

MAZE GENERATOR & MAZE SOLVER

Maze Generation Using Prim’s Minimum Spanning Tree & Maze Solving Using Dead End Filling by Danny Piper



Contents

[MAZE GENERATOR & MAZE SOLVER 0](#_Toc25779487)

[Analysis 4](#_Toc25779488)

[Background to Project 4](#_Toc25779489)

[Research Log 5](#_Toc25779490)

[Diary of Research 5](#_Toc25779491)

[FINISH ME 6](#_Toc25779492)

[Evidence of Analysis 7](#_Toc25779493)

[Further Research 7](#_Toc25779494)

[Alternate Algorithms 7](#_Toc25779495)

[Solving Algorithms 7](#_Toc25779496)

[Generating Algorithms 8](#_Toc25779497)

[Types of Maze 8](#_Toc25779498)

[Perfect Mazes 9](#_Toc25779499)

[Braid Mazes 9](#_Toc25779500)

[Combo Mazes 9](#_Toc25779501)

[Data Volumes 10](#_Toc25779502)

[Modelling of the Problem 11](#_Toc25779503)

[Maze Generation 11](#_Toc25779504)

[Existing Solutions 12](#_Toc25779505)

[Examples of Maze Generators 13](#_Toc25779506)

[http://www.billsgames.com/mazegenerator/ 13](#_Toc25779507)

[https://xefer.com/maze-generator 15](#_Toc25779508)

[Examples of Maze Solvers 16](#_Toc25779509)

[https://scratch.mit.edu/projects/70718980/ 16](#_Toc25779510)

[Proposed Solution Details 18](#_Toc25779511)

[Maze Solver – Input 19](#_Toc25779512)

[Maze Solver – Processing 20](#_Toc25779513)

[Data Flow 21](#_Toc25779514)

[Maze Generator – Input 22](#_Toc25779515)

[Maze Generator – Processing 22](#_Toc25779516)

[Maze Generator – Output 22](#_Toc25779517)

[Prototyping 23](#_Toc25779518)

[Maze Solver Prototype 23](#_Toc25779519)

[Objectives 24](#_Toc25779520)

[Program Objectives 24](#_Toc25779521)

[Maze Generator 24](#_Toc25779522)

[Maze Solver 25](#_Toc25779523)

[Research Objectives 27](#_Toc25779524)

[Maze Generator 27](#_Toc25779525)

[Maze Solver 27](#_Toc25779526)

[Documented Design – Generic 28](#_Toc25779527)

[Graph to Image Mapping 28](#_Toc25779528)

[Image to Graph Mapping 28](#_Toc25779529)

[Image rendering 29](#_Toc25779530)

[Documented Design – Maze Generator 30](#_Toc25779531)

[High Level Overview 30](#_Toc25779532)

[User Interface Design 31](#_Toc25779533)

[Algorithms 33](#_Toc25779534)

[Prim’s Algorithm 33](#_Toc25779535)

[Kruskal’s Algorithm 35](#_Toc25779536)

[Sorting algorithms 37](#_Toc25779537)

[Quicksort 37](#_Toc25779538)

[Pseudocode: 37](#_Toc25779539)

[Insertion Sort 38](#_Toc25779540)

[Pseudocode: 38](#_Toc25779541)

[Bubble Sort 39](#_Toc25779542)

[Pseudocode: 39](#_Toc25779543)

[Counting Sort 40](#_Toc25779544)

[Pseudocode: 40](#_Toc25779545)

[Sort Time Comparison 41](#_Toc25779546)

[Loop Finding Algorithms 42](#_Toc25779547)

[Disjoint Sets (Union/Find Algorithm) 42](#_Toc25779548)

[Graph Traversal – Breath First / Depth First 42](#_Toc25779549)

[Documented Design – Maze Solver 43](#_Toc25779550)

[High Level OverviewSolving the Maze 43](#_Toc25779551)

[Multi-Threading 44](#_Toc25779552)

[User Interface Design 45](#_Toc25779553)

[45](#_Toc25779554)

[Algorithms 46](#_Toc25779555)

[Dead End Filling 46](#_Toc25779556)

[Technical Solution – General 47](#_Toc25779557)

[Problems Encountered 47](#_Toc25779558)

[Technical Solution – Maze Generator 48](#_Toc25779559)

[Problems Encountered When Creating the Solution 48](#_Toc25779560)

[Technical Solution – Maze Solver 49](#_Toc25779561)

[Problems Encountered When Creating the Solution 49](#_Toc25779562)

# Analysis

## Background to Project

In Further Maths, during the modelling with algorithms module, I was learning about Prim’s minimum spanning tree algorithm. Whilst thinking of uses of this algorithm I concluded it could be used to generate a maze with no loops. The modelling with algorithms module in Further Maths is about the application of algorithms for complex problems, most of the modelling with algorithms is about graph theory and related algorithms. Graph theory has extensive use with problems involving paths, nodes and network flows.

To test the generator a solver will be required so I have decided to create a program for both and analyse effectiveness and complexity for the solver as the generators complexity is known at 0(n2) because it uses Prim’s algorithm for the generation of the minimum spanning tree.

## Research Log

### Diary of Research

<http://www.billsgames.com/mazegenerator/>

This site has a maze generator that I have analysed later (page 17).

Last Accessed on: 8/09/2019

<https://xefer.com/maze-generator>

This site has a maze generator that I have analysed later (page 19).

Last Accessed on: 8/09/2019

<https://scratch.mit.edu/projects/70718980/>

Last Accessed on: 8/09/2019

<https://www.youtube.com/watch?v=rop0W4QDOUI>

Last Accessed on: 15/09/2019

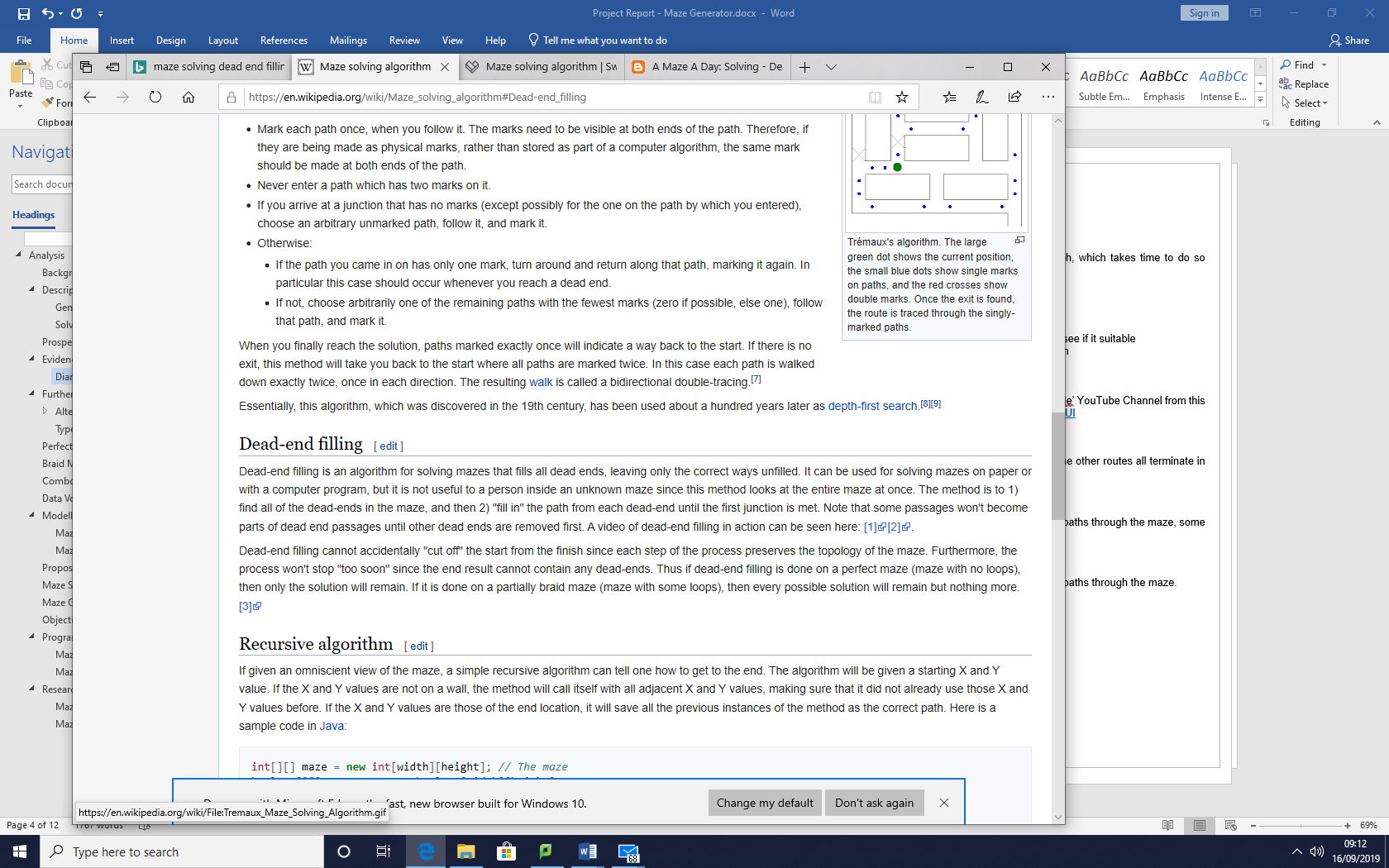
<https://github.com/mikepound/mazesolving>

Last Accessed on: 15/09/2019

### FINISH ME

## Evidence of Analysis

To solve a maze, dead end filling algorithms can be used, they remove all of the dead ends and leave the path from the start to end of the maze behind.

Dead end filling algorithms (for maze solving) have one paragraph of material on Wikipedia so is not researched very deeply. Below is a screenshot of the Wikipedia article.Many pages about this algorithm have it referenced as a paragraph for example <https://swuecho.fandom.com/wiki/Maze_solving_algorithm> mentions the algorithm (last accessed 16/09/2019) an exact copy of Wikipedia’s content. This article is very short <https://amazeaday.blogspot.com/2009/03/solving-dead-end-filling.html> (accessed on 16/09/2019) and has no new information, it describes the algorithm very briefly. Below is the entire article.

|  |
| --- |
| Basically you find all the dead ends and fill them, working backwards down the passages that leads to the dead ends till you reach a junction. If filling the maze leads to a junction that is a dead end you fill that junction and continue backfilling. Keep doing this until all passsages lead to a dead end are filled. What is left (not filled) is the solution path |

Dead end filling algorithms remove the dead ends from graphs until the paths to traverse the maze are left.

In comparison to maze generation very little research has been carried out, both Prim’s and Kruskal’s algorithms have their own Wikipedia pages ([https://en.wikipedia.org/wiki/Kruskal%27s](https://en.wikipedia.org/wiki/Prim%27s_algorithm)\_algorithm and <https://en.wikipedia.org/wiki/Prim%27s_algorithm>), this is a good indicator to how much research has been carried out.

## Further Research

### Alternate Algorithms

#### Solving Algorithms

* Dijkstra’s & A\* Dijkstra’s
  + Requires mapping of the image to a graph, which takes time to do so increases the time taken
  + Not suitable for divide and conquer parallelisation techniques
* Trémaux's algorithm
  + Is suitable for divide and conquer techniques

<https://intranet.psc.ac.uk/computing_linear/documents/Computer%20Science%20Project/AQA%20Booklets/Project%20Maze%20Solving.pdf>

#### Generating Algorithms

* Kruskal’s
  + Requires a sort of arcs by weight
  + Not suitable for divide and conquer parallelisation techniques
  + Followed by loop detection for each arc to see if it suitable
    - Loop detection is a complex algorithm

### Types of Maze

These maze types are sourced from the ‘Computer Phile’ YouTube Channel from this video <https://www.youtube.com/watch?v=rop0W4QDOUI>

|  |  |
| --- | --- |
| Perfect Mazes A perfect maze has one path from start to finish and the other routes all terminate in a dead end. |  |
| Braid Mazes A braid maze has minimal dead ends (sometimes during generation of this type of maze requires a dead end to be connected fully) and there are several paths through the maze, some shorter than others. An example of a braid maze is to the right. |  |
| Combo Mazes A combo maze has dead ends and loops with several paths through the maze. An example of a combo maze is below. |  |

## Data Volumes

An image of x by y is imported and a byte array of size x\*y bytes + 16 bytes is made, an image object in java is made. Overall the system will most likely use around 1 GB of RAM for a large data set.

This program will have hardware limitations for large amounts images due to the amount of RAM it will require and compute time. Expected hardware requirements are:

* 4GB Ram for small datasets and 8GB for large data sets and 16GB for huge datasets.
* 4 Core / 8 threads processor such as an AMD Ryzen 5 1600

Since the program uses java the following additional JRE parameters are recommended to make the JRE run with enough RAM for the task:

* (Java -jar <filename>) -Xmx2G
* E.g.:
  + Java -jar mazeSolver.jar -Xmx2G

Given a huge dataset alternate JRE parameters are recommended

* (Java -jar <filename>) -Xmx10G
* E.g.:
  + Java -jar mazeSolver.jar -Xmx10G

## Modelling of the Problem

A maze can be mapped to a graph and a graph to a maze.

### Maze Generation

This project will make extensive use of graph theory which has a lot of unique terminology, which will be described when appropriate.

Currently Relevant Terminology:

Node – a junction of sorts, the diagram below will make this clearer.

Arc – a connection between two nodes.

Weight (of an arc) – A numerical value assigned to a node. Given a graph that was used to model a road network then the weight of a node can be used to describe the time taken to traverse.

Connected (a connected graph) – The graph has a path from any one point to another.

Tree – A set of arcs that connect all nodes together with no loops

Minimum Spanning Tree – A tree that has the least total weight

On the next page is the topological view of the basic graph structure for a 4x4 maze, observe how there are 16 nodes and 27 arcs, each arc is given a random weight, a positive integer then a minimum spanning tree can be created. The minimum spanning tree can be generated by using Kruskal’s Algorithm or Prim’s Algorithm.

An example graph with nodes:

|  |
| --- |
| n nodes with n-1 arcs going across  Arc  Node |

A minimum spanning tree is the tree with least weight, a tree has no loops and is connected. By generating a minimum spanning tree from this graph, a maze with no exit and no entrance is made. By adding an entrance/ exit in post (after image mapping) it means that a solvable, loop less maze is made.

To generate a minimum spanning tree from a connected graph either Prim’s or Kruskal’s algorithms are applied.

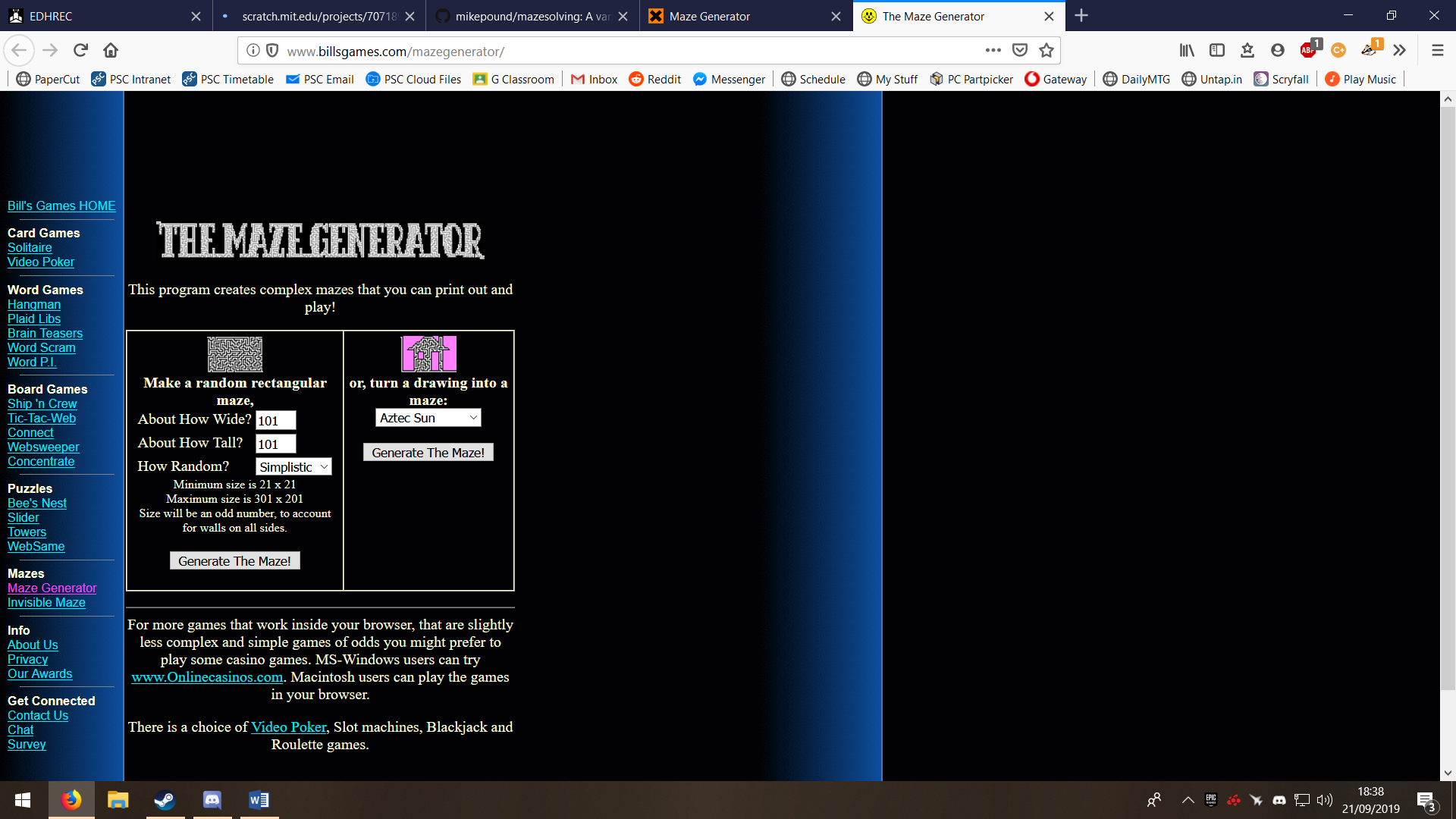
Here there are 24 arcs for 16 nodes (a grid of 4x4) which is r(r-1) + c(c-1) arcs where c is the number of columns, r the number of rows.

## Existing Solutions

The following are examples of programs which can solve or generate mazes a brief description of the pros and cons of each program.

### Examples of Maze Generators

#### <http://www.billsgames.com/mazegenerator/>

Below is an image of the user interface of *billsgames.com*‘s maze generator, and an image of the output.

Pros

* A large amount of customisation for the maze output

Cons

* Not HTTPS
* Has a maximum maze size of 301x301
* Not always solvable, the sample output has a disconnected exit so has no path through the maze. The circled area shows the disconnected exit path

|  |
| --- |
| Your Maze   * This is the sample output for *billsgames.com*’s mazegenerator |

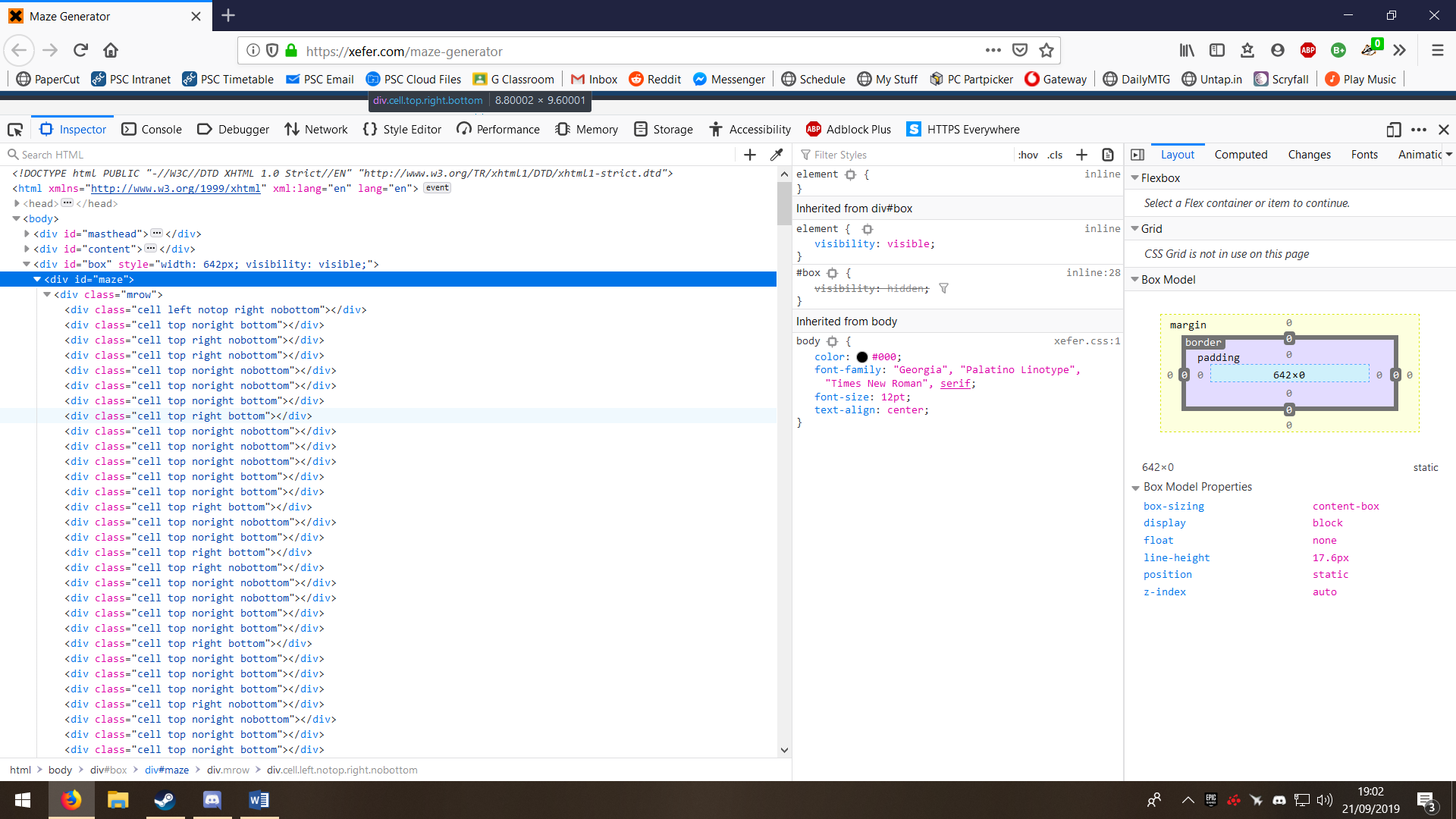
#### <https://xefer.com/maze-generator>

This maze generator has a very clean and minimal user interaface however it does not store the maze as an image. The program uses html to render the maze in the form of lots of *div* elements which uses a lot of RAM.

Pros

* Has no maximum size

Cons

* Is slow for large mazes, 1000x1000 took a few minutes
* Used 2GB of RAM for the 1000x1000 image
* Cannot save the maze as an image

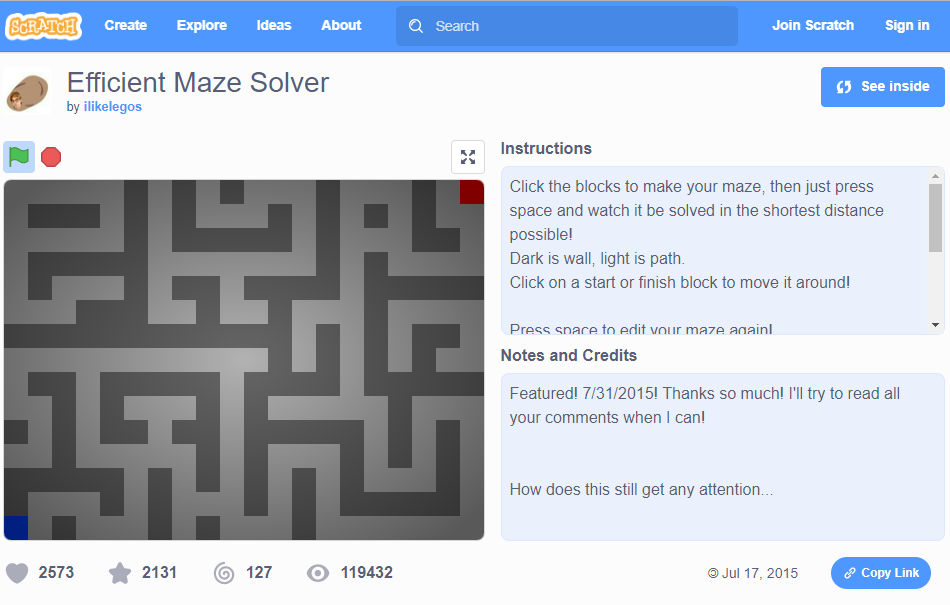
### Examples of Maze Solvers

#### <https://scratch.mit.edu/projects/70718980/>

(last accessed 8/09/2019)

This maze solver has an animation of the generation of the path. However is made in scratch so no image maze can be loaded.

Here is a screenshot of the maze solver, as it is in scratch the source code can be loaded. The solver uses graph traversal to generate a solution and highlights it in pink. Unfortunately, since last accessing the project it has been changed and no longer works (when accessed on 25/11/2019).



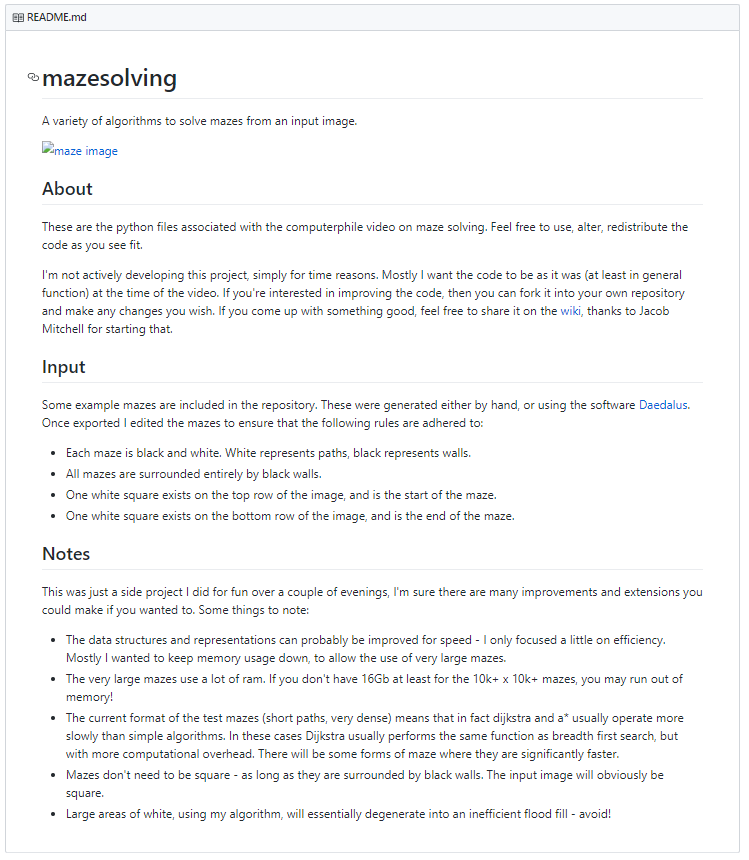
Below is a link to a video discussing many ways to solve a maze and is more focused on the technical aspect of maze solving rather than the algorithms. The video is detailed and shows python explains the solution to maze solving which he has linked to.

<https://www.youtube.com/watch?v=rop0W4QDOUI>

* Discusses many ways in which to solve a maze
  + Dijkstra’s, and A\* Dijkstra’s
    - Discusses image to graph mapping
  + Trémaux's algorithm (Depth first/ breadth first)

<https://github.com/mikepound/mazesolving> is the link to the solution used in the video

* It is written in python and uses different algorithms and approaches to compute the output image. Below is a screenshot of the readme of the program. The sample mazes for this project are generated with <http://www.astrolog.org/labyrnth/daedalus.htm> “The world's most sophisticated Maze program! :-)”



## Proposed Solution Details

Both the maze generator and solver are going to be programmed with Java SE 12 using the eclipse IDE and will require the GUI library javafx to be executed. I decided to use javafx over the built in library, called swing, because it is more modern and works better.

### Maze Solver – Input

An image will be loaded from a file. A GUI will prompt the user to select the image file. The output file is also selected by the user here.

Some input mazes will provide better results than others, the table below describes the output of varying maze types.

|  |  |
| --- | --- |
| A perfect maze will generate the best output. As the top left image shows a path through the maze (highlighted in red) is left behind. All perfect mazes will only have one path through them so will produce output that is similar to this. |  |
| Given a none-perfect maze such as a braid or combo maze then there are several paths through the maze and the output will not be clear to the user because all of the paths are shown highlighted in the same colour. This could mean that a braid maze the entire image is highlighted. To the right is example output for a combo maze. |  |

### Maze Solver – Processing

An image is loaded, translated to a 2D byte array, each byte has the pixels binary value. One instance of the 2D byte array is present. For each real core a thread is generated, and the following algorithm ran to completion on a chunk of the image that they are assigned:

Terminology:

NODEPIXEL – A pixel that represents a node in a graph, the x and y index value will be odd.

NONFILLED – A wall that has not been tagged as filled

CONNECTION – A node that is neighbouring the node in question with an arc connecting it to the node in question

ChangesMade = true  
While ChangesMade do  
begin  
ChangesMade = false  
 For each NONFILLED NODEPIXEL do  
 If NODEPIXEL has one CONNECTION then  
 begin  
 Fill NODEPIXEL  
 ChangesMade = true  
 end  
end

After the threads have finished running the algorithm the main thread runs the algorithm to finish it off. All the while a render thread will show graphically to the user what is occurring as it solves, this will be cool to watch and let the user know far through the processing the image is.

The dead-end filling algorithm can undergo multi-threaded, parallel execution by dividing the image into chunks and treating each as a separate image whilst fetching the data from the same 2D array.

In order for multiple formats to be loaded the program must interpret the format for example a path of width X has a different algorithm for one of width Y, so a standardised input where all paths of width one will be far more practical to program.

### Data Flow

Image enters the system via the GUI module, is sent to the solver module. The solver module solves it and saves an image, the output image is selected by the user in the GUI.

### Maze Generator – Input

A GUI will be used to get the information required for maze generation from the user. The following data will be obtained and passed to the generation module.

|  |  |  |
| --- | --- | --- |
| Data Name | Data Type | Validation Required |
| Entrance point Y | Unsigned integer | Must be 32 bit |
| Exit point Y | Unsigned integer | Must be 32 bit |
| Graph width | Unsigned integer | Must be 32 bit |
| Graph height | Unsigned integer | Must be 32 bit |
| File to save output to | File name | Must be a valid name (by windows standards) |
| Whether Primm’s or Kruskal’s algorithm will be used for generation | Boolean | Cannot be null |
| Sorting algorithm (if Kruskal’s algorithm is selected) | Enumerable  {bubble sort, insertion sort, quicksort, counting sort} | Cannot be null |
| Whether or not to use procedural generation for Primm’s algorithm (if Primm’s algorithm is selected) | Boolean | Cannot be null |

### Maze Generator – Processing

The appropriate algorithm is executed, and a real-time render preview of the maze is shown to the user. The path will be white and the walls black.

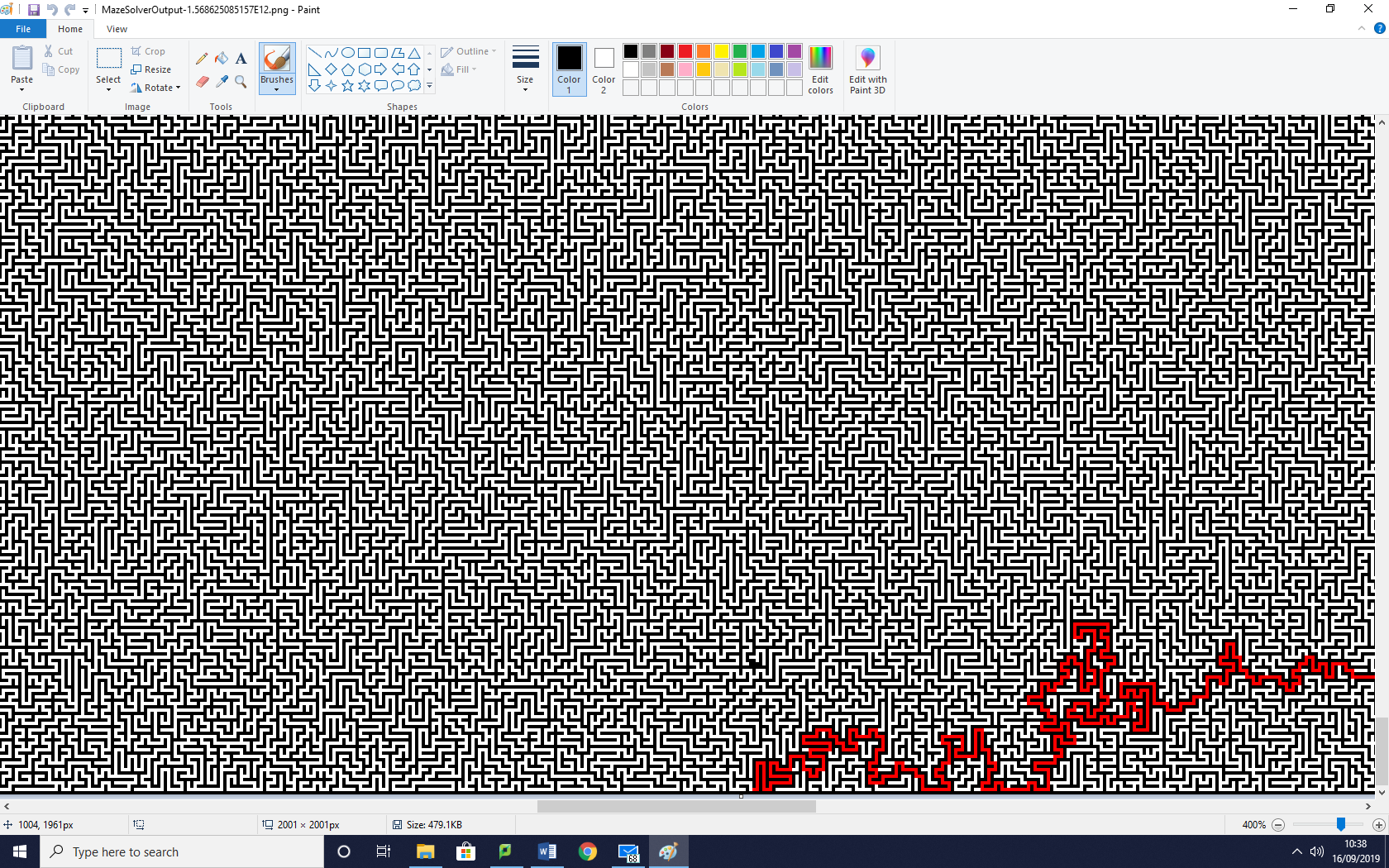
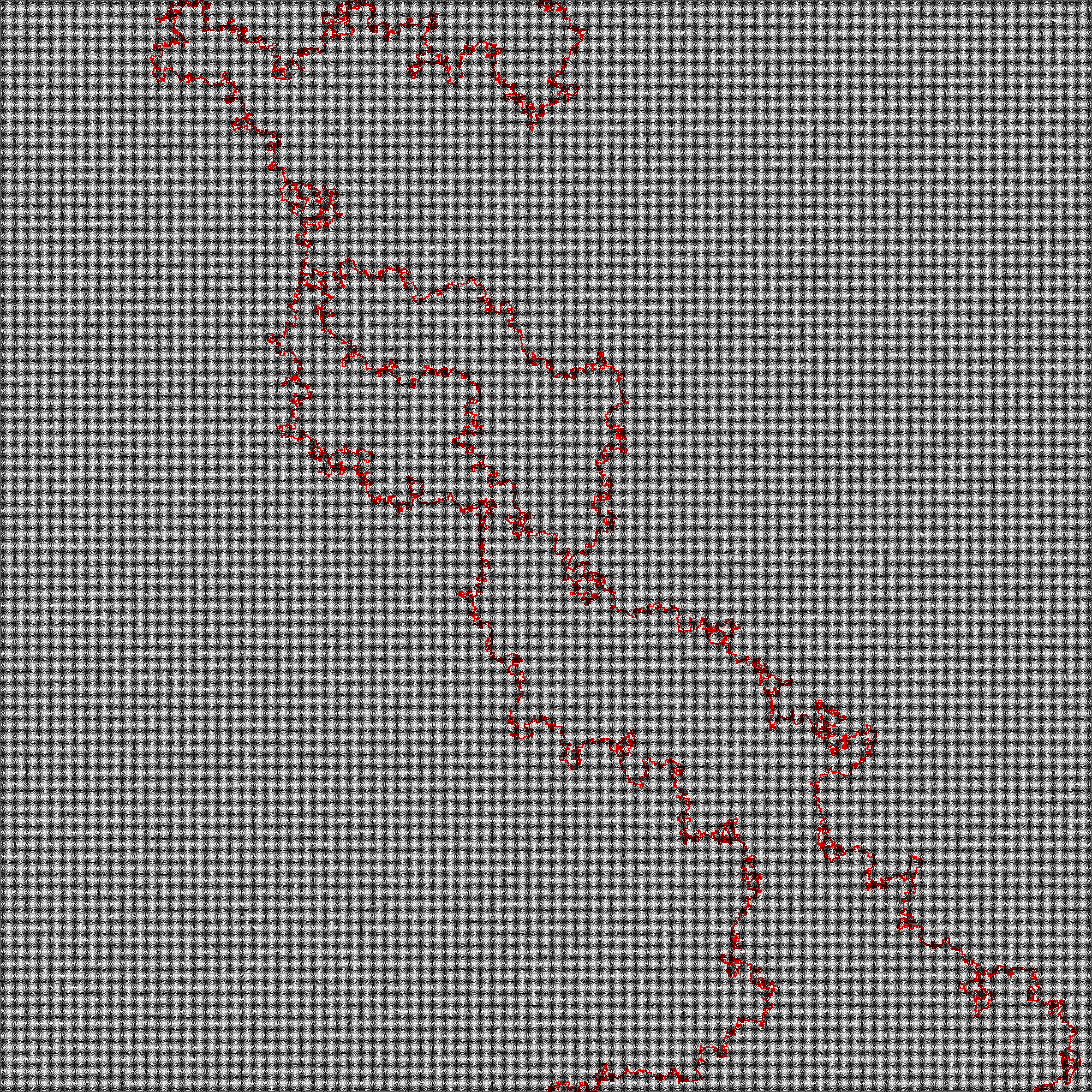
### Maze Generator – Output

An 8 bit image of size (2 \* graph width + 1) x (2 \* graph height + 1) is saved with the maze saved inside of it. The path will be white and the walls black.

## Prototyping

### Maze Solver Prototype

The prototype can load an image and solve the maze then save it as an output image. However, there are some bugs related to premature termination, excessive memory usage and the inability to scroll about large images. The prototype is multithreaded and does show a live render of potential paths.

Under certain circumstances the program terminates early and shows a path which branches towards a dead end and ends suddenly (example below). This is caused by the program crashing for an unknown reason.

The image to the right shows the branching path on the output image. The image is 2001 x 2001 so may not display or print correctly other than the red path on a grey looking background. This output contains the snapshot from above.

# Objectives

The overall objective is to create a maze generator, a maze solver and to research complexity of the algorithms that I make use of which have not been documented in detail (dead end filling).

## Program Objectives

### Maze Generator

\*Graph, a Graph is a collection of nodes with arcs connecting them.

1. Input
   1. Parameters
      1. Graph\* width  
         Graph\* height  
         Entrance location  
         Exit location  
         File name
   2. Data is validated and the user will be notified of each invalid entry
2. Processing
   1. Show an expected completion time in the GUI whilst processing
      1. Based off of the complexity of Prim’s algorithm and the time for each matrix iteration
   2. Graph Generation
      1. A width by height grid of nodes connected by arcs of random weight comprises the graph
      2. Each node is given a unique ID that has a 1:1 mapping between the x and y coordinates and node ID
      3. The Graph is stored as a 2D array of IDS and a 2D array of weight values
   3. Minimum Spanning Tree Creation
      1. The application of Prim’s algorithm
      2. Generates a loop-less, minimum spanning tree where each node has a connection to another
   4. Graph to Image Mapping
      1. A 2D Boolean array is made, maze path is true, maze wall false
      2. The minimum spanning tree nodes are mapped
      3. Each node’s ID is turned to coordinate and is mapped to the 2D Boolean array
      4. Pixel interpolation to generate a connected graph in the image
         1. 2 adjacent (down/ right) nodes are compared and the average x and average y are used to create the true coordinate to mapped to the Boolean array
      5. Entrance and exit addition to the 2D Boolean array
      6. The coordinate is user-defined, defaults to (0,1) and (0, width\*2+1)
   5. The image is then made with a 1-bit colour depth (hence the Boolean 2D array) and the Boolean array values are mapped to the image directly
3. Storage
   1. The image is then saved
      1. If a file under that name is already present it will ask for a new name and attempt to save again
      2. If an I/O error occurs, it will try again once
         1. If it still errors it will ask the user to insert a new file name or to exit without saving
         2. Exit without saving will have a confirmation button to stop accidents (hopefully it will stop them)

### Maze Solver

* + 1. Input
       1. An image of standard format (png, no lossy compression)
          1. Given an I/O error whilst reading the program will ask for new input
    2. Processing
       1. Multithreaded processing
       2. Turn the image into a 2D byte array
       3. All 0xFFFFFF is turned to 0xFF0000
       4. 0xFF0000 is a maze path 0xFFFFFF is a filled path
       5. The number of cores on the system is recorded

A thread per real core is generated

Each thread is assigned a part of the image

Each thread applies the algorithm to the area of the image

* + - 1. Once finished the main thread is notified
      2. The main thread waits for all the threads to finish
      3. The main thread applies the algorithm is applied to the whole image until it finishes
      4. The algorithm scans for dead ends and fills until none remains
      5. Whilst the solve is underway the render thread will render the image ever frame to the GUI
      6. Time taken is output in console (not visible by default).
    1. Output
       1. The taken to solve in milliseconds
    2. Storage
       - 1. The image is saved with a red path showing how to navigate the maze

The filename is generated from the hash of the time

## Research Objectives

### Maze Generator

1. Identify why Kruskal’s algorithm is not suitable for this workload
   1. Analyse the efficiency of sorting algorithms for use prior to Kruskal’s algorithm
      1. Investigate the bubble sort, insertion sort and quicksort
         1. Program each using java
         2. Run each of the algorithms on arrays of length n of strings of length 5 with random lowercase characters, the same array for each sort to obtain times in milliseconds of the compute times required
         3. Export the data to a text file with | as the delimiter
         4. Analyse the algorithms
   2. Identify the different algorithms for finding loops in graphs
      1. Analyse each one’s efficiency and complexity to program
2. Conclude why Prim’s algorithm will be superior for MST (minimum spanning tree) generation

### Maze Solver

1. Using output generated from the maze solver, obtain data on time taken for compute for mazes of size nxn
2. Using this data and logic to attempt to find the complexity of the algorithm
3. Create a video of a large dataset being solved to demonstrate the efficiency of the algorithm

# Documented Design – Generic

## Graph to Image Mapping

The following is pseudo code to map a graph to an image.

|  |
| --- |
| Image ← new image with 8-bit colour depth  ImageWidth ← graphWidth\*2 +1  ImageHeight ← graphHeight\*2 +1  For each pixel in image, set colour to 0x000000  For each node in graph{  Image[node.xID\*2+1][node.yID\*2+1] ← 0xFFFFFF  } |

## Image to Graph Mapping

The image will be mapped to an *adjacency matrix*. An *adjacency matrix* can then me mapped to a graph.

|  |
| --- |
| adjMat ← 2D array of booleans with width equal to (imageWidth/2)-1, and height equal to (imageHeight/2)-1  set all values in adjMat to false  X ← 1  While x < imageWidth {  Y←1  While y <imageHeight{  If image[x+1][y] == 0xFFFFFF {  adjMat[node.ID][connectingNode.ID] ← true  }  If image[x][y+1] == 0xFFFFFF {  adjMat[node.ID][connectingNode.ID] ← true  }  Y←Y+2  }  x←x+2  } |

## Image rendering

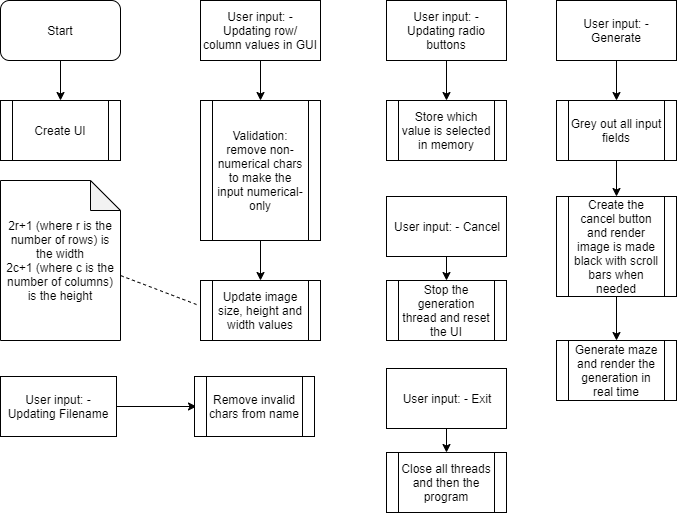
To render an image, I will use Java’s built in libraries to minimise the amount of dependencies. Pseudo code of how I will render an image onto the screen.

|  |
| --- |
| Panel **←** new double-buffered panel  //Render an image  Procedure renderImage (ImageToRender) {  Image **←** ImageToRender  ScaleImageToPanelSize  RenderScaledImageToPanel  }  //Scale an image, integer only  Function scaleImage(Image, scaleFactor) {  If scaleFactor <= 0 then  throw new InvalidSFError  else  ScaledImage **←** byte[image x \* scale][image y \* scale]  For x = 0 to imageWidth\*scale {  For y = 0 to imageHeight\*scale {  ScaledImage[x][y] **←** ScaledImage[x div scale][y div scale];  }  }  } |

As a demonstration I have created a prototype called ‘*Graphics*’ which is a basic physics simulation of a bouncing ball. It runs at the framerate of the monitor. The runnable file is called ‘*test1.jar*’ and has source within the *test1* folder. After consideration I have concluded that integer nearest neighbour will not be suitable and it too slow when single threaded.

# Documented Design – Maze Generator

## High Level Overview



## User Interface Design

I have created a mock-up of the user interface in RAD studio (because it has a fantastic design tool), below is the mock-up.

|  |
| --- |
| Each group of radio buttons (sorting algorithms) and (generation algorithms) are independent.  If Kruskal’s algorithm is not selected then the sorting algorithms menu area will be greyed out to show that they are not being used  This is where the program’s parameters are keyed-in by the user.  If the image of the maze being generated is too large to be displayed with a scale factor greater than or equal to 1.0 then scroll bars and zoom tools will be shown to the user.  As the generation algorithm is executing, the commitment of an arc to the Minimum Spanning Tree can be mapped in real time to a 2D array representing an image in memory and displayed to the user. |

When the generate button is pressed, a cancel button is created next to the generate button and the generate button is greyed out and made unclickable.

If the user can only input valid chars to input fields then it reduces the amount of validation required is greatly reduced. All that needs to be checked is that the filename can be used and that the OS allows writing to the folder and that there is enough space to write the image to. Given any errors appropriate error messages will be displayed.

## Algorithms

### Prim’s Algorithm

Prim’s algorithm has complexity of O(n2) for an adjacency matrix method (expanded on later). Given a more advanced implementation such as using Fibonacci Heaps then the complexity can be modelled as O(V log V + E log V) = O(ElogV) (E being the amount of edges (arcs) and V being the amount of vertices (nodes) ) source: <https://www.cs.auckland.ac.nz/software/AlgAnim/prim.html> last accessed 18/09/2019

Applying Prim’s algorithm is simplest with an adjacency matrix and as such prototyping will be done with an adjacency matrix. An adjacency matrix can be modelled as a 2D array of size nxn where n is the number of nodes. An arc from node 1 to node 2 will have the weight plotted at [1][2] and [2][1] as it is bidirectional and of equal weight for each direction. Below is an example of an adjacency matrix and its corresponding graph.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Graph:  25  30  28  11  15  18  12  21 | | | | | | |
| Adjacency Matrix: | | | | | | |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 0 | 25 | 12 | 0 | 0 | 0 |
| 2 | 25 | 0 | 0 | 30 | 21 | 0 |
| 3 | 12 | 0 | 0 | 28 | 0 | 15 |
| 4 | 0 | 30 | 28 | 0 | 0 | 11 |
| 5 | 0 | 21 | 0 | 0 | 0 | 18 |
| 6 | 0 | 0 | 15 | 11 | 18 | 0 |

After the application of Prim’s algorithm (shown later) the following output will be produced:

*1-3 12, 3-6 15, 4-6 11, 1-2 25, 6-5 18*

Total weight 81

All minimum spanning tree algorithms will always top when n-1 arcs have been selected as n-1 arcs is how many arcs are required for a minimum spanning tree of n nodes.

To produce the minimum spanning tree the first row of the matrix is removed. The first column is assigned as a pivot column. The value with the least weight is found, its row number is a column number, its column is now a pivot and the row is removed after the weight and connection is noted. Then for instead of searching just the first column, all pivot columns are searched and the algorithm repeated until n-1 nodes are found, where n is the amount of nodes.

### Kruskal’s Algorithm

Kruskal’s algorithm is a simpler than Prim’s algorithm to understand.

The arcs are placed into an array and sorted in ascending order by weight, for each arc in the list it is added to the minimum spanning tree list if the following conditions are met: there are less than (n-1) elements in the list and the addition of the arc will not create a loop (loop finding algorithms on page 29).

The sorting algorithm that I will use for Kruskal’s algorithm will be the quicksort because of its performance in comparison to other sorts, this comparison can be found on page 27.

Using the same graph as Prim’s (the graph is copied below) the execution of Kruskal’s is as following:

|  |
| --- |
| Graph:  25  30  28  11  15  18  12  21 |

Unsorted data:

{1-2 25, 1-3 12, 2-5 21, 2-4 30, 3-4 28, 3-6 15, 6-4 11, 5-6 18}

After execution a recursive quicksort (detailed on page 25) the following sorted data will be produced:

Sorted data:

{4-6 11, 1-3 12, 3-6 15, 5-6 18, 2-5 21, 1-2 25, 3-4 28, 2-4 30}

|  |  |
| --- | --- |
| Node Details (x-y weight) | Part of MST (Minimum Spanning Tree) (True or false) |
| 4-6 11 | True |
| 1-3 12 | True |
| 3-6 15 | True |
| 5-6 18 | True |
| 2-5 21 | True |
| 1-2 25 | End n-1 arcs on MST |
| 3-4 28 | - |
| 2-4 30 | - |

#### Sorting algorithms

##### Quicksort

The quicksort has a complexity of 0(n log n) on average best and 0(n2) at worst because the input will be random the average complexity will be used a lot. Source: [https://www.cs.auckland.ac.nz/courses/compsci220s1c/lectures/2016S1C/CS220-Lecture10.pdf last accessed 21/09/2019](https://www.cs.auckland.ac.nz/courses/compsci220s1c/lectures/2016S1C/CS220-Lecture10.pdf%20last%20accessed%2021/09/2019)

To execute a quicksort on a list the list’s size is taken if it is 0 or 1 then the list is returned as it is. Otherwise the pivot is found, the first element is used as the pivot as it ideal for a random dataset (the middle element is used for a non-randomised dataset).

The value of the pivot value is found and then the list is iterated (excluding the pivot element), for each element if it is less than or equal to the pivot it goes into list a, otherwise it goes to list b.

Following this, lists a and b are quick sorted then are merged together. Merging the lists has the pivot value added to list a then list b added to list a (list a is then returned). This will result in the output of a list sorted in ascending order.

###### Pseudocode:

|  |
| --- |
| Function quicksort (list data) returns list  If length of data <= 1 then return data  Pivot index = length of list div 2  List a, b = new list of integers  For x in data excluding data [pivot]  If x < data [pivot]  Add x to list a  Else  Add x to list b  A = quicksort a  B = quicksort b  Add pivot value to a  Add of b to a  Return a |

##### Insertion Sort

Execution of insertion sort is less efficient than the quicksort and has a complexity of 0(n2) because it has a loop within a loop each of which progress linearly.

The insertion sort works by generating a blank list and adding the first element of the unsorted list to the new list. Then for each element inserting it into the correct place. This involves the iteration of the blank list to find the correct place to insert.

###### Pseudocode:

|  |
| --- |
| Function insertion\_Sort (list data) returns list  Output = list  Output[0] = data[0]  Max = 1  For x in data  For counter = 0 to max  Changed = false  If x < output [counter] then  Insert x to index counter of output list  Changed = true  Else if counter == max then  Add x to output list  Increment max by 1  Return output |

##### Bubble Sort

Bubble sorting is the go-to of bad algorithms and has a complexity of 0(n2) just like the insertion sort but runs very poorly compared to the insertion sort so is always a bad choice of sorting algorithm. For more data on the performance of the bubble sort see below, the ‘Sort Time Comparison’.

The bubble sort iterates through the data and for each element iterates on the elements ahead of it in the array and when it finds one greater than its value it swaps place with it, until a pass with no swaps is executed the algorithm continues to loop.

###### Pseudocode:

|  |
| --- |
| Function bubbleSort(array data) returns array  Changed = true  While changed  Changed = false  For I = 0 to length data  For J = I to length data  If data[i] > data[j] then  Swap elements I and J  Changed = true  Return data |

## Counting Sort

The counting sort requires data to have discrete categories for it to work. Categories are mapped to the index of an array to count how many entries of that category exist, this is often done with a hash map or with integer categories. The data is then iterated through and for each value the counter for its category is increased. After the data has been iterated through the output array or list is then constructed. This has complexity O(n + k), n = the number of elements in the input array, k = the number of categories. Source: <https://www.geeksforgeeks.org/counting-sort/> last accessed (25/11/2019).

###### Pseudocode:

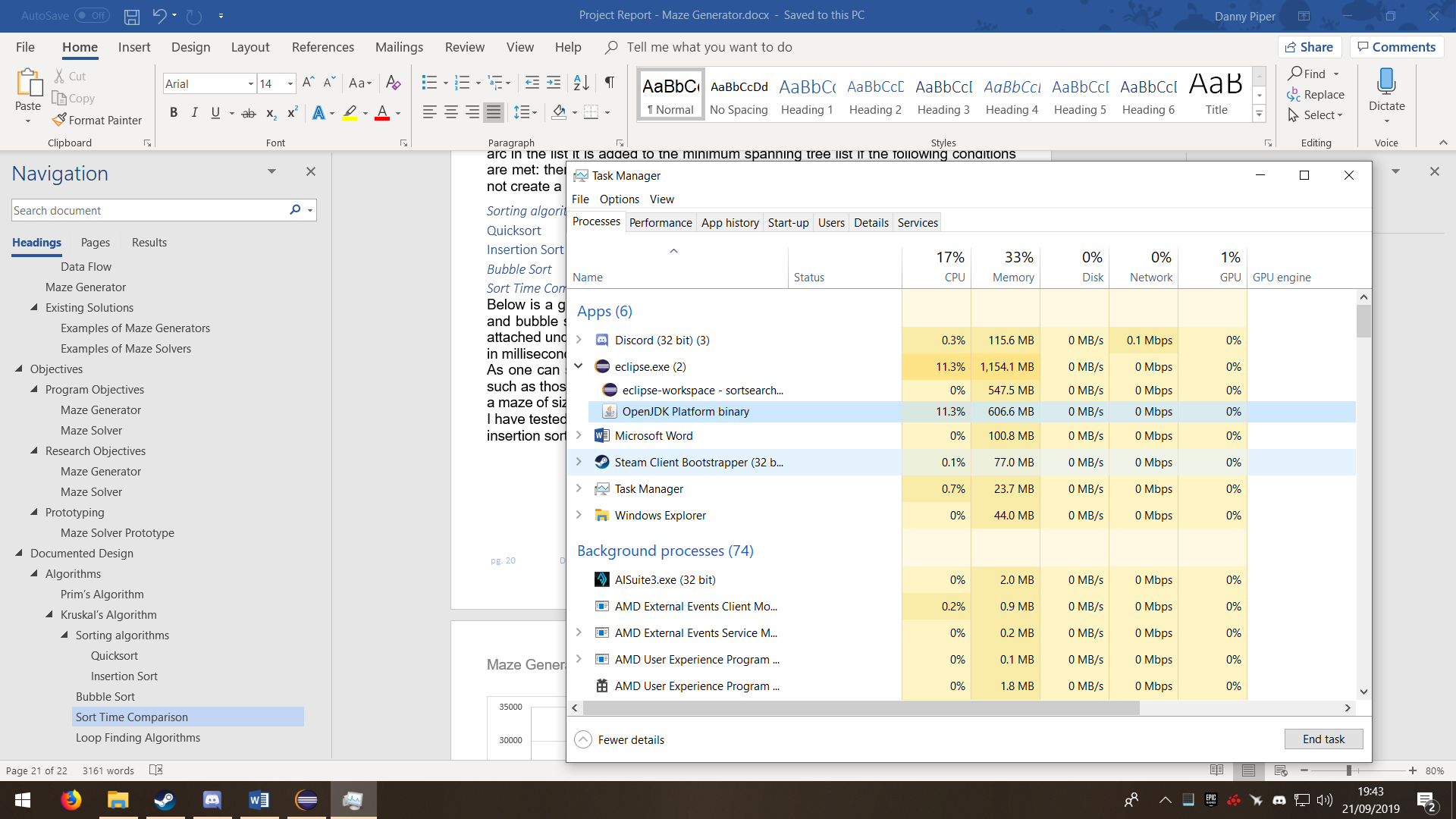
|  |
| --- |
| //Sorting an array of chars  Function countingSort(array of char: data) returns array of char  countingArray = array of integer[256]  set all values in countingArray to 0  for char c in data  countingArray [ord(c)]+=1    outputArray = array of char[data.length]  counter = 0  value = 0  //Add the data in the correct quantities to the outputArray  while value < 256  innerCounter = 0  while InnerCounter < count  outputArray [counter] = chr(countArray[value])  counter+=1  innerCounter+=1  value+=1  return outputArray |

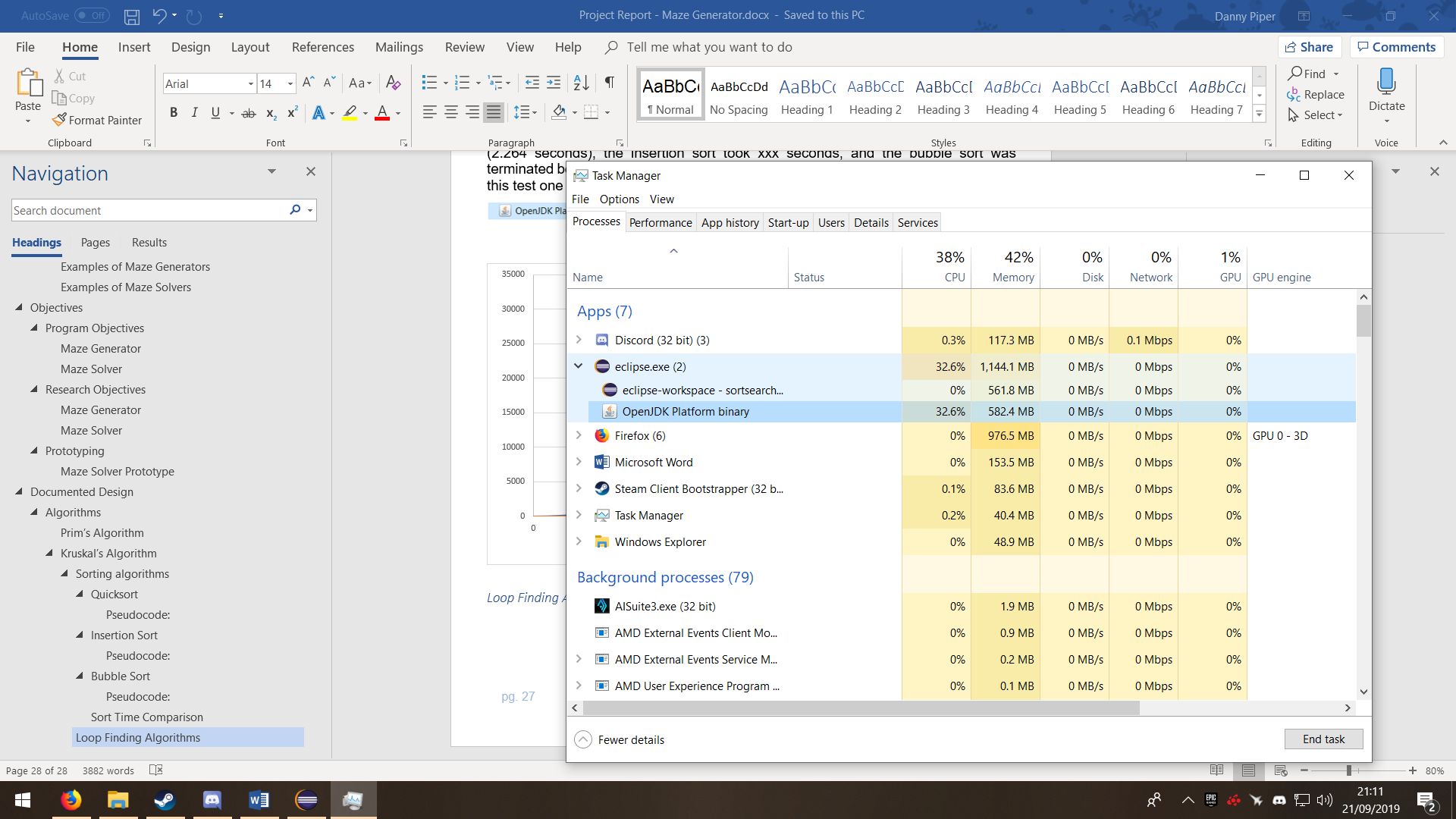
##### Sort Time Comparison

Below is a graph of to show the performance differences of quicksort, insertion sort and bubble sort, the following is experimental data of mine and the source code is attached under the program name of ‘*sortSearch’*. On the y-axis the time taken to sort in milliseconds is given and the bottom is the number of elements in the array tested.

As one can see the bubble sort is incredibly slow and ineffective for large datasets such as those required for a maze. A maze of size nxn there are 2x(n-1)2 arcs. Given a maze of size 1000, there are

I have since tested for a dataset of this size and the quicksort took 2264 milliseconds (2.264 seconds), the insertion sort took over an hour before the program was terminated, and the bubble sort was terminated after an hour and a half. Whilst executing the bubble test one of my twelve cores was maxed out on CPU time for the entire time.



During the insertion sort test 4 of more cores were maxed out on CPU time for the execution period. 

#### Loop Finding Algorithms

## Disjoint Sets (Union/Find Algorithm)

A disjoint set is a set of sets which have null unions (no overlap between sets). This can be implemented with an array of pointers to point to parent nodes. The union algorithm finds the sets of the start node and end node of an arc (the arc that could make the graph cyclic) and finds the sets of each, if they are part of the same set then the union of sets would not be equal to null so the addition of the arc would create a cycle. The find algorithm will continue to go to the parents in the disjoint list until it reaches the top of the tree, it will then return the ID of that node for the union algorithm to use. Adding an arc to the disjoint set has the end node of the arc pointing at the start node pointing at the grandparent node (highest parent).

## Graph Traversal – Breath First / Depth First

The graph is traversed and for each node a Boolean stores whether or not the node has already been visited. If a node is visited whilst traversing the graph then the graph is cyclic. Due to the high complexity of graph traversal I shall not be implementing it in my solution.

# Documented Design – Maze Solver

### High Level OverviewSolving the Maze

#### Multi-Threaded Program Design

The program will create many threads and divide and conqueror paradigm to shorten compute time.

|  |  |  |  |
| --- | --- | --- | --- |
| Thread Name | Instances of Thread | Thread Description | Thread Started By |
| *mainThread* | 1 | This thread starts the other threads and manages the computation | Starting the program |
| *renderThread* | 1 | This thread renders the current maze preview | *mainThread* |
| *solverThread* | X, where X is the number of processor cores on the system. | This thread solves a pre-assigned chunk of the maze | *mainThread* |

Multi-Threading the system will be effective at enhances the performance because it can utilise more of the computational power that is available. Multi-threading will cause problems with thread timings; to combat this the thread objects will have Booleans as whether or not they have finished execution. By doing this it means that timings are easier to manage.

## User Interface Design

|  |
| --- |
| When the solve button is clicked it will grey out the filename input box, itself and the select file button before starting to solve the image. A cancel button is then placed.  The text box here allows the user to insert the filename that the image file with the highlighted path is saved to.  This button opens a user prompt where they select the maze image file to solve.  Render Space. This shows the render of the maze as it is being solved including the paths being considered highlighted in red. |

## Algorithms

### Dead End Filling

Given a graph, each node with 1 connection (other than the start and end nodes) are removed. This is repeated until nodes with 1 connection (again excluding the start and end nodes are the maze) have been removed.

Pseudo code for dead end filling.

|  |
| --- |
| Changed **← true**  While changed {  Changed **← false**  For each pixel{  If pixel is white and has one connection{  make it black  changed **← true**  }  }  } |

# Technical Solution – General

## Problems Encountered

|  |  |
| --- | --- |
| Problem Description | Project Solution |
| Swing (Java’s built in GUI library) being hard to use and buggy. | Changed to using the library JavaFX and the JRE required to run the program. (JRE SE 13 is now needed compared to SE 1.8). |
| Project not running on the college System due to an incompatible JRM and lack of JavaFX | By running the following command to launch my program and having a local copy of JDK (and the JRE) 13 and JavaFX  /jdk-13/bin/java.exe --module-path "javafx/sdk/lib" --add-modules java.desktop --add-modules javafx.controls --add-modules javafx.base --add-modules javafx.graphics --add-modules javafx.swing -jar mazeSolver.jar -Xms1G -Xss50M -d64 |

# Technical Solution – Maze Generator

## Problems Encountered When Creating the Solution

|  |  |
| --- | --- |
| Problem Description | Problem Solution |
| Render times for a scale of one being the same as render times with a different scale. This shouldn’t happen because the image needs less processing | An if statement that checks the scale is used before the render.    The images are set to null afterwards to save on memory. |
| Crashes when non-numerical values are put into the input-boxes | The Solution was to validate the input. |

# Technical Solution – Maze Solver

## Problems Encountered When Creating the Solution

|  |  |
| --- | --- |
| Problem Description | Problem Solution |
| RAM Usage – The program needs 1.5GB RAM for large datasets and is too fast on small datasets to test properly. | Test on my PC at home rather than the college computers because I have 16GB RAM (10GB free after Windows takes its fair share) as opposed to the 4GB on the college systems with about 1GB free. |