Maze Generation

Abdallah Muhammed

The National Mathematics and Science College

Table of Contents

Analysis

Problem Background:

Students at the National Mathematics and Science College have recently begun a puzzle club where they meet to solve a combination of different puzzles such as mazes and crosswords. The club has grown in popularity, causing new and existing members to become increasingly interested in mazes and how to solve them using various methods and mediums. Some students in the club would prefer to work on paper, while others would like to apply their programming skills to solve far more complex mazes than possible to solve on paper.

Due to the wide range of skills in the club, the students have been having difficulties finding mazes of varying difficulty that will be enjoyable for both amateur and experienced solvers, also due to the sudden influx of students into the club. There has been a scarcity in the number of puzzles available for the club members to solve. Similarly, the programmers in the puzzle club have also been having problems finding adequately complex puzzles that they can write programs to solve. There is no unified data structure used to represent the mazes in memory. So, the current system that the programmers use is to recreate the maze using python, which is very inefficient and time-consuming. Also, each student has their own method of representing the maze, so the data structure for mazes cannot be shared between different students.

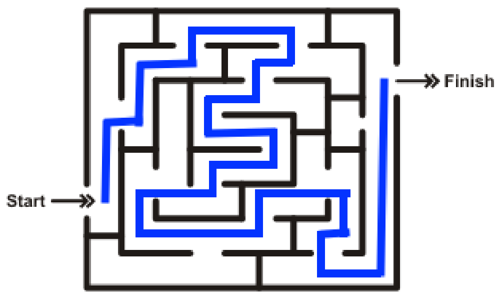
Furthermore, the head of Computing at the college, Mr C, is looking for new materials and interactive methods of teaching the computer science students about abstract data types and how to use such data types when writing a program. Currently, in the A-Level syllabus, students learn about stacks and queues. There is a multitude of resources available to teach about these abstract data types in the syllabus, but Mr C would like to expand beyond the syllabus and show students that they can create new abstract data types while simultaneously providing the opportunity to improve their Object-oriented programming (OOP) and programming skills in general.

I am looking to create an application that can generate mazes of varying difficulties and solve the maze, showing the shortest path. The application should also be able to export the mazes as images to be printed. It should also have the capability to export the internal data structure of the maze so other students can use it in their programs.

Research:

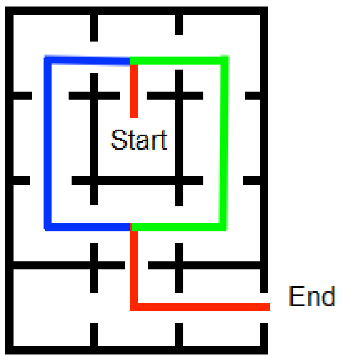
Types of Mazes:

Simply connected:



This is a maze that only has on path from start to finish, so the shortest path is the only solution.

Multi-connected maze:



This is a maze that has more than one solution. Therefore, the shortest path is the most efficient way to solve the maze.

Maze generation algorithm:

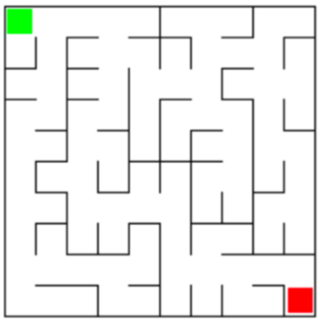
Prim's randomized algorithm:

Prim’s algorithm is a greedy algorithm; this means it uses problem-solving heuristics to find the most optimal choice at each stage of the program’s execution. Prim’s algorithm finds the minimum spanning tree of a weighted undirected graph. Therefore, it forms a tree that includes all the vertices and therefore, the total weight of the edges in the tree are minimized, by building the tree one vertex at a time and starting at an arbitrary initial vertex.

The algorithm was first developed by a Czech mathematician Votêch Jarník in 1930 but was not rediscovered and published until 1959 by computer scientist Robert C. Prim.

The version of Prim’s algorithm used to create the maze is randomized therefore it creates variations in the weights of the edges, this means the edges closer to the starting vertex have a smaller effective weight than those located further away from the starting vertex. The randomized version of Prim’s algorithm also differs from the classical Prim’s algorithm because instead of storing a list of edges, the randomized algorithm stores a list of adjacent cells. If the randomly chosen cell has multiple edges that connect to the existing maze, then one of the unselected edges will be selected at random. This will cause it to branch more than the edge-based version, which therefore produces a more complex maze.

Example of a maze created with Prim's randomized algorithm:



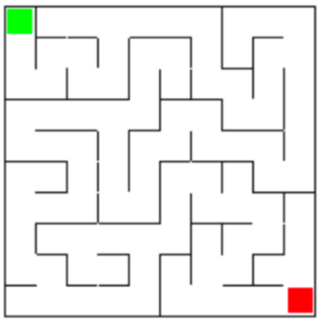
The maze created by Prim's randomized algorithm is more vertically biased and has more dead ends, which makes it more complex compared to a maze created using the recursive backtracking algorithm below.

Recursive backtracking method:

The recursive backtracking maze creation algorithm uses a depth-first search in tandem with backtracking to produce the maze. It recursively follows the routine of being passed a cell, then marking the cell as visited and while the current cell has any unvisited adjacent cells a random adjacent cell is chosen and the wall between the new cell and the current cell is removed. This is repeated until the maze is complete.

In the case that all the adjacent cells have been visited the algorithm backtracks until it reaches a cell that has an unvisited adjacent cell. Then the same process above is repeated until all the cells have been visited and the maze is complete.

Example of a maze created with Recursive backtracking algorithm:



The maze created by the recursive backtracking algorithm is more horizontally biased and has fewer dead ends compared to the maze created using Prim's randomized algorithm above.

Maze-solving algorithm:

A* search algorithm:

A* is a graph traversal and path searching algorithm, that has a space complexity of O(b^d). This is due to all the generated nodes being stored in memory. On the other hand, A* is more efficient than other graph traversals and path algorithms because it uses heuristics. it considers the position/location of the end node while searching for it and hence it searches less nodes to reach the end.

Python Libraries:

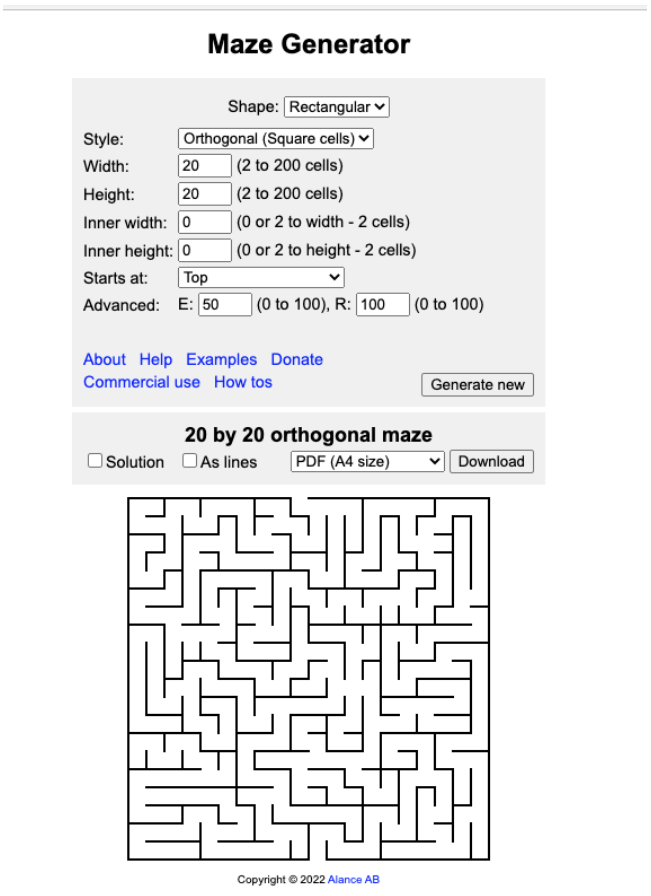
Pygame

Pygame is a set of python modules designed for writing video games. Pygame provides functionality that allows for the creation of fully featured games and multimedia programs in python. It includes python libraries for graphics and sound.

Pygame was written by Pete Shinners to replace PySDL after its development stalled in the early 200s. Pygame has been a community project since then and it is released under the free software GNU Lesser General Public License (which allows Pygame to be distributed with both open source and commercial software)

Description of current system:

System 1:



Maze Generator

Shape: Rectangular

Style: Orthogonal (Square cells)
Width: 20 (2 to 200 cells)
Height: 20 (2 to 200 cells)
Inner width: 0 (0 or 2 to width - 2 cells)
Inner height: 0 (0 or 2 to height - 2 cells)
Starts at: Top
Advanced: E: 50 (0 to 100), R: 100 (0 to 100)

20 by 20 orthogonal maze

Copyright © 2022 Alance AB

Advantages:

- The user can select the width and height of the maze
- The user can select where the start cell is located
- The user can select the shape of the maze
- The user can select the size of the individual cell
- The user can choose the style of the maze
- The maze can be exported as a PDF
- The user can see the solution of the maze by toggling the solution button
- There is a limit to how large the user can make the maze

Disadvantages:

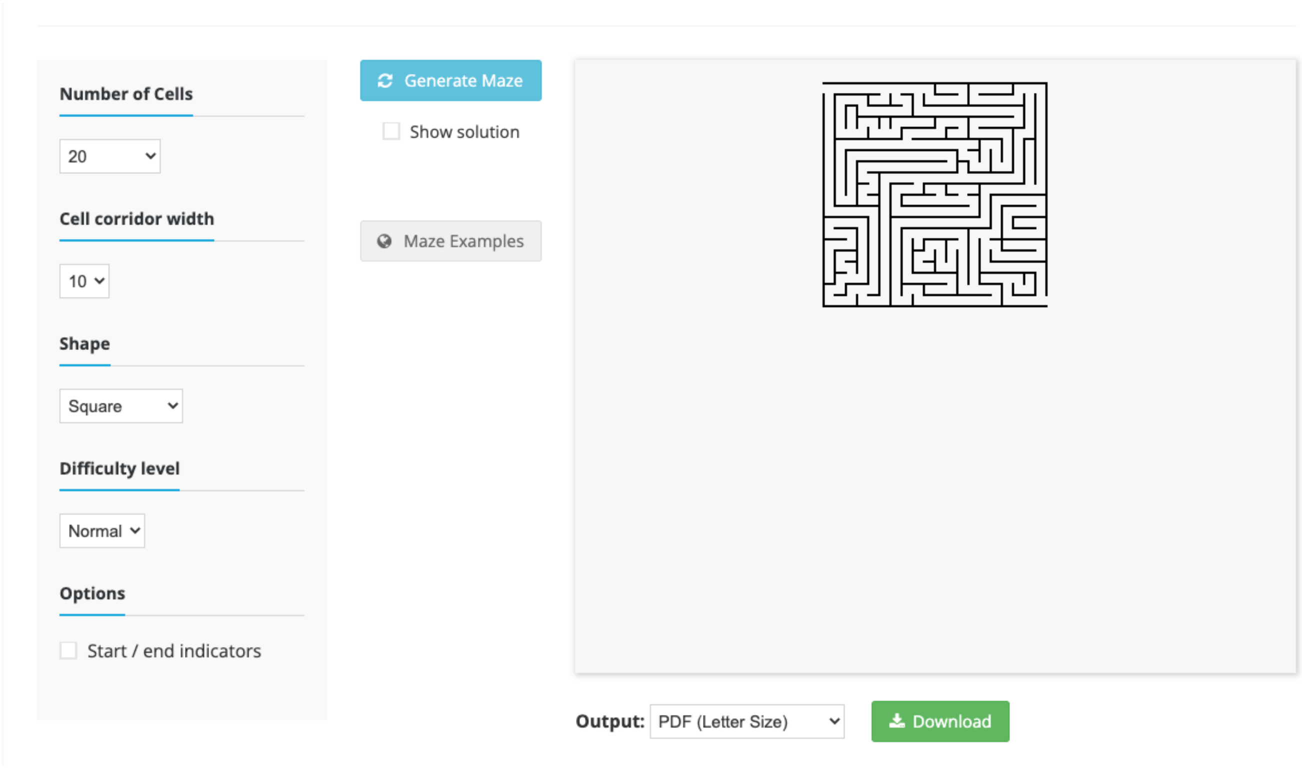
- The ending cell is chosen at random
- The maze cannot be exported as an image
- The maze cannot be exported as a text file with an abstract memory representation of the maze
- The maze creation algorithm cannot be chosen

- The maze solving algorithm cannot be chosen
- There is no visual simulation as to how the maze is created
- There is also no visual simulation to how the maze is solved

Overall:

The program above provides the user with various methods to alter the maze to their specification. On the other hand, the program does not provide the user with multiple maze creation and maze solving algorithms and it does not provide an abstract memory representation of the maze that the user can use in their own programs. The program could also be more interactive by providing a visual simulation to how the maze is created and solved.

System 2:



Number of Cells: 20
Cell corridor width: 10
Shape: Square
Difficulty level: Normal
Options: Start / end indicators
Output: PDF (Letter Size)

Advantages:

- The user can select the size of the maze
- The program can identify where the starting cell and ending cell are located, which makes it easier for the user to locate
- The user can select the shape of the maze
- The user can select the size of the individual cell in pixels
- The maze can be exported as a PDF
- The user can select the difficulty of the maze

- The user can see the solution of the maze by toggling the solution button
- The program also provides example mazes for the user to see
- The starting cell and ending cell are set in predetermined locations:
    - Start cell in the top left
    - End cell in the bottom right

Disadvantages:

- The maze cannot be exported as an image
- The maze cannot be exported as a text file with an abstract memory representation of the maze
- The maze creation algorithm cannot be chosen
- The maze solving algorithm cannot be chosen
- There is no visual simulation as to how the maze is created
- There is also no visual simulation to how the maze is solved

Overall:

The program above provides the user with various methods to alter the maze to their specifications such as altering the difficulty and the size of the maze. On the other hand, the program does not provide the user with multiple maze creation and maze solving algorithms and it does not provide an abstract memory representation of the maze that the user can use in their own program. The program could also be more interactive by providing a visual simulation to how the maze is created and solved.

Requirements gathering (interviews with students):

Questions:

1. What issues do you have when writing programs to solve a maze?
2. What aspects do you think are important in a maze generator program?
3. How big would you like the mazes generated to be?
4. Would you like the maze generation and solving to be animated?
5. How would you like the maze to exported so you can use it in your own programs?

Summary of interview with student Ilan Iwumbwe:

Ilan’s main problem when writing programs to solve mazes was the lack of flexibility on the type of mazes that can be created. He also stated that he values having different algorithms to create the mazes and having the ability to choose the size of the maze. On the other hand, he also stated that he prefers for the maze to have different levels of complexity compared to a maximum size. Lastly, he would like for the maze creation and solving to be animated and for the program to have an option to export the maze to be used in his own programs.

The following objectives can be gathered from Ilan's interview:

- The program should have different maze creation algorithms, to vary the type of mazes that the user can create
- The program should allow the user to choose the size of the maze that should be created
- The program should also include a difficulty setting for the user to select
- The maze creation and maze solving should be animated
- The program should be able to export the internal representation of the maze as a text file

Summary of interview with student Sofia Negreskul:

Sofia's main problem when writing programs to solve mazes was the lack of complex enough mazes that could be solved. She also stated that she values seeing the solution of the maze and she prefers large and complex mazes such as a 50x50 maze, which take a relatively long time to solve by hand. Lastly, she would like the maze creation and solving to be animated and for the program to have an option to export the maze. This should also come with a class that has subroutines which she can use in her own program to access the maze data.

The following objectives can be gathered from Sofia's interview:

- The program should allow the user to choose the size of the maze that should be created
- The program should also include a difficulty setting for the user to select
- The program should be able to solve the maze
- The maze creation and maze solving should be animated
- The program should be able to export the internal representation of the maze as a text file
- The program should also include a class in which the user can use subroutines to utilise the exported maze's data

Objectives

Overall objective:

The project will allow a user to generate mazes of varying difficulties and solve the mazes, showing the shortest path. The application would also be able to export the mazes as images to be printed. It will also have the capability to export the internal data structure of the maze so other students can use it in their programs.

SMART objectives:

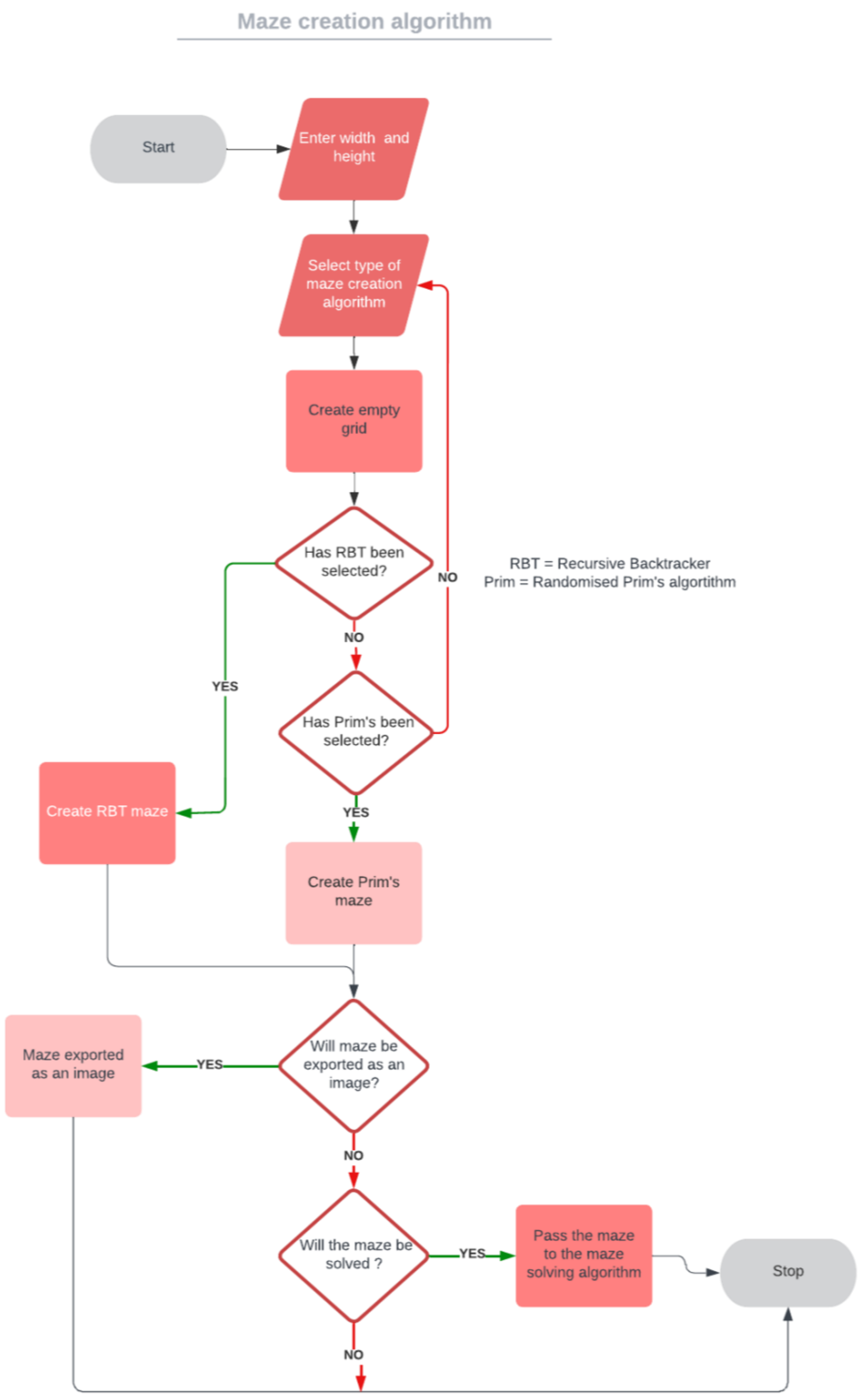
1. The user should have a user interface where they can:
   a) Select what maze generation algorithm they want to use

        i) The user should be able to use the recursive backtracking algorithm
        ii) The user should be able to use Prim's randomized algorithm
    b) Select if they want the maze to be solved
        i) The colour of the solution should be a complementary colour to the grid colour so the solution of the maze can be easily seen
    c) Select the grid size
        i) The size of each cell should change depending on the grid size
        ii) The user should be limited to how large they can make the maze
            (1) Ensure each individual cell and wall is still visible by the user
    d) The user should be able to select the difficulty of the maze if grid size is not given
    e) Select if the maze should be animated
    f) Export the maze as a picture or as text file with an abstract memory representation of the maze

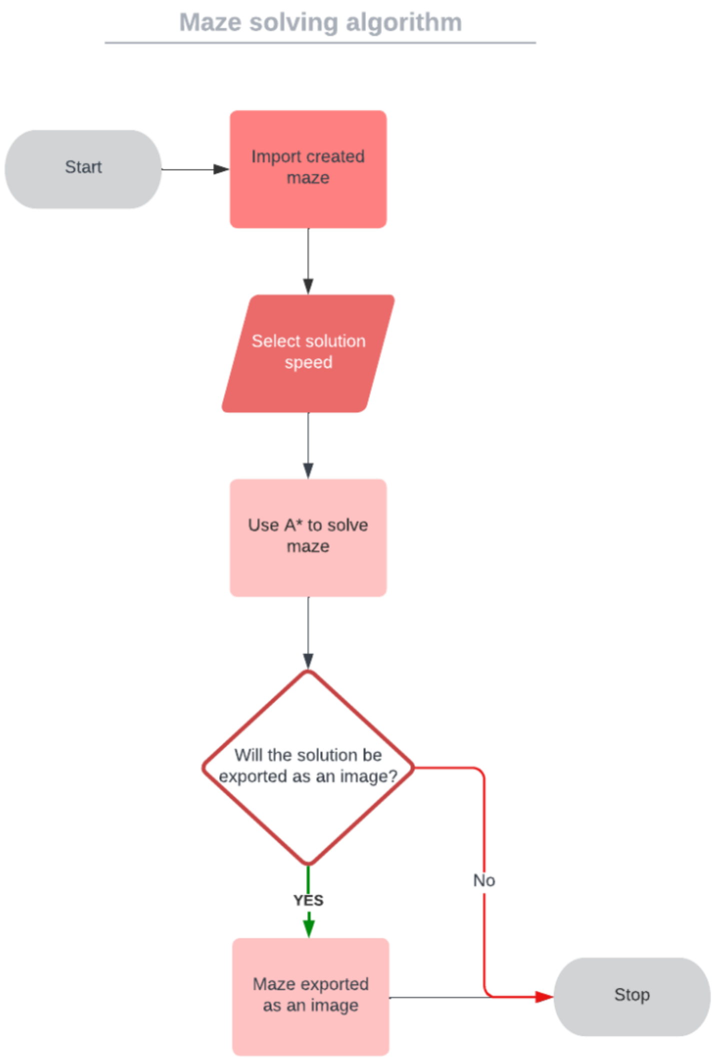
2) The starting cell and ending cell should be set and shown to the user at the beginning of program execution
    a) The start cell should be in the top left hand conner of the maze
    b) The ending cell should be in the bottom right hand conner of the maze
3) Be able to store a maze in a suitable abstract data structure making use of OOP principles:
    a) The user should be able to export this memory representation of the maze as a text file
    b) The user should be able to use a class which contains subroutines to process the maze data
4) Be able to implement a recursive backtracking maze generation algorithm to create a simply connected maze
    a) The maze generation should be able to be animated on the GUI by the user
5) Be able to implement Prim's randomized algorithm for maze creation of a simply connect maze
    a) The maze generation should be able to be animated on the GUI by the user
6) Be able to implement the A* shortest path algorithm to solve the maze
    a) The maze solving should be able to be animated on the GUI by the user

Initial Modelling

Below you can see the basic outline for how the maze generation algorithm should work and the type of data that should be collected from the user before and during program execution. It also includes an outline of the process used to create the maze.


Maze creation algorithm
Start
Enter width and height
Select type of maze creation algorithm
Create empty grid
Has RBT been selected?
NO
YES
NO
RBT = Recursive Backtracker
Prim = Randomised Prim's algortithm
Has Prim's been selected?
YES
Create RBT maze
Create Prim's maze
Will maze be exported as an image?
YES
Maze exported as an image
NO
Will the maze be solved ?
YES
Pass the maze to the maze solving algorithm
Stop
NO

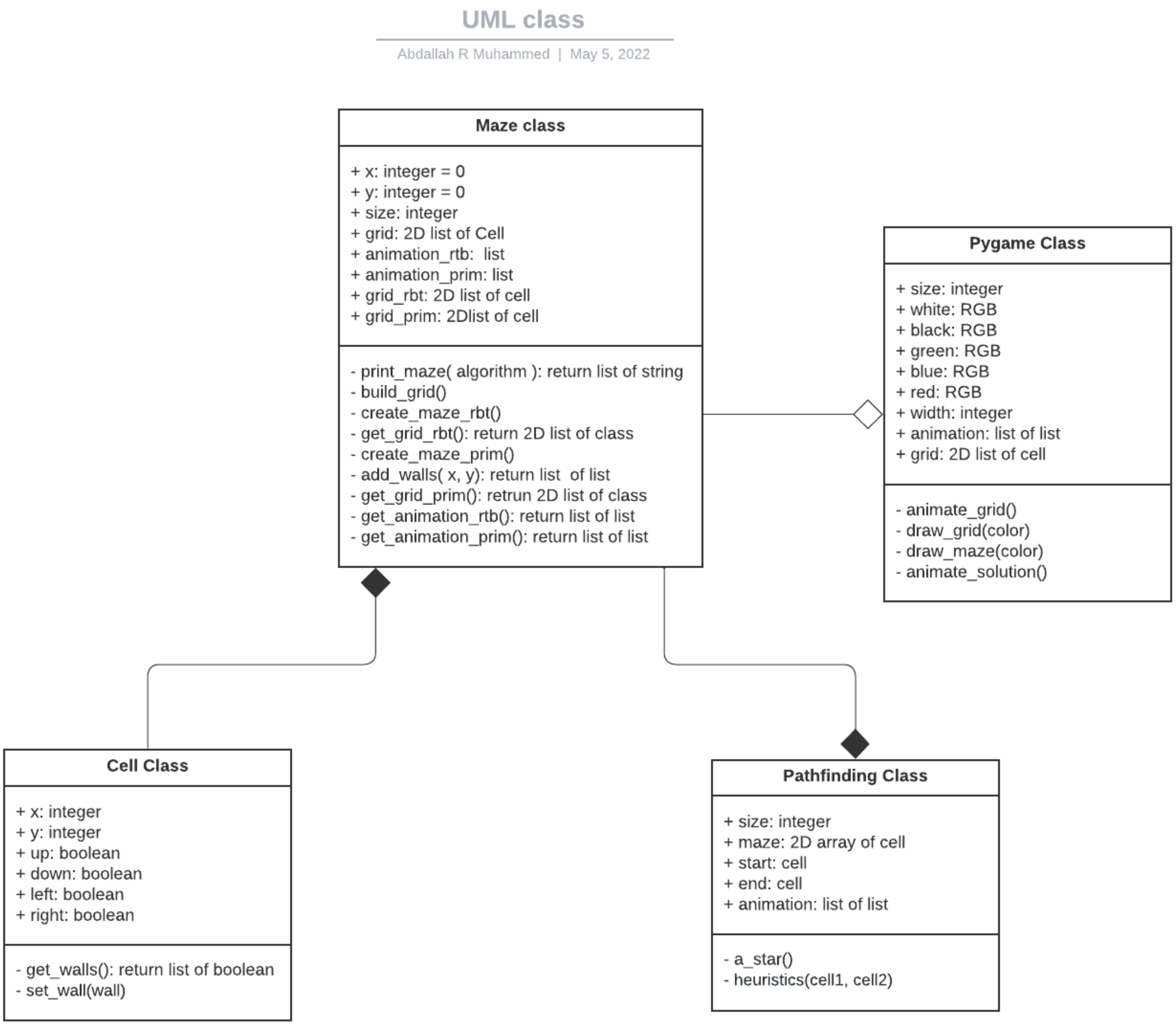
Below you can see the basic outline for how the maze solving algorithm should work and the type of data that should be collected from the user before and during program execution. It also includes an outline of the process used to solve the maze.



Documented Design

Overview of design

In this section, I will break down how the maze program works, and the algorithms used, it will be written in python and utilise OOP principles. The maze program will be able to generate mazes of varying difficulty and solve the maze, showing the shortest path. The application should also be able to export the mazes as images to be printed. It should also have the capability to export the internal data structure of the maze so other students can use it in their programs. The maze is made from four main sections which are broken-down below.



Breakdown of program main sections

Maze creation

The maze creation will be the main part of the program and is made up of the Maze class shown in the UML diagram above. It initialises itself by creating a 2D empty grid depend on the size requested by the user. Each coordinate of the 2D grid is assigned as a cell class, which is explained below in the maze storage section. After each coordinate is assigned a cell class the add_walls method is used to add all four walls to each coordinate. The grid is now passed to either Prim's randomized algorithm or the randomised backtracking algorithm. Both Prim's and the recursive backtracking algorithm are broken down in greater detail below.

The Maze class also has methods to return the mazes created by both Prim's and the recursive backtracking algorithm to be used in other classes. The methods return a 2D array of cells the same size as the grid array. Also, the Maze class contains methods to return a list of walls removed used to animate the maze using the Pygame class. Lastly, the maze class has a method to print the maze onto the console for easy debugging.

Example of console printout for 5x5 maze:

```
+--+--+--+--+--+
|              |
+  +--+  +--+  +
|  |  |  |  |  |
+  +  +  +  +--+
|  |  |        |
+  +  +--+--+--+
|     |        |
+  +  +  +--+--+
|  |           |
+--+--+--+--+--+
```

Maze solving

The maze is solved using the A-star pathfinding algorithm and utilises the pathfinding class in the UML diagram above. A* is an informed search algorithm that considers the location of the goal while searching for it and hence it searches fewer nodes to reach to the goal compared to a breadth first search or a depth first search algorithm. The A* algorithm is broken down in further detail below.

Maze storage

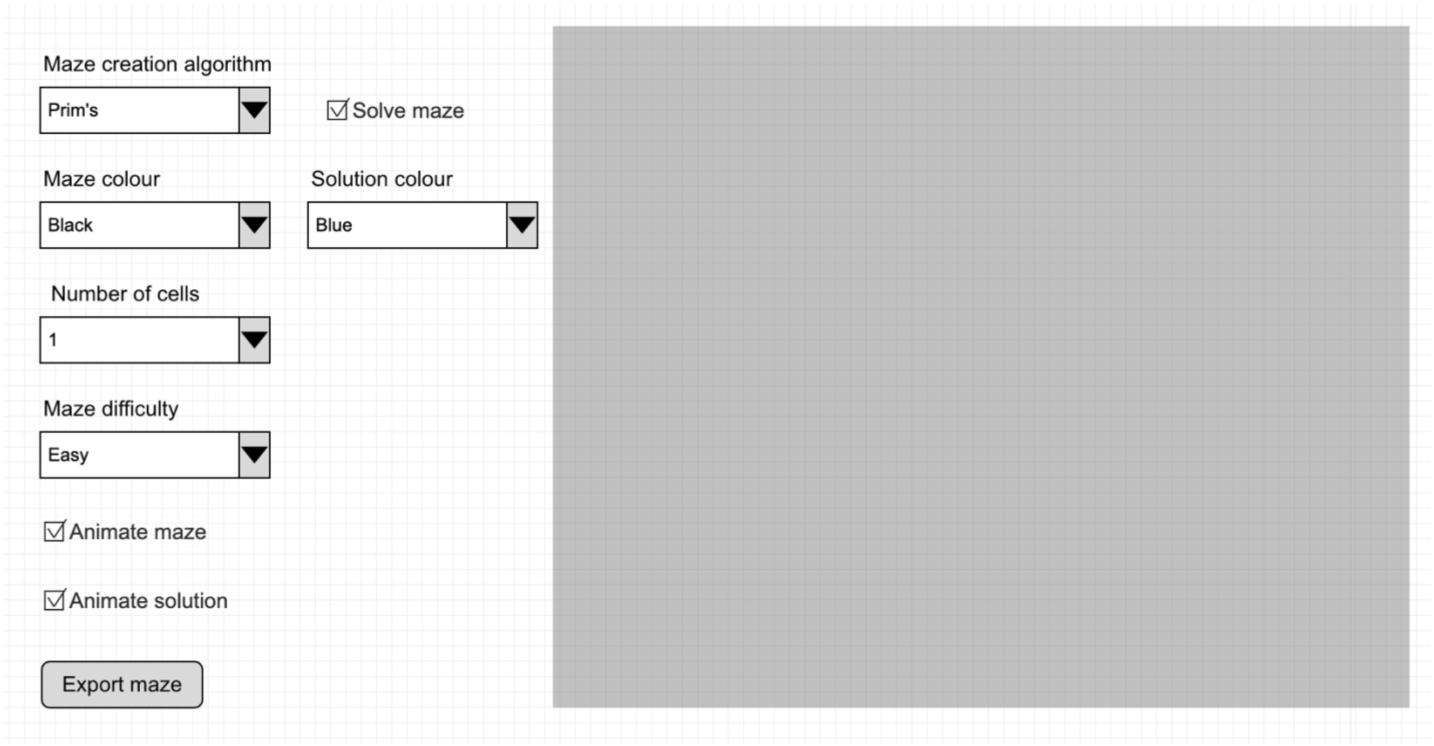
The maze is stored using 2D arrays made up of the cell class. The cell class stores the coordinate of each cell and four of Boolean values indicating if the top, bottom, left, and right wall are present.

The cell class also contains a method to change the state of the top, bottom, left, and right to either true or false. The second method that is in the cell class contains is the get_walls method which returns a list of the walls present for a particular cell.

For example, **get_walls (1,1)** will return **[True, True, True, False].** The list is formatted as **[top, bottom, left, right]**

GUI for user

The layout for the GUI is shown below:



- The grey box is where the maze will be displayed
- The maze creation algorithm dropdown has two options:
    - Prim’s
    - Recursive backtracking
- The maze and solution colour have the following options:
    - Black
    - Blue
    - White
    - Green
    - Red
- The number of cells ranges from 1 to 100
- The maze difficulty has the following options:

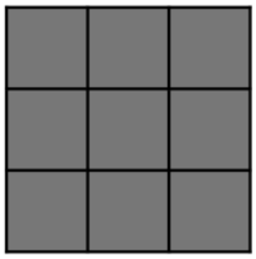
- Easy
- Medium
- Hard

How will I represent a maze?

The maze is represented using 2D arrays and the cell class. The cell class contains a method to change the state of the top, bottom, left, and right to either true or false. The second method that is in the cell class contains is the get_walls method which returns a list of the walls present for a particular cell.

Example with 3x3 maze:

Initially a grid is created which will be represented as a 2D array of coordinates:



```
Grid= [
        [(1,1), (1,2), (1,3)],
        [(2,1), (2,2), (3,3)],
        [(3,1), (3,2), (3,3)]
       ]
```

Then each coordinate is assigned a cell class:

```
Grid= [
                [(1,1) type class, (1,2) type class, (1,3) type class],
                [(2,1) type class, (2,2) type class, (3,3) type class],
                [(3,1) type class, (3,2) type class, (3,3) type class]
        ]
```

```
(1,1) class:
        Top wall = True
        Bottom wall = True
        Left wall = True
        Right wall = True
```

Maze Creation algorithms:

Recursive backtracking:

The recursive backtracking maze creation algorithm recursively follows the routine of being passed a cell, then marking the cell as visited and while the current cell has any unvisited adjacent cells a random adjacent cell is chosen and the wall between the new cell and the current cell is removed. This is repeated until the maze is complete.

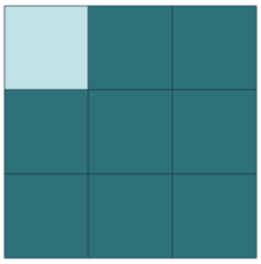
In the case that all the adjacent cells have been visited the algorithm backtracks until it reaches a cell that has an unvisited adjacent cell. Then the same process above is repeated until all the cells have been visited and the maze is complete.

Pseudo code:

```
Given a current cell as a parameter
Mark the current cell as visited
While the current cell has any unvisited neighbour cells
Choose one of the unvisited neighbours
Remove the wall between the current cell and the chosen cell
Invoke the routine recursively for a chosen cell
```

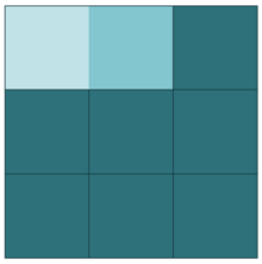
Example with 3 x 3 grid:

Select a random cell:



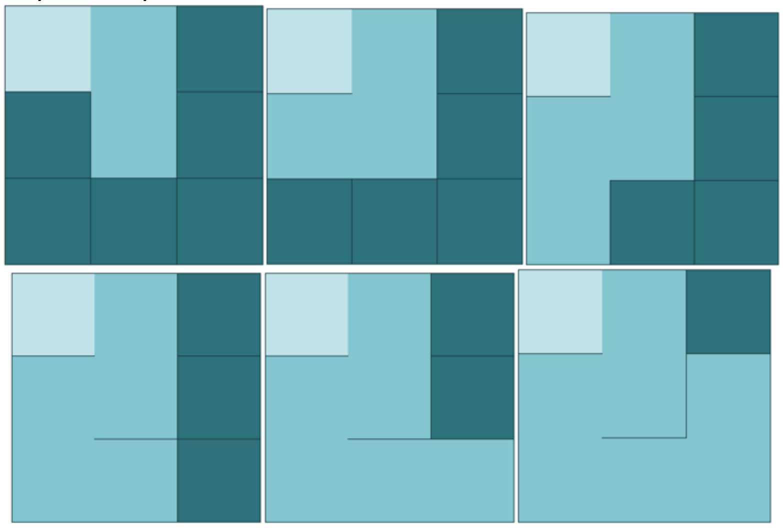
Add cell to the visited list

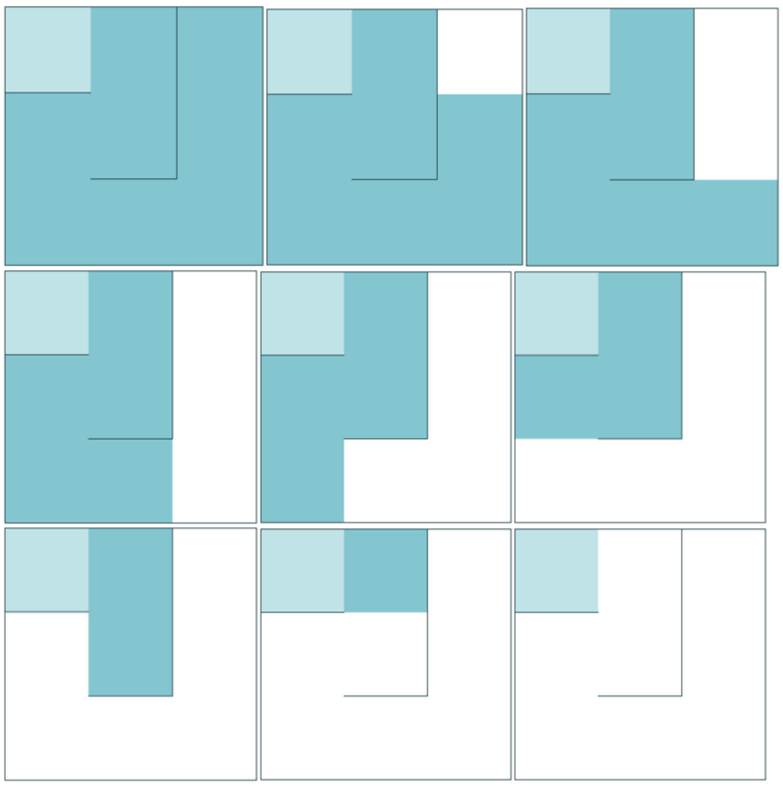
Select a random adjacent cell



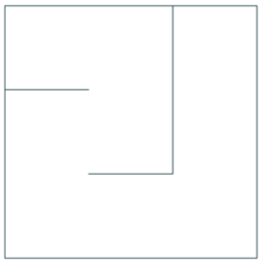
If the new cell not in the visited list add the new cell to the visited list remove the wall in-between the two cells, if the new cell is in the visited list select a new adjacent cell and repeat step 2

Repeat step 2 until the visited list is full





Final maze:



Modular testing

Test Number	Test	Evidence	Result
1	Can the recursive backtracking algorithm produce a simply connected maze?	Image 1.3 Page. 19	Successful
2	Can the recursive backtracking algorithm produce a 100x100 maze?	Image 1.1 Page. 18	Successful
3	Can the recursive backtracking algorithm produce a 2x2 maze?	Image 1.2 Page. 19	Successful
4	Can a maze created with recursive backtracking be animated?	Video 1	Successful
5	Can the recursive backtracking algorithm adjust the cell width depending on maze size?	Image 1.1 Image 1.3 Page. 18 & 19	Successful

6	Can the user see the start and end cell?	Image 1.3 Page. 19	Successful
7	Can the recursive backtracking algorithm create an easy maze?	Image 1.3 Page. 19	Successful
8	Can the recursive backtracking algorithm create a medium maze?	Image 1.4 Page. 19	Successful
9	Can the recursive backtracking algorithm create a hard maze?	Image 1.5 Page. 20	Successful

Image 1.1

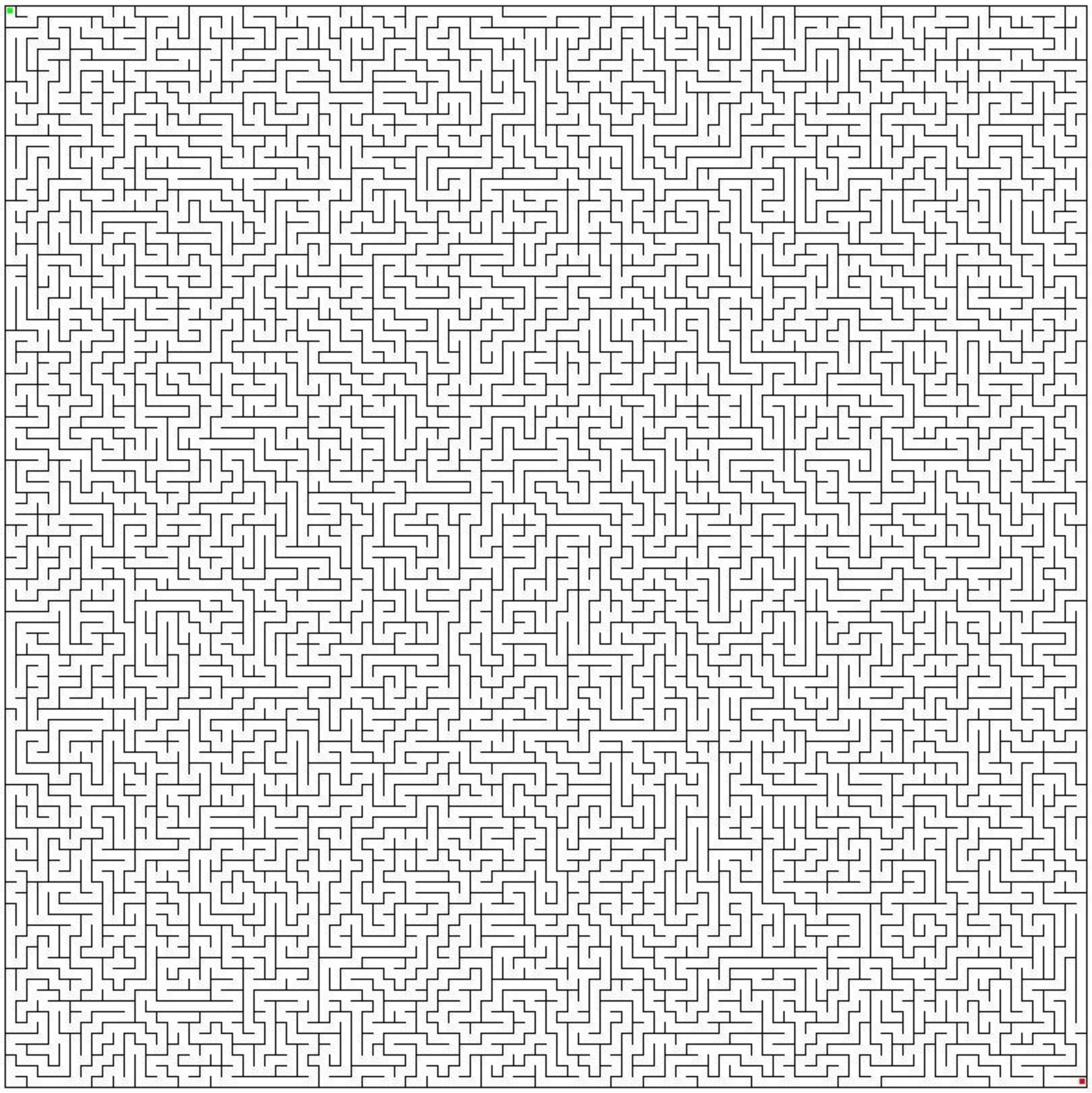


Image 1.2

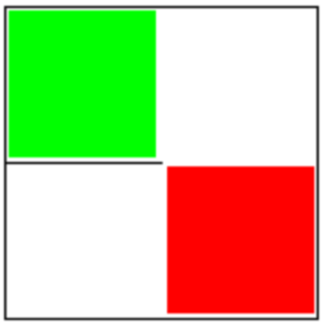


Image 1.3

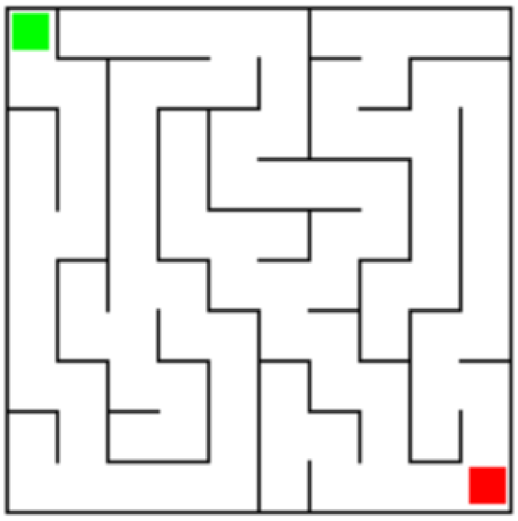


Image 1.4

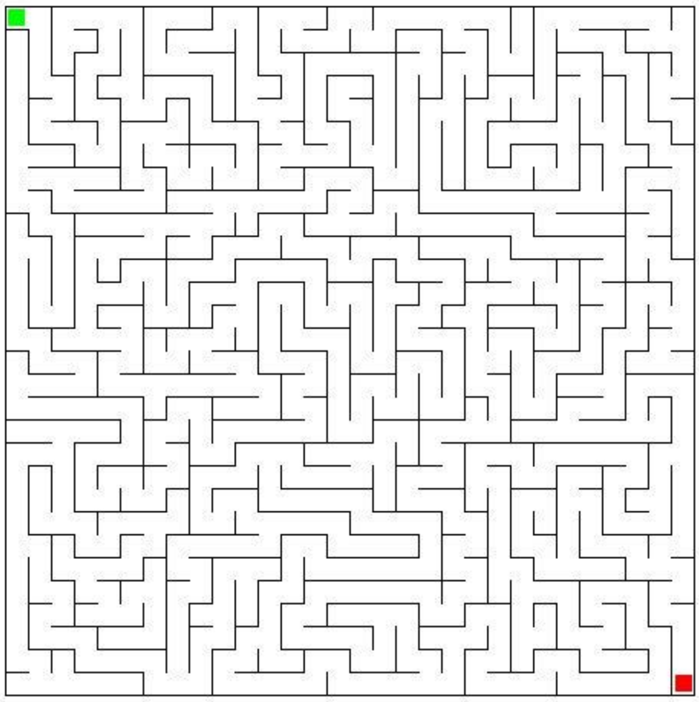
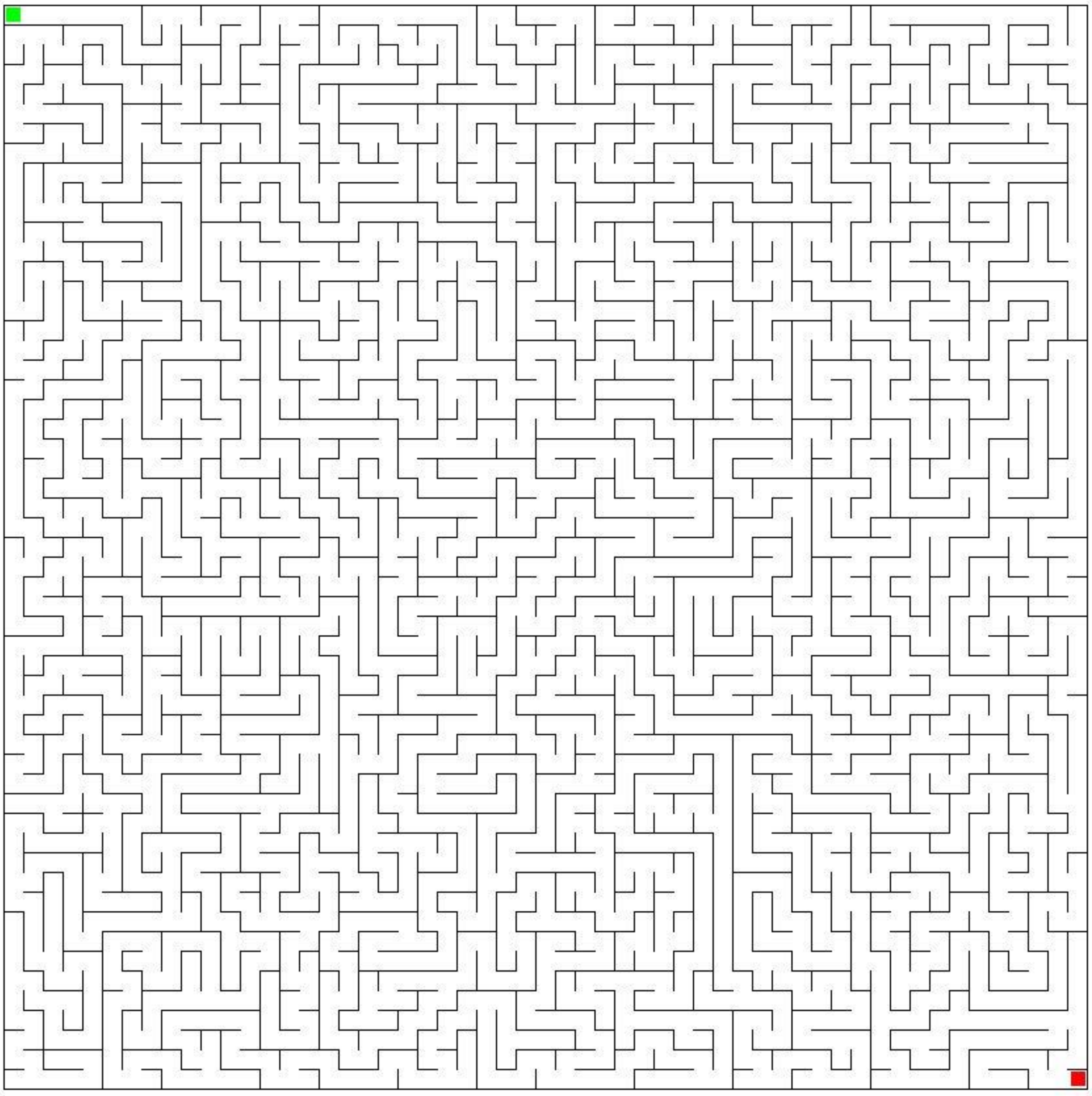


Image 1.5



Prim’s randomized algorithm:

Prim’s randomized algorithm stores a list of adjacent cells. Then a random adjacent cell is chosen, If the randomly chosen cell has multiple edges that connect to the existing maze, then one of the unselected edges will be selected at random. Else, a new random cell is chosen.

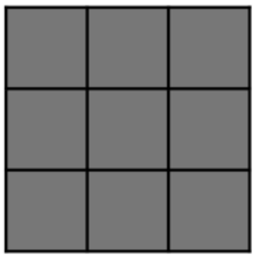
Pseudo code:

```
Start with a grid full of walls.
Pick a cell, mark it as part of the maze. Add the walls of the cell to the wall list.
While there are walls in the list:
```

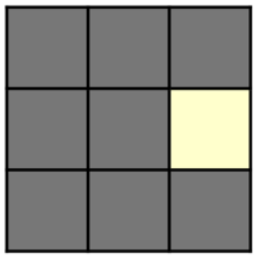
Pick a random wall from the list. If only one of the cells that the wall divides is visited, then:
Make the wall a passage and mark the unvisited cell as part of the maze.
Add the neighbouring walls of the cell to the wall list.
Remove the wall from the list

Example with 3 x 3 grid:

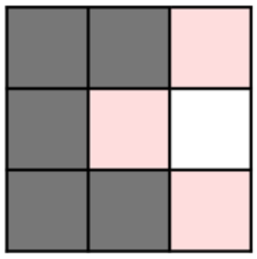
Start with an empty 3 x 3 grid:



1. Select a random cell:

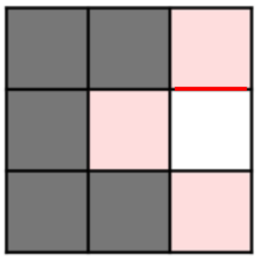


2. Add adjacent walls to the wall list:

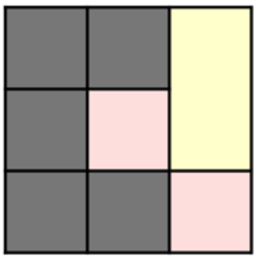


All the walls connecting the white and the red cells are added to the walls list.
The current cell is also added to the visited list.

3. Select a random wall from the walls list:

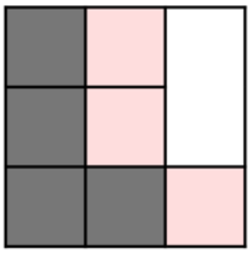


4. Check if both cells connected to selected wall are not in visited

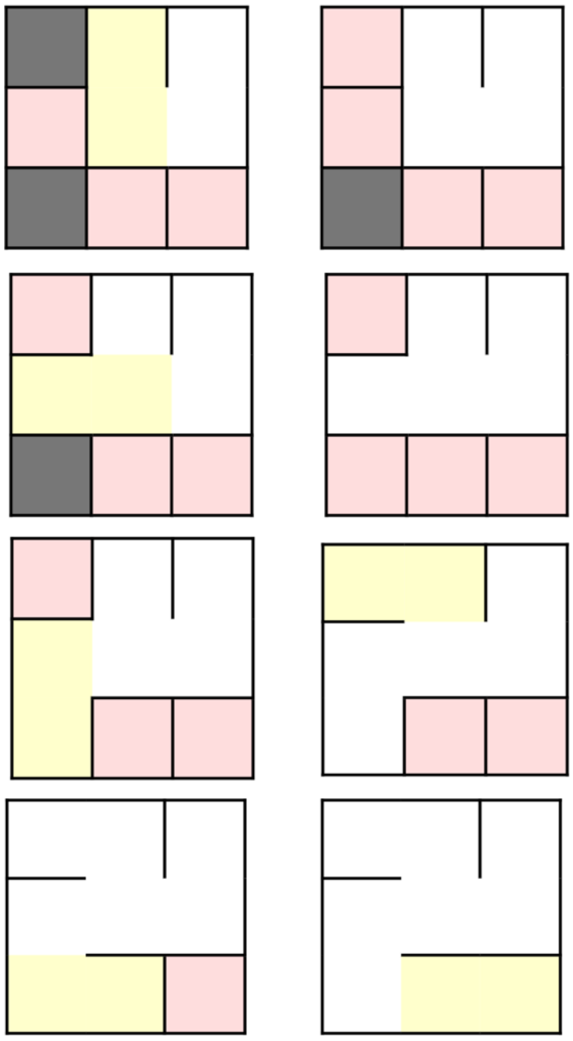


- If false, remove the wall from both cells and the walls list and add the new cell to the visited list
- If true, remove selected wall from walls list and select a new wall from the walls list, then repeat step 4.

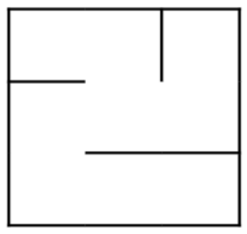
5. Add the walls of the new cell to the wall list:



6. Repeat step 3 to 5 until the walls list is empty:



Final maze:



Modular testing

Test Number	Test	Evidence	Result
1	Can Prim’s randomized algorithm produce a simply connected maze?	Image 2.3 Page. 26	Successful
2	Can Prim’s randomized algorithm produce a 100x100 maze?	Image 2.1 Page. 25	Successful
3	Can Prim’s randomized algorithm produce a 2x2 maze?	Image 2.2 Page. 25	Successful
4	Can a maze created with Prim’s be animated?	Video 2	Successful
5	Can Prim’s randomized algorithm adjust the cell width depending on maze size?	Image 2.1 Image 2.3 Page. 25 & 26	Successful
6	Can the user see the start and end cell?	Image 2.3 Page. 26	Successful
7	Can Prim’s randomized algorithm create an easy maze?	Image 2.3 Page. 26	Successful
8	Can Prim’s randomized algorithm create a medium maze?	Image 2.4 Page. 26	Successful
9	Can Prim’s randomized algorithm create a hard maze?	Image 2.5 Page. 27	Successful

Image 2.1

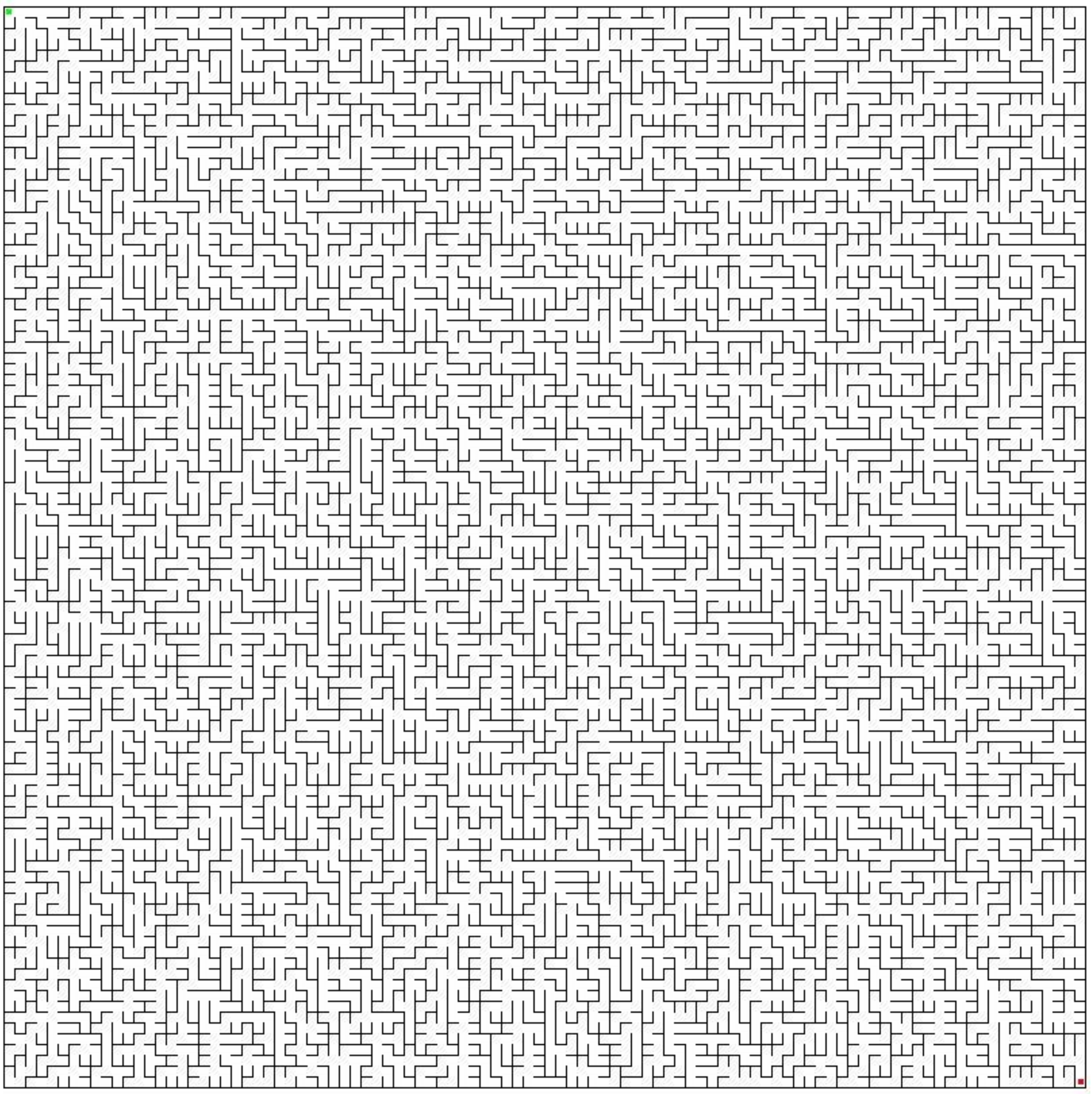


Image 2.2

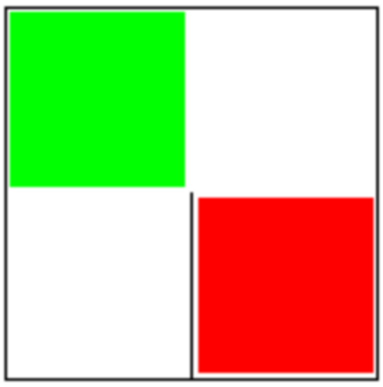


Image 2.3

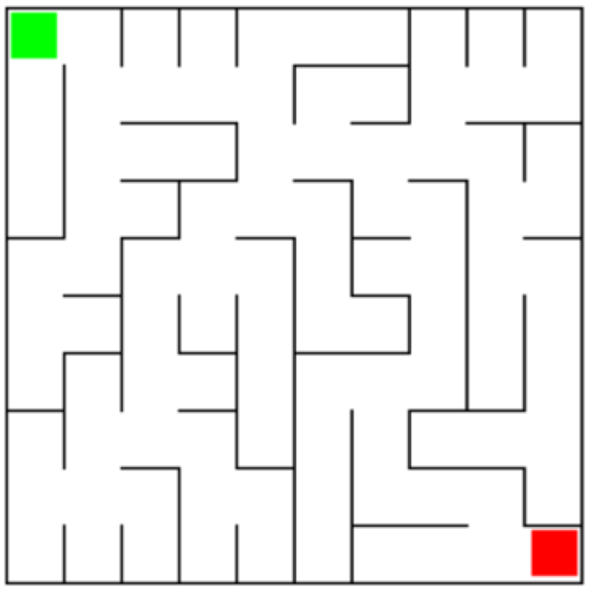


Image 2.4

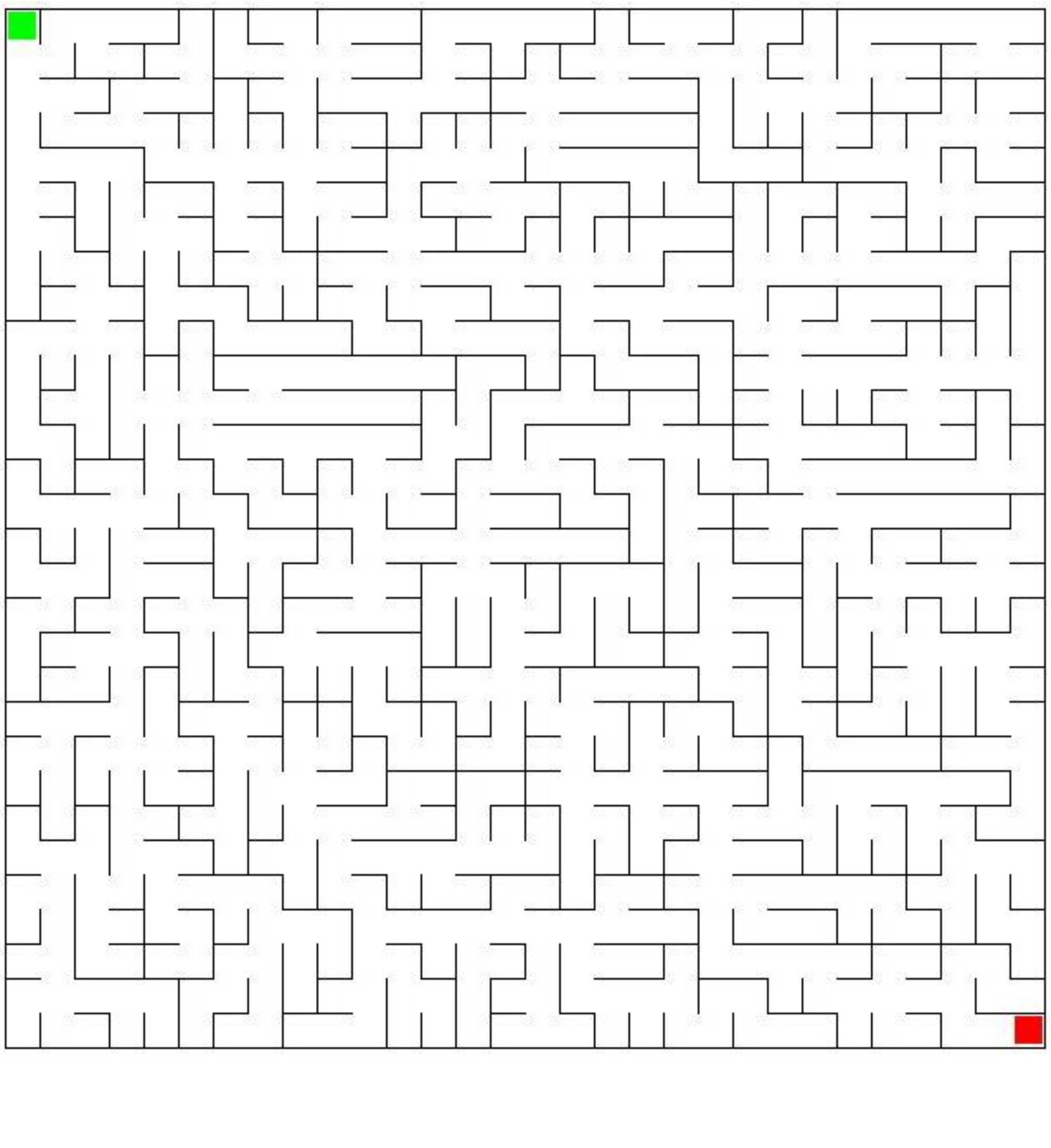
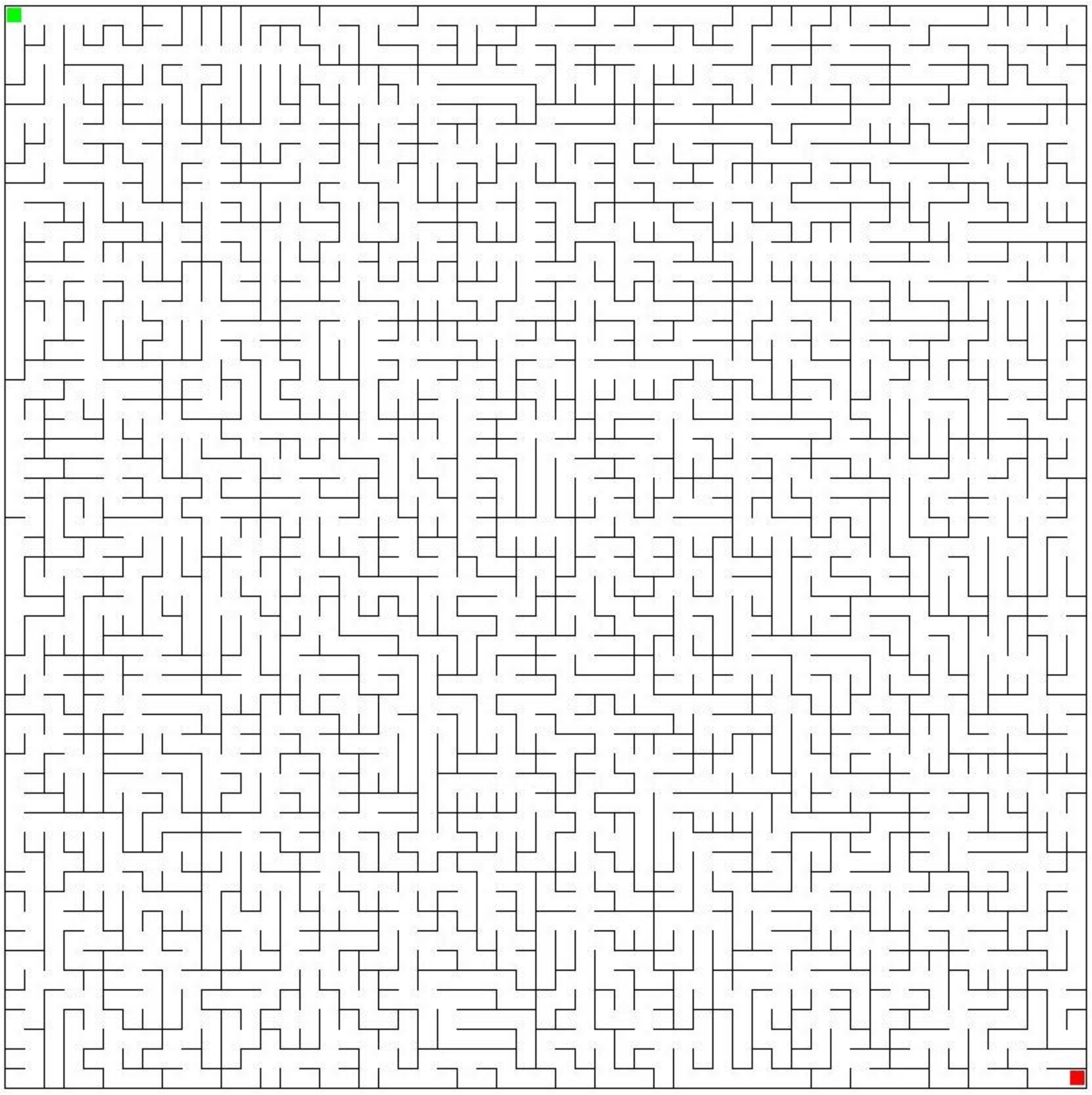


Image 2.5



Maze Solving algorithm

A* algorithm

A* is an informed search algorithm that considers the location of the goal while searching for it and hence it searches fewer nodes to reach to the goal compared to a breadth first search or a depth first search algorithm. It does this by assigning a cost to each cell and allows the algorithm to choose the path with the minimum overall cost.

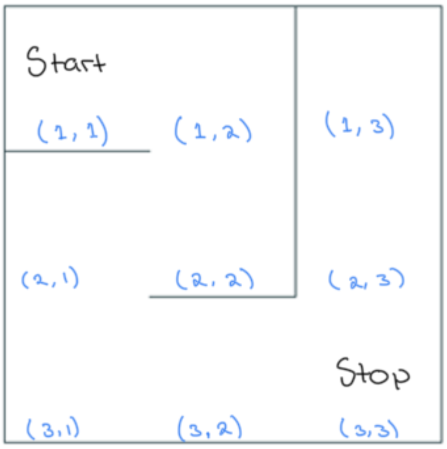
The cost of each cell is defined as the following function:

F(n) = g(n) + h(n)

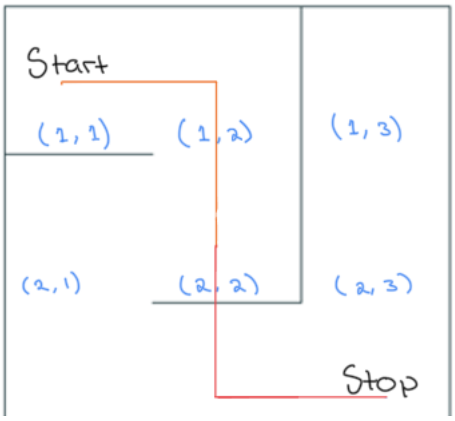
The cost function is defined of two parts; g(n) which represents the actual cost to reach cell n from the start cell and h(n) which represents the heuristics cost to reach the end cell from cell n.

Example with 3x3 grid:

Firstly, we start with an already created maze



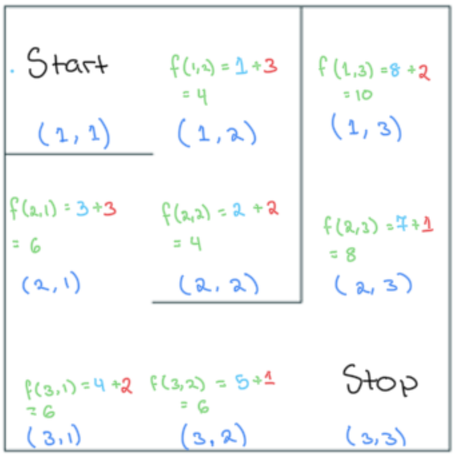
Let’s choose cell (2,2) to calculate the cost to reach it from the start



F (2,2) = g (2,2) + h (2,2)

g (2,2) = 2 (shown in orange)
This is the number of steps needed to be taken to reach cell (2,2) from the start cell
h (2,2) = 2 (shown in red)

This is the Manhattan distance an approximate of the number of steps needed to reach the end.
This is then repeated for all the cells in the grid



Once all the cost has been found for each cell in the grid, a path with the minimum cost can be found

Test Number	Test	Evidence	Result
1	Can A* solve a maze produced by Prim’s randomized algorithm?	Image 3.1 Page. 31	Successful
2	Can A* solve a maze produced by the recursive backtracking algorithm?	Image 3.3 Page. 31	Successful
3	Can A* solve an easy Prim’s Maze?	Image 3.2 Page. 30	Successful
4	Can A* solve a medium Prim’s Maze?	Image 3.5 Page. 32	Successful
5	Can A* solve a hard Prim’s Maze?	Image 3.6 Page. 32	Successful
6	Can A* solve an easy recursive backtracking, Maze?	Image 3.4 Page. 31	Successful
7	Can A* solve a medium recursive backtracking, Maze?	Image 3.7 Page. 33	Successful
8	Can A* solve a hard recursive backtracking, Maze?	Image 3.8 Page. 33	Successful
9	Can A* solving a Prim’s Maze be animated?	Video 3	Successful
10	Can A* solving a recursive backtracking Maze be animated?	Video 4	Successful

Image 3.1

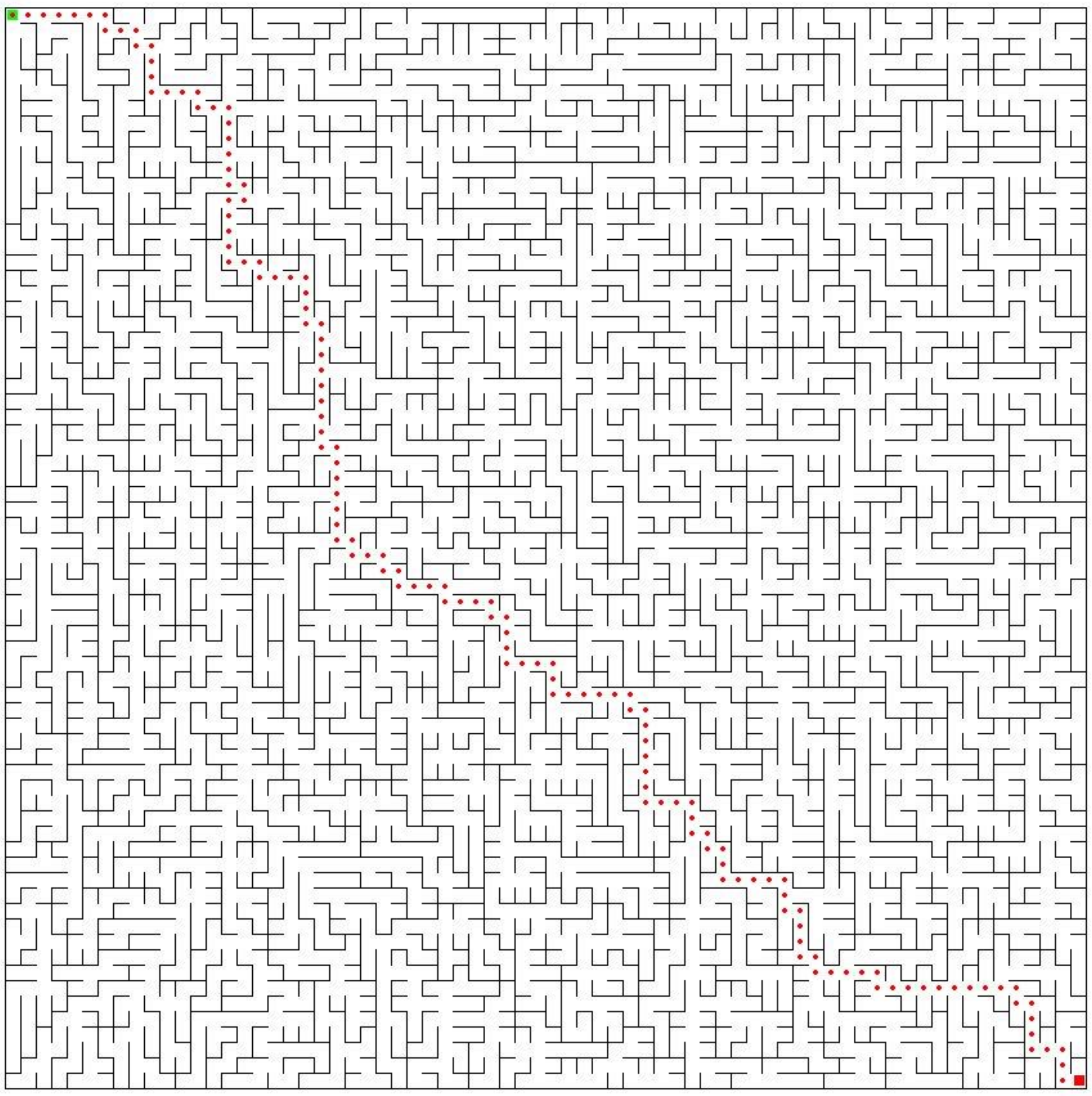


Image 3.2

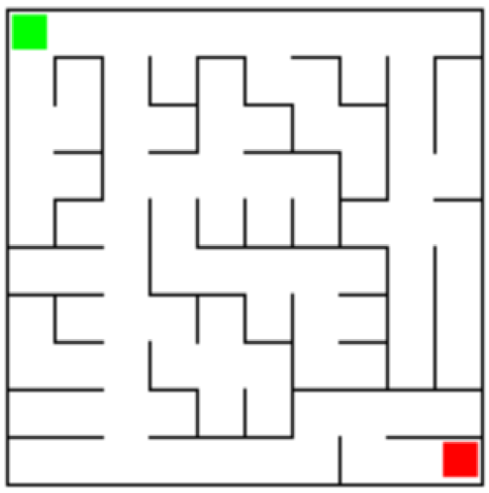


Image 3.3

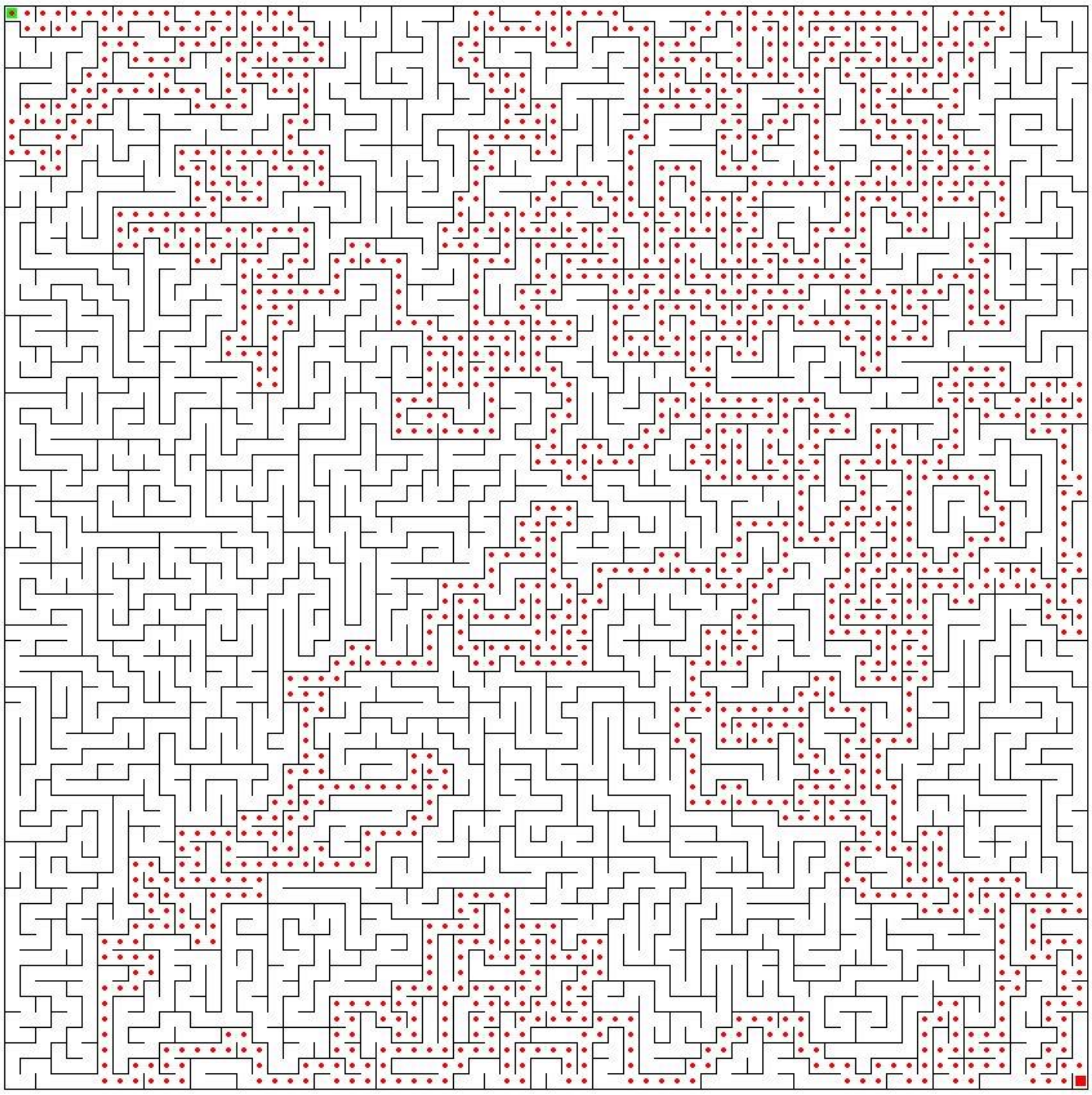


Image 3.4

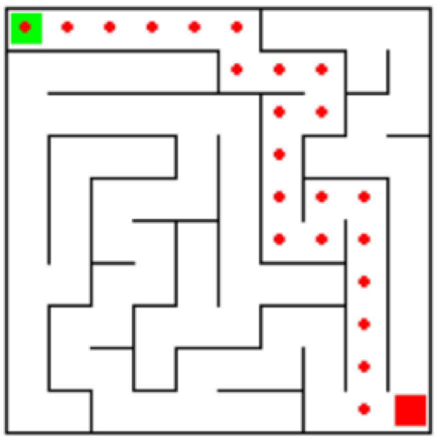


Image 3.5

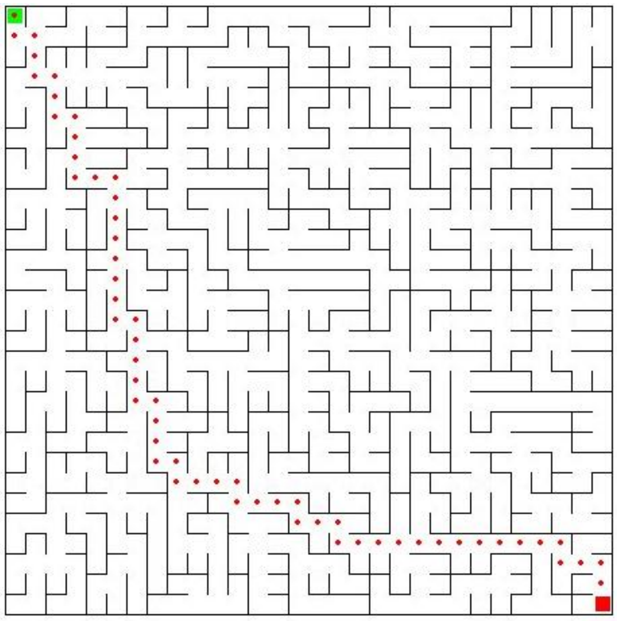


Image 3.6

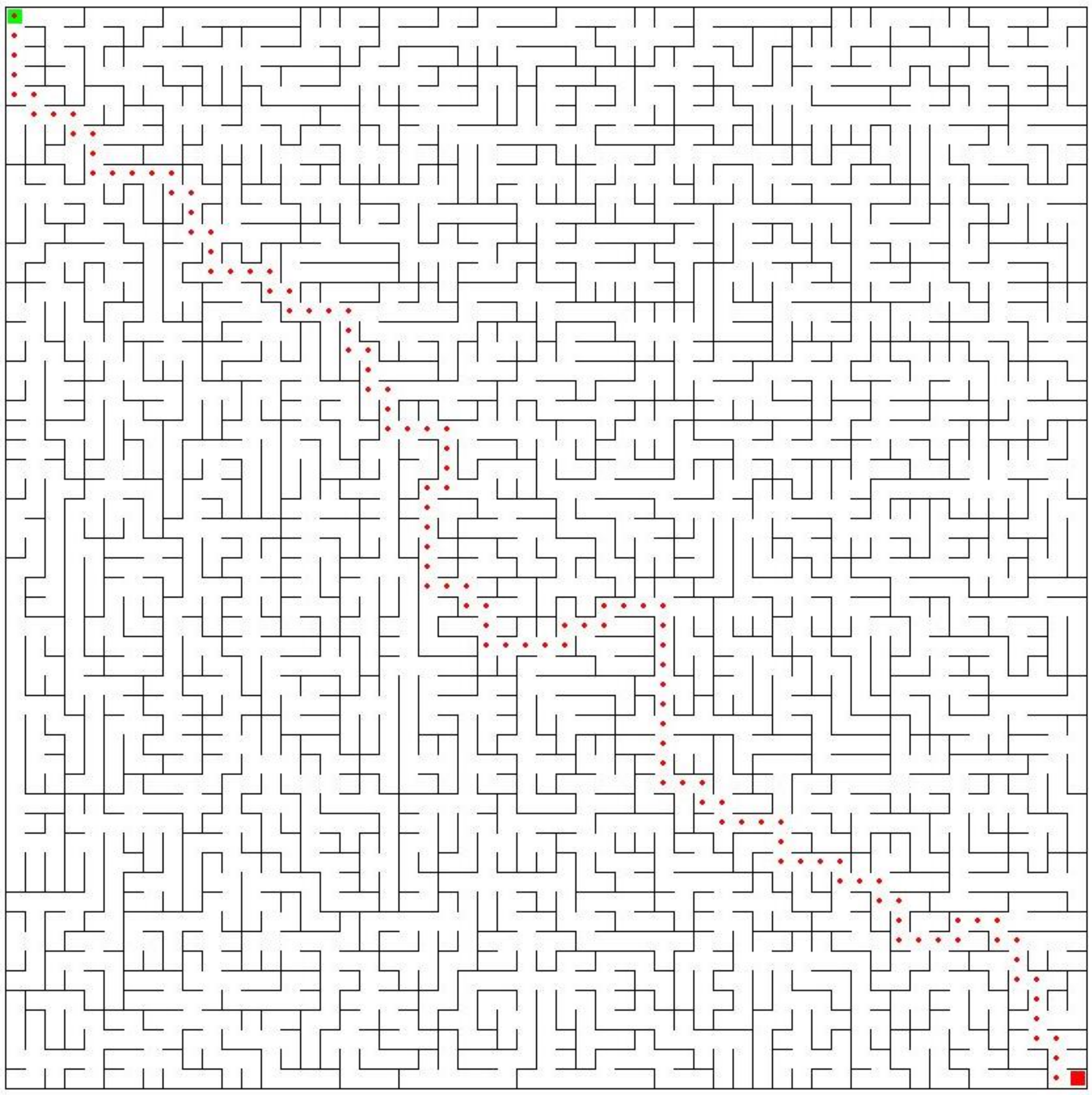


Image 3.7

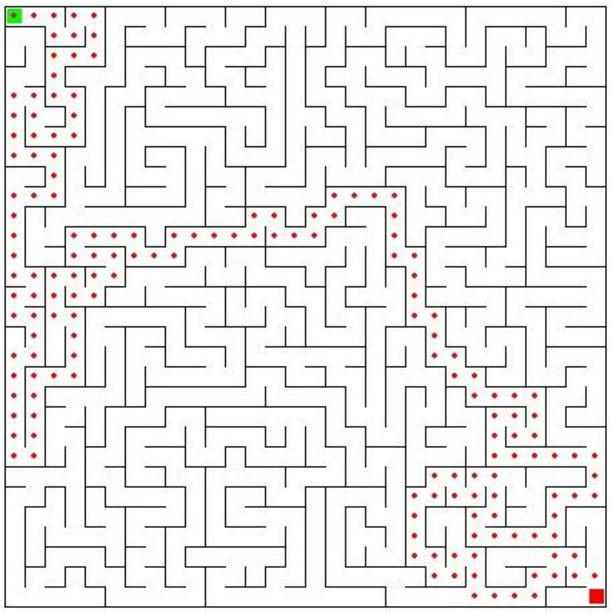
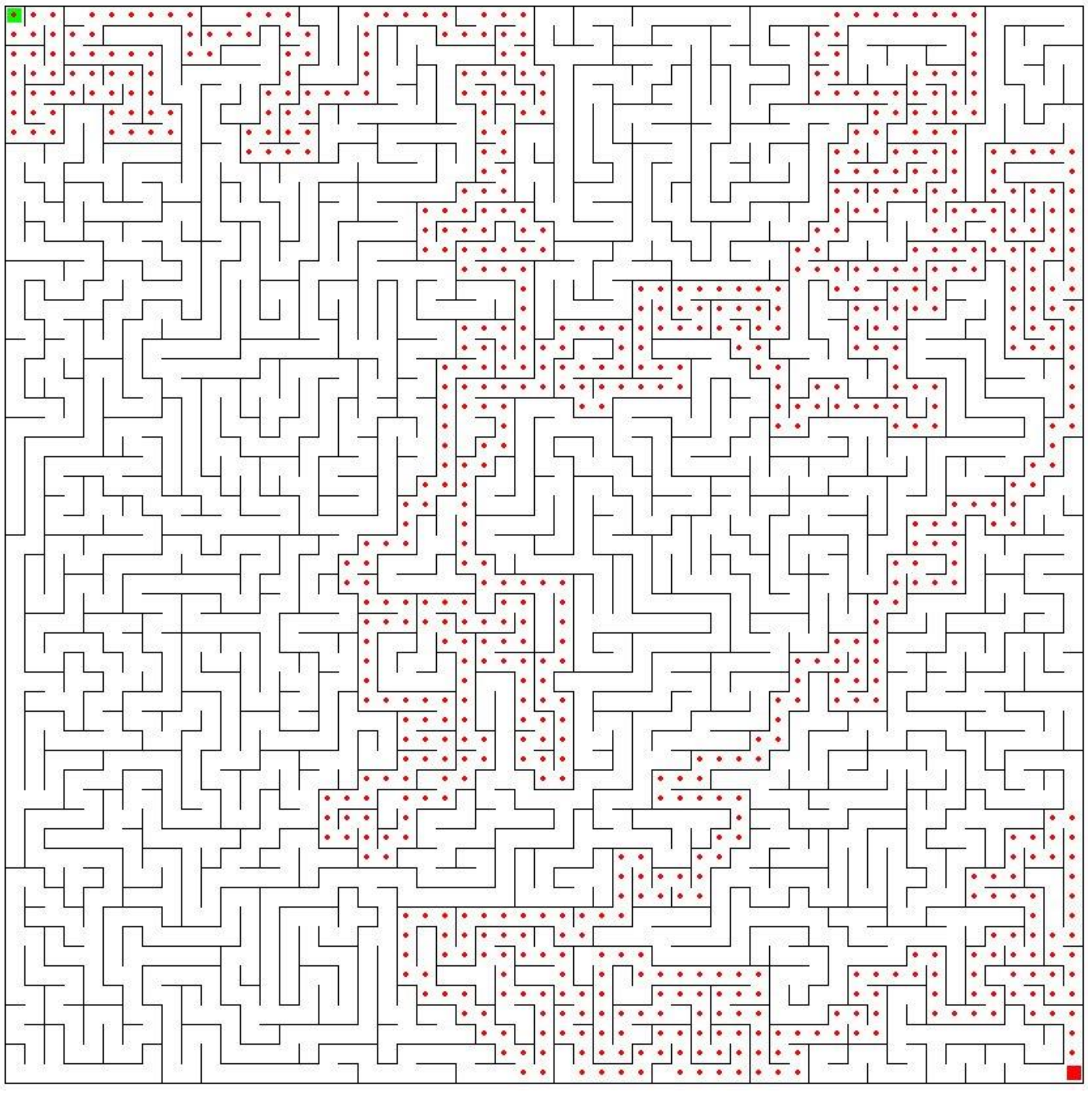


Image 3.8



Animation of Maze

Pygame will be used to display and animate the maze. Pygame is a set of python modules designed for writing video games. Pygame provides functionality that allows for the creation of fully featured games and multimedia programs in python. It includes python libraries for graphics and sound.

To animate the maze creation algorithms Pygame will firstly draw a grid on the screen, then it will draw the maze on the grid by following a list of walls removed in order from the maze creation algorithm. The Pygame class goes through the list provided by the maze creation algorithms and draws over the walls in the same colour as to background, this produces an animation effect. After the maze animation is complete Pygame redraws the maze without the animation effect to produce a crisp image.

Similarly, to animate the maze solving algorithm Pygame will first start with the maze already drawn on the screen. Then Pygame will go through a list provided by the maze solving algorithm and draw a point on the route the algorithm follows while solving the maze.

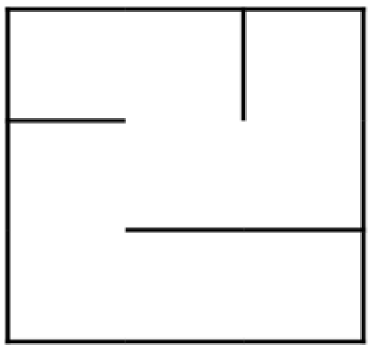
Test Number	Test	Evidence	Result
1	Can a maze created with Prim’s be animated?	Video 2	Successful
2	Can a maze created with recursive backtracking be animated?	Video 1	Successful
3	Can A* solving a Prim’s Maze be animated?	Video 3	Successful
4	Can A* solving a recursive backtracking Maze be animated?	Video 4	Successful

Exporting the Maze

The maze can either be exported as an image or a text file. To export both the maze and solution as an image. Pygame is used to capture the Pygame window, and it is saved in a folder on the user’s device.

The maze can also be exported as a text file which is saved on the user’s device an example for a 3x3 maze is shown below:

Text file:



```
(0, 0), [True, True, True, False]
(0, 1), [True, False, False, True]
(0, 2), [True, False, True, True]
(1, 0), [True, False, True, False]
(1, 1), [False, True, False, False]
(1, 2), [False, True, False, True]
(2, 0), [False, True, True, False]
(2, 1), [True, True, False, False]
(2, 2), [True, True, False, True]
```

Each line of the text file outputs the coordinate of the cell and the walls present. The walls are formatted in the order **[top, bottom, left, right].**

Test Number	Test	Evidence	Result
1	Can a text file with the representation of the maze be produced?	Video 5	Successful

Testing

Modular testing during design

Recursive backtracking:

Test Number	Test	Evidence	Result
1	Can the recursive backtracking algorithm produce a simply connected maze?	Image 1.3 Page. 19	Successful
2	Can the recursive backtracking algorithm produce a 100x100 maze?	Image 1.1 Page. 18	Successful
3	Can the recursive backtracking algorithm produce a 2x2 maze?	Image 1.2 Page. 19	Successful
4	Can a maze created with recursive backtracking be animated?	Video 1	Successful
5	Can the recursive backtracking algorithm adjust the cell width depending on maze size?	Image 1.1 Image 1.3 Page. 18 & 19	Successful
6	Can the user see the start and end cell?	Image 1.3 Page. 19	Successful
7	Can the recursive backtracking algorithm create an easy maze?	Image 1.3 Page. 19	Successful

8	Can the recursive backtracking algorithm create a medium maze?	Image 1.4 Page. 19	Successful
9	Can the recursive backtracking algorithm create a hard maze?	Image 1.5 Page. 20	Successful

Prim's randomized algorithm:

Test Number	Test	Evidence	Result
1	Can Prim's randomized algorithm produce a simply connected maze?	Image 2.3 Page. 26	Successful
2	Can Prim's randomized algorithm produce a 100x100 maze?	Image 2.1 Page. 25	Successful
3	Can Prim's randomized algorithm produce a 2x2 maze?	Image 2.2 Page. 25	Successful
4	Can a maze created with Prim's be animated?	Video 2	Successful
5	Can Prim's randomized algorithm adjust the cell width depending on maze size?	Image 2.1 Image 2.3 Page. 25 & 26	Successful
6	Can the user see the start and end cell?	Image 2.3 Page. 26	Successful
7	Can Prim's randomized algorithm create an easy maze?	Image 2.3 Page. 26	Successful
8	Can Prim's randomized algorithm create a medium maze?	Image 2.4 Page. 26	Successful
9	Can Prim's randomized algorithm create a hard maze?	Image 2.5 Page. 27	Successful

A* algorithm:

Test Number	Test	Evidence	Result
1	Can A* solve a maze produced by Prim's randomized algorithm?	Image 3.1 Page. 31	Successful
2	Can A* solve a maze produced by the recursive backtracking algorithm?	Image 3.3 Page. 31	Successful
3	Can A* solve an easy Prim's Maze?	Image 3.2 Page. 30	Successful
4	Can A* solve a medium Prim's Maze?	Image 3.5 Page. 32	Successful
5	Can A* solve a hard Prim's Maze?	Image 3.6 Page. 32	Successful
6	Can A* solve an easy recursive backtracking, Maze?	Image 3.4 Page. 31	Successful
7	Can A* solve a medium recursive backtracking, Maze?	Image 3.7 Page. 33	Successful

8	Can A* solve a hard recursive backtracking, Maze?	Image 3.8 Page. 33	Successful
9	Can A* solving a Prim's Maze be animated?	Video 3	Successful
10	Can A* solving a recursive backtracking Maze be animated?	Video 4	Successful

Animation of Maze:

Test Number	Test	Evidence	Result
1	Can a maze created with Prim's be animated?	Video 2	Successful
2	Can a maze created with recursive backtracking be animated?	Video 1	Successful
3	Can A* solving a Prim's Maze be animated?	Video 3	Successful
4	Can A* solving a recursive backtracking Maze be animated?	Video 4	Successful

Exporting the Maze:

Test Number	Test	Evidence	Result
1	Can a text file with the representation of the maze be produced?	Video 5	Successful

Testing of whole system

Video link:

Test Number	Test	Evidence	Result
1	Can the user select the maze creation algorithm to use? • Is user input validated?	00:10 – 00:18 in video	Successful
2	Can the user select a difficulty level? • Is user unput validated?	00:18 – 00:32 in video	Successful
3	Can the program print the maze onto the console?	00:32 – 00:49 in video	Successful
4	Can the program animate the maze?	00:48 – 01:14 in video	Successful
	Can the program solve the maze and is the solution animated?	01:14 – 01:38in video	Successful

5	Can the program save the maze as an image?	04:37 – 05:45 in video	Successful
7	Can the maze be exported as a text file?	06:30 – 06:54 in video	Successful
8	Can the user create a new maze after the completion of one?	00:00 – 04:03 in video	Successful

Evaluation

General evaluation of solution:

The program was able to complete most of the objectives set out at the beginning of the project. It allows the user to alter the program to their specifications by allowing them to make certain choices such as choosing the size of the maze and choosing the maze creation algorithm. It also solves the maze and animates the maze to give an interactive experience to the user. Lastly, the program also exports the maze as a text file and image.

On the other hand, the program has only a console interface and not a graphical interface which can make it difficult for a user to use. Also, the program only produces simply connected mazes which limits the number and complexity of the mazes that can be produced. Therefore, by adding a graphical user interface and by altering the program to produce multi-connected mazes, the overall user experience can be improved.

Evaluation against objectives:

1) The user should have a user interface where they can:
   a) Select what maze generation algorithm they want to use
      i) The user should be able to use the recursive backtracking algorithm
      ii) The user should be able to use Prim's randomized algorithm

   Evaluation:

   This was achieved, to make it easier for the user a graphical user interface should be implemented.

   b) Select if they want the maze to be solved
      i) The colour of the solution should be a complementary colour to the grid colour so the solution of the maze can be easily seen

   Evaluation:

   This was achieved, the user was able to solve the maze and the solution was very clear and easy to see.

   c) Select the grid size
      i) The size of each cell should change depending on the grid size

ii) The user should be limited to how large they can make the maze
(1) Ensure each individual cell and wall is still visible by the user

Evaluation:

This was achieved, the user was able to input the size of the maze, but it was limited to values between 2 and 100. This ensured that each individual cell can be clearly seen.

d) The user should be able to select the difficulty of the maze if grid size is not given

Evaluation:

This was achieved, the user was able to choose between creating an easy, medium, and hard maze.

e) Select if the maze should be animated

Evaluation:

This was achieved by using Pygame.

f) Export the maze as a picture or as text file with an abstract memory representation of the maze

Evaluation:

This was achieved, to improve this, other options could be added such as allowing the user to export the maze as a csv or pdf.

2) The starting cell and ending cell should be set and shown to the user at the beginning of program execution

a) The start cell should be in the top left hand conner of the maze

Evaluation:

This was achieved by not allowing the user to change the position of the start cell and by using Pygame to colour in the start cell.

b) The ending cell should be in the bottom right hand conner of the maze

Evaluation:

This was achieved by not allowing the user to change the position of the end cell and by using Pygame to colour in the end cell.

3) Be able to store a maze in a suitable abstract data structure making use of OOP principles:

a) The user should be able to export this memory representation of the maze as a text file

Evaluation:

This was achieved by using the cell class to write the coordinates and walls of the cell as a line in a text file.

4) Be able to implement a recursive backtracking maze generation algorithm to create a simply connected maze
    a) The maze generation should be able to be animated on the GUI by the user

    Evaluation:

    This was achieved by using Pygame.

5) Be able to implement Prim's randomized algorithm for maze creation of a simply connect maze
    a) The maze generation should be able to be animated on the GUI by the user

    Evaluation:

    This was achieved by using Pygame.

6) Be able to implement the A* shortest path algorithm to solve the maze
    a) The maze solving should be able to be animated on the GUI by the user

    Evaluation:

    This was achieved by using Pygame.

User feedback:

Ilan's evaluation

The UI is really good. It's a nice looking one as well. Users might get confused on what the blue vs red dots mean, so maybe a key could help. I'm guessing blue is for the cells that were visited. All prompts are working well. The maze animation is really cool. The image saving feature is really cool as well. It might be better to scale the size of the grid with the size of the window, for example if I make my window smaller, the grid is cut in half. Might be useful to give users choice to choose file name when saving, as saving with the same filename will overwrite the previous save. Perhaps allow them to choose the path as well.

Reflection upon user feedback:

Overall, Ilan seemed to like the program, but he mentioned certain aspects that could be improved such as the UI and file name. To improve the UI as Ilan stated a key could be added to allow the user easily to identify the colours used in the program, also the user should have more control over the customization of the Pygame window. Also, the file saving system can be improved to allow the user to choose the name of the file, so they can save the data for multiple mazes.

Extensions / Improvements:

Although the program achieved all the objectives set out, to improve the project a graphical user interface could be added, and the program could be altered to produce multi-connected mazes. Producing multi-connected mazes could be achieved by altering both Prim’s and the recursive backtracking algorithm to remove walls randomly from the maze to create multiple paths to the end cell. The GUI could be added by using Pygame to create the layout of the GUI and linking it to the program.

Appendix:

Copy of program:

```
import pygame
import sys
import random
import time
from pygame.locals import *

# Constants
screen_size = width, height = 870, 860
black = (0, 0, 0)
grey = (160, 160, 160)
blue = (255, 192, 203)
FPS = 30


def main():
   consoleUI()


def consoleUI():
   print("Hello, this is a maze creation and solving program")
   createq = input("Would you like to create a maze (Y/N)---").upper()
   while createq not in ["Y", "N"]:
      print("Invalid input")
      createq = input("Please enter Y or N ---").upper()
   if createq == "N":
      sys.exit()
   print("What type of maze would you like to create?")
   maze_type = input("Type P for Prim's and R for RBT ---").upper()
   while maze_type not in ["P", "R"]:
      print("Invalid input")
      maze_type = input("Please enter P or R ---").upper()
```

```
    global size
    global cell_width
    use_diff = input("Would you like to select a difficulty level (Y/N)---").upper()
    while use_diff not in ["Y", "N"]:
        print("Invalid input")
        use_diff = input("Please enter Y or N ---").upper()
    if use_diff == "Y":
        diff = input("(E)asy, (M)edium, (H)ard ---").upper()
        while diff not in ["E", "M", "H"]:
            print("Invalid input")
            diff = input("Please enter E, M or H ---").upper()
        if diff == "E":
            maze_size = 10
        elif diff == "M":
            maze_size = 30
        else:
            maze_size = 55
    else:
        maze_size = int(input("Enter maze size ---"))
        while maze_size < 2 or maze_size > 100:
            print("Invalid input")
            maze_size = int(input("Please enter a whole number between 2 and
100 ---"))
    size = maze_size
    cell_width = (width - 5) // size
    if cell_width > 15:
        cell_width = 15
    m = Maze()
    if maze_type == "P":
        m.create_maze_prim()
        cprint = input("Would you like to print maze to console? (Y/N) ---
").upper()
        while cprint not in ["Y", "N"]:
            print("Invalid input")
            cprint = input("Please enter Y or N ---").upper()
        if cprint == "Y":
            m.print_maze("prim")
        animate = input("Would you like to animate maze? (Y/N) ---").upper()
        while animate not in ["Y", "N"]:
            print("Invalid input")
            animate = input("Please enter Y or N ---").upper()
        if animate == "Y":
            time.sleep(3)
            Pygame(m.get_grid_prim(), m.get_animation_prim()).animate_grid()
        else:
            Pygame(m.get_grid_prim(), m.get_animation_prim()).draw_maze(blue)
            pygame.event.get()
            pygame.display.update()
        pygame.image.save(window, "Maze.jpg")
        solve = input("Would you like to solve maze? (Y/N) ---").upper()
        while solve not in ["Y", "N"]:
            print("Invalid input")
            solve = input("Please enter Y or N ---").upper()
        if solve == "Y":
            x = Pathfinding(m.get_grid_prim())
            x.a_star()
            Pygame(m.get_grid_prim(), m.get_animation_prim()).draw_maze(blue)
            pygame.display.update()
            animate = input("Would you like to animate solution? (Y/N) ---
").upper()
            while animate not in ["Y", "N"]:
                print("Invalid input")
                animate = input("Please enter Y or N ---").upper()
            if animate == "Y":
                time.sleep(3)
                Pygame(m.get_grid_prim(), x.get_visited()).animate_solution()
            Pygame(m.get_grid_prim(), x.get_path()).draw_solution()
            pygame.display.update()
    if maze_type == "R":
        m.create_maze_rbt()
        cprint = input("Would you like to print maze to console? (Y/N) ---
").upper()
        while cprint not in ["Y", "N"]:
            print("Invalid input")
            cprint = input("Please enter Y or N ---").upper()
        if cprint == "Y":
            m.print_maze("rbt")
        animate = input("Would you like to animate maze? (Y/N) ---").upper()
        while animate not in ["Y", "N"]:
            print("Invalid input")
            animate = input("Please enter Y or N ---").upper()
        time.sleep(3)
        if animate == "Y":
            Pygame(m.get_grid_rbt(), m.get_animation_rtb()).animate_grid()
        else:
            Pygame(m.get_grid_rbt(), m.get_animation_rtb()).draw_maze(blue)
            pygame.event.get()
            pygame.display.update()
        pygame.image.save(window, "Maze.jpg")
        solve = input("Would you like to solve maze? (Y/N) ---").upper()
        while solve not in ["Y", "N"]:
            print("Invalid input")
            solve = input("Please enter Y or N ---").upper()
        if solve == "Y":
            x = Pathfinding(m.get_grid_rbt())
            x.a_star()
            Pygame(m.get_grid_rbt(), m.get_animation_rtb()).draw_maze(blue)
            pygame.display.update()
            animate = input("Would you like to animate solution? (Y/N) ---
").upper()
            while animate not in ["Y", "N"]:
                print("Invalid input")
                animate = input("Please enter Y or N ---").upper()
            if animate == "Y":
                time.sleep(3)
                Pygame(m.get_grid_rbt(), x.get_visited()).animate_solution()
            Pygame(m.get_grid_rbt(), x.get_path()).draw_solution()
            pygame.display.update()
    save = input("Would you like save maze as image? (Y/N) ---").upper()
    while save not in ["Y", "N"]:
        print("Invalid input")
        save = input("Please enter Y or N ---").upper()
    if save == "Y":
        pygame.image.save(window, "Maze.jpg")
    export = input("Would you like to export maze? (Y/N) ---").upper()
    while export not in ["Y", "N"]:
        print("Invalid input")
        export = input("Please enter Y or N ---").upper()
    if export == "Y":
        if maze_type == "R":
            exportMaze(m.get_grid_rbt())
        else:
            exportMaze(m.get_grid_prim())
    s_maze = input("Would you like to create another maze? (Y/N) ---").upper()
    while s_maze not in ["Y", "N"]:
        print("Invalid input")
        s_maze = input("Please enter Y or N ---").upper()
    if s_maze == "Y":
        window.fill(black)
        consoleUI()
    sys.exit()

def exportMaze(maze):
    f = open("maze.txt", "w")
    for row in range(len(maze)):
        for column in range(len(maze)):
            cell = maze[row][column]
            line = f"({cell.x},{cell.y}),{cell.get_walls()}\n"
            f.write(line)
    f.close()



class Cell:  # Stores certain attributes for each cell in the maze
    def __init__(self, x, y):
        # Stores the coordinates of the cell
        self.x = x
        self.y = y
        # Initially sets all the walls of the cell to True
        self._up = True
        self._down = True
        self._left = True
        self._right = True

    def get_walls(self):  # Returns the list of walls a cell has
        return [self._up, self._down, self._left, self._right]

    def set_wall(self, wall):  # Removes a certain wall from a cell
        if wall == "up":
            self._up = False
        elif wall == "down":
            self._down = False
        elif wall == "left":
            self._left = False
        elif wall == "right":
            self._right = False
        if wall == 0:
            self._up = False
        elif wall == 1:
            self._down = False
        elif wall == 2:
            self._left = False
        elif wall == 3:
            self._right = False


class Maze:
    def __init__(self):
        self.x = 0
        self.y = 0
        self.size = size
        self.grid = []
        self.animation_rtb = []
        self.animation_prim = []
        self.build_grid()
        self.grid_rbt = self.grid
        self.grid_prim = self.grid

    def print_maze(self, algorithm=None):
        vstring = "+"
        hstring = ""
        string = ""
        if algorithm == "rbt":
            grid = self.grid_rbt
        elif algorithm == "prim":
            grid = self.grid_prim
        else:
            grid = self.grid
        for row in range(self.size):
            for column in range(self.size):
                walls = grid[row][column].get_walls()
                if walls[2]:
                    hstring += "|  "
                else:
                    hstring += "   "
                if walls[0]:
                    vstring += "--+"
                else:
                    vstring += "  +"
            string += vstring + "\n" + hstring + "|\n"
            hstring = ""
            vstring = "+"
        print(string + "+--" * self.size + "+")

    def build_grid(self):
        for i in range(self.size):
            self.grid.append([])
            for j in range(self.size):
                cell = Cell(i, j)
                self.grid[i].append(cell)

    def create_maze_rbt(self):
        stack = []
        visited = []
        coords = [(x, y) for x in range(self.size) for y in range(self.size)]
        x = 0
        y = 0
        stack.append((x, y))
        visited.append((x, y))
        while len(stack) > 0:
            path = []
            if (x + 1, y) not in visited and (x + 1, y) in coords:
                path.append("right")

            if (x - 1, y) not in visited and (x - 1, y) in coords:
                path.append("left")

            if (x, y + 1) not in visited and (x, y + 1) in coords:
                path.append("down")
```

```python
            if (x, y - 1) not in visited and (x, y - 1) in coords:
                path.append("up")

            if len(path) > 0:
                current_path = random.choice(path)
                self.animation_rtb.append((x, y, current_path))
                if current_path == "right":
                    self.grid_rbt[y][x].set_wall(current_path)
                    x += 1
                    self.grid_rbt[y][x].set_wall("left")
                    visited.append((x, y))
                    stack.append((x, y))

                elif current_path == "left":
                    self.grid_rbt[y][x].set_wall(current_path)
                    x -= 1
                    self.grid_rbt[y][x].set_wall("right")
                    visited.append((x, y))
                    stack.append((x, y))

                elif current_path == "up":
                    self.grid_rbt[y][x].set_wall(current_path)
                    y -= 1
                    self.grid_rbt[y][x].set_wall("down")
                    visited.append((x, y))
                    stack.append((x, y))

                elif current_path == "down":
                    self.grid_rbt[y][x].set_wall(current_path)
                    y += 1
                    self.grid_rbt[y][x].set_wall("up")
                    visited.append((x, y))
                    stack.append((x, y))

            else:
                x, y = stack.pop()

    def get_grid_rbt(self):
        return self.grid_rbt
```

```
def create_maze_prim(self):
    coords = [(x, y) for x in range(self.size) for y in range(self.size)]
    walls = []
    visited = []
    cell = self.grid[0][0]
    x, y = cell.x, cell.y
    walls = walls + self.add_walls(x, y)
    visited.append((x, y))
    while len(walls) > 0:
        path = random.choice(walls)
        walls.pop(walls.index(path))
        visitNum = 0
        x, y = path[0], path[1]
        if path[-1] == "up":
            if (x, y - 1) not in visited and (x, y - 1) in coords:
                self.animation_prim.append(path)
                self.grid_prim[y][x].set_wall(path[-1])
                y -= 1
                self.grid_prim[y][x].set_wall("down")
                visited.append((x, y))
                walls = walls + self.add_walls(x, y)
                visitNum += 1
        elif path[-1] == "down":
            if (x, y + 1) not in visited and (x, y + 1) in coords:
                self.animation_prim.append(path)
                self.grid_prim[y][x].set_wall(path[-1])
                y += 1
                self.grid_prim[y][x].set_wall("up")
                visited.append((x, y))
                walls = walls + self.add_walls(x, y)
                visitNum += 1
        elif path[-1] == "left":
            if (x - 1, y) not in visited and (x - 1, y) in coords:
                self.animation_prim.append(path)
                self.grid_prim[y][x].set_wall(path[-1])
                x -= 1
                self.grid_prim[y][x].set_wall("right")
                visited.append((x, y))
                walls = walls + self.add_walls(x, y)
                visitNum += 1
            elif path[-1] == "right":
                if (x + 1, y) not in visited and (x + 1, y) in coords:
                    self.animation_prim.append(path)
                    self.grid_prim[y][x].set_wall(path[-1])
                    x += 1
                    self.grid_prim[y][x].set_wall("left")
                    visited.append((x, y))
                    walls = walls + self.add_walls(x, y)
                    visitNum += 1

    def add_walls(self, x, y):
        walls = []
        cell_walls = self.grid_prim[y][x].get_walls()
        if cell_walls[0] and y != 0:
            walls.append((x, y, "up"))
        if cell_walls[1] and y != self.size - 1:
            walls.append((x, y, "down"))
        if cell_walls[2] and x != 0:
            walls.append((x, y, "left"))
        if cell_walls[3] and x != self.size - 1:
            walls.append((x, y, "right"))
        return walls

    def get_grid_prim(self):
        return self.grid_prim

    def get_animation_rtb(self):
        return self.animation_rtb

    def get_animation_prim(self):
        return self.animation_prim


class Pygame:
    def __init__(self, maze, animation):
        self.size = size
        self.white = (255, 255, 255)
        self.black = black
        self.width = cell_width
        self.animation = animation
        self.grid = maze
        self.green = (0, 255, 0,)
        self.blue = (255, 192, 203)
        self.red = (255, 0, 0)
        self.lblue = (45, 213, 200)
        self.dblue = (1, 95, 96)

    def animate_grid(self):
        color = self.blue
        self.draw_grid(color)
        for cell in self.animation:
            x, y = cell[0] + 1, cell[1] + 1
            x, y = x * self.width, y * self.width
            # print(cell)
            time.sleep(.001)
            if cell[-1] == "up":
                pygame.draw.line(window, self.black, [x, y], [x + self.width, y])  #
Draws top wall of cell
                pygame.event.get()
                pygame.display.update()

            if cell[-1] == "down":
                pygame.draw.line(window, self.black, [x + self.width, y + self.width],
                                 [x, y + self.width])  # Draws bottom wall of cell
                pygame.event.get()
                pygame.display.update()

            if cell[-1] == "left":
                pygame.draw.line(window, self.black, [x, y + self.width], [x, y])  #
Draws left wall of cell
                pygame.event.get()
                pygame.display.update()

            if cell[-1] == "right":
                pygame.draw.line(window, self.black, [x + self.width, y],
                                 [x + self.width, y + self.width])  # Draws right wall of cell
                pygame.event.get()
                pygame.display.update()
        self.draw_maze(color)
    def draw_grid(self, color):
        for row in range(self.size):
            for column in range(self.size):
                x, y = row + 1, column + 1
                x, y = x * self.width, y * self.width
                pygame.draw.line(window, color, [x, y], [x + self.width, y])  # Draws
top wall of cell
                pygame.draw.line(window, color, [x + self.width, y + self.width],
                                 [x, y + self.width])  # Draws bottom wall of cell
                pygame.draw.line(window, color, [x, y + self.width], [x, y])  # Draws
left wall of cell
                pygame.draw.line(window, color, [x + self.width, y],
                                 [x + self.width, y + self.width])  # Draws right wall of cell
        pygame.draw.rect(window, self.green, (self.width + 2, self.width + 2,
self.width - 4, self.width - 4), 0)
        pygame.draw.rect(window, self.red,
                         (self.width * self.size + 3, self.width * self.size + 2, self.width - 4,
self.width - 4), 0)

    def draw_maze(self, color):
        for row in range(self.size):
            for column in range(self.size):
                cell = self.grid[row][column]
                walls = cell.get_walls()
                x, y = cell.x + 1, cell.y + 1
                y, x = x * self.width, y * self.width
                if walls[0]:
                    pygame.draw.line(window, color, [x, y], [x + self.width, y])  # Draws
top wall of cell
                if walls[1]:
                    pygame.draw.line(window, color, [x + self.width, y + self.width],
                                     [x, y + self.width])  # Draws bottom wall of cell
                if walls[2]:
                    pygame.draw.line(window, color, [x, y + self.width], [x, y])  # Draws
left wall of cell
                if walls[3]:
                    pygame.draw.line(window, color, [x + self.width, y],
                                     [x + self.width, y + self.width])  # Draws right wall of cell
        pygame.draw.rect(window, self.green, (self.width + 2, self.width + 2,
self.width - 4, self.width - 4), 0)
        pygame.draw.rect(window, self.red,
                    (self.width * self.size + 3, self.width * self.size + 2, self.width - 4,
self.width - 4), 0)

    def animate_solution(self):
        color = self.lblue
        for cell in self.animation:
            time.sleep(.007)
            x, y = cell[0] + 1, cell[1] + 1
            y, x = x * self.width, y * self.width
            width = self.width / 2
            pygame.draw.circle(window, color, (x + width, y + width), 2, 2)
            pygame.event.get()
            pygame.display.update()

    def draw_solution(self):
        color = self.red
        for cell in self.animation:
            x, y = cell[0] + 1, cell[1] + 1
            y, x = x * self.width, y * self.width
            width = self.width / 2
            pygame.draw.circle(window, color, (x + width, y + width), 2, 2)


class Pathfinding:
    def __init__(self, maze):
        self.size = size
        self.maze = maze
        self.path = []
        self.visited = 0

    def a_star(self):
        start = (0, 0)
        end = self.maze[self.size - 1][self.size - 1]
        self.path.append((0, 0))
        found = False
        visited = [(0, 0)]
        p_route = []
        w = 0
        while not found:
            adj = []
            adj = self.get_adjacent_cells(self.path[-1][0], self.path[-1][1])
            cost = []
            coords = []
            for cell in adj:
                if cell not in visited:
                    cost.append(self.heuristics(self.maze[cell[0]][cell[1]], end))
                    coords.append(cell)
            if len(cost) > 0:
                p_route.append(self.path[-1])
            elif len(p_route) > 0:
                coord = p_route.pop()
                popped = self.path.pop()
                while coord != popped:
                    popped = self.path.pop()
                self.path.append(coord)
                adj = self.get_adjacent_cells(coord[0], coord[1])
                cost = []
                coords = []
                for cell in adj:
                    if cell not in visited:
                        cost.append(self.heuristics(self.maze[cell[0]][cell[1]], end))
                        coords.append(cell)
            if len(cost) > 0:
                self.path.append(coords[cost.index(min(cost))])
            visited.append(self.path[-1])
            w += 1
            if visited[-1] == (self.size - 1, self.size - 1):
                found = True
                self.visited = visited
                break

    def heuristics(self, cell1, cell2):
        # Returns the manhattan distance between the two cells
        return abs(cell1.x - cell2.x) + abs(cell1.y - cell2.y)

    def get_adjacent_cells(self, x, y):
        adj_cells = []
        walls = self.maze[x][y].get_walls()
```

```
        if walls[0] == False:
            adj_cells.append((x - 1, y))
        if walls[1] == False:
            adj_cells.append((x + 1, y))
        if walls[2] == False:
            adj_cells.append((x, y - 1))
        if walls[3] == False:
            adj_cells.append((x, y + 1))
        return adj_cells

    def get_path(self):
        return self.path

    def get_visited(self):
        return self.visited


if __name__ == '__main__':
    # initialize pygame
    pygame.init()
    clock = pygame.time.Clock()
    window = pygame.display.set_mode((width, height))
    window.fill(black)
    pygame.display.set_caption("Test 1")

    # Calls the Main function
    main()

    # Mainloop
    while True:
        # Sets frame rate to 60 FPS
        clock.tick(FPS)
        for event in pygame.event.get():
            if event.type == QUIT:
                sys.exit()
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_RETURN:
                    sys.exit()

        # Updates the display
        pygame.display.update()

# main()
```