 maze generated by each algorithm (K,P,RD,RB going clockwise from top-left)

Fig. 12 shows a small maze generated by each of the algorithms, allowing us to compare their qualities. Both Kruskals and Prims generate mazes with a similar appearance. From my samples, the former seems to generate mazes with longer straight paths than the latter. Kruskals has, on average, more junctions, as more walls tend to be removed before the algorithm stops. Recursive Division / Backtracking mazes look quite different, as the recursion used determines the qualities of the generated maze. Recursive Backtracking produces mazes with long, winding paths, and very few junctions, leading to very erratic solutions, with a tendency to fill a large proportion of the maze's area. Mazes generated through Recursive Division, however, are much more organised. Within a Recursive Division maze, the distinct sections can be seen clearly.

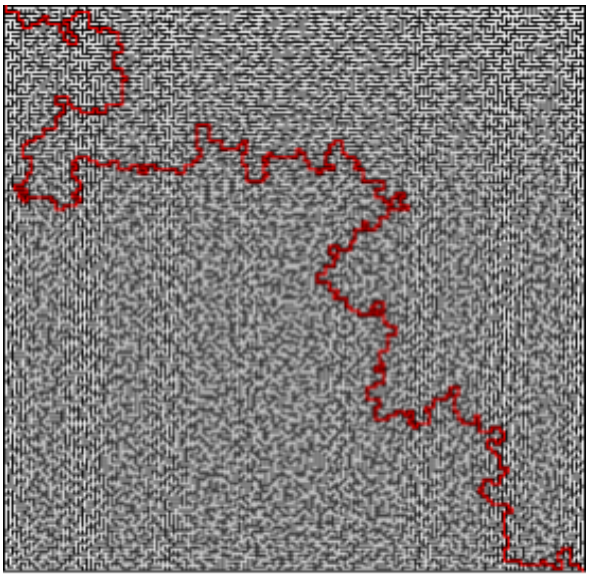


Figure 13: A 140x140 maze generated using Kruskal's algorithm

While with smaller mazes, it is quite unlikely for the path not to simply be a straight line from top-left to bottom-right (particularly referring to the first two algorithms), when you increase the size (Fig. 13, 14), I noticed that significant detours are much more common. These are no fault with the algorithm, but just a result of using random generation.

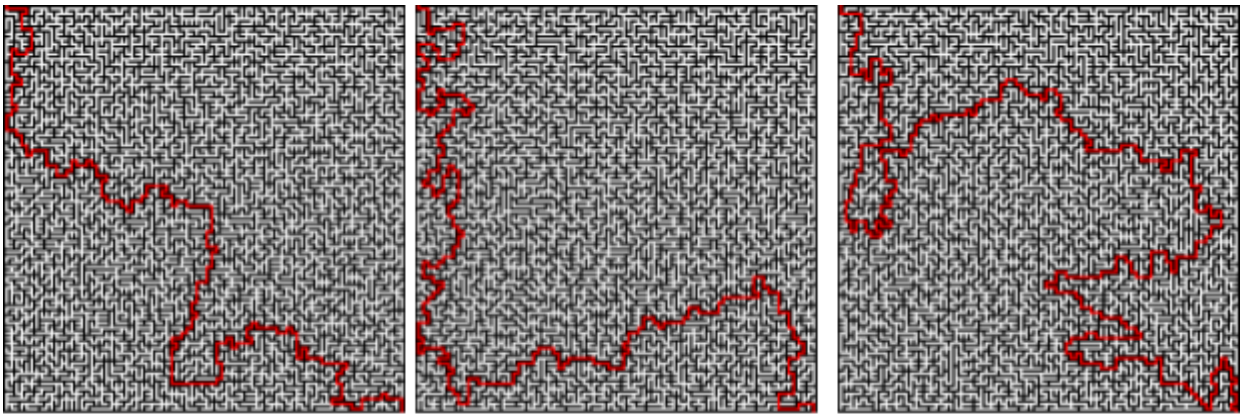


Figure 14: A sample of three 80x80 mazes generated using Kruskal's algorithm

Fig. 15 shows a maze from each algorithm with the dimensions 100x50. As previously mentioned, Kruskals generates more zig-zag paths at this size. The Prims maze creates very direct paths, as the straight sections seem to favour this aspect ratio. Again, the mazes generated with Recursive Backtracking demonstrate longer, winding solutions (https://www.astrolog.org/labyrnth/algrithm.htm calls this quality “river”) - with a rectangular maze, the sheer length of the solutions is only amplified. Non-square mazes generated with Recursive Division are interesting as they often appear “weighted”: having more junctions and splits to one side; and longer, uninterrupted, straight paths on the other. They also sometimes have a slight bias towards horizontal paths, which I can only assume is caused by the increased likelihood with a maze of such dimensions.

In terms of generating mazes for humans to solve, either Kruskals or Prims seems most appropriate, as they generate the “friendliest” mazes - they follow a logical progression from the entrance to the exit. The other two algorithms are only really suitable at smaller sizes, as they can get out of hand as the size increases. At large sizes, the solutions with these algorithms become either very excessive or just painfully trivial (Fig. 16, 17).

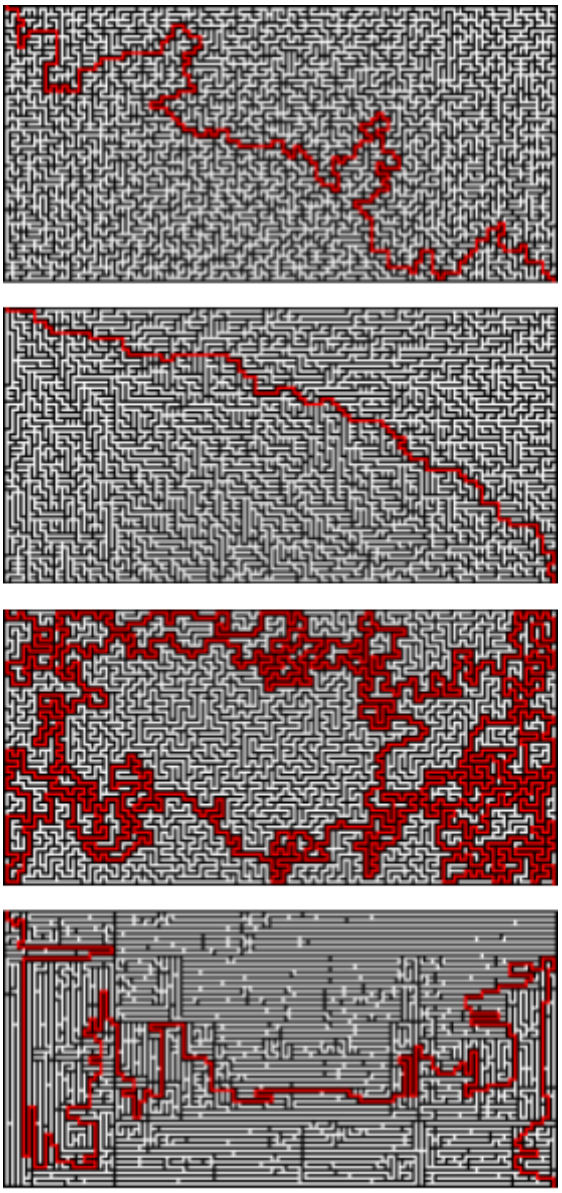


Figure 15: A 100x50 maze generated by each algorithm (K,P,RB,RD top-to-bottom)

Figure 16: A 200x200 maze generated using Recursive Division

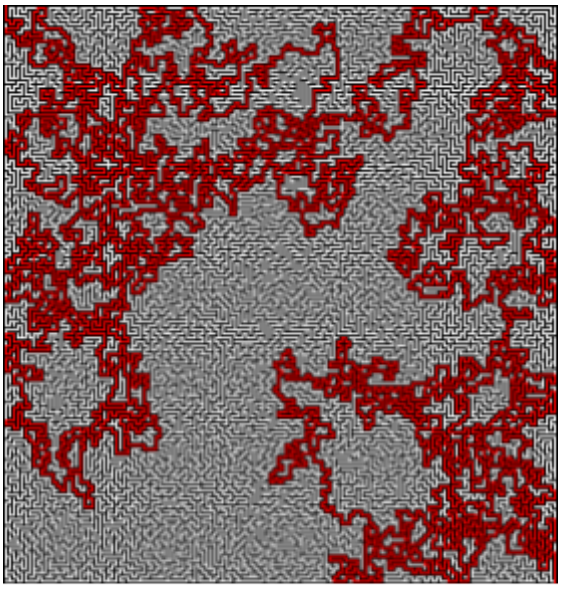


Figure 17: A 140x140 maze generated using Recursive Backtracking

Improvements:

I systematically saved older iterations of my program to document the changes made during the developmental process. Most of the changes were simply to improve the speed of the algorithms, by implementing them in a more efficient way.

The main improvement made with Kruskal's algorithm was the introduction of the union find structure, which drastically improved the speed of the set manipulation. In comparison to my previous method, which was quite slow and memory intensive, the efficiency of the set manipulation was drastically improved. In addition, before the introduction of the improved sets, the algorithm would run until all the walls were exhausted, so effectively had a constant runtime. The new system was able to improve upon this.

The Prim's algorithm went through several re-writes, as I found more elegant ways to clear excess cells from the *activeCells* array. This step was required to make sure the chosen cell was valid, and would not break the program. Originally, I was using a for loop to simply iterate through each cell in the array and check if it should be removed - this became a problem when generating at larger sizes. I eventually realized that the only cells that may need to be checked are those neighbouring the cell where the changes were made. This was much more effective as each iteration only required 4 cells to be checked.

Recursive Backtracking was fairly easy to implement - I just had to ensure that data was added/removed from the stack at the right point within the loop. Both the recursive algorithms required me to create a separate stack to circumvent Python's stack limit.

With Recursive Division, I was initially unsure of how to store the properties of each chamber. My first idea was to store all the coordinates within each chamber - this was obviously very inefficient, memory-wise. I decided to create a new structure to store the chambers, by just storing the bounds of each chamber. This still allowed me to choose points within the chamber, as I could just check if the x and y coordinates were within the right range, and created less data to store in the stack.

Some optimisation was also required with the UI. I was experiencing intermittent crashes after generating mazes (more often with the larger mazes), and I concluded it was a memory issue related to the bitmaps I was storing to display in the GUI. Eventually I discovered that the maze which was previously displayed on the screen was not actually being cleared off the screen, and newly generated mazes were just superimposed above the previous one. To fix this, I had to ensure that a reference to the bitmap was being stored so I could clear it before generating and displaying a new maze.

End User Evaluation:

Documented below is a short interview with my end-user on the suitability of my application, where we discuss if I have met our objectives.

Q: After testing the program, are you satisfied that your requirements have been met?
A: *I would say yes. The four unique generation algorithms mean we can easily tailor the mazes produced for different projects. The solution is also very useful, as it allows us to quickly decide if a maze is suitable.*

Q: In comparison to your previous methods, how has this streamlined your design process?
A: *This program is definitely a significant improvement on our previous workflow, which was predominantly manual (using CAD software). The main issue before was ensuring the maze had a feasible solution - we often had to work backwards from an intended route in order to circumvent this, but in turn this would lead to mazes which were not particularly challenging. This system has also dramatically reduced the time taken to design a maze.*

Q: What improvements, if any, would you suggest for the interface?
A: *Perhaps an option to scale up the mazes in the preview - the more intricate mazes can be difficult to make out, so having each unit be maybe 2 or 3 pixels would help.*

Conclusion:

I have been able to successfully create a program in accordance with my objectives, and my end-user is satisfied that they have been met. All 4 algorithms were implemented efficiently, allowing me to investigate and compare them in a visual way, and consider whether each algorithm is appropriate for humans to try to solve. There are some improvements which could be made in the future, such as the implementation of more algorithms, or modifications to the interface.