A-Level Computer Science 7517

Non-exam Assessment Project

Adam Biggs

Table of Contents

Analysis 7
- Identification and Background 7
- Defining Terms 10
  - Mathematical Terms 10
  - Maze Specific Terms 10
- Graph Representation of a Maze 11
- Memory Requirements 12
  - Maze 12
- Algorithms to Implement 14
  - Maze Creation Algorithms 14
  - Maze Solving Algorithms 15
- Preliminary Research 16
- Objectives for Potential Solution 18
- Data Flow Diagram 19
- Candidate technologies 19
  - Hand Drawn on Paper 19
- Feasibility of Potential Solutions 20
  - C# and VB.NET Languages and the Microsoft .NET Framework 20
  - Java 21
  - PHP & Cascading Style Sheets (CSS) 21
  - Windows Presentation Foundation (WPF) 22
- Chosen Solution 22
- References 22

Design 23
- Overall System Design 23
  - Inputs 23
  - Outputs 23
  - Processes 24
  - Storages 24
- Class Definitions 25
  - CellEdge and CellWall, CellPassage Inheritance 26
- Class Diagram 27
- Modular Structure of the System 28
- Flow of User Control 29

Identification of Appropriate Storage Media 30
Images 30
Text File Structure 31
Algorithms 33
Validation Algorithms 33
Creation of Text Files 33
Maze Generation Algorithms 36
Maze Solver Algorithms 38
Class functions and Procedures 39
Display Algorithms 40
Complexity Calculation 41
Human-Computer Interface 42
Security & Integrity of Data 43
System security 43
Test Strategy 43
Implementation 44
MazeLibary Project 45
Core 45
Cell 45
CellEdge 48
Cell Passage 49
Cell Wall 50
Direction 50
Grid 51
Events Arguments 56
CellPassageCreatedEventArgs 56
CellWallCreatedEventArgs 57
HighlightPathAddedNewCellEventArgs 58
RoutePathAddedNewCellEventArgs 59
Helpers 60
ConsoleHelper 60
DirectionHelper 63
Maze Algorithms 65
Algorithm Base 65
AStarAlgorithm 67
BinaryTreeAlgorithm 71
DrunkardsWalkAlgorithm 74
HuntAndKillAlgorithm 78

KruskalsAlgorithm 81
PrimsAlgorithm 84
RecursiveBackTrackerAlgorithm 86
RecursiveDivisionAlgorithm 91
SidewinderAlgorithm 96
SimpleRectangleAlgorithm 100
WallFollowerAlgorithm 101
WilsonsAlgorithm 107
Mazer Project 111
Main Windows and Dialogs 111
Multi-Threading 111
MazeGeneratorWindow 111
MazeGeneratorWindow.xaml 112
MazeGeneratorWindow.cs 116
MazeSolverWindow 134
MazeSolverWindow.xaml 134
MazeSolverWindow.cs 138
MazeStartUpWindow 155
MazeStartUpWindow.xaml 155
MazeStartUpWindow.cs 155
View Models 157
MazeGeneratorWindowViewModel 157
MazeSolverWindowViewModel 162
ResultsWindowViewModel 167
Model 169
AlgorithmMetric 169
Helpers 171
PathHelper 171
SaveImageHelper 173
Enums 176
MazeAlgorithm 176
Converters 177
HalfValueConverter 177
RectConverter 178
MazeLibraryUnitTests 179
AlgorithmUnitTests 179
AStarAlgorithmUnitTest 179
BinaryTreeAlgorithmUnitTest 179

DrunkardsWalkAlgorithmUnitTest 180
HuntAndKillAlgorithmUnitTest 182
KruskalsAlgorithmUnitTest 182
PrimsAlgorithmUnitTest 183
RecursiveBackTrackerAlgorithmUnitTest 184
RecursiveDivisionAlgorithmUnitTest 185
SidewinderAlgorithmUnitTest 186
SimpleRectangleAlgorithmUnitTest 187
WallFollowerAlgorithmSolveLeftUnitTest 188
WallFollowerAlgorithmSolveRightUnitTest 189
WilsonsAlgorithmUnitTest 189
CoreUnitTests 190
CellEdgeUnitTest 190
CellUnitTests 191
DirectionHelperUnitTests 192
GridUnitTests 193
Helpers 196
AlgorithmTestBase 196
GridFactory 198
Test Data Files 200
Test Maze 17X13 (RecursiveBacktracker) [.maz] 200
Test Maze 25X20 (RecursiveDivision) [.maz] 203
Test Images 211
Test Maze 17X13 (RecursiveBacktracker) 211
Test Maze 25X20 (RecursiveDivision) 212
Testing 213
Test strategy 213
Unit Testing 213
Manual Testing 213
Core Unit Testing 214
AlgorithmTestBase 214
Figures 230
Algorithm Unit Testing 233
UI Manual Testing 257
Button Testing 257
Window Initialisation 257
UI Canvas Manual Testing 260
Generation 260

Solving 262
Figures 264
Other Manual UI Tests 296
Generating 296
Solving 296
Figures 297
Evaluation 302
Meeting Objectives 302
The Results 304
Generator 304
Complexity 305
Solver 306
Suggested Improvements 309
Conclusion 309

Analysis

Identification and Background

Puzzles can be complex or trivial. There are hundreds of examples of puzzles ranging from the quite simplistic such as Snakes and Ladders or Noughts and Crosses through to much more complex examples such as Chess or the Rubik's Cube. A particularly interesting category of puzzle are mazes since they are a puzzle where the complexity is not always constant.

A maze's complexity is defined at the time of its creation. If created one way it can be very simple if created another way it can be very complicated. No matter what way a maze is created the goal is the same, to reach the end from the start in as short a time as possible while having fun as you go. I will, therefore, develop an investigatory program that will allow the exploration into the fascinating world of the maze and their many forms.

Mazes come in many shapes and sizes. They can by very large or very small, complex or quite trivial, ordered or a jumbled mess, 2D or 3D and so on. Even the examples listed above can be expanded further, 3D mazes could be several 2D ones stacked, mazes set on the faces of some larger shape like a cube, or even be recursively built using some sort of fractal behaviour.

Many people enjoy creating mazes and like to challenge their friends and family with the mazes that they come up with. Teachers may also set a maze creation or solving challenge to their students. However, the creation of a maze can be time-consuming and a manual process. Maze creation is often undertaken haphazardly through randomly drawing lines or alternatively quite carefully and precisely on graph paper. Someone creating a maze by hand has to make each hallway a specific size while also providing an interesting and challenging path to follow to reach the end.

When mazes are to drawn precisely a plan or recipe can be followed to produce a maze that is both challenging and fun. In computer science and mathematics, these recipes are referred to as algorithms. There are many maze algorithms one might use to produce interesting mazes of varying complexity. These include the Recursive Backtracker algorithm, Prim's algorithm, Drunkard's Walk algorithm to name just a few. Additionally, the same maze algorithms can often be used to solve mazes as well as create them.

Starting with the least complex algorithm, the Drunkard's Walk, we can understand how it helps a person generate a maze by considering the following key algorithm steps: randomly draw a new point to add to the maze until the maze has a specified size. This algorithm is often used when creating non-looping mazes which is a concept that I will come back to later.

The Recursive Backtracker algorithm is an extension of the Drunkard's Walk algorithm. It follows the same basic approach but with a more efficient plan of action when it gets blocked. When a person has randomly created a path that happens to get blocked (which might happen if the path spirals into itself or into a previously created wall) then the person moves back along the existing path to a place where they will be able to once more draw a new point that extends the path into available adjacent space. This tends to create longer offshoot paths than the Drunkard's Walk algorithm. It is faster algorithm in comparison to the Drunkards Walk since it does not rely on randomness when it gets stuck.

Lastly, those that use Prim's algorithm draw mazes differently, for this algorithm isn't based on the Drunkard's Walk. Firstly, a point in the available space is chosen as a starting point. Then all the

available spaces around that point are noted down. Next, a random number generator is used to pick between these neighbour points and when one is chosen it is linked to the set of drawn points, in this case, the starting point. This process is then repeated for the new group of available spaces now with each point next to the two linked points. This then continues until there are no more available spaces to draw in.

When a person wants to create a maze and is provided with descriptions of maze algorithms as in the previous description they can easily be dissuaded from even starting. Often the time taken to create a maze by hand using one of these algorithm is so long it would be near impossible to create one of any large and truly interesting size. A significant problem when creating mazes by hand is the challenge of having to carry out simple but very many repeating steps accurately. It takes a lot of time and patience to create a maze by hand and one cannot even be sure that the result will meet the criteria for complexity. A maze may turn out to be a very easy one where a more difficult one was required, similarly, the maze produced may turn out to be far harder to solve than what was required for the intended audience.

As previously mentioned, a person needing to solve a maze can also use one of the maze creation algorithms to go about solving it. This is done by simply ticking off spaces that have been visited instead of creating new parts of the maze. For example, the Recursive Backtracker now works by randomly feeling out ways through the maze space by space and returning to where it can start a new path each time it fails to find the finish. Likewise, with the creation of a maze, the solving of a maze can easily take a lot of time and resources. Short and simple mazes can be solved in a matter of minutes but large and complex mazes may take hours or even days to solve by hand.

No matter what method a person uses to create or solve a maze there is always a question of what would be the best algorithm to use that will meet the required criteria. The most typical criteria are the size the final maze will be and the difficulty or complexity of the solution to the maze. Other criteria might be the time it takes for the maze to be created or even how pleasing to the eye the final maze is when it is finished.

My project will be to create a computer program which will allow the exploration into the fascinating world of mazes. It will allow the generation and solving of any maze. It will contain implementations of many of the most well-known maze algorithms already discussed and several more in addition.

- The maze creation and generation aspect of the solution will allow the user to select a desired maze creation algorithm as well as a maze size.
- Once a maze has been created it will allow the user to view it and check its features visually.
- Mazes will be able to be saved and reloaded later.
- The solution will provide a sense of the complexity of the maze as a number and provide accurate details of how long it took to make.
- The solution will provide a mechanism for solving mazes. The user should be able to select any of the algorithms to use for solving the maze.
- Maze solutions will be able to be displayed on the maze. The correct solution to the maze will be highlighted. The elapsed time required to for the solution to be found will be included.
- The ability to compare algorithm performances laying out the times in a graph or similar diagram will be included.
- A mechanism to allow the user to save the maze image to a file (in bitmap or png format) with a specified user-defined name will be included.

Potential users of the solution that I build could be Game designers, many of which require randomly generated mazes or dungeons of varying size and complexity. Computer scientists testing the limitations of the artificial intelligence in robots who might need randomly generated mazes of a specific size and complexity might also find the solution useful. Scientists wishing to test the intelligence of animals by sending them through a real 3D maze may use this solution to plan an appropriate 2D maze with sufficient complexity and size.

Defining Terms

The following terms define the common names and phrases used throughout the solution.

Mathematical Terms

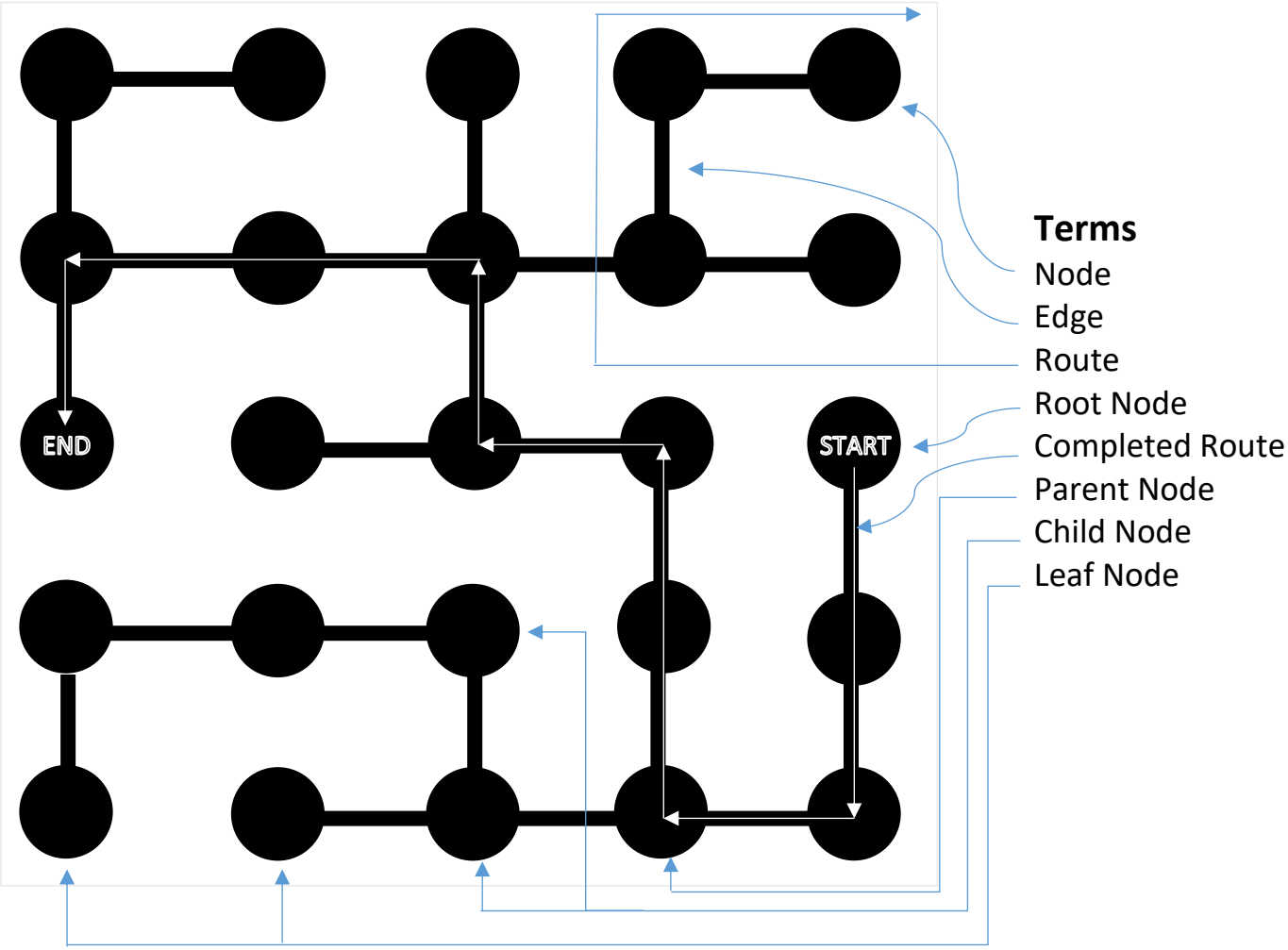
Term	Description
Graph	A mathematical structure that models relationships between pairs of objects.
Undirected Graph	A graph where the relationship between nodes is two way.
Tree or Spanning Tree	A special instance of a graph that has no loops or cycles.
Node / Vertex / Cell	An object in a graph.
Arc / Edge	A join of relationship between nodes.
Root Node	The start node in a tree structure from which all other nodes branch off from.
Route	A path through a maze in terms of an adjacency list of nodes.
Completed Route	The path that completes the maze starting from the root node and creating a pathway to the end node.
Cycle	A part of a graph structure that has multiple routes to get from one node to another. A tree graph cannot have any of these.
Parent Node	A node in a tree that other child nodes branch down from.
Child Node	A node in a tree that has nodes above it in the hierarchy.
Leaf Node	A node with no other node beneath it.

Maze Specific Terms

Term	Description
Maze	An undirected graph of nodes and edges modelled as a tree.
Complexity	A numerical value given to score how complex or complicated a maze is to solve. The calculation of the Complexity value is defined in the Design section.

Graph Representation of a Maze

Maze generation will rely on the creation of an undirected graph. This is a tree of nodes and edges spanning out from a root or start node and producing several routes. One route as a completed route will reach the finish node. Any node will hold a collection of child nodes each of which will have their own collection of child nodes.



Memory Requirements

In the following section, I estimate the typical amount of memory that will be required to store a small, medium and large maze in memory. I will assume that an object reference takes 4 bytes (32 bits) of memory.

Class Name	Data Name	Data Type	Data Size	Total Size
Maze	Array of Nodes	Array of References	4 Bytes x Size of Array	See Below
Node	North Edge	Reference	4 Bytes	16 Bytes
	East Edge	Reference	4 Bytes	
	South Edge	Reference	4 Bytes	
	West Edge	Reference	4 Bytes	
Edge	Start Node	Reference	4 Bytes	9 Bytes
	End Node	Reference	4 Bytes	
	Edge Type	Boolean	1 Byte	

- Each node has an object reference to four edges, each of 4 bytes.
- Each edge has an object reference to a start node and an end node each of 4 bytes. In addition, an edge also has an Edge Type which is a Boolean of size 1 byte which is used to indicate if there is a wall between two nodes.

Maze Grid Size	Rows	Columns	Number of Nodes	Maze Node Array (Bytes)	Node Data (Byes)	Edge Data (Bytes)	Total Data Size (Bytes)
Small	5	5	25	100	400	900	1400
Medium	10	10	100	400	1600	3600	5600
Large	25	25	625	2500	10000	22500	35000

The maximum expected size of a created maze is large (25 rows by 25 columns) and would require approximately 35000 Bytes of space or 35 Kb when rounding to the nearest Kb.

Algorithms to Implement

Maze Creation Algorithms

The following table provides a brief description of the maze creation algorithms along with their relative complexity.

Algorithm Name	Description	Relative Complexity
Drunkard's Walk	Moves randomly from node to node creating edges between them. Every time it moves to a node that hasn't been visited yet, it adds that node to the spanning tree, along with the edge used to reach it. And when all the nodes have been visited, the maze is done.	Low
Wilson's Drunkard's Walk	Moves randomly from node to node creating edges between them. Once a completed route is found from a specified start node to a specified end node, it is added to the spanning tree, and the process repeats. Except for each time around, any point already in the tree is a valid ending node so subsequent iterations complete much more quickly.	Low
Binary Tree	For each node in the graph, it randomly decides if it should add an edge north or an edge to the west. The first row will be a straight line of connected nodes as there is no way to create an edge north so it will create nodes only west.	Low
Sidewinder	This algorithm looks at groups of adjacent nodes. Each group of nodes is added to a connected set of the spanning tree, then one node is chosen at random in each group to create an edge northwards.	Low
Hunt and Kill	Starts by applying the Drunkard's Walk, but with one constraint; it may never revisit a previously visited node. This process would eventually create spiralled paths into the side of the maze or into itself. To solve this, the program then searches through the entire maze row by row from top to bottom until it finds an unvisited node next to a visited node. It then connects them and continues the path from this node still following the constraint above. When no unvisited nodes are detected the maze is done.	Medium
Recursive Backtracker	Starts also by applying the Drunkard's Walk with the constraint that it may never revisit an already visited node. The program also stores a list of each node it visits in a stack placing each new node on the top. Once it has spiralled into itself or the edge of the maze, it pops the top nodes off the stack until if finds one that it can make a new route from.	Medium
Recursive Division	This algorithm works by chopping the field of all connected nodes in half, by removing all the links between nodes bar one, and then chopping each new sub-group in half and so on until the maze is created.	Low

Algorithm Name	Description	Relative Complexity
Krustal's	Works by randomly connecting adjacent nodes with one restraint which is that two connections or more cannot be made between two nodes that are already in the same group. This occurs until the maze is complete.	Low
Prim's	A root node is chosen at random and is added to the spanning tree of visited nodes. Then it marks all its adjacent nodes as "frontier nodes". On the subsequent passes, it chooses one of the frontier nodes at random and makes a link to it, adding it to the spanning tree and marking all its adjacent nodes as frontier nodes. This process continues until there are no more frontier nodes.	Medium

Maze Solving Algorithms

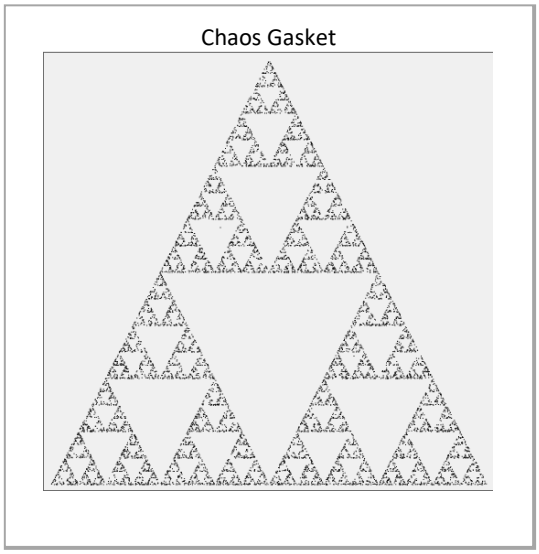
The following table gives a description for each of the maze solving algorithms.

Algorithm Name	Description
Drunkard's Walk	Moves randomly from node to node adding the new node to a stack. If it visits a previously visited node, it removes the latest node from the stack. When the latest node is the end node, the maze is done.
Recursive Backtracker	Starts also by applying the Drunkard's Walk with the constraint that it may never revisit an already visited node. The program also stores a list of each node it visits in a Stack placing each new node on the top. Once it has spiralled into itself or the edge of the maze, it pops the top nodes off the stack until if finds one that it can make a new route from.
A* or A-Star	It solves mazes using a heuristic calculation of the distance the current node is from the end node in terms of the Manhattan distance (the x and y distances added). The solver will, using this information, prefer moving along edges where the next nodes Manhattan distance is the lowest compared to all other nodes that can be reached. This algorithm is often considered the fastest algorithm when getting the shortest path along a graph, but in this application, it may not always be the best.
Follow Walls (Left)	Continually priorities taking the left edge from every node it visits to go next. If that node has no left edge, it then tries forwards, then right, then the way it came. This eventually gets it to the end node.
Follow Walls (Right)	Continually priorities taking the right edge from every node it visits to go next. If that node has no right edge, it then tries forwards, then left, then the way it came. This eventually gets it to the end node.

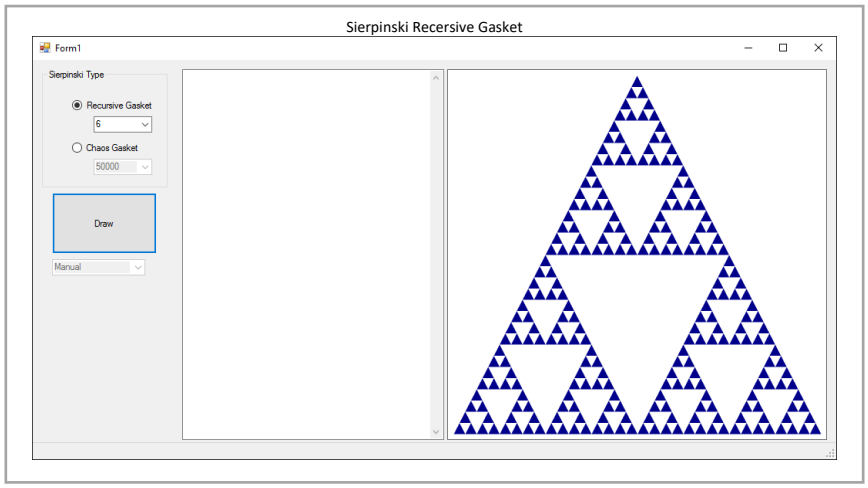
Preliminary Research

Before starting the design and implementation of the maze creation and solver application I first need to find the best technologies and approaches to use. To this end, I have decided to start by creating a Sierpinski Gasket generation application. I have done this as a research exercise so that I can learn more about algorithm implementation and to practice my C# programming skills. I also want to evaluate the various technologies that I am interested in using in my final solution.

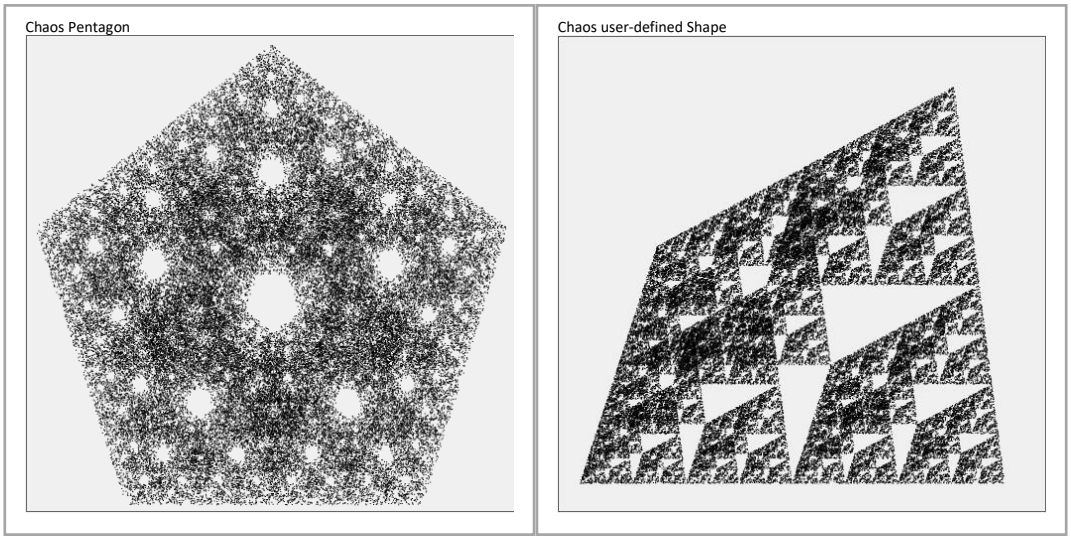
The Sierpinski Gasket application is, in itself, very interesting and I have learnt much about recursive algorithms and the creation of various fractal patterns. A screenshot of the research programs output in its initial stages as I developed it is shown below.



I have proceeded to look for alternative tasks involving Sierpinski style shapes to further my knowledge of the technologies involved. I read about the chaos method of creation and so incorporated that into the program as well. These tasks were complex to understand and it was here that I have learnt much about how to use generic classes such as List<T> and Dictionary<TKey, TValue> that are provided in the .NET Framework libraries. I have decided to use these in my maze solution. I have also discovered and used the PointF data type a floating point structure for storing an X and Y coordinate conveniently together. A screenshot of the research programs user interface is included below.



I expanded the program to allow for the options of creating a Chaos Square Gasket, the Chaos Pentagon Gasket and the finally the creation of any chaos shape determined by clicking on a PictureBox control with the mouse to select the corners of the desired Sierpinski Gasket shape.



The program's creation of the recursive triangle gasket generation works by using a function that takes three points (x and y coordinates as floating point numbers) and finds the nine new coordinates from the original triangle to make three new triangles inside it, this process then continues for the number of iterations a user has specified.

In comparison, the chaos gasket works by using a random point on the picture box as the starting position and gets the three points for the points of the triangle. Then through random number generation, it chooses one of these points to calculate the midpoint, colours this pixel black and then continues from this new position. This process, when applied a multiple hundreds of times slowly, generates the pattern of the Sierpinski Gasket. The concept is extended for each of the other types of chaos shape, the only difference being the coordinates of the corners of the shape.

Another important factor that came out of this exercise was that I have learnt the benefits of writing clear and concise code that can be used multiple times in the same program. I have learnt much about recursion and the value of using subroutines and functions in implementing this research program.

This preliminary research has helped me to formulate the approach that I now plan to follow when I design and implement the maze creation and solving solution later.

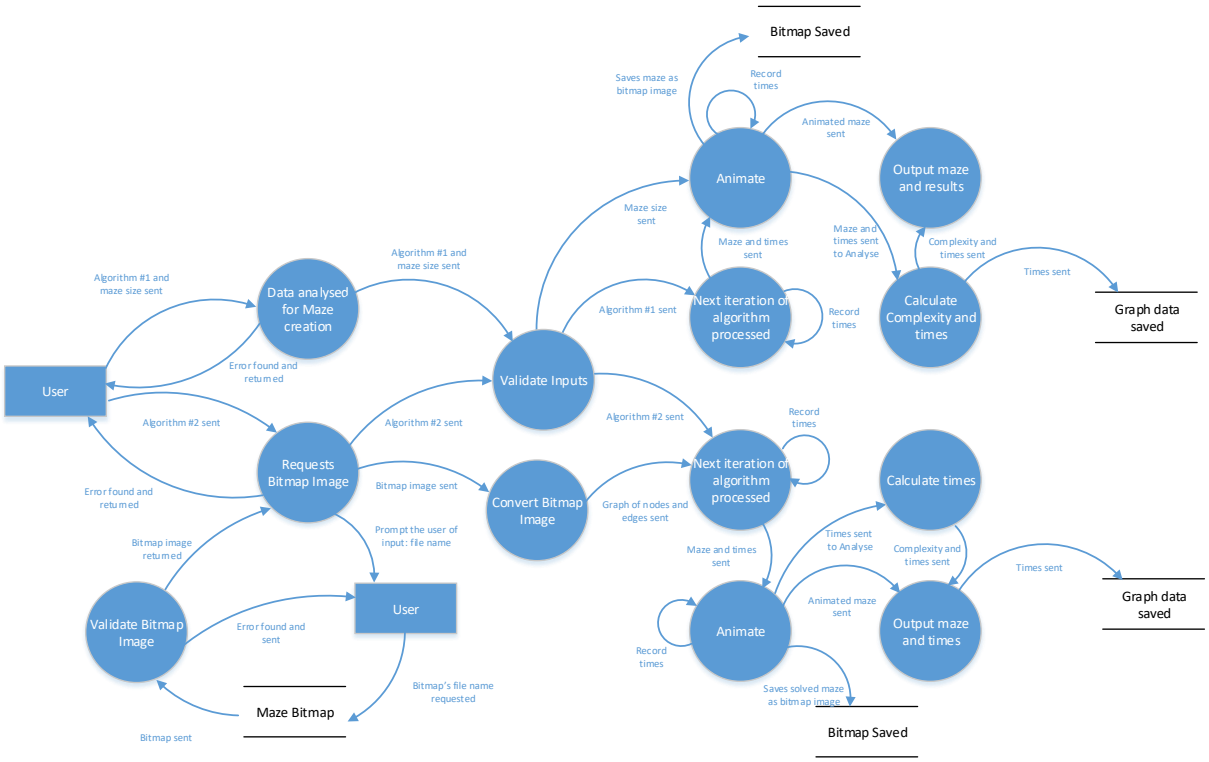
Objectives for Potential Solution

The following table details the main objectives for the maze exploration application that I intend to build that have resulted from my analysis.

Objective Number	Description
1	Creation of a user interface, which includes the creation of a start-up menu, a maze creation dialog window and a maze solver dialog window. Should include a simple way to select algorithms as well as any other required buttons or UI elements needed by the various windows.
2	Inputs into the user interface must be validated to ensure smooth running of the solution. Inputs such as the entering of maze size, for example, must take only integer values and should also be validated for appropriate maximum and minimum values.
3	All algorithms written must produce undirected tree graph for generation, and produce a clear route from a designated start to a designated end for solving.
4	The generation of a maze should be shown on screen, perhaps updating after each new addition to the maze. This should be clear and flow smoothly without screen flicker or pausing background processes.
5	All algorithms must not require extensive time to generate or solve mazes. This only applies when an algorithm takes more than a minute for a small grid size (e.g. 10 by 10).
6	All algorithms should, within reason, be fair. Realistically some algorithms may apply methods speeding up processes before or after algorithms create or solve a maze, where others don't, placing them at an unfair advantage. This should be avoided where possible.
7	The program must calculate the time taken to create and solve the mazes. It should be capable of creating a histogram (either directly in the application or possibly in graph in Microsoft's Excel) which can be viewed in the application itself.
8	The solution must be able to save mazes generated as a bitmap and also be able to open these maze bitmap files in the application.
9	The solution should display a complexity value for the mazes that it creates.

Data Flow Diagram

The following diagram details the flow of data that will be required in my solution.



Candidate technologies

Hand Drawn on Paper

It is possible for the creation of mazes to be done by hand using pen and paper. The careful user could even incorporate the use of the algorithms that I have previously explained. The major constraints to a person using this method to create mazes this way is the time it would take. For example, if a person were to try and make a maze that was 10 nodes by 10 nodes using the Sidewinder algorithm it would take tens of minutes as they repeatedly draw new lines and remove old ones. Even if an algorithm like the Recursive Backtracker were used a lot of time would be wasted retracing nodes in the current route. If a person required a maze of size 1000 by 1000 an exponentially larger time requirement would be needed.

The calculation of the maze complexity also requires considering every node in each route and assessing it would also take a further length of time.

It is obvious to me that maze creation is a task more suited to computers than humans. The task requires repeatedly completing simple steps of a larger task and to doing them quickly yet with no error. This is a job ideal for a computer.

Feasibility of Potential Solutions

C# and VB.NET Languages and the Microsoft .NET Framework

C# is a Microsoft developed language with a similar syntax to C++ and other members of the C language family. VB.NET is Microsoft's .NET language that is based on the BASIC language syntax but with many object-oriented features included. The relative performance and complexities between C# and VB.NET are negligible. They both share the same .NET Intermediate Language (IL), the same Common Language Runtime (CLR) and the same class library (Microsoft .NET Framework). In fact, code written and compiled into libraries in one language can easily be used from the other.

The choice between the two is more one of a personal syntax preference than for any other reason. It might be argued that C# has a marginally more concise syntax than VB.NET and has less clutter on the screen. C# is my personal preference for this reason.

I am sure I could use either C# or VB.NET for my overall approach to creating a potential solution with equal success. C# (and VB.Net) when used in conjunction with the Microsoft .NET Framework have more than enough Object-Oriented features and functionality to allow for an efficient and maintainable program to be produced.

I personally favour programming using Visual Studio for an integrated development environment (IDE) using the C# and the .NET Framework libraries. This is due to me having built up some experience of using these technologies during my Computer Science studies at school.

There is plentiful documentation and support for fundamental data structures such as Lists, Stacks, Dictionaries etc. and I know that all of these will be useful in implementing my intended solution.

Learning to program in a new language seems a pointless endeavour when a suitable framework already exists, especially one that I am already familiar with. This is unless the alternatives language offers a real benefit to my intended solution.

Java

There is much in command between Java and C#. Both Java and C# share similar C-style language syntax. Both are statically and strongly typed languages which enables the compiler to report many errors at compile time rather than run-time. Both are highly Object-Oriented languages, are class-based and both are designed with semi-interpretation or runtime compilation in mind. They also both use garbage collectors to manage memory usage when reclaiming variables that have gone out of scope unlike ANSI C or C++.

The following table is a comparison between Java and C#.

Language Feature	Java	C#
Classes	Required	Required
Abstract Classes	Yes	Yes
Interfaces	Yes	Yes
Implicit Conversions	No	Yes
Explicit Conversions	No	Yes
Operator Overloading	No	Yes
Cross Platform	Yes	No (On the way to becoming so)
Personal Knowledge in the programming language.	None	Several years programming in the .NET Framework and approximately a year programming in C#.

As shown in this table, both programming languages would be sufficient to develop the maze solution and to a high level of design quality and professionalism. The only point of which I have left to judge each language is the one that I am most familiar with. As this program does not need to be cross platform as is suggested in the objectives, there is no reason to consider Java as a potential language to use.

PHP & Cascading Style Sheets (CSS)

PHP is typically used as a server side scripting language but can also be used as a standalone language. It is true that the creation of the solution as a web based solution would allow this program to reach a far wider audience. Deployment of the finished solution would also be no problem since all users would simply require a modern web browser to use it. The potential to have several volunteers testing the functionality of the code each day would go far in finalising the project to a high quality which is also positive for a web based solution written in PHP.

However, there are several disadvantages when developing in PHP and CSS. Negatives include animation would be tricky to do while showing the maze generation on screen. Constant refreshing of the web browser is a danger. There is no requirement to have the solution available to anyone over the internet.

Also using the PHP language might place a constraint on available memory. This might limit the processing capabilities of the solution considerably, making it both unreliable and slower when running the algorithms for larger maze sizes. Finally, by developing this application as a standalone

desktop application there is no dependency on a web server or network always being available which is needed in a web based solution

Windows Presentation Foundation (WPF)

Windows Presentation Foundation (WPF) is a technology that is readily available as part of the Microsoft .NET Framework libraries. It offers a full graphical interface where user interface elements can be linked to their respected event handlers using any of the .NET languages such as C#. The user interface can made built using a designer view from within the Visual Studio IDE .The same GUI could also be made using the Microsoft .NET Windows Forms technologies but I favour using WPF in this case due to the graphical nature of the intended solution.

Chosen Solution

I will write the logic of the application in the Visual Studio's C# programming language and make extensive use of the .NET Framework using as many Object-Oriented programming features as I can. I will implement the user interface as a Window Presentation Foundation (WPF) application which has code support to animation and vector based graphics.

References

The following section list the references that I have found useful in this project.

1. Sierpinski Gasket - https://en.wikipedia.org/wiki/Sierpinski_triangle

2. Algorithm is not a 4-letter Word - http://www.jamisbuck.org/presentations/rubyconf2011/index.html

3. Maze Generator - http://www.mazegenerator.net/

4. Comparison of C Sharp and Java - https://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Java

5. Windows Presentation Foundation (WPF) - https://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx

6. What is PHP? - http://php.net/manual/en/intro-whatis.php

Design

The following sections include my design (including descriptions and explanations) of the classes and algorithms that will be required to implement the intended application. A proposal for the user interface design is included and a consideration of the security aspects of the application is included. Finally, to ensure the code that I write meets the needed objectives, the details of the test strategy that I will be using is also included.

Overall System Design

The following tables describe the inputs, outputs, processes and storages that I will build to create the overall system.

Inputs

Input Name	Description
Algorithm Generate Maze	Input algorithms to generate the maze.
Algorithm Solve Maze	Input algorithms to solve a pre-entered maze.
Maze Size	Input the maze size for the generation of the maze.
Maze Conversion	Input a text file to convert to a graph in memory.

Outputs

Output Name	Description
Generate Maze	Output the generated maze bitmap.
Solved Maze	Output the solved maze bitmap, with a completed route path shown.
Complexity	Output complexity of the maze.
Time Generated Creation	Output time taken for the maze to be created.
Time Generated Display	Output time taken for the maze to be displayed.
Time Solved Creation	Output time taken for the maze to be solved.
Time Solved Display	Output time taken for the maze to be displayed, with the completed route.

Processes

Process	Details
Validate #1	Validate inputs of “Algorithm Generate Maze” and “Maze Size”. Also, validate “Algorithm Solve Maze”
Validate #2	Validate inputs of “Maze Conversion”.
Generate Maze Using Specified Algorithm	Create the maze of size “Maze Size” using the “Algorithm Generate Maze”.
Solve Maze Using Specified Algorithm	Solve the maze using the algorithm “Algorithm Solve Maze”.
Calculate Complexity	Calculate complexity using the number of branches and the number of nodes in each route of the maze.
Add Times	Record the time taken to produce or solve the maze to a running value of each of “Time Generated Creation”, “Time Generated Display”, “Time Solved Creation” and “Time Solved Display”.
Render Maze To Screen	Take the current graph from memory and draw it on the screen.
Save Generated Maze As Bitmap	Convert the maze into a bitmap representation and save it to disk using a Save File dialog.
Save Solved Maze As Bitmap	Convert the solved maze into a bitmap representation and save it to disk using a Save File dialog.
Save Generated Maze As Text File	Convert the maze into a Text File representation and save it to disk using a Save File dialog.

Storages

Storage	Details
Text File	A Text File representation of an in-memory maze object. Used for the long-time persistence of mazes. Using this a maze can be saved and then reloaded from disk during another session.
Bitmap #1	The bitmap picture of a generated maze.
Bitmap #2	The bitmap picture of a solved maze which shows its solution.

Class Definitions

The following table describes the various classes that have been designed to represent all the information required to represent a maze.

Class Definition	Function
CellEdge - startCell : Cell - endCell : Cell - direction : Direction + StartCell + EndCell + Direction + Initialise(startCell, endCell, direction)	Represents the connection between two cells. CellEdge requires a start cell, end cell and direction to be a valid edge between two cells. CellEdge is never directly created. It is a base class for CellPassage and CellWall. CellEdges of type CellWall or CellPassage are created and then added to the private collection “edges” in the Cell class.
CellPassage	Represents a cell passageway to another cell. The CellPassage class is inherited from the CellEdge class.
CellWall	Represents a cell wall between itself and another cell. The CellWall class is inherited from the CellEdge class.
Cell - name : string - point : Point - edges : CellEdge Array - intialisedEdgeCount : int - random : Random + IsFullyInitialised + Name + Point + Cell(name : String, point : Point) + GetEdge(direction : Direction) + SetEdge(direction : Direction, cellEdge : CellEdge) + RandomUninialisedDirection()	Represents a cell in the grid of a maze. Each Cell knows about its position as a point in the grid and holds a CellEdge array. The variable initialisedEdgeCount holds the number of edges the cell has that have either a CellWall or CellPassage assigned to them. Cell can optionally store a name which might be the cells grid position to help with debugging.

Class Definition	Function
<<Enumeration>> Direction North East South West	This enumerated type holds each of the four possible directions that can be used in a maze. CellEdges use this to indicate the direction between the start cell and the destination cell.
DirectionHelper - random : Random Integer - count : Integer - point : Point - opposites : Direction + RandomDirection() + GetOffset(direction : Direction) +GetOpposite(direction : Direction)	The DirectionHelper class is a helper class which is used to help with the Direction enumeration. It can calculate the offset to a Cell in each direction. It can return the opposite direction for a given direction. It also knows the total number of directions that are available.
Grid - Cells : Cell (Two Dimensional) Array - name : String - random : Random - size : Size + ContainsPoint(point : Point) + CreateCell(point : Point) + CreatePassage(startCell : Cell, endCell : Cell, direction : Direction) + CreateWall(startCell : Cell, endCell : Cell, direction : Direction) + RandomPoint()	Represents a maze in the form of a 2D array of Cells. The Grid knows its own size. The Grid supports the main routines for adding Cells and CellEdges to the maze that it represents as it is being created or traversed.

CellEdge and CellWall, CellPassage Inheritance

During the design effort, I have decided intention that would indicate whether a CellEdge is a wall or a passage by using **inheritance**. This is simply an implementation detail. For the various maze algorithms to know which type of CellEdge they are dealing with they can simply use the C# "is" language keyword. A typical example looks like this in the code.

```
if(cellEdge is CellPassage)
{
  //Do something
}
```

An alternative approach would have been to introduce a CellEdgeType property to the CellEdge class itself. This would have been an enumerated type indicating if the CellEdge is a wall or a passage.

Whenever an algorithm needed to know what type of CellEdge it was dealing with it could simply examine the property for its value. A typical example of code would look like this below.

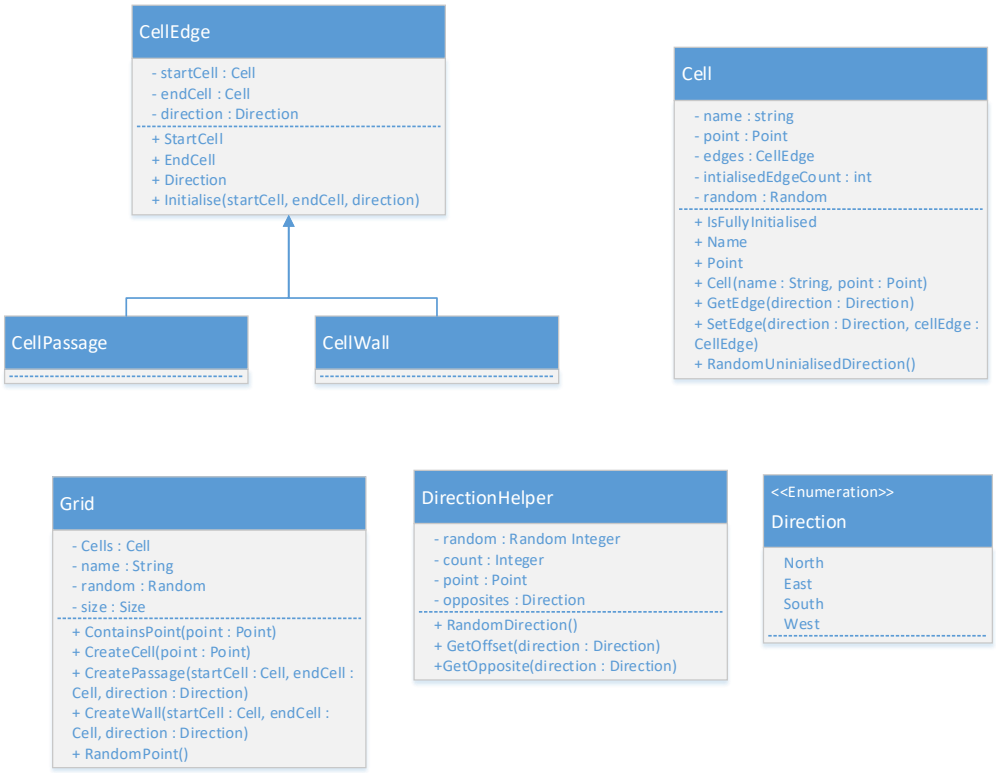
```
if(cellEdge.CellEdgeType == CellEdgeType.Passage)
{
  //Do something
}
```

As can be seen, there is little to decide between these two approaches. I chose to use the first or inheritance design in my application.

Class Diagram

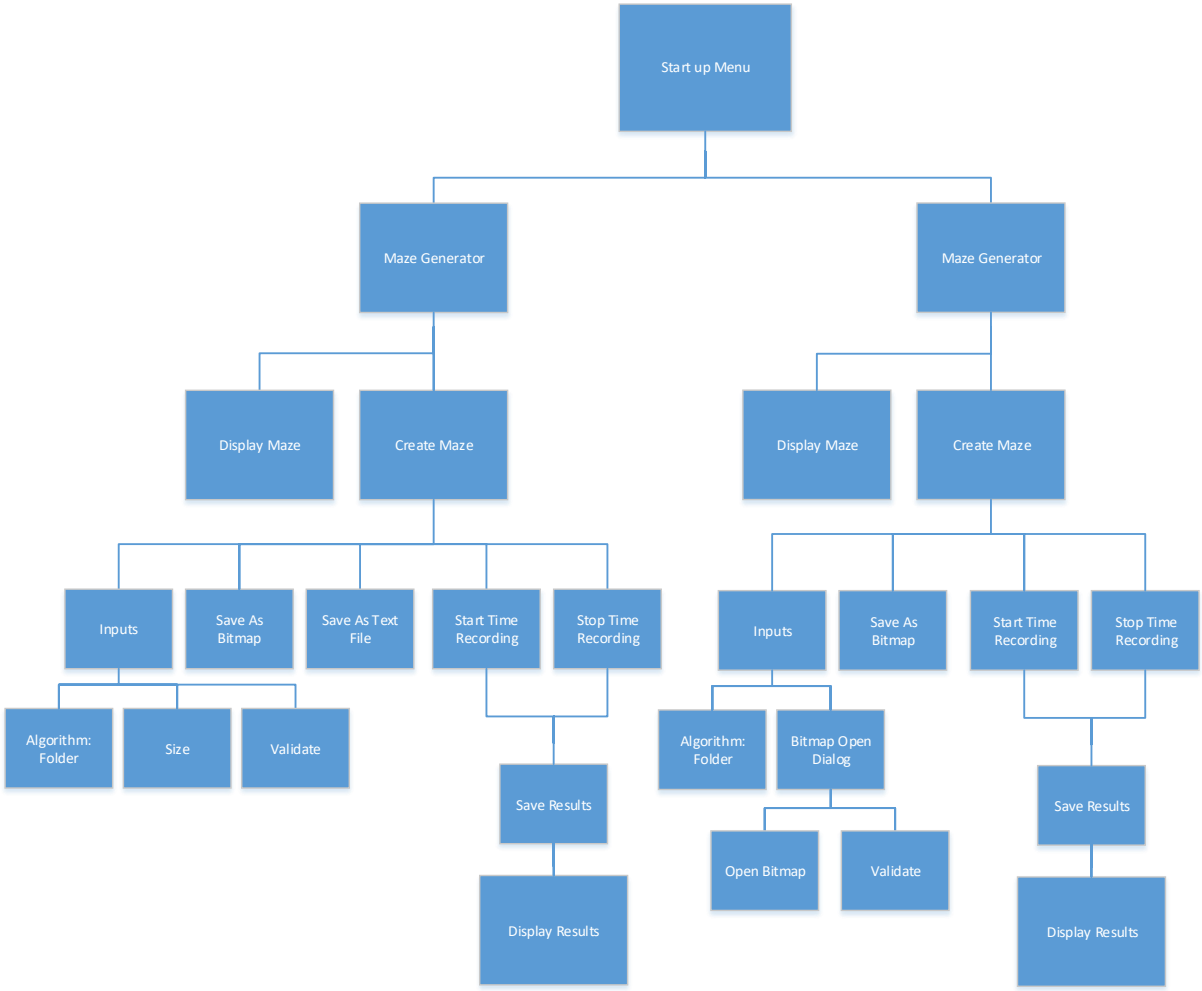
Below is a diagram showing how the CellPassage and CellWall classes inherit from the CellEdge base class. The other classes used in the design are also included. The diagram only shows the major classes that will be used in this program to represent the maze itself.

The various algorithms used to generate and solve the mazes make extensive use of these classes. For a textual description of these classes see the Class Definition section above.



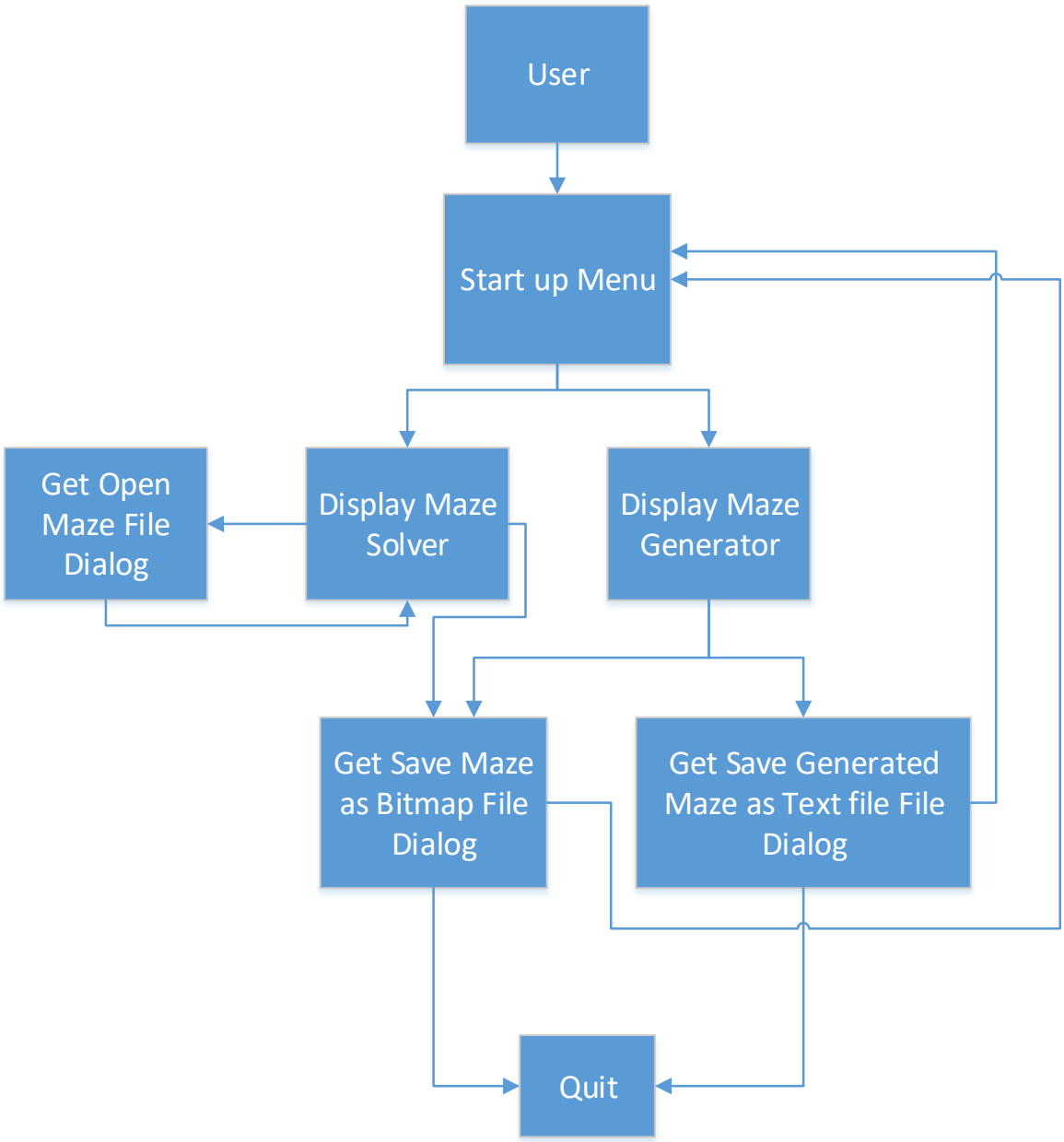
Modular Structure of the System

This is a hierarchy chart of the modular system design, showing the conceptual way menu structures and methods will be accessed from each other in the final solution.



Flow of User Control

Below is a system chart of how the menu structure will be presented to the user. It shows how each dialog will be reached and where the flow of control will return to when windows or dialogs are closed.



Identification of Appropriate Storage Media

In terms of the program itself, once finalised, the application will be compiled into an executable (EXE) and some dynamic link libraries (DLL's). It will be run by clicking on the EXE file. The application will be stored on the hard disk drive of the computer that it is running on. The total size required for this is estimated to be less than 300 KB. This is small because the application will be using the Microsoft .NET Framework which will already be installed on the computer as a prerequisite and so I have not included in the calculation for this storage requirement.

A folder for storing any generated and solved maze images will also be created. The storage requirements for these are small but depend on the number of images saved by the user. The user will be expected to carry out their own housekeeping in this folder if storage space on the computer is running low.

The application analysis requires two mechanisms for storing mazes on disk. These are in the form of Images and Text Files.

Images

The analysis calls for the application to be able to save images of both newly created mazes and solved mazes including their solutions. This will be done by using a **File Save** dialog and the built-in features of Image class provided by the Microsoft .NET Framework libraries.

The maximum size of a maze image will often be far below the maximum theoretical size that an image can be. For example, the largest size that a Bitmap image can be is in the region of 65,000 by 65,000 pixels. At a colour depth of 8 bits per pixel this gives a storage requirement of up to 17 GB. In practice, my application will draw mazes on the screen and will be much smaller than this upper limit. A typical maze drawn on the screen will be no larger than 1024 by 768 pixels with a colour depth of 8 bits per pixel which should require only 368 KB per picture.

I have concluded that this application is not expected to have any higher than normal storage requirements.

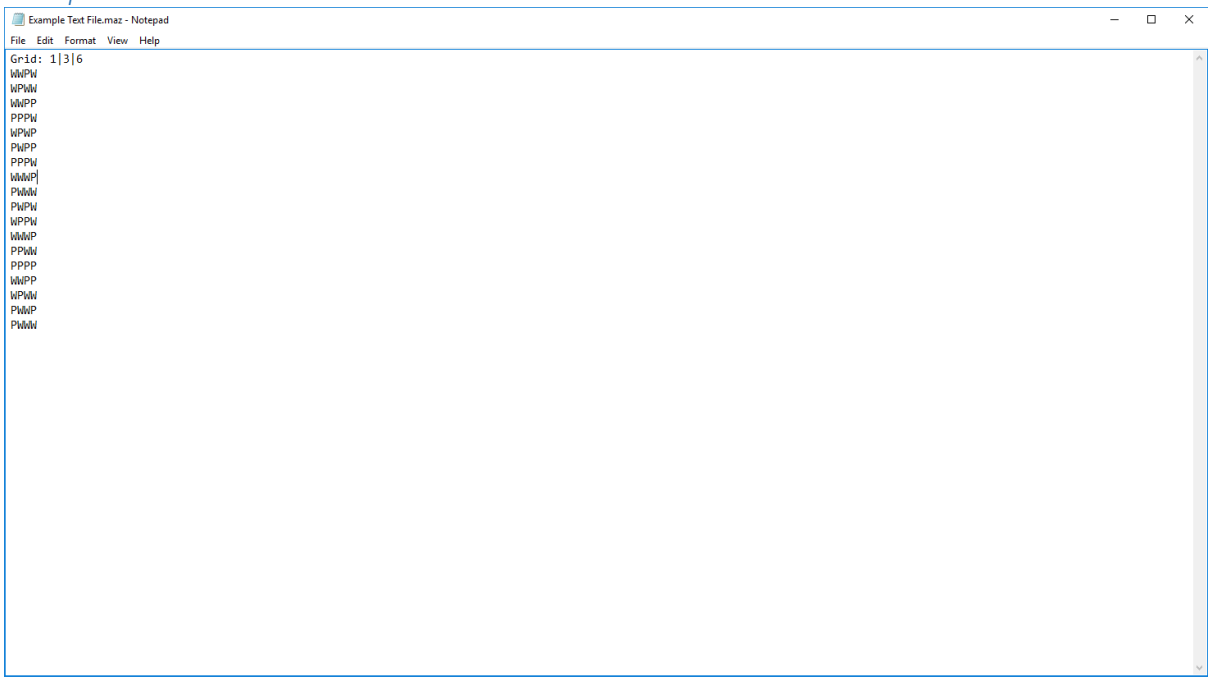
Text File Structure

The program will be able to convert the maze object in memory into a text file format for writing to disk. It will also be able to read the text file back from disk and recreate the maze from the information in the file.

The file structure for saving mazes that the program generates is described in the table below. Further examples of this file structure will be included in the Test section later.

<table>
<tr><th>Line Number</th><th>Example of Stored Information</th><th>Description</th></tr>
<tr><td>1</td><td>“Maze_1”|3|6</td><td>This line uses a vertical bar character as a delimiter to hold three pieces of information.
The first part is quoted characters which form the name of the stored maze and can be any length but must not include vertical bar character.
The second part is the Width of the maze.
The third part is the Height of the maze.</td></tr>
<tr><td>2</td><td>WPWW</td><td>This line constitutes the first cell in the maze, at position (0, 0). Each character refers to the direction of the edge connection. It is written as North, East, South, and West in this order. The meanings of the characters are shown below:
<table>
<tr><th>Character</th><th>Meaning</th></tr>
<tr><td>W</td><td>Edge is a Wall</td></tr>
<tr><td>P</td><td>Edge is a Passageway</td></tr>
</table></td></tr>
<tr><td>3</td><td>WPWP</td><td>This line refers to the cell at position (1, 0) and it has a Wall edge North, a Passageway both East and West, and a Wall South.</td></tr>
<tr><td>4</td><td>WWPP</td><td>Cell position (2, 0) is the last cell on this row. This is known since line 1 holds the Width of each row which in this example is 3.
The next line, therefore, must be on the next row as is shown in the coordinates of the next cell.</td></tr>
<tr><td>5</td><td>WPPW</td><td>This cell has a position of (0, 1).</td></tr>
<tr><td>…</td><td>…</td><td>…</td></tr>
<tr><td>18</td><td>WWWP</td><td>The last cell has a position (2, 5).</td></tr>
</table>

Example Text File



Checking Maze File Structure

A simple file check can be used when reading a maze text file. The final line number should always be the Width multiplied by the Height. This is 3 * 6 giving 18 in this example. If this rule does not hold true then the file being read is corrupt in some way and cannot be read.

Algorithms

I have included only pseudocode for the major parts of the application. The pseudocode included is for the algorithms and routines that are required in multiple areas of the program or are particularly perplexing. The text "[Maze name].txt" in the following sections refers to the text file that the user will choose either in an Open File dialog or a Save File dialog.

Validation Algorithms

The validation algorithms will provide the methods for validating the various inputs by the user. They include checking that the grid size specified is within the allowed range. The validation for checking the structure of a maze text file is correct. The validation of the algorithm selection is unneeded as the selection is done through a drop-down control. This greatly reduces the chance of error compared with a user manually entering the name of an algorithm to choose it.

Validate Entered Size

```
Input → gridHeightInput
Input → gridWidthInput

If gridWidthInput < 2 OR gridHeightInput < 2
      Output message "This Size is too small"
      Return false
End if
Return true
```

Validate Text File

```
Open "[Maze name].txt" with StreamReader
If line[0] in "[Maze name].txt" is not in the form: "[Maze name]|[Maze
Width]|[Maze Height]"
      Return false
End if
countLines ← 0
gridWidth ←(CAST as integer)  all first line characters from first
       '|' (exclusive) to the second '|' (exclusive)
gridHeight ←(CAST as integer)  all first line characters from
        second '|' (exclusive) to the end of the line
totalExpectedLines ← gridHeight * gridWidth
For each line in "[Maze name].txt" apart from the first
      If line.Length is not 4
            Return false
      End if
      If line.Contains('W' OR 'B') = false OR if line.Contains('p')     =
false
            Return false
      End if
      countLines ← countLines + 1
End for each
If countLines doesn't = totalExpectedLines
      Return false
End if
Return true
```

Creation of Text Files

This algorithm is responsible for ensuring that a maze can be converted into a text file to be saved on the hard drive. The one afterwards is for the conversion back. In the listing "[Maze name].txt" refers to a text file the user will either open or save. The text "[Maze Width]" and "[Maze Height]" represent the entered dimensions of the maze.

Create Text File

```
{Copy blank.txt and paste in “Generated Mazes” folder, rename it “[Maze
name].txt”}
 Open [Maze name].txt with StreamWriter
Sb ← “[Maze name]|[Maze Width]|[Maze Height]”
Append Sb to [Maze name].txt
For each line in “[Maze name].txt”
      For each edge in cell[i]
            If edge is borderEdge
                  Append to text file “B”
            else If the edge is Wall
                  Append to text file “W”
            else
                  Append to text file “P”
            end if
      End foreach
End foreach
Save and close “[Maze name].txt”
```

Convert Text File

```
Open [Maze name].txt with StreamReader
If ValidateTextFile = false
      return null
end if
gridName ← all first line characters until it reaches the first:'|'
        (exclusive)
gridWidth ← (CAST as integer) all first line characters from first
         '|' (exclusive) to the second '|' (exclusive)
gridHeight ← (CAST as integer) all first line characters from
          second '|' (exclusive) to the end of the line
Create new grid of name: gridName, width: gridWidth and height: gridHeight
For 0→TotalLines while totalLines <= gridHeight * gridWidth
      CurrentCell ← Grid.CreateCell(i,j)
      For 0→i while i < 3 step +1
            If line[at i] = 'B'
                  Grid.CreateWall(currentCell, null (CAST as Direction)i)
            End if
            If line[at i] = 'P' AND (i = 0 OR i = 3)
                  Grid.CreatePassage(currentCell, grid.GetCell(i,f-1),
(CAST as Direction)i))
            End if
            If line[at i] = 'W' AND (i = 0 OR i = 3)
                  Grid.CreateWall(currentCell, grid.GetCell(i-1,f), (CAST
as Direction)i))
            End if
      End for
      GetNextLine()
End for
```

Convert Grid to Image

```
currentX ← 0
currentY ← 0
for 0→i while i < grid.Size.Height step +1
      for 0→j while j < grid.Size.Width step +1
            currentCell ← getCell(at i,at j)
            for 0→f while f<3 step +1
                  if currentCell.GetEdge[(CAST as Direction)f] =
            CellWall OR = CellBorder
                        CASE based on f
                              Case 0
                                    Shape horizontal north
                              position(currentX, currentY)
                              Case 1
                                    Shape vertical east
                                    position(currentX, currentY)
                              Case 2
                                    Shape horizontal south
                                    position(currentX, currentY)
                              Case 3
                                    Shape vertical west
                                    position(currentX, currentY)
                        End case
                  End if
            currentX ← currentX + 30
            if currentX >= grid.Size.Width * 30
                  currentX ← 0
                  currentY ← currentY + 30
            end if
      end for
end for
```

Maze Generation Algorithms

Pseudocode for the algorithms for both a test grid and for the Drunkard's Walk are included. I have not included pseudocode for the many other algorithms since the others will be very similar. I feel that including them will not further to help me at this stage to produce ideas for how each algorithm works. The algorithms that I will provide are those that are either very different from those written in other languages or where I could not find similar creations at all. This entire theme is much the same for the pseudo code for solving mazes, as both parts stem from the same tree.

Simple Rectangle Algorithm (used only for Testing)

*This algorithm is a testing algorithm that I will create first, it will not feature as a solvable maze and be only used to show that the creation of a maze will be possible using the classes and methods as described. It simply produces a graph where every cell in the grid has a cellPassage to every other cell and border cells having border walls in their appropriate directions.

```
grid of set height
for 0→i while i < grid.Size.Height step +1
      for 0→j while j < grid.Size.Height step +1
            currentCell ← grid.CreateCell(at i, at j)
            possiblePoints Array ← new point array (of points:
      currentCell.Point.X, currentCell.Point.Y – 1,
      currentCell.Point.X + 1, currentCell.Point.Y,
      currentCell.Point.X, currentCell.Point.Y + 1,
      currentCell.Point.X - 1, currentCell.Point.Y)
            for 0→f while f < number of possible directions step +1
                  if grid.ContainsPoint(at possiblePoints[index f] is
            false)
                        grid.CreateWall(from currentCell, to null, in
                        direction)
                  end if
            end for
            if grid.ContainsPoint(at possiblePoints[index 0]) is true
            grid.CreatePassage(from currentCell, to
      grid.GetCell(at possiblePoints[index 0])), in
      Direction.North)
            end if
            if grid.ContainsPoint(at possiblePoints[index 3]) is true
            grid.CreatePassage(from currentCell, to
      grid.GetCell(at possiblePoints[index 3])), in
      Direction.West)
            end if
      end for
end for
```

Drunkards Walk Algorithm

```
Get a premade grid of set height and width

Create list of cells called activeCells
currentCell ← grid.CreateCell(at Grid.RandomPoint())
cellCount ← grid.height * grid.width
Repeat While activeCell.Count < cellCount
      direction ← DirectionHelper.GetRandomdirection()
      offset ← DirectionHelper.GetOffset(at direction)
      point.X ← currentCell.X-Coordinate + offset.X-coordinate
      point.Y ← currentCell.Y-Coordinate + offset.Y-coordinate
            if grid.ContainsPoint(at point) is true
                  neighbourCell ← grid.GetCell(at point)
                  if neighbourCell is null
                        neighbourCell ← grid.CreatCell(at point)
                        cellEdge ← currentCell.GetEdge(in direction)
                        if cellEdge is null
                              grid.CreatePassage(from currentCell, to
neighbourCell, in direction)
                        currentCell ← neighbourCell
                        end if
                  else
                        cellEdge ← currentCell.GetEdge(in direction)
                        if cellEdge is null
                              grid.CreateWall(from currentCell, to
neighbourCell, in direction)
                              currentCell ← neighbourCell
                        end if
                  end if
            else
                  cellEdge ← currentCell.GetEdge(in direction)
                  if cellEdge is null
                        grid.CreateWall(from currentCell, to null, in
direction)
                        currentCell ← neighbourCell
                  end if
            if activeCells.Contains(this cell: currentCell) is false AND
currentCell.IsFullyIntialised is true
                  activeCells ← currentCell
            end if
      end if
end while
```

Maze Solver Algorithms

In these algorithms the start position is always the top left cell and the end point is always the bottom right cell.

This simplification of the start and end points will be used throughout each solver algorithm's code. The reason why I have chosen these points is because they are two of the furthest points from each other, so it would be expected that the route from one to the other would be the longest it could be. **Since all my grids are spanning trees it is guaranteed that there is a route between any two points, therefore the start and end points are completely arbitrary**.

Drunkard's Walk Algorithm

```
startCell ← grid.GetCell(0, 0)
endCell ← grid.GetCell(grid.Size.Width-1, grid.Size.Height-1)
currentCell ← startCell
add currentCell to activeCells
add currentCell to visitedCells
repeat while currentCell doesn't = endCell
      direction ← get one Random Passage from currentCell
      offset ← DirectionHelper.GetOffset(at direction)
      point.X ← currentCell.X-Coordinate + offset.X-coordinate
      point.Y ← currentCell.Y-Coordinate + offset.Y-coordinate
      neighbourCell ← cell at point
      if visitedCells doesn't contain neighbourCell
            add neighbourCell to visitedCells
      end if
      if activeCells doesn't contain neighbourCell
            add neighbourCell to activeCells
      else
            remove the last value from activeCells
      end if
      currentCell ← neighBbourCell
end while
```

Wall follower (Left)

```
startCell ← grid.GetCell(0, 0)
endCell ← grid.GetCell(grid.Size.Width-1, grid.Size.Height-1)
currentCell ← startCell
cameFrom ← North
repeat while currentCell doesn't = endCell
      leftDirection ← Get left direction {based from currentCell and last
direction}
      offset ← DirectionHelper.GetOffset(at leftDirection)
      point.X ← currentCell.X-Coordinate + offset.X-coordinate
      point.Y ← currentCell.Y-Coordinate + offset.Y-coordinate
      neighbourCell ← cell at point

      if activeCells doesn't contain currentCell
            add currentCell to activeCells
      else
            remove currentCell from activeCells
      end if

      if visitedCells doesn't contain currentCell
            add currentCell to visitedCells
      end if

      camefrom ← opposite direction to leftDirection
end while
```

Class functions and Procedures

Any underscores found before a variable in any of these class functions and procedures highlight to me that these variables will be created as private variables.

Grid's Create Passage

Use of this will require a starting cell and an ending cell and the direction to create it in.

```
cellPassage1 ← new CellPassage()
cellPassage1.Initialise(from startCell, to endCell, in direction)

cellPassage2 ← new CellPassage()
cellPassage2.Initalise(from endCell, to startCell, in
DirectionHelper.GetOpposite(in direction))
Grid's Create Wall
cellWall1 ← new CellPassage()
cellWall1.Initalise(from startCell, to endCell, in direction)
if endCell isn't null
      cellWall2 ← new CellPassage()
      cellWall2.Initalise(from endCell, to startCell, in
DirectionHelper.GetOpposite(in direction))
end if
```

Grid's Create Cell

Outside of this function, _cells will have to be declared as a two-dimensional cell array of maximum dimension length of grid width (will be properly called _size.Width in this instant) for the X-coordinate and grid height (_size.Height) for the Y-coordinate.

```
DirectionHelper.GetOpposite
Name ← "{i},{j}"
Point ← new Point(at i, at j)
Cell ← new Cell(name, at point)
_cells[at i, at j] ← cell
Return cell
```

Grid's Contains Point

```
If point.X >= 0 && point.Y >= 0 && point.X < _size.Width && point.Y <
_size.Height
      Return true
End if
Return false
```

Cell's Set Edge

```
{direction is passed in}
_edges[at direction (cast as an integer)] ← cellEdge
_intialisedEdgeCount ← _intialisedEdgeCount + 1
```

Cell's Get Edge

```
{direction is passed in}
Return  _edges[at direction (cast as an integer)]
```

Cell's Is Fully Initialised

```
Get (if _intialisedEdgeCount = DirectionHelper.Count
      return true
else if
      return false
end if)
```

Cell's Get Random Uninitialised Direction

```
directionList as new list of Directions
for 0→i while i < DirectionHelper.Count(), step + 1
      if _edges[at i] = null
            directionList ← i (cast as a Direction)
      end if
end for
index ← _randomNumber(between 0 and directionList.Count)
return directionList[at index]
```

Display Algorithms

Output grid

```
{grid passed in}
for 0→i while i < grid.Size.Height step +1
      for 0→j while j < grid.Size.Width step +1
            cell ← grid.GetCell(x, y)
            if cell is not null
                  northWall ← cell.GetEdge(in northern Direction)
                  if northWall = cellWall
                        Generate next cell position's display
                  End if
                  eastWall ← cell.GetEdge(in eastern Direction)
                  if eastWall = cellWall
                        Generate next cell position's display
                  End if
                  southWall ← cell.GetEdge(in southern Direction)
                  if southWall = cellWall
                        Generate next cell position's display
                  End if
                  westWall ← cell.GetEdge(in western Direction)
                  if westWall = cellWall
                        Generate next cell position's display
                  End if
            End if
      End for
End for
```

Time Results

The timings of the performance of the algorithms is achieved by using the Stopwatch class provided in the Microsoft .NET Foundation libraries.

Start Recording

```
stopwatch.start()
```

Stop Recording

```
Stopwatch.stop()
```

Complexity Calculation

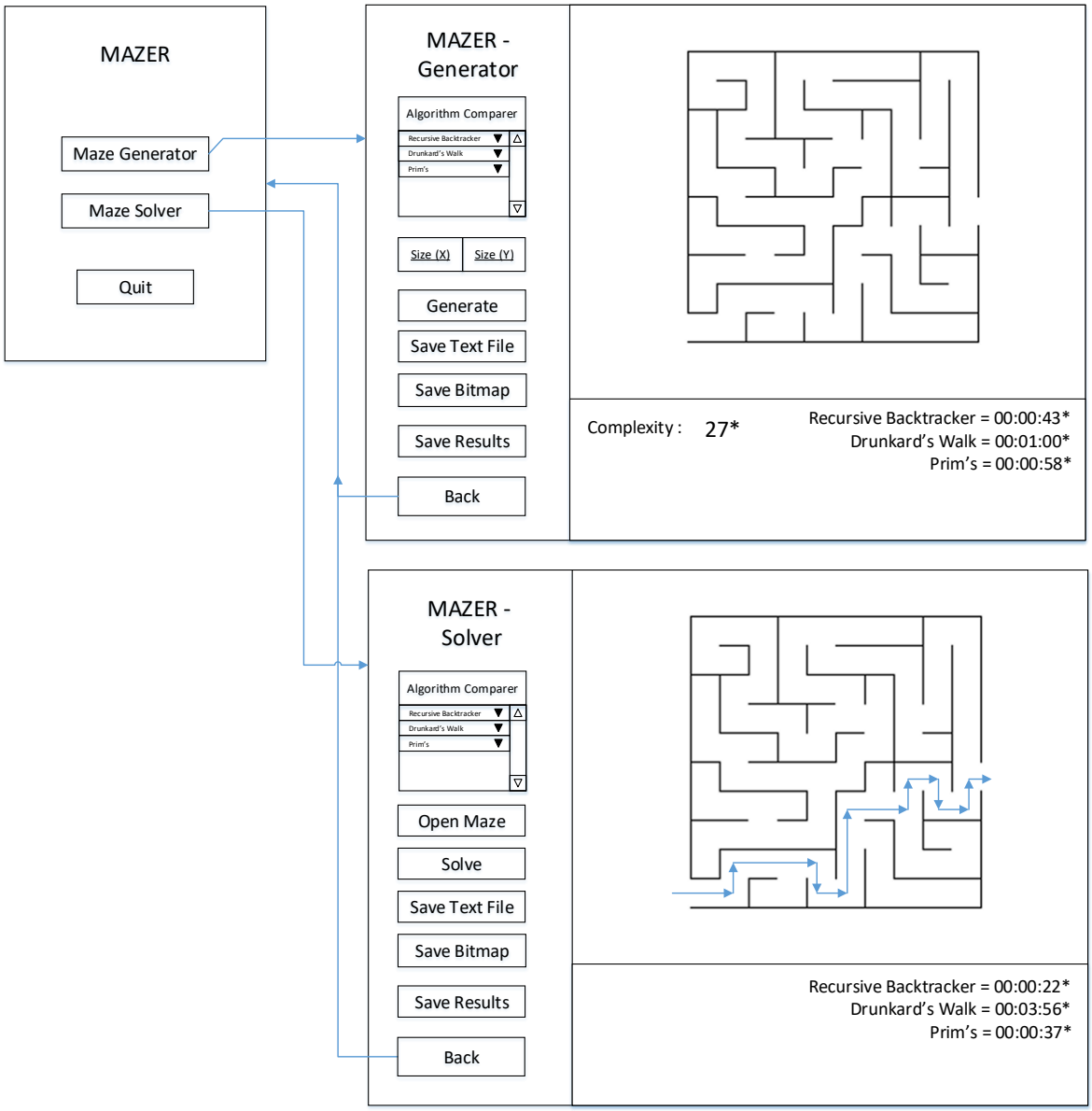
The complexity of a grid is calculated by adding the number of the cells with two passageways, to the number with three passageways and then also to the number with four passageways. This total is then divided by the number of cells with only one passageway.

By adding together the number of cells that have the rarer cell edge structure, I gather how many cells give signs of making a more complex maze. Then by dividing by the number of cells that have the cell edge structure a complex maze has less of I produce a number which represents how complex the maze actually is.

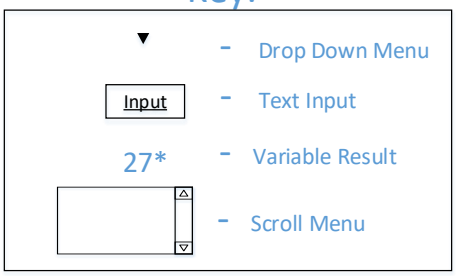
```
_grid = maze to calculate complexity of
For each cell in _grid
      For 0→i while i < DirectionHelper.Count(), step + 1

            If cell.getEdge(i (cast as a Direction)) is a cellPassage
                  Counter + 1 → counter
            End if
      End for
      If counter = 1
            onePassage + 1 → onePassage
      Else If counter = 2
            twoPassage + 1 → twoPassage
      Else If counter = 3
            threePassage + 1 → threePassage
      Else If counter = 4
            fourPassage + 1 → fourPassage
      End if
End foreach
Complexity = (twoPassage + threePassage + fourPassage) ÷ onePassage
Return complexity
```

Human-Computer Interface

Below is the layout of the UI interface for my solution. Dialog interfaces have not been drawn as they will be the Save File Dialogs and Open File Dialogs controls from the Microsoft .NET Framework libraries.



Key:



Security & Integrity of Data

The security and integrity of data is not really an issue for this application. There is no personally identifiable or any other data of a sensitive nature used by the application. The data that is transferred from a text file and converted to a graph the program can use is checked and validated to ensure that it does not crash the application. The program will catch any errors found during transferring files and output an error message to the user.

System security

The required solution as described in the objectives requires no special considerations for security. No data of a private or financial nature is ever handled by the application so not special security needs to be built in. There is no data that would be covered by the Data Protection Act. Without a need to incorporate this security, there is no reason to add any authentication or authorization capabilities into the application.

Test Strategy

Throughout the development of the application, I plan to use a pen and paper method to prove each part of the maze is created in a correct manner and in a way that the program could then use to convert it to an image to show the user and/ or can be stored to a text file.
In addition, I will also test the program using Unit Testing by making use of the testing capabilities of Visual Studio. Unit Testing involves writing many small tests containing simple assertions. Each test is based on the requirements and expected behaviour of the code unit that is under test. After making changes to the code I will re-run all my unit tests. This will keep a high level of quality and give me an early warning if I make a breaking change to my code.

Implementation

I decided to call my application **Mazer.** The following sections include the code written while implementing the design for my Mazer program. As was decided in the analysis and design the implementation makes extensive use of object-oriented features of the C# language. These include the use of Classes, **Polymorphism**, **Inheritance** and **Encapsulation**. I also use the generic collection classes such as List<T>, Dictionary<TKey, TValue> etc which are provided by the Microsoft .NET Framework libraries. The Mazer application uses the Windows Presentation Foundation (WPF) also found in the .NET libraries to provide a modern and professional user interface with smooth animation effects.

My Mazer solution is made up from three projects in Visual Studio. These are

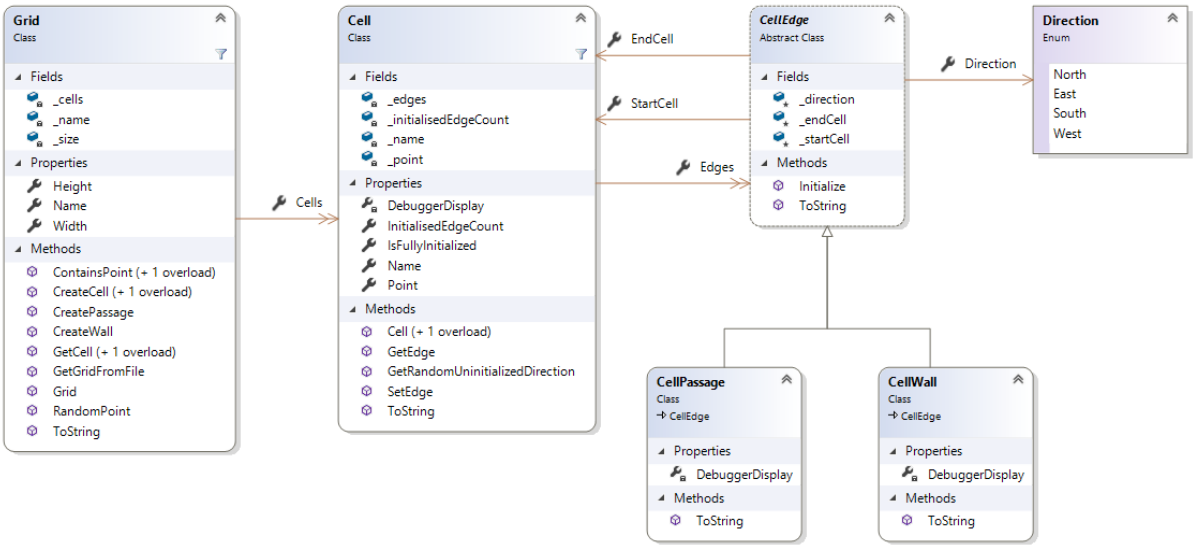
Project Name	Project Type	Description
MazeLibrary	Class Library	A reusable class library for exploring mazes. Includes many algorithms for solving and creating mazes. There is no user interface found in here since it is concerned with the tasks of dealing with mazes.
MazeLibraryUnitTests	Class Library	A Unit Test library for the automatic testing of the objects provided in the MazeLibrary project.
Mazer	Windows Application	The main application. This makes use of WPF to provide the user interface. It uses the objects and algorithms of the MazeLibrary to do its work.

I will include a class diagram, description and the C# code for each of the objects I created in the implementation of my Mazer application in the following sections.

MazeLibary Project

Core

This core code provides the functionality of creating and manipulating a Grid object. The Grid (also known as a graph in mathematics) has a collection of Cells. Each Cell has a collection of four CellEdges. CellEdges can be of two types: CellPassage and CellWall. Each CellEdge knows its StartCell, EndCell and its Direction. These objects underpin the rest of the solution and this is why I have named this group of objects "Core".



Cell

The Cell is a node, or vertex, which links edges together to create a graph which is represented in the Grid class. The Cell implementation demonstrates a good example of **Polymorphic** behaviour in the ToString() routine. When the list of CellEdges are being iterated over to add to the return value to StringBuilder it is the correct ToString() routine that is being called on either a CellWall or a CellPassage depending on its type. This works because these classes supply an **overridden** version of the ToString() routine which is declared **virtual** in the CellEdge base class.

Variable	Description
`_name`	The name of a Cell. This string allows Cells to be named which is useful when debugging or referring to nodes in a graph.
`_point`	The cell's position in the Grid. It represents the X, Y coordinate of the Cell in a single structure.
`_edges`	An array of CellEdges of length 4. One CellEdge is stored for each of the four directions in the order North, East, South, West. It may contain a null for any Direction if a CellEdge has not been assigned in that Direction yet.
`_initialisedEdgeCount`	Stores the current number of CellEdges that are not null in the _edges array.

```
using MazeLibrary.Helpers;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;

namespace MazeLibrary
{
    //Attribute: A helper telling what to display when the debugger views this class
    [DebuggerDisplay("{DebuggerDisplay}")]
    public class Cell
    {
        private static Random _random = new Random();

        //Encapsulation: Private storage with public properties and functions
        private string _name;
        private Point _point;
        private CellEdge[] _edges = new CellEdge[DirectionHelper.Count];
        private int _initialisedEdgeCount;

        //Constructor for Cell from a name and a point
        public Cell(string name, Point point)
        {
            _name = name;
            _point = point;
        }

        //Constructor for Cell from a name, x coordinate and then a y coordinate
        public Cell(string name, int x, int y)
        {
            _name = name;
            _point = new Point(x, y);
        }

        //Properties for getting the name or point at a cell
        public string Name
        {
            get { return _name; }
        }

        public Point Point
        {
            get { return _point; }
        }

        //Property for getting the name of a cell to be displayed in the output tab
when debugging code
        private string DebuggerDisplay
        {
            get { return String.Format("Cell: {0}", Name); }
        }


        //Method for replacing whatever value was in that element of the _edges array
with a new cellEdge and a direction.
        //_initialisedEdgeCount stores the number of edges that a cell has, it should
never go above 4
        public void SetEdge(Direction direction, CellEdge cellEdge)
        {
            //Only increase the initialised edge count if the edge that is being set
is currently null. If
            //it is not null then we are changing an edge that was already initialised
so the count does not
            //need to be increased.
            if (_edges[(int)direction] == null)
```

```
            {
                _initialisedEdgeCount += 1;
            }

            _edges[(int)direction] = cellEdge;
        }

        //Function for returning a specific edge name of a direction from a cell
        public CellEdge GetEdge(Direction direction)
        {
            return _edges[(int)direction];
        }

        //Debugging and testing for creating the maze in string form using the console
        public string ToString(int level = 0)
        {
            StringBuilder sb = new StringBuilder();
            string indent = new string(' ', 2 * level);
            sb.AppendLine(String.Format("{0}Cell: {1} ({2})", indent, _name, _point));
            foreach (CellEdge cellEdge in _edges)
            {
                if (cellEdge != null)
                {
                    //Polymorphic behaviour... the cellEdge calls the correct
Display()
                    //routine depending on whether this particular cellEdge is a
CellWall
                    //or CellPassage derived class.
                    sb.AppendLine(cellEdge.ToString(level + 1));
                }
            }
            return sb.ToString();
        }

        //Returns true if the current cell has exactly 4 edges, all of which aren't
null
        public bool IsFullyInitialized
        {
            get { return _initialisedEdgeCount == DirectionHelper.Count; }
        }

        //Function returning all edges of a cell that are null, hence if there aren't
any then a problem has occurred
        public Direction GetRandomUninitializedDirection()
        {
            List<Direction> directionList = new List<Direction>();
            for (int i = 0; i < DirectionHelper.Count; i++)
            {
                if (_edges[i] == null)
                {
                    directionList.Add((Direction)i);
                }
            }
            int index = _random.Next(0, directionList.Count);
            return directionList[index];
        }

        //Returns the number of edges a cell has
        public int InitialisedEdgeCount
        {
            get { return _initialisedEdgeCount; }
        }
```

```
    }
}
```

CellEdge

CellEdge is the connector between a given Cell and one of its four possible neighbours.

Variable	Description
_startCell	The Cell starting a connection of a passageway or wall from one Cell to another Cell.
_endCell	The Cell ending a connection of a passageway or wall from one Cell to another Cell.
_direction	The direction the connection is going in to form the link between the StartCell and the EndCell. This is a value from the enumerated type Direction and can take one of the values (North, East, South or West).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MazeLibrary
{
    public abstract class CellEdge
    {
        //protected because these variables can be used by this class and those
inherited from it.
        // Another case of Encapsulation here as well
        protected Cell _startCell;
        protected Cell _endCell;
        protected Direction _direction;

        //Properties for setting and getting a start cell, an end cell and a direction
        public Cell StartCell
        {
            get { return _startCell; }
            set { _startCell = value; }
        }

        public Cell EndCell
        {
            get { return _endCell; }
            set { _endCell = value; }
        }

        public Direction Direction
        {
            get { return _direction; }
            set { _direction = value; }
        }

        //CellEdge requires this information when it is created
        public void Initialize(Cell startCell, Cell endCell, Direction direction)
        {
            _startCell = startCell;
            _endCell = endCell;
            _direction = direction;
```

```
            startCell.SetEdge(direction, this);
        }

        //virtual lets you override this function in the derived classes
        public virtual string ToString(int level)
        {
            return null;
        }
    }
}
```

Cell Passage

Cell Passage is a connection from one cell to another which allows movement between those cells in either direction. These are displayed as an empty gap between the cells. If HideGridLines is turned off in the user interface then they are displayed as a line of red dashes perpendicular to the passage direction.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

//
// Object Orientation is PIE in the sky!
//
// P = Polymorphism
// I = Inheritance
// E = Encapsulation
//
namespace MazeLibrary
{
    //Inheritance: The ':' means this CellPassage class is derived from a base class
    //CellEdge. Or said another way... CellPassage is inherited from CellEdge.
    [DebuggerDisplay("{DebuggerDisplay}")]
    public class CellPassage : CellEdge
    {
        public override string ToString(int level)
        {
            string indent = new string(' ', 2 * level);
            return String.Format("{0}Passage: {1} {2} ({3})", indent, _startCell.Name,
_endCell.Name, _direction);
        }

        private string DebuggerDisplay
        {
            get { return String.Format("Passage: {0} {1} {2}", _startCell.Point,
_endCell.Point, _direction); }
        }
    }
}
```

Cell Wall

Cell Wall is a connection from one cell to another which disallows movement between those cells in either direction. These are displayed as a black line perpendicular to the direction the CellEdge is going in

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MazeLibrary
{
    [DebuggerDisplay("{DebuggerDisplay}")]
    //The ':' means this CellWall class is derived from a base class CellEdge
    public class CellWall : CellEdge
    {
        private string DebuggerDisplay
        {
            get { return String.Format("Wall: {0} {1} {2}", _startCell.Point, _endCell
== null ? "[Border]" : _endCell.Point.ToString(), _direction); }
        }

        //The 'override' means we are overriding the function in the base class that
        //has the same name.
        public override string ToString(int level)
        {
            string indent = new string(' ', 2 * level);
            return String.Format("{0}Wall: {1} {2} ({3})",
                indent,
                _startCell.Name,
                _endCell == null ? "Border" : _endCell.Name,
                _direction);
        }
    }
}
```

Direction

The Direction enumerable type is used to allow conversion of an integer to a direction, useful for making the source code more readable.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MazeLibrary
{
    //An enumerable class for setting the 4 directions as integer values 0 through 3
    public enum Direction
    {
        North,
        East,
        South,
        West,
    }
}
```

Grid

The Grid is used to hold the **2 Dimensional array** of Cells. It represents a mathematical graph of nodes which I have called Cells.

Variable	Description
_name	The name of the Grid. This is used to show a friendly readable name for the grid and to offer up a default name when the user is saving the Grid to disk.
_size	The width and height of the Grid stored in a single structure.
_cells	A 2-dimensional array of Cell's with dimensions the same dimensions as specified in _size.

```
using MazeLibrary.Helpers;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MazeLibrary
{
    public class Grid
    {
        //Encapsulation of variables, methods and properties within a class shown here
        private string _name;
        private Random _random = new Random();
        private Size _size;
        private Cell[,] _cells;

        //Constructor for a grid, i.e. this information is required to initialise a
new grid
        public Grid(string name, int width, int height)
        {
            _name = name;
            _size = new Size(width, height);
            _cells = new Cell[_size.Width, _size.Height];
        }

        //Properties for getting grid width, grid height and both getting or setting a
grid name
        public int Width
        {
            get { return _size.Width; }
        }

        public int Height
        {
            get { return _size.Height; }
        }

        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }
```

```
        //Procedure for creating a passageway between two cells on the grid, from
startCell to endCell in the direction given.
        //Notice it also creates the another passage in reverse from endCell to
startCell in the opposite direction.
        public void CreatePassage(Cell startCell, Cell endCell, Direction direction)
        {
            CellPassage cellPassage1 = new CellPassage();
            cellPassage1.Initialize(startCell, endCell, direction);

            CellPassage cellPassage2 = new CellPassage();
            cellPassage2.Initialize(endCell, startCell,
DirectionHelper.GetOpposite(direction));
        }

        //Procedure for creating a Wall between two cells on the grid, from startCell
to endCell in the direction given.
        //Notice it also creates the another Wall in reverse from endCell to startCell
in the opposite direction.
        public void CreateWall(Cell startCell, Cell endCell, Direction direction)
        {
            CellWall cellWall1 = new CellWall();
            cellWall1.Initialize(startCell, endCell, direction);
            if (endCell != null)
            {
                CellWall cellWall2 = new CellWall();
                cellWall2.Initialize(endCell, startCell,
DirectionHelper.GetOpposite(direction));
            }
        }

        //Function to return true if a specified point is within the bounds of the
grid's height and width
        public bool ContainsPoint(Point p)
        {
            if (p.X >= 0 && p.Y >= 0 && p.X < _size.Width && p.Y < _size.Height)
            {
                return true;
            }
            return false;
        }

        //Returns bool from ContainsPoint where the parameter is a point rather than a
x and y coordinate
        public bool ContainsPoint(int x, int y)
        {
            return ContainsPoint(new Point(x, y));
        }

        //function to return a random point on the grid within its' bounds
        public Point RandomPoint()
        {
            Point randomPoint = new Point(_random.Next(0, _size.Width),
_random.Next(0, _size.Height));
            return randomPoint;
        }

        //Function for creating a cell on the grid with its coordinates as its name,
for ease of debugging, at the specified width and height
        public Cell CreateCell(int i, int j)
        {
            string name = String.Format("({0}, {1})", i, j);
```

```
            Point point = new Point(i, j);
            Cell cell = new Cell(name, point);
            _cells[i, j] = cell;
            return cell;
        }

        //runs the previous CreateCell function but allowing a point to be passed in
instead two integers
        public Cell CreateCell(Point p)
        {
            return CreateCell(p.X, p.Y);
        }

        //Function returns the cell at the specified x and y coordinates
        public Cell GetCell(int i, int j)
        {
            return _cells[i, j];
        }

        //function returns the cell at a specified point
        public Cell GetCell(Point p)
        {
            return GetCell(p.X, p.Y);
        }

        //This code relates only to outputting the maze as a string in console, to
start testing if the WPF application would be possible to create
        public string ToString(int level = 0)
        {
            StringBuilder sb = new StringBuilder();
            string indent = new string(' ', 2 * level);
            sb.AppendLine(String.Format("{0}Grid: {1}", indent, _name));
            for (int i = 0; i < _size.Width; i++)
            {
                for (int j = 0; j < _size.Height; j++)
                {
                    Cell cell = GetCell(i, j);
                    if (cell != null)
                    {
                        sb.Append(cell.ToString(level + 1));
                    }
                }
            }
            return sb.ToString();
        }

        //Function for converting a text file format into a maze it can use
        public static Grid GetGridFromFile(string filename)
        {

            using (StreamReader reader = new StreamReader(filename, true))
            {


                //Array stores the first line of the text file in three parts, where
seporated by a verticle bar
                string[] splitter = reader.ReadLine().Split('|');

                //Creates the grid called the first part, width the second part, and
height the third part
                Grid grid = new Grid(splitter[0], Convert.ToInt32(splitter[1]),
Convert.ToInt32(splitter[2]));
```

```
                //e.g. "Maze1|5|6"

                //Prepare Grid Cells and Borders
                for (int y = 0; y < grid.Height; y++)
                {
                    for (int x = 0; x < grid.Width; x++)
                    {
                        //Creates a cell at every position on the grid and...
                        Cell currentCell = grid.CreateCell(x, y);

                        //fills its north direction in with a wall if it is at the
north border
                        if (y == 0)
                        {
                            grid.CreateWall(currentCell, null, Direction.North);
                        }

                        //fills its south direction in with a wall if it is at the
south border
                        if (y == grid.Height - 1)
                        {
                            grid.CreateWall(currentCell, null, Direction.South);
                        }

                        //fills its west direction in with a wall if it is at the west
border
                        if (x == 0)
                        {
                            grid.CreateWall(currentCell, null, Direction.West);
                        }

                        //fills its east direction in with a wall if it is at the east
border
                        if (x == grid.Width - 1)
                        {
                            grid.CreateWall(currentCell, null, Direction.East);
                        }
                    }
                }

                //line equals the next line in the file
                string line = reader.ReadLine();

                //steps through every cell in the grid
                for (int y = 0; y < grid.Height; y++)
                {
                    for (int x = 0; x < grid.Width; x++)
                    {
                        //Only gets the directions from the currentCell going North or
East, this ensures at no point will it try to replace an already set edge
                        for (int i = 0; i < 4; i = i + 3)
                        {
                            Cell currentCell = grid.GetCell(x, y);
                            Direction direction = (Direction)i;
                            System.Drawing.Point offset =
DirectionHelper.GetOffset(direction);
                            System.Drawing.Point point = new
System.Drawing.Point(currentCell.Point.X + offset.X, currentCell.Point.Y + offset.Y);

                            //Only continues if the new neighbour point is on the
grid, as off the grid values have already been done
                            if (grid.ContainsPoint(point))
```

```
                            {
                                //Note: the text file format lists every cell's edges
in the order of the enum class Direction and also when saving
                                //files it uses only the char's 'P' and 'W'
                                if (line[i] == 'P')
                                {
                                    grid.CreatePassage(currentCell,
grid.GetCell(point), direction);
                                }
                                else if (line[i] == 'W')
                                {
                                    grid.CreateWall(currentCell, grid.GetCell(point),
direction);
                                }
                                else
                                {
                                    //Error thrown if an unexpected character is in
the file
                                    throw new Exception("Uninitialized edge
detected?");
                                }
                            }
                        }
                        //Gets the next line
                        line = reader.ReadLine();
                    }
                }
                //The grid will at this stage be complete and be a completed maze
therefore it returns it
                return grid;
            }
        }
    }
}
```

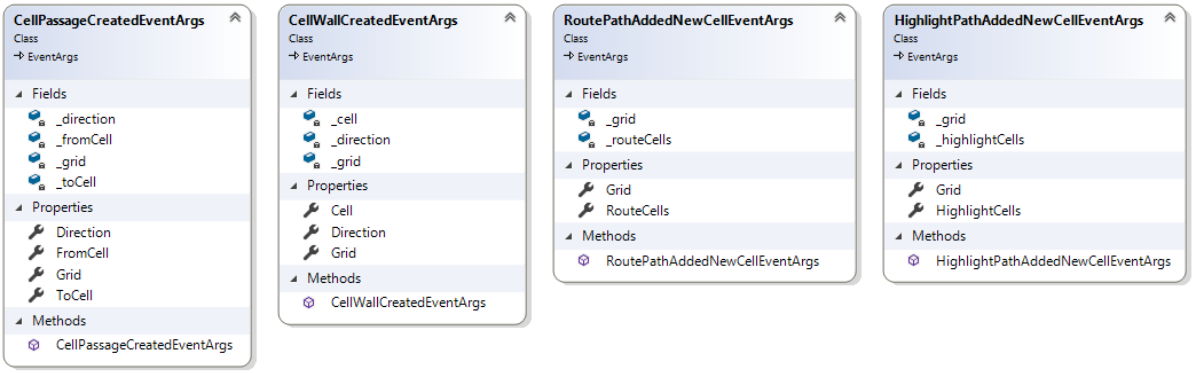
Events Arguments

Event arguments are used in the Mazer program when it signals to the generator or solver windows that the canvas needs to be updated, provided the relevant checkboxes are turned off.

The event argument for cell passage created and cell wall created are used in the Mazer maze creation code.

The route path added new cell event argument is used in the Mazer maze solution finding code.
The highlight path added new cell event argument is used by both maze creation and solver algorithms to show what cells are currently being considered. This allows the user to see an animated display of progress on the screen.

These event arguments are all required to run the UI part of the program at the same time as when algorithms are running. It does this through **multithreading**, which I will write more on in the generator and solver window sections. These event arguments carry event information from the algorithm thread to the user interface thread enabling the user interface to remain responsive when the algorithms are running.



CellPassageCreatedEventArgs

Event argument for when a passageway is created between two cells. Used to update the UI.

```
using MazeLibrary;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MazeLibrary.Events
{
    //Event argument for getting the UI canvas to update with the latest information,
namely the grid it's working on,
    //the cells the passage came from and is going and the direction
    public class CellPassageCreatedEventArgs : EventArgs
    {
        private Grid _grid;
        private Cell _fromCell;
        private Cell _toCell;
        private Direction _direction;

        //Constructor, the event needs this information
        public CellPassageCreatedEventArgs(Grid grid, Cell fromCell, Cell toCell,
Direction direction)
        {
            _grid = grid;
```

```
            _fromCell = fromCell;
            _toCell = toCell;
            _direction = direction;
        }

        //Properties for getters for grid, fromCell, toCell and direction
        public Grid Grid
        {
            get { return _grid; }
        }

        public Cell FromCell
        {
            get { return _fromCell; }
        }

        public Cell ToCell
        {
            get { return _toCell; }
        }

        public Direction Direction
        {
            get { return _direction; }
        }
    }
}
```

CellWallCreatedEventArgs

Event argument for when a wall is created between two cells or a cell and a nonexistent cell off the grid (null). Used to update the UI.

```
using MazeLibrary;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MazeLibrary.Events
{
    //Event argument for getting the UI canvas to update with the latest information,
namely the grid it's working on,
    //the cell the wall came from and the direction
    public class CellWallCreatedEventArgs : EventArgs
    {
        private Grid _grid;
        private Cell _cell;
        private Direction _direction;

        //Constructor, the event requires this information
        public CellWallCreatedEventArgs(Grid grid, Cell cell, Direction direction)
        {
            _grid = grid;
            _cell = cell;
            _direction = direction;
        }

        //Properties for getting grid, cell and direction
        public Grid Grid
        {
            get { return _grid; }
```

```
        }

        public Cell Cell
        {
            get { return _cell; }
        }

        public Direction Direction
        {
            get { return _direction; }
        }
    }
}
```

HighlightPathAddedNewCellEventArgs

Event argument for when the highlight path is updated by adding a new cell to it. Used to update the UI.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MazeLibrary.Events
{
    //Event argument for getting the UI canvas to update with the latest information,
namely the grid its working on and the latest version of the highlight path
    public class HighlightPathAddedNewCellEventArgs : EventArgs
    {
        private Grid _grid;
        private List<Cell> _highlightCells;

        //Constructor for the event requiring this information
        public HighlightPathAddedNewCellEventArgs(Grid grid, List<Cell>
highlightCells)
        {
            _grid = grid;
            _highlightCells = highlightCells;
        }

        //Properties for getting grid and the highlight cells list
        public Grid Grid
        {
            get { return _grid; }
        }

        public List<Cell> HighlightCells
        {
            get { return _highlightCells; }
        }
    }
}
```

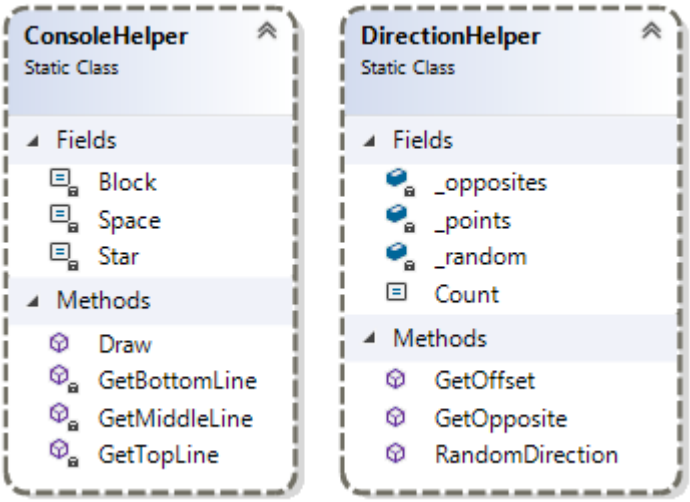
RoutePathAddedNewCellEventArgs

Event argument for when the route path is updated by adding a new cell to it. Used to update the UI.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MazeLibrary.Events
{
    //Event argument for getting the UI canvas to update with the latest information,
namely the grid its working on and the latest version of the route path
    public class RoutePathAddedNewCellEventArgs : EventArgs
    {
        private Grid _grid;
        private List<Cell> _routeCells;

        //Constructor for this event requiring this information
        public RoutePathAddedNewCellEventArgs(Grid grid, List<Cell> routeCells)
        {
            _grid = grid;
            _routeCells = routeCells;
        }

        //Properties for getting grid and the routeCells list
        public Grid Grid
        {
            get { return _grid; }
        }

        public List<Cell> RouteCells
        {
            get { return _routeCells; }
        }
    }
}
```

Helpers

The helper classes in this solution are all static classes meaning these classes are never created by the user. Their routines are available via the class name.
ConsoleHelper can be used to create a character only version of a maze (no longer used, but useful for testing). The DirectionHelper class provides useful help routines for handling the Direction enumerated type.



ConsoleHelper

Class produces a console ready character only version of the maze. This is no longer used however it is still useful for testing the generation of the mazes quickly in console applications, or the debug window.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MazeLibrary.Helpers
{
    public static class ConsoleHelper
    {
        //Code used in the intial testing of the sollution checking if it was a
feasible sollution to pursue
        private const string Block = "  "; // Form walls
        private const string Space = "  "; // Form passageways
        private const string Star = "**";  // Form still being built areas

        public static string Draw(Grid grid)
        {
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < grid.Height; i++)
            {
                //Each cell and it's edges are built as a 3X3 space which fit my
constants (where there are 18 character spots)
                //All corners of the 3X3 area are blocks as these would technically
not exist
                StringBuilder topLine = new StringBuilder();
                StringBuilder middleLine = new StringBuilder();
                StringBuilder bottomLine = new StringBuilder();
                for (int j = 0; j < grid.Width; j++)
                {
```

```
                Cell cell = grid.GetCell(j, i);

                if (cell == null)
                {
                    //If the cell doesn't exist yet then fill this spot with stars
                    topLine.AppendFormat("{0}{0}{0}", Star);
                    middleLine.AppendFormat("{0}{0}{0}", Star);
                    bottomLine.AppendFormat("{0}{0}{0}", Star);
                }
                else
                {
                    //Functions getting each line for the current cell
                    topLine.Append(GetTopLine(cell));
                    middleLine.Append(GetMiddleLine(cell));
                    bottomLine.Append(GetBottomLine(cell));
                }
            }
            //Appends each line to sb
            sb.AppendLine(topLine.ToString());
            sb.AppendLine(middleLine.ToString());
            sb.AppendLine(bottomLine.ToString());

            //Resets each StringBuilder for the next loop
            topLine.Clear();
            middleLine.Clear();
            bottomLine.Clear();
        }
        //Returns the string, originally would be send to console
        return sb.ToString();
    }

    //This is how each cell was displayed, shown in more detail below
    // aa, cc, gg and ii are corners so just become blocks
    // ee is where the cell is so becomes a space
    // aabbcc
    // ddeeff
    // gghhii

    private static string GetTopLine(Cell cell)
    {
        //Gets each edge for the current cell
        CellEdge northCellEdge = cell.GetEdge(Direction.North);
        CellEdge eastCellEdge = cell.GetEdge(Direction.East);
        CellEdge southCellEdge = cell.GetEdge(Direction.South);
        CellEdge westCellEdge = cell.GetEdge(Direction.West);

        StringBuilder sb = new StringBuilder();
        // aa
        sb.Append(Block);

        // bb
        if (northCellEdge is CellWall)
        {
            sb.Append(Block);
        }
        else
        {
            sb.Append(Space);
        }

        // cc
        sb.Append(Block);
```

```
        return sb.ToString();
    }

    private static string GetMiddleLine(Cell cell)
    {
        CellEdge northCellEdge = cell.GetEdge(Direction.North);
        CellEdge eastCellEdge = cell.GetEdge(Direction.East);
        CellEdge southCellEdge = cell.GetEdge(Direction.South);
        CellEdge westCellEdge = cell.GetEdge(Direction.West);

        StringBuilder sb = new StringBuilder();

        // dd
        if (westCellEdge is CellWall)
        {
            sb.Append(Block);
        }
        else
        {
            sb.Append(Space);
        }

        // ee
        sb.Append(Space);

        // ff
        if (eastCellEdge is CellWall)
        {
            sb.Append(Block);
        }
        else
        {
            sb.Append(Space);
        }

        return sb.ToString();
    }

    private static string GetBottomLine(Cell cell)
    {
        CellEdge northCellEdge = cell.GetEdge(Direction.North);
        CellEdge eastCellEdge = cell.GetEdge(Direction.East);
        CellEdge southCellEdge = cell.GetEdge(Direction.South);
        CellEdge westCellEdge = cell.GetEdge(Direction.West);

        StringBuilder sb = new StringBuilder();
        //gg
        sb.Append(Block);

        // hh
        if (southCellEdge is CellWall)
        {
            sb.Append(Block);
        }
        else
        {
            sb.Append(Space);
        }

        // ii
        sb.Append(Block);
```

```
            return sb.ToString();
        }
    }
}
```

DirectionHelper

This class is used to provide extra functions to use in conjunction with the Direction class. IT provides several helpful routines such as returning random directions and the opposite direction for a given direction.

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MazeLibrary.Helpers
{
    //A class to help the grids usage of the enum class Direction
    public static class DirectionHelper
    {
        private static Random _random = new Random();

        //A point array storing the 4 directions interms of coordinates
        private static Point[] _points = new Point[]
        {
            new Point(0, -1), //N
            new Point(1, 0),  //E
            new Point(0, 1),  //S
            new Point(-1, 0), //W
        };

        //A Direction array for storing the opposite directions to the enum class
Direction
        private static Direction[] _opposites = new Direction[]
        {
            Direction.South,
            Direction.West,
            Direction.North,
            Direction.East,
        };

        //Assignment of the number 4 into a constant count. This is for the 4 possible
directions
        public const int Count = 4;

        //Function returning a random direction by choosing a random number between 0
and 3 (inclusive) and casting it as a Direction
        public static Direction RandomDirection()
        {
            int randomNumber = _random.Next(0, Count);
            return (Direction)randomNumber;
        }

        //Function for returning the corresponding _points point based on the passed
Direction argument
        public static Point GetOffset(Direction d)
        {
            return _points[(int)d];
        }
```

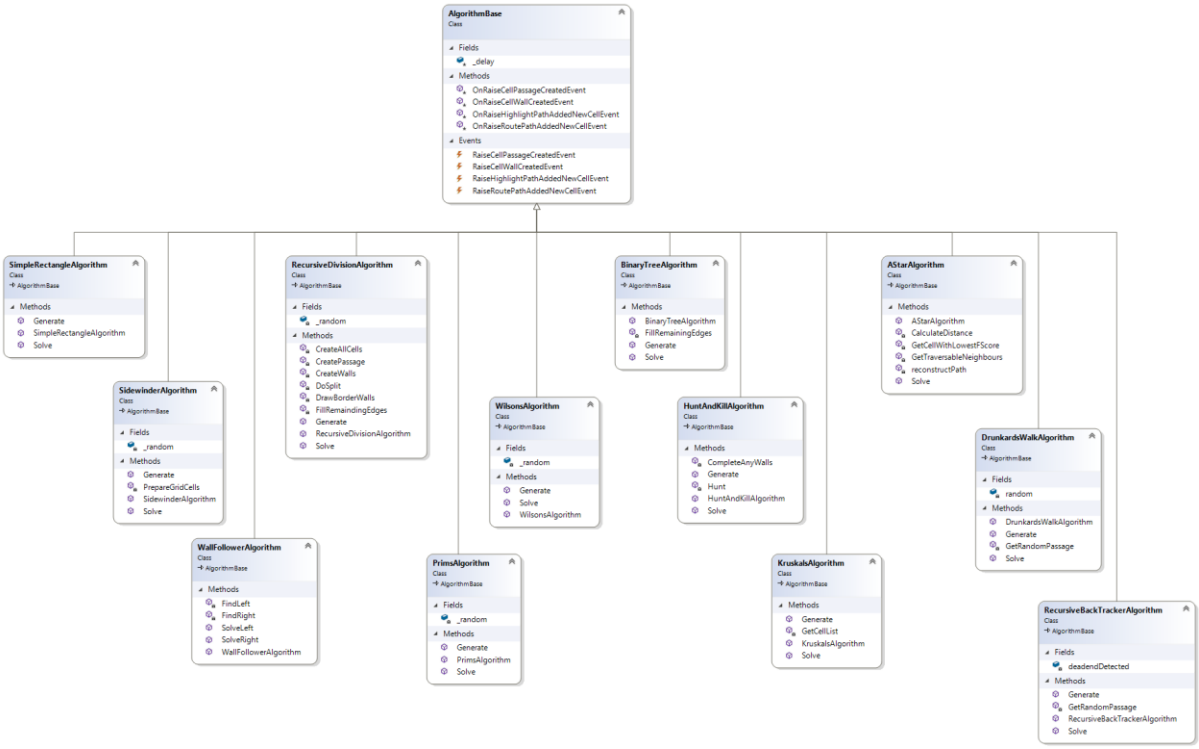
```
        //Function returning the corrasponding _opposites direction based on the
passed Direction arguement
        public static Direction GetOpposite(Direction direction)
        {
            return _opposites[(int)direction];
        }
    }
}
```

Maze Algorithms

The following section describes all the algorithms in my Mazer program. Each algorithm includes the generator code and if applicable the solver code.

These algorithms demonstrate many examples of **traversing tree graphs** which is a fundamental requirement of any maze generator or solver program.

This section also demonstrates the use of **inheritance** since every algorithm inherits the event handlers from an Algorithm Base class which massively reduces the amount of repeated code that would otherwise be needed.



Algorithm Base

Class containing event handling functions. These are connected to by the main Mazer windows and dialogs. When a particular event is fired by an algorithm (as it is running) the appropriate event arguments class is created, populated and then passed to the listening code in the UI. The UI can then update the screen without blocking the algorithm thread from continuing. This results in a very smooth animation and UI experience for the user.

```
using MazeLibrary.Events;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MazeLibrary.Algorithms
{
    //AlgorithmBase is a class for allowing events I've made to pass to the UI code,
while whichever algorithms currently running to continue being made
    public class AlgorithmBase
    {
        //Event handlers for each event: wall created, passage created or the
routepath getting a new cell added
        public event EventHandler<CellWallCreatedEventArgs> RaiseCellWallCreatedEvent;
```

```
        public event EventHandler<CellPassageCreatedEventArgs>
RaiseCellPassageCreatedEvent;
        public event EventHandler<RoutePathAddedNewCellEventArgs>
RaiseRoutePathAddedNewCellEvent;
        public event EventHandler<HighlightPathAddedNewCellEventArgs>
RaiseHighlightPathAddedNewCellEvent;

        //A delay will feature in every algorithm forcing the algorithm to pause for
the milliseconds chosen, allowing the canvas to refresh and the user time to view any
changes
        protected int _delay;

        //For raising a cell wall created event arg
        protected virtual void OnRaiseCellWallCreatedEvent(CellWallCreatedEventArgs e)
        {
            EventHandler<CellWallCreatedEventArgs> handler =
RaiseCellWallCreatedEvent;
            //Ensures that the event will only update the ui if an actual change has
occurred
            if (handler != null)
            {
                handler(this, e);
            }
        }

        //For raising a cell passage created event arg
        protected virtual void
OnRaiseCellPassageCreatedEvent(CellPassageCreatedEventArgs e)
        {
            EventHandler<CellPassageCreatedEventArgs> handler =
RaiseCellPassageCreatedEvent;
            //Ensures that the event will only update the ui if an actual change has
occurred
            if (handler != null)
            {
                handler(this, e);
            }
        }

        //For raising a route being added to event arg
        protected virtual void
OnRaiseRoutePathAddedNewCellEvent(RoutePathAddedNewCellEventArgs e)
        {
            EventHandler<RoutePathAddedNewCellEventArgs> handler =
RaiseRoutePathAddedNewCellEvent;
            //Ensures that the event will only update the ui if an actual change has
occurred
            if (handler != null)
            {
                handler(this, e);
            }
        }

        //For raising a highlightpath being added to event arg
        protected virtual void
OnRaiseHighlightPathAddedNewCellEvent(HighlightPathAddedNewCellEventArgs e)
        {
            EventHandler<HighlightPathAddedNewCellEventArgs> handler =
RaiseHighlightPathAddedNewCellEvent;
            if (handler != null)
            {
                handler(this, e);
```

```
            }
        }
    }
}
```

AStarAlgorithm

The A* Algorithm has only a solver. It solves mazes using a heuristic calculation of the distance the current cell is from the end cell in terms of the Manhatten distance (x and y distances added). The solver will, using this information, prefer moving along passageways where the next cells Manhatten distance is the lowest compared to all other cells that can be reached. This algorithm is often considered the fastest algorithm when getting the shortest path along a graph. This algorithm also features the use of the generic class **Dictionary<TKey, TValue>** which was very useful in storing all the information required for each cell in a easy to look up way.

```
using MazeLibrary.Events;
using MazeLibrary.Helpers;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace MazeLibrary.Algorithms
{
    public class AStarAlgorithm : AlgorithmBase
    {
        //https://en.wikipedia.org/wiki/A*_search_algorithm#Pseudocode

        public AStarAlgorithm(int delay)
        {
            //This constructor simply forces throughout the code a delay or pause for
the chosen time from the user
            _delay = delay;
        }

        public void Solve(ref Grid grid, out List<Cell> activeCells, out List<Cell>
closedSet)
        {
            {
                Cell startCell = grid.GetCell(0, 0);
                Cell endCell = grid.GetCell(grid.Width - 1, grid.Height - 1);

                //The set of cells already evaluated
               closedSet = new List<Cell>();

                //Set of currently discovered cells which are not evaluated yet.
                //Currently only the startCell is known
                List<Cell> openSet = new List<Cell>();
                openSet.Add(startCell);

                //For each cell, which cell it came from to get there, required to
record the path to any cell in the closedSet
                Dictionary<Cell, Cell> cameFrom = new Dictionary<Cell, Cell>();

                //To create a value for each cell in the grid, I set each cells
cameFrom to be null
                for (int y = 0; y < grid.Height; y++)
                {
                    for (int x = 0; x < grid.Width; x++)
```

```
                    {
                        Cell cell = grid.GetCell(x, y);
                        cameFrom[cell] = null;
                    }
                }

                //For each cell the 'cost' of getting there from the startCell. Every
cell starts with a default of infinity (largest number the integer can hold)
                Dictionary<Cell, int> gScore = new Dictionary<Cell, int>();
                for (int y = 0; y < grid.Height; y++)
                {
                    for (int x = 0; x < grid.Width; x++)
                    {
                        Cell cell = grid.GetCell(x, y);
                        gScore[cell] = Int32.MaxValue;
                    }
                }

                //Set the startCell's gScore to be zero as the cost to get from the
start to itself is zero
                gScore[startCell] = 0;

                //For each cell the 'cost' of getting to it from the startCell added
to the distance from it to the endCell.
                //Every cell starts with a default of infinity (largest number the
integer can hold)
                Dictionary<Cell, int> fScore = new Dictionary<Cell, int>();
                for (int y = 0; y < grid.Height; y++)
                {
                    for (int x = 0; x < grid.Width; x++)
                    {
                        Cell cell = grid.GetCell(x, y);
                        fScore[cell] = Int32.MaxValue;
                    }
                }

                //The fScore for getting from the start to the end is calculated in
the CalculateDistance function
                fScore[startCell] = CalculateDistance(startCell, endCell);

                //An indefinite loop used to run through every cell in openSet (as it
increases and decreases), until there are no more cells in the grid to evaluate
                while (openSet.Count > 0)
                {
                    OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, closedSet));

                    //currentCell equals the cell in the currently unevaluated cells
in openSet with the lowest distance from its self to the endCell.
                    Cell currentCell = GetCellWithLowestFScore(openSet, fScore);

                    //If true the algorithm has found the endCell. It must now
reproduce the specific route that got to that endCell in the following function
                    if (currentCell == endCell)
                    {
                        activeCells = reconstructPath(grid, cameFrom, currentCell);
                        return;
                    }

                    //Because the algorithm is now evaluating the lowest fScore cell:
currentCell, it must remove the cell from the openSet (unevaluated cells list) and add
it to the closedSet (evaluated cells list)
```

```
                    openSet.Remove(currentCell);
                    closedSet.Add(currentCell);

                    //Function returning a list to the cells around the currentCell
with a passageway between them
                    List<Cell> neighbours = GetTraversableNeighbours(grid,
currentCell);

                    //Runs through each of these neighbour cells individually
                    foreach (Cell neighbour in neighbours)
                    {

                        //If this neighbour has already been evaluated; ignore it
                        if (closedSet.Contains(neighbour))
                        {
                            continue;
                        }

                        //A temporary gScore for this neighbour is recorded. This is
calculated as the currentCell's gScore + 1
                        int tenativeGScore = gScore[currentCell] + 1;

                        //Should openSet not contain this neighbour cell it adds it as
it is an unevaluated cell it now knows about
                        if (!openSet.Contains(neighbour))
                        {
                            openSet.Add(neighbour);

                            //Raises an event to update the ui, namly the pen drawing
the path. Delays after to allow a user to see the change
                            //  OnRaiseRoutePathAddedNewCellEvent(new
RoutePathAddedNewCellEventArgs(grid, openSet)); //...Ugly
                            if (_delay > 0)
                            {
                                Thread.Sleep(_delay);
                            }
                        }
                        //If the openSet contains this neighbour, but the neighbour's
gScore is lower than or equal to the newly calculated gScore, it will be ignored
                        else if (tenativeGScore >= gScore[neighbour])
                        {
                            continue;
                        }

                        //As this neighbour is not in closedSet and has been added to
openSet it will be the next cell used.
                        //cameFrom for this neighbour will be currentCell
                        cameFrom[neighbour] = currentCell;

                        //gScore for this neighbour is the new gScore
                        gScore[neighbour] = tenativeGScore;

                        //fScore for this neighbour is it's gScore + the heuristic to
the end
                        fScore[neighbour] = gScore[neighbour] +
CalculateDistance(neighbour, endCell);
                    }
                }
                //Should the program have evaluated every cell, with no endCell being
found it throws an error as there must be no way to reach the endCell
                throw new Exception(string.Format("Error: unable to find endCell, at:
{0}", endCell.Name));
```

```
            }
        }

        private Cell GetCellWithLowestFScore(List<Cell> openSet, Dictionary<Cell, int>
fScore)
        {
            //Function compares all the fScores of cells in the openSet.
            // To setup intial variables I set the value of infinity (or as close as I
can get using int32s) to the currently lowest fScore cell: unknown
            int lowestFScore = Int32.MaxValue;
            Cell cellWithLowestFScore = null;

            //Simple foreach checking and comparing each cell of openSet with the next
one in it,
            // storing the cell with the currently lowest fScore in
cellWithLowestFScore with its fScore in lowestFScore
            foreach (Cell cell in openSet)
            {
                int currentFScore = fScore[cell];
                if (currentFScore < lowestFScore)
                {
                    lowestFScore = currentFScore;
                    cellWithLowestFScore = cell;
                }
            }
            //Returns the cell with the lowest fScore
            return cellWithLowestFScore;
        }

        private List<Cell> reconstructPath(Grid grid, Dictionary<Cell, Cell> cameFrom,
Cell currentCell)
        {
            List<Cell> totalPath = new List<Cell>();

            //The while loop below will not add the endCell as no cameFrom leads from
it, so it must be added first
            totalPath.Add(currentCell);

            //Ensures the dictionary contains the next cell, which should always be
true
            while (cameFrom.ContainsKey(currentCell))
            {
                currentCell = cameFrom[currentCell];

                //If null, or is the first index, break out of the loop and return the
path
                if (currentCell == null)
                    break;
                totalPath.Add(currentCell);

                //Updates ui canvas with route everytime a new cell is added to
totalPath and delays so the user can see it clearly
                OnRaiseRoutePathAddedNewCellEvent(new
RoutePathAddedNewCellEventArgs(grid, totalPath));
                if (_delay > 0)
                {
                    Thread.Sleep(_delay);
                }
            }
            return totalPath;
        }
```

```
        private List<Cell> GetTraversableNeighbours(Grid grid, Cell currentCell)
        {
            //Simply checks each of the four cells around currentCell to see if they
have a passageway to it and adds this cell to a list data structure to be returned.
            //It doesn't matter if the only cell returned is currentCell's cameFrom
cell as the returned cell is check if it is in both openSet and closedSet later on
            List<Cell> neighbours = new List<Cell>();
            for (int i = 0; i < DirectionHelper.Count; i++)
            {
                if (currentCell.GetEdge((Direction)i) is CellPassage)
                {
                    Point offset = DirectionHelper.GetOffset((Direction)i);
                    Point point = new Point(currentCell.Point.X + offset.X,
currentCell.Point.Y + offset.Y);
                    Cell neighbourCell = grid.GetCell(point);

                    neighbours.Add(neighbourCell);
                }
            }

            return neighbours;
        }

        private int CalculateDistance(Cell startCell, Cell endCell)
        {
            //Simply returns the Manhattan or rectilinear Distance between the two
cells, calculated as is shown
            return Math.Abs(endCell.Point.X - startCell.Point.X) +
Math.Abs(endCell.Point.Y - startCell.Point.Y);
        }
    }
}
```

BinaryTreeAlgorithm

The Binary Tree Algorithm only has maze generation code. It moves through every cell in the grid, top left to bottom right, and randomly decides if it should add a passageway North, adding a wall East or vice versa.

```
using MazeLibrary.Events;
using MazeLibrary.Helpers;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace MazeLibrary.Algorithms
{
    public class BinaryTreeAlgorithm : AlgorithmBase
    {
        //http://www.jamisbuck.org/presentations/rubyconf2011/index.html#binary-tree-
demo

        //Constructor updates the value currently in Delay in AlgorithmBase, in case
the user has changed it between runs of the program
        public BinaryTreeAlgorithm(int delay)
        {
            _delay = delay;
        }
```

```
        public void Generate(Grid grid)
        {
            Random random = new Random();

            //two definate iteration loops for addressing every cell in the grid.
            for (int y = 0; y < grid.Height; y++)
            {
                for (int x = 0; x < grid.Width; x++)
                {
                    Cell currentCell = grid.CreateCell(x, y);

                    OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, new List<Cell>(new Cell[] { currentCell })));

                    //possiblePoints is an array storing every point adjecent to the
currentCell
                    Point[] possiblePoints = new Point[] { new
Point(currentCell.Point.X, currentCell.Point.Y - 1), //North
                        new Point(currentCell.Point.X + 1, currentCell.Point.Y),
//East
                        new Point(currentCell.Point.X, currentCell.Point.Y + 1),
//South
                        new Point(currentCell.Point.X - 1, currentCell.Point.Y)};
//West

                    //Creates walls from the current cell to a null value if the
endpoint is not on the grid
                    for (int i = 0; i < DirectionHelper.Count; i++)
                    {
                        if (grid.ContainsPoint(possiblePoints[i]) == false)
                        {
                            grid.CreateWall(currentCell, null, (Direction)i);
                            OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, currentCell, (Direction)i));
                        }
                    }

                    //A random number to decide if:
                    //Case 0 (randSelected = 0); will create a passage north and a
wall west, thus ending the current hallway
                    //Case 1 (randSelected = 1); will create a wall north and a
passage west, thus continueing the current hallway
                    int randSelected = random.Next(0, 2);

                    if (randSelected == 0 && grid.ContainsPoint(possiblePoints[0]) &&
grid.ContainsPoint(possiblePoints[3]))
                    {
                        grid.CreatePassage(currentCell,
grid.GetCell(possiblePoints[0]), Direction.North);
                        OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, currentCell, grid.GetCell(possiblePoints[0]),
Direction.North));
                        grid.CreateWall(currentCell, grid.GetCell(possiblePoints[3]),
Direction.West);
                        OnRaiseCellWallCreatedEvent(new CellWallCreatedEventArgs(grid,
currentCell, Direction.West));
                    }
                    else if (randSelected == 1 &&
grid.ContainsPoint(possiblePoints[3]) && grid.ContainsPoint(possiblePoints[0]))
                    {
```

```
                        grid.CreatePassage(currentCell,
grid.GetCell(possiblePoints[3]), Direction.West);
                        OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, currentCell, grid.GetCell(possiblePoints[3]),
Direction.West));
                        grid.CreateWall(currentCell, grid.GetCell(possiblePoints[0]),
Direction.North);
                        OnRaiseCellWallCreatedEvent(new CellWallCreatedEventArgs(grid,
currentCell, Direction.North));
                    }

                    if (_delay > 0)
                    {
                        Thread.Sleep(_delay);
                    }
                }
            }
            //Once every cell has been addressed as much as it can be, the remaining
CellEdges must be filled otherwise the displaying algorithm in the Mazer>ViewModels
            //cannot display it
            FillRemainingEdges(grid);
        }

        private void FillRemainingEdges(Grid grid)
        {
            //another set of two definate iteration loops addressing every cell in the
grid
            for (int y = 0; y < grid.Height; y++)
            {
                for (int x = 0; x < grid.Width; x++)
                {
                    Cell currentCell = grid.GetCell(x, y);

                    //addresses each CellEdge of the currentCell
                    for (int i = 0; i < 4; i++)
                    {
                        //Casts i as a Direction: direction
                        Direction direction = (Direction)i;
                        //Gets the edge corrasponding to that direction from the
currentCell
                        CellEdge cellEdge = currentCell.GetEdge(direction);

                        //if this cellEdge is not a wall or a passage, i.e. is null
it...
                        if (!(cellEdge is CellWall) && !(cellEdge is CellPassage))
                        {
                            Point offset = DirectionHelper.GetOffset(direction);
                            Point point = new Point(currentCell.Point.X + offset.X,
currentCell.Point.Y + offset.Y);
                            //...finds if the point in that direction is on the grid
or not.
                            if (grid.ContainsPoint(point))
                            {
                                //if it is it creates a passage to it
                                Cell neighbourCell = grid.GetCell(point);

                                grid.CreatePassage(currentCell, neighbourCell,
direction);
                                //OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, currentCell, neighbourCell, direction));
                            }
                            else
```

```
                                {
                                    //if it isn't it creates a wall in its direction
                                    grid.CreateWall(currentCell, null, direction);
                                    //OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, currentCell, direction));
                                }
                            }
                        }
                    }
                }
            }

        public List<Cell> Solve(Grid _grid)
        {
            throw new NotImplementedException();
        }
    }
}
```

DrunkardsWalkAlgorithm

The Drunkard's Walk Algorithm has both maze generation and solver code. The generator works by randomly moving in the grid creating passageways as it goes. Every time it is about to create a passageway where a loop would be created, it creates a wall instead. The solver works by randomly moving around the maze, recording its path from the start, until the end cell is found.

```
using MazeLibrary.Events;
using MazeLibrary.Helpers;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace MazeLibrary.Algorithms
{
    public class DrunkardsWalkAlgorithm : AlgorithmBase
    {
        //http://www.jamisbuck.org/presentations/rubyconf2011/index.html#aldous-
broder-demo

        private Random random = new Random();

        //Constructor updates the value currently in Delay in AlgorithmBase, in case
the user has changed it between runs of the program
        public DrunkardsWalkAlgorithm(int delay)
        {
            _delay = delay;
        }

        public void Generate(Grid grid)
        {
            List<Cell> activeCells = new List<Cell>();

            Cell currentCell = grid.CreateCell(grid.RandomPoint());
            activeCells.Add(currentCell);

            int cellCount = grid.Height * grid.Width;
            while (activeCells.Count < cellCount)
```

```
            {
                OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, new List<Cell>(new Cell[] { currentCell })));

                //Finds a random cell neighbouring currentCell
                Direction direction = DirectionHelper.RandomDirection();
                Point offset = DirectionHelper.GetOffset(direction);
                Point point = new Point(currentCell.Point.X + offset.X,
currentCell.Point.Y + offset.Y);

                //If the point is not on the grid, it cannot create a cell there and
must instead create a wall in the direction variable's direction
                if (grid.ContainsPoint(point))
                {
                    //Attempts to get the cell where the point is...
                    Cell neighbourCell = grid.GetCell(point);

                    //If it is null (doesn't exist yet)...
                    if (neighbourCell == null)
                    {
                        //creates the cell and...
                        neighbourCell = grid.CreateCell(point);
                        CellEdge cellEdge = currentCell.GetEdge(direction);
                        //if there is no wall or passageway between currentCell and
neighbourCell...
                        if (cellEdge == null)
                        {
                            //creates a passageway, then updates the ui
                            grid.CreatePassage(currentCell, neighbourCell, direction);
                            OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, currentCell, neighbourCell, direction));
                        }

                        //Uses neighbourCell as the next cell to build from
                        currentCell = neighbourCell;
                    }
                    else
                    {
                        //If neighbour cell already exists, check the edge between the
two cells. If it is not set yet set it as a wall and update the ui
                        CellEdge cellEdge = currentCell.GetEdge(direction);
                        if (cellEdge == null)
                        {
                            grid.CreateWall(currentCell, neighbourCell, direction);
                            OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, currentCell, direction));
                        }

                        //Uses neighbourCell as the next cell to build from
                        currentCell = neighbourCell;
                    }
                }
                else
                {
                    //Creates a wall at the border, should a wall already be on that
edge it it doesn't waste time replacing it with a new one
                    CellEdge cellEdge = currentCell.GetEdge(direction);
                    if (cellEdge == null)
                    {
                        grid.CreateWall(currentCell, null, direction);
                        OnRaiseCellWallCreatedEvent(new CellWallCreatedEventArgs(grid,
currentCell, direction));
```

```
                    }
                }

                //A list being used to keep track of all cells with all their edges
set
                if (!activeCells.Contains(currentCell) &&
currentCell.IsFullyInitialized)
                {
                    activeCells.Add(currentCell);
                }

                //Delaying the ui updates so a user can view them
                if (_delay > 0)
                {
                    Thread.Sleep(_delay);
                }
            }
        }

        public void Solve(ref Grid grid, out List<Cell> activeCells, out List<Cell>
visitedCells)
        {
            //Sets startCell and endCell as the top left and bottom right
respectfully. The currentCell is startCell
            Cell startCell = grid.GetCell(0, 0);
            Cell endCell = grid.GetCell(grid.Width - 1, grid.Height - 1);
            Cell currentCell = startCell;

            //Adding currentCell to list of explored cells
            activeCells = new List<Cell>();
            visitedCells = new List<Cell>();
            activeCells.Add(currentCell);
            visitedCells.Add(currentCell);
            OnRaiseRoutePathAddedNewCellEvent(new RoutePathAddedNewCellEventArgs(grid,
activeCells));

            //Stops searching when the goal is found
            while (currentCell != endCell)
            {
                // OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, activeCells));

                //Gets a random direction and the cell that direction moves it to from
currentCell.
                //This is done by the following function which ensures that the
returned direction is through a passageway
                Direction direction = GetRandomPassage(grid, currentCell);

                Point offset = DirectionHelper.GetOffset(direction);
                Point point = new Point(currentCell.Point.X + offset.X,
currentCell.Point.Y + offset.Y);
                Cell nextCell = grid.GetCell(point);

                if (!visitedCells.Contains(nextCell))
                {
                    visitedCells.Add(nextCell);
                    OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, visitedCells));
                }

                //If the list doesn't contain the cell it adds it and updates the UI,
otherwise it removes the latest addition to the list.
```

```
                //(Stops multiples of the same cell being added)
                if (!activeCells.Contains(nextCell))
                {
                    activeCells.Add(nextCell);

                    OnRaiseRoutePathAddedNewCellEvent(new
RoutePathAddedNewCellEventArgs(grid, activeCells));
                }
                else
                {
                    activeCells.RemoveAt(activeCells.Count() - 1);
                    OnRaiseRoutePathAddedNewCellEvent(new
RoutePathAddedNewCellEventArgs(grid, activeCells));
                }

                //Delays the ui after any update
                if (_delay > 0)
                {
                    Thread.Sleep(_delay);
                }

                currentCell = nextCell;
            }

        }

        private Direction GetRandomPassage(Grid grid, Cell currentCell)
        {
            //Creates a list of all passageways from currentCell to it's neighbours
            List<Direction> directionList = new List<Direction>();

            for (int i = 0; i < DirectionHelper.Count; i++)
            {
                if (currentCell.GetEdge((Direction)i) is CellPassage)
                {
                    directionList.Add((Direction)i);
                }
            }
            //chooses one cell from that list and returns the direction to it.
            int index = random.Next(0, directionList.Count);
            return directionList[index];
        }
    }
}
```

HuntAndKillAlgorithm

The Hunt and Kill Algorithm has only maze generation code. It starts by applying the Drunkard's Walk, but with one constraint; it may never revisit a previously visited cell. This process would eventually create spiralled paths into the side of the maze or into itself. To overcome this the program then searches through the entire maze row by row from top to bottom until it finds an unvisited cell next to a visited cell. It then connects them and continues the path from this cell still following the constraint above. When no unvisited cell are detected the maze is done.

```
using MazeLibrary.Events;
using MazeLibrary.Helpers;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace MazeLibrary.Algorithms
{
    public class HuntAndKillAlgorithm : AlgorithmBase
    {
        //http://www.jamisbuck.org/presentations/rubyconf2011/index.html#hunt-and-
kill-demo

        //Constructor updates the value currently in Delay in AlgorithmBase, in case
the user has changed it between runs of the program
        public HuntAndKillAlgorithm(int delay)
        {
            _delay = delay;
        }

        ////Global 'stuck' variable used to tell the main function if the algorithm
has reached a dead end
        //static bool stuck;
        public void Generate(Grid grid)
        {
            Cell currentCell = grid.CreateCell(grid.RandomPoint());
            List<Cell> tempListCells = new List<Cell>();

            while (currentCell != null)
            {
                //OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, new List<Cell>(new Cell[] { currentCell })));



                if (!tempListCells.Contains(currentCell))
                {
                    tempListCells.Add(currentCell);
                    OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, tempListCells));
                }


                //If currentCell is fully initialised i.e has all its edges as non-
null values then the algorithm must find a new cell to process next.
                //Otherwise if it isn't it can process this one
                if (!currentCell.IsFullyInitialized)
                {
```

```
                    //In order to ensure no edge with a value is being replace with a
new value, direction is given a null value from currentCell
                    Direction direction =
currentCell.GetRandomUninitializedDirection();
                    Point offset = DirectionHelper.GetOffset(direction);
                    Point point = new Point(currentCell.Point.X + offset.X,
currentCell.Point.Y + offset.Y);

                    if (grid.ContainsPoint(point))
                    {
                        Cell neighbourCell = grid.GetCell(point);

                        //As the last, if statement filtered the possibility of the
neighbour being off the grid, it is safe to now ask if that cell is null
                        if (neighbourCell == null)
                        {
                            //Creates the edge between the old cell and the new
neighbour cell
                            neighbourCell = grid.CreateCell(point);
                            grid.CreatePassage(currentCell, neighbourCell, direction);
                            OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, currentCell, neighbourCell, direction));

                            //procedure to complete any other edges, if possible, for
the currentCell
                            CompleteAnyWalls(currentCell, grid);

                            currentCell = neighbourCell;
                        }
                        else
                        {
                            //Neighbour is already made, so a wall must seporate them,
or a loop may be made
                            grid.CreateWall(currentCell, neighbourCell, direction);
                            OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, currentCell, direction));
                        }
                    }
                    else
                    {
                        //Neighbour is off the grid
                        grid.CreateWall(currentCell, null, direction);
                        OnRaiseCellWallCreatedEvent(new CellWallCreatedEventArgs(grid,
currentCell, direction));
                    }
                }
                else
                {
                    //currentCell is done, such as if it has reached a deadend of 3
walls and one passageway
                    currentCell = Hunt(grid);
                    //Debug.WriteLine("New hunted at {0}", currentCell.Name);
                    //tempListCells.Clear();
                }

                if (_delay > 0)
                {
                    Thread.Sleep(_delay);
                }
            }
        }
```

```
        private Cell Hunt(Grid grid)
        {
            //Checks every cell, from the top-left to the bottom-right
            for (int y = 0; y < grid.Height; y++)
            {
                for (int x = 0; x < grid.Width; x++)
                {
                    Cell currentCell = grid.GetCell(x, y);

                    if (currentCell == null)
                    {
                        //Tests every direction from the null cell
                        for (int i = 0; i < DirectionHelper.Count; i++)
                        {
                            Direction direction = (Direction)i;
                            Point offset = DirectionHelper.GetOffset(direction);
                            Point point = new Point(x + offset.X, y + offset.Y);
                            if (grid.ContainsPoint(point))
                            {
                                Cell neighbourCell = grid.GetCell(point);
                                //The next if statement forces the program to only use
currentCell if it's neighbour is part of the already made maze structure.
                                //After this, if it isn't null, the generation wilol
continue from this cell next time round
                                if (neighbourCell != null)
                                {
                                    //Creates currentCell and
                                    currentCell = grid.CreateCell(x, y);

                                    grid.CreatePassage(currentCell, neighbourCell,
direction);
                                    OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, currentCell, neighbourCell, direction));
                                    if (_delay > 0)
                                    {
                                        Thread.Sleep(_delay);
                                    }

                                    CompleteAnyWalls(currentCell, grid);

                                    return currentCell;
                                }
                            }
                        }
                    }
                }
            }
            //There are no more non-null cells or all remaining cells are off the grid
it returns null to say 'there are no more cells left to use'
            return null;
        }

        private void CompleteAnyWalls(Cell currentCell, Grid grid)
        {
            //Checks every egde from the currentCell
            for (int i = 0; i < DirectionHelper.Count; i++)
            {
                Direction direction = (Direction)i;

                //Ensures the edge is not already set before it tries to replace it
                if (currentCell.GetEdge(direction) == null)
                {
```

```
                    Point offset = DirectionHelper.GetOffset(direction);
                    Point point = new Point(currentCell.Point.X + offset.X,
currentCell.Point.Y + offset.Y);
                    if (grid.ContainsPoint(point))
                    {
                        Cell neighbourCell = grid.GetCell(point);

                        //If that cell is on the grid and the cell isn't null, so is
part of the maze structure a wall is created to stop any loops being made later
                        if (neighbourCell != null)
                        {
                            grid.CreateWall(currentCell, neighbourCell, direction);
                            OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, currentCell, direction));
                        }
                    }
                    else
                    {
                        //If the cell isn't on the grid a border wall must be made
                        grid.CreateWall(currentCell, null, direction);
                        OnRaiseCellWallCreatedEvent(new CellWallCreatedEventArgs(grid,
currentCell, direction));
                    }
                }
                if (_delay > 0)
                {
                    Thread.Sleep(_delay);
                }
            }
        }

        public List<Cell> Solve(Grid grid)
        {
            throw new NotImplementedException();
        }

    }
}
```

KruskalsAlgorithm

The Kruskals Algorithm has only a maze generator code. This algorithm works by choosing a point at random to create a cell from. It then creates a passage to an adjacent cell. It then adds these two cells to a 2D multidimensional list forming groups of cells. If the neighbour is already part of a group, and the start cell is not part of a group, it adds the start cell to the already made group. If both cells are in their own groups then it adds all the cells from one of the groups to the other and deletes the first. Walls are created when otherwise a cycle would be created.

```
using MazeLibrary.Events;
using MazeLibrary.Helpers;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace MazeLibrary.Algorithms
{
    public class KruskalsAlgorithm : AlgorithmBase
    {
```

```
        //http://www.jamisbuck.org/presentations/rubyconf2011/index.html#kruskal-demo
        //Works by making passages between any random cell and its neighbour then
places these two in a new list,
        //if one of those cells belongs to another list it adds them both to that
list.
        //If both cells belong to different lists it merges both lists into one list

        //Constructor updates the value currently in Delay in AlgorithmBase, in case
the user has changed it between runs of the program
        public KruskalsAlgorithm(int delay)
        {
            _delay = delay;
        }

        public void Generate(Grid grid)
        {
            List<List<Cell>> cellListList = new List<List<Cell>>();
            List<Cell> completedCellList = new List<Cell>();
            int numberOfCells = grid.Width * grid.Height;

            //Only stops the algorithm when the number of completed cells is equal to
the maximum number of cells in the grid
            while (completedCellList.Count < numberOfCells)
            {
                Point randomPoint = grid.RandomPoint();
                Cell currentCell = grid.GetCell(randomPoint);
                if (currentCell == null)
                {
                    currentCell = grid.CreateCell(randomPoint);
                }

                //OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, new List<Cell>(new Cell[] { currentCell })));

                //Checks if the randomCell is part of a already made list
                List<Cell> currentCellList = GetCellList(currentCell, cellListList);
                if (currentCellList == null)
                {
                    //It isn't so make a new one
                    currentCellList = new List<Cell>();
                    currentCellList.Add(currentCell);
                    cellListList.Add(currentCellList);
                }


                if (!currentCell.IsFullyInitialized)
                {
                    Direction direction =
currentCell.GetRandomUninitializedDirection();
                    Point offset = DirectionHelper.GetOffset(direction);
                    Point point = new Point(currentCell.Point.X + offset.X,
currentCell.Point.Y + offset.Y);
                    if (grid.ContainsPoint(point))
                    {
                        //Simple part, if neighbour is null create the cell
                        Cell neighbourCell = grid.GetCell(point);

                        if (neighbourCell == null)
                        {
                            neighbourCell = grid.CreateCell(point);
                        }
```

```
                        OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, new List<Cell>(new Cell[] { currentCell,
neighbourCell })));

                        //NeighbourCell has already been added to the cellList so a
wall must be created to ensure no loops are made later
                        if (currentCellList.Contains(neighbourCell))
                        {
                            grid.CreateWall(currentCell, neighbourCell, direction);
                            OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, currentCell, direction));
                        }
                        else
                        {
                            //It isn't part of the currentCells' list so it starts the
process of adding neighbourCell to it
                            grid.CreatePassage(currentCell, neighbourCell, direction);
                            OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, currentCell, neighbourCell, direction));

                            //Checks if the neighbourCell has its own list
                            List<Cell> neighborCellList = GetCellList(neighbourCell,
cellListList);
                            if (neighborCellList == null)
                            {
                                //It doesn't so just add it
                                currentCellList.Add(neighbourCell);
                            }
                            else
                            {
                                //It does so add all cells in the neighbour's list to
the currentCells list and delete the neighbours list, removing it from the list of
lists
                                foreach (Cell cell in neighborCellList)
                                {
                                    currentCellList.Add(cell);
                                }
                                cellListList.Remove(neighborCellList);
                            }
                        }
                    }
                    else
                    {
                        //Edge becomes a border
                        grid.CreateWall(currentCell, null, direction);
                        OnRaiseCellWallCreatedEvent(new CellWallCreatedEventArgs(grid,
currentCell, direction));
                    }
                }
                else
                {
                    //Adds the completed cell to the list of completed cells
                    if (!completedCellList.Contains(currentCell))
                    {
                        completedCellList.Add(currentCell);
                    }
                }

                if (_delay > 0)
                {
                    Thread.Sleep(_delay);
                }
```

```
            }
        }

        private List<Cell> GetCellList(Cell cell, List<List<Cell>> cellListList)
        {
            //function returning the cellList which contains the requested cell
            foreach (List<Cell> cellList in cellListList)
            {
                if (cellList.Contains(cell))
                {
                    return cellList;
                }
            }
            return null;
        }


        public List<Cell> Solve(Grid _grid)
        {
            throw new NotImplementedException();
        }
    }
}
```

PrimsAlgorithm

Prims algorithm has only maze generation code. A root cell is chosen at random and is added to the spanning tree of visited cell. Then it marks all its adjacent cells as “frontier cells”. On the subsequent passes, it chooses one of the frontier cell at random and makes a passage to it, adding it to the spanning tree and marking all its adjacent cell as frontier cells. This process continues until there are no more frontier cells.

```
using MazeLibrary.Events;
using MazeLibrary.Helpers;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace MazeLibrary.Algorithms
{
    public class PrimsAlgorithm : AlgorithmBase
    {
        //http://www.jamisbuck.org/presentations/rubyconf2011/index.html#prim-demo

        //Constructor updates the value currently in Delay in AlgorithmBase, in case
the user has changed it between runs of the program
        public PrimsAlgorithm(int delay)
        {
            _delay = delay;
        }
        private Random _random = new Random();

        public void Generate(Grid grid)
        {
            //Discovered cells where at least one edge from it is null
            List<Cell> activeCells = new List<Cell>();

            //Discovered cells where all edges from  it are completed
```

```
            List<Cell> completedCells = new List<Cell>();

            Cell currentCell = grid.CreateCell(grid.RandomPoint());
            activeCells.Add(currentCell);

            int cellCount = grid.Height * grid.Width;

            //Only stops the algorithm when the number of completed cells is equal to
the maximum number of cells in the grid
            while (completedCells.Count < cellCount)
            {
                //The next cell it will prgress from is chosen randomly from the list
of active cells
                currentCell = activeCells[_random.Next(0, activeCells.Count)];//check

                OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, activeCells));

                //Gets a null direction from the currentCell to get a neighbour cell
to it later, if necessary
                Direction direction = currentCell.GetRandomUninitializedDirection();
                Point offset = DirectionHelper.GetOffset(direction);
                Point point = new Point(currentCell.Point.X + offset.X,
currentCell.Point.Y + offset.Y);


                if (grid.ContainsPoint(point))
                {
                    Cell neighbourCell = grid.GetCell(point);
                    if (neighbourCell == null)
                    {
                        neighbourCell = grid.CreateCell(point);
                        //As the cell is on the grid and as it hasn't been created
yet, a passage can be made to it
                        grid.CreatePassage(currentCell, neighbourCell, direction);
                        OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, currentCell, neighbourCell, direction));
                    }
                    else
                    {
                        //As the cell has been made already, a wall must be placed to
prevent creation of a loop
                        grid.CreateWall(currentCell, neighbourCell, direction);
                        OnRaiseCellWallCreatedEvent(new CellWallCreatedEventArgs(grid,
currentCell, direction));
                    }

                    //Checks if the currentCell is fully initialised if so removes it
from the activeCells and then adds it to the completedCells list
                    if (currentCell.IsFullyInitialized)
                    {
                        activeCells.Remove(currentCell);
                        completedCells.Add(currentCell);
                    }


                    currentCell = neighbourCell;
                }
                else
                {
                    //As the cell is off the grid a border wall must be placed
                    grid.CreateWall(currentCell, null, direction);
```

```
                    OnRaiseCellWallCreatedEvent(new CellWallCreatedEventArgs(grid,
currentCell, direction));
                }

                //If the new cell hasn't been added to the known cells and isn't fully
intialised then add it
                if (!activeCells.Contains(currentCell) &&
!currentCell.IsFullyInitialized)
                {
                    activeCells.Add(currentCell);
                }

                //Checks if the new currentCell is fully initialised if so removes it
from the activeCells and then adds it to the completedCells list
                if (currentCell.IsFullyInitialized)
                {
                    activeCells.Remove(currentCell);
                    completedCells.Add(currentCell);
                }

                //Delays so the UI can update and user can see any changes
                if (_delay > 0)
                {
                    Thread.Sleep(_delay);
                }

            }
        }

        public List<Cell> Solve(Grid _grid)
        {
            throw new NotImplementedException();
        }
    }
}
```

RecursiveBackTrackerAlgorithm

The Recursive Backtracker Algorithm has both maze generation and a solver code. It starts by applying the Drunkard's Walk with the constraint that it may never revisit an already visited cell. It also stores a list of each cell it visits in a generic **List<T>** (which is actually being used as a stack) placing each new cell on the top. Once it has spiralled into itself or the edge of the maze, it pops the top cells off the stack until if finds one that it can make a new route from.

```
using MazeLibrary.Events;
using MazeLibrary.Helpers;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace MazeLibrary.Algorithms
{
    public class RecursiveBackTrackerAlgorithm : AlgorithmBase
    {
        //http://www.jamisbuck.org/presentations/rubyconf2011/index.html#recursive-
backtracker-demo
```

```
        //Constructor updates the value currently in Delay in AlgorithmBase, in case
the user has changed it between runs of the program
        public RecursiveBackTrackerAlgorithm(int delay)
        {
            _delay = delay;
        }

        public void Generate(Grid grid)
        {
            //The list activeCells functions as a stack in the running of this
algorithm
            List<Cell> activeCells = new List<Cell>();
            Cell cell = grid.CreateCell(grid.RandomPoint());
            activeCells.Add(cell);
            //Debug.WriteLine(String.Format("First == ({0},{1}) ", cell.Point.X,
cell.Point.Y));

            while (activeCells.Count > 0)
            {
                OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, activeCells));

                //currentCell is now equal to the latest value in the list
                Cell currentCell = activeCells[activeCells.Count - 1];
                //Debug.Write(String.Format("Cell ({0},{1}): ", currentCell.Point.X,
currentCell.Point.Y));

                if (currentCell.IsFullyInitialized)
                {
                    //Removes the currentCell if it is fully intialised, or more
simply is at a dead end, etc.
                    activeCells.RemoveAt(activeCells.Count - 1);
                }
                else
                {
                    //If currentCell is not fully intialised then create an edge
between the currentCell
                    //and one of it's neighbours
                    Direction direction =
currentCell.GetRandomUninitializedDirection();
                    Point offset = DirectionHelper.GetOffset(direction);
                    Point point = new Point(currentCell.Point.X + offset.X,
currentCell.Point.Y + offset.Y);

                    if (grid.ContainsPoint(point))
                    {
                        //If neighbourCell doesn't exist make the cell and a passage
otherwise make a wall
                        Cell neighbourCell = grid.GetCell(point);
                        if (neighbourCell == null)
                        {
                            neighbourCell = grid.CreateCell(point);
                            activeCells.Add(neighbourCell);
                            //Debug.WriteLine(String.Format("Adding ({0},{1})",
point.X, point.Y));
                            grid.CreatePassage(currentCell, neighbourCell, direction);
                            OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, currentCell, neighbourCell, direction));
                        }
                        else
                        {
```

```
                            //Debug.WriteLine(String.Format("Neighbour Exists
({0},{1}) ", point.X, point.Y));
                            grid.CreateWall(currentCell, neighbourCell, direction);
                            OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, currentCell, direction));
                        }
                    }
                    else
                    {
                        //Create the border wall
                        //Debug.WriteLine(String.Format("Off Grid ({0},{1}) ",
point.X, point.Y));
                        grid.CreateWall(currentCell, null, direction);
                        OnRaiseCellWallCreatedEvent(new CellWallCreatedEventArgs(grid,
currentCell, direction));
                    }
                }

                if (_delay > 0)
                {
                    Thread.Sleep(_delay);
                }
            }
        }

        static bool deadendDetected;
        public void Solve(ref Grid grid, out List<Cell> activeCells, out List<Cell>
visitedCells)
        {
            Cell startCell = grid.GetCell(0, 0);
            Cell endCell = grid.GetCell(grid.Width - 1, grid.Height - 1);
            Cell currentCell = startCell;

            //activeCells will hold the final path once completed and inactiveCells
will hold all cells the program
            //will not have to recheck
            activeCells = new List<Cell>();
            List<Cell> inactiveCells = new List<Cell>();
            visitedCells = new List<Cell>();
            activeCells.Add(currentCell);
            visitedCells.Add(currentCell);
            OnRaiseRoutePathAddedNewCellEvent(new RoutePathAddedNewCellEventArgs(grid,
activeCells));

            while (currentCell != endCell)
            {
                deadendDetected = false;



                //Function returning a direction from currenctCell which is on a null
edge. Also this function checks
                //if the currentCells is now completed
                Direction direction = GetRandomPassage(grid, currentCell,
inactiveCells);

                Point offset = DirectionHelper.GetOffset(direction);
                Point point = new Point(currentCell.Point.X + offset.X,
currentCell.Point.Y + offset.Y);
                Cell nextCell = grid.GetCell(point);
                if (!visitedCells.Contains(nextCell))
                {
```

```
                    visitedCells.Add(nextCell);
                    OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, visitedCells));
                }



                if (!activeCells.Contains(nextCell))
                {
                    //Adds the nextCell if it isn't already in the list
                    activeCells.Add(nextCell);
                    OnRaiseRoutePathAddedNewCellEvent(new
RoutePathAddedNewCellEventArgs(grid, activeCells));

                    if (_delay > 0)
                    {
                        Thread.Sleep(_delay);
                    }
                }
                else if (deadendDetected == true)
                {
                    //Stuck is a boolean for telling this part of code if the cell is
stuck
                    //in a deadend, if so it must add that cell to inactiveCells so it
never tries to go
                    //there again and remove it from activeCells as it doesn't lead to
the endCell
                    inactiveCells.Add(activeCells[activeCells.Count - 1]);
                    activeCells.RemoveAt(activeCells.Count - 1);

                    OnRaiseRoutePathAddedNewCellEvent(new
RoutePathAddedNewCellEventArgs(grid, activeCells));
                    //Next it sets nextCell to the latest cell in activeCells, making
it step backwards
                    nextCell = activeCells[activeCells.Count - 1];

                    if (_delay > 0)
                    {
                        Thread.Sleep(_delay);
                    }

                    //All in all this if statement follows the procedure of popping an
item off the activeCell's stack
                    //adding it to the inactiveCell's stack, and then moves to the
last cell
                }
                else
                {
                    //If activeCells does contain nextCell but it isn't stuck make
nextCell equal the latest cell in activeCells.
                    //This makes sure the program does every path from a crossroads
before backtracking, otherwise it could skip
                    //over a path
                    nextCell = activeCells[activeCells.Count - 1];
                }

                currentCell = nextCell;
            }


        }
```

```
        private Direction GetRandomPassage(Grid grid, Cell currentCell, List<Cell>
inactiveCells)
        {
            Random random = new Random();
            List<Direction> directionList = new List<Direction>();

            //Collects all directions from currentCell...
            for (int i = 0; i < DirectionHelper.Count; i++)
            {
                //...that are passageway edges and...
                if (currentCell.GetEdge((Direction)i) is CellPassage)
                {
                    Point offset = DirectionHelper.GetOffset((Direction)i);
                    Point point = new Point(currentCell.Point.X + offset.X,
currentCell.Point.Y + offset.Y);
                    Cell testCell = grid.GetCell(point);

                    //...arn't part of the cells it doesn't need to recheck,
inactiveCells
                    if (!inactiveCells.Contains(testCell))
                    {
                        directionList.Add((Direction)i);
                    }
                }
            }

            if (directionList.Count == 1)
            {
                //The only direction it can go is where it came from as this is the
only passageway, therefore it is at a dead end
                deadendDetected = true;
            }

            //Returns a random direction from the list
            int index = random.Next(0, directionList.Count);
            return directionList[index];
        }



    }
}
```

RecursiveDivisionAlgorithm

The Recursive Division Algorithm has only maze generation code. This algorithm works by chopping the current grid area of all connected cells in half, by adding walls between cells. Next one random cell is chosen to have its wall replaced with a passage. The algorithm proceeds by then chopping each new sub-group in half and so on until the maze is created. This algorithm relies heavily on the use of **recursion** to implement the process of 'chopping' chopped areas over and over until nothing is left. Splitting the steps of doing the recursive process into smaller and more manageable pieces is a good example of **decomposition** of code into smaller units of work.

```
using MazeLibrary.Events;
using MazeLibrary.Helpers;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace MazeLibrary.Algorithms
{
    //http://www.jamisbuck.org/presentations/rubyconf2011/index.html#recursive-
division-demo
    public class RecursiveDivisionAlgorithm : AlgorithmBase
    {
        private static Random _random = new Random();

        //Constructor updates the value currently in Delay in AlgorithmBase, in case
the user has changed it between runs of the program
        public RecursiveDivisionAlgorithm(int delay)
        {
            _delay = delay;
        }

        public void Generate(Grid grid)
        {
            CreateAllCells(grid);
            DrawBorderWalls(grid);

            //Recursive procedure splitting the grid into a new portion over and over
until it is finished
            DoSplit(grid, 0, grid.Width, 0, grid.Height);

            //Due to the algorithm having some complications, which will be addressed
in the evaluation,
            //some edges aren't set they can, however, all be set to the same edge
type in this function
            FillRemaindingEdges(grid);
        }

        private void CreateAllCells(Grid grid)
        {
            //Does as the name suggests
            for (int y = 0; y < grid.Height; y++)
            {
                for (int x = 0; x < grid.Width; x++)
                {
                    grid.CreateCell(x, y);
                }
            }
        }
```

```
        private void DrawBorderWalls(Grid grid)
        {
            int width = grid.Width;
            int height = grid.Height;

            //procedures filling in all border cell's border edges with walls, thus
filling the entire row or column at once
            CreateWalls(grid, 0, width, 0, Direction.East);
            CreateWalls(grid, 0, width, height, Direction.East);
            CreateWalls(grid, 0, height, 0, Direction.South);
            CreateWalls(grid, 0, height, width, Direction.South);
        }

        private void DoSplit(Grid grid, int x0, int x1, int y0, int y1)
        {
            int width = x1 - x0;
            int height = y1 - y0;

            if (width >= height)
            {
                //The space is only recursively divided if there is room, i.e. more
than one cell's distance
                if (x1 - x0 > 1)
                {
                    //Width is greater than or equal to the height - Divide Vertically

                    int cut = _random.Next(x0 + 1, x1);

                    //forms all the walls vertically, (in terms of a grid makes walls
east or west but
                    //we perceive them as being vertical in the ui)
                    CreateWalls(grid, y0, y1, cut, Direction.South);
                    if (cut > 0 & cut < grid.Width)
                    {
                        //Must create one way through the wall which will be this
point:
                        CreatePassage(grid, cut, _random.Next(y0, y1),
Direction.West);
                    }

                    if (_delay > 0)
                    {
                        Thread.Sleep(_delay);
                    }

                    if (cut - x0 > 1)
                    {
                        //Divide region left of the cut as there is room on this side
                        DoSplit(grid, x0, cut, y0, y1);
                    }

                    if (x1 - cut > 1)
                    {
                        //Divide region right of the cut as there is room on this side
                        DoSplit(grid, cut, x1, y0, y1);
                    }
                }
            }
            else
            {
```

```
                //The space is only recursively divided if there is room, i.e. more
than one cell's distance
                if (y1 - y0 > 1)
                {
                    //Height is greater than or equal to the width - Divide
horizontally

                    int cut = _random.Next(y0 + 1, y1);

                    //forms all the walls vertically, (in terms of a grid makes walls
east or west but
                    //we perceive them as being vertical in the UI)

                    CreateWalls(grid, x0, x1, cut, Direction.East);
                    if (cut > 0 & cut < grid.Height)
                    {
                        //Must create one way through the wall which will be this
point:
                        CreatePassage(grid, _random.Next(x0, x1), cut,
Direction.North);
                    }

                    if (_delay > 0)
                    {
                        Thread.Sleep(_delay);
                    }

                    if (cut - y0 > 1)
                    {
                        //Divide region above the cut as there is room on this side
                        DoSplit(grid, x0, x1, y0, cut);
                    }

                    if (y1 - cut > 1)
                    {
                        //Divide region below the cut as there is room on this side
                        DoSplit(grid, x0, x1, cut, y1);
                    }
                }
            }
        }

        private void FillRemaindingEdges(Grid grid)
        {
            for (int y = 0; y < grid.Height; y++)
            {
                for (int x = 0; x < grid.Width; x++)
                {
                    Cell currentCell = grid.GetCell(x, y);

                    //Checks every direction from currentCell
                    for (int i = 0; i < 4; i++)
                    {
                        Direction direction = (Direction)i;
                        CellEdge cellEdge = currentCell.GetEdge(direction);
                        if (cellEdge == null)
                        {
                            Point offset = DirectionHelper.GetOffset(direction);
                            Point point = new Point(currentCell.Point.X + offset.X,
currentCell.Point.Y + offset.Y);
                            if (grid.ContainsPoint(point))
                            {
```

```
                                Cell neighbourCell = grid.GetCell(point);

                                //As the cellEdge is not set and is on th grid it can
be replaced with a passageway
                                grid.CreatePassage(currentCell, neighbourCell,
direction);
                                OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, currentCell, neighbourCell, direction));
                            }
                        }
                    }
                }
            }
        }

        private void CreatePassage(Grid grid, int x, int y, Direction direction)
        {
            //cell equals one of the cells on the wall section just created
            Cell cell = grid.GetCell(x, y);
            Point offset = DirectionHelper.GetOffset(direction);
            Point point = new Point(cell.Point.X + offset.X, cell.Point.Y + offset.Y);
            if (grid.ContainsPoint(point))
            {
                Cell neighbour = grid.GetCell(point);
                if (neighbour != null)
                {
                    //Overrides one of the walls with a passageway, doublechecking
that the neighbour is indeed existant
                    grid.CreatePassage(cell, neighbour, direction);
                    OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, cell, neighbour, direction));
                }
            }
        }

        private void CreateWalls(Grid grid, int start, int end, int offset, Direction
direction)
        {
            Cell cell = null;
            Cell neighbour = null;
            //Steps through every cell in the perpendicular to the variable
"direction"
            //filling in walls in the "direction" variable's direction
            for (int i = start; i < end; i++)
            {
                switch (direction)
                {
                    //As direction is either east or west cellEdges are being filled
in with walls going up or down,
                    //thus making a straight vertical wall
                    case Direction.East:
                    case Direction.West:
                        if (offset >= 0 && offset < grid.Height)
                        {
                            cell = grid.GetCell(i, offset);
                        }

                        if (offset > 0 && offset <= grid.Height)
                        {
                            neighbour = grid.GetCell(i, offset - 1);
                        }
```

```
                        if (cell != null)
                        {
                            grid.CreateWall(cell, neighbour, Direction.North);
                            OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, cell, Direction.North));
                        }
                        else
                        {
                            grid.CreateWall(neighbour, cell, Direction.South);
                            OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, neighbour, Direction.South));
                        }

                        if (_delay > 0)
                        {
                            Thread.Sleep(_delay);
                        }
                        break;

                    //As direction is either north or south cellEdges are being filled
in with walls going leaft or right,
                    //thus making a straight horizontal wall
                    case Direction.North:
                    case Direction.South:
                        if (offset >= 0 && offset < grid.Width)
                        {
                            cell = grid.GetCell(offset, i);
                        }

                        if (offset > 0 && offset <= grid.Width)
                        {
                            neighbour = grid.GetCell(offset - 1, i);
                        }

                        if (cell != null)
                        {
                            grid.CreateWall(cell, neighbour, Direction.West);
                            OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, cell, Direction.West));
                        }
                        else
                        {
                            grid.CreateWall(neighbour, cell, Direction.East);
                            OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, neighbour, Direction.East));
                        }

                        if (_delay > 0)
                        {
                            Thread.Sleep(_delay);
                        }
                        break;
                }
            }
        }

        public List<Cell> Solve(Grid _grid)
        {
            throw new NotImplementedException();
        }
    }
}
```

SidewinderAlgorithm

The Sidewinder Algorithm has only maze generator code. This algorithm looks at groups of adjacent cells. Each group of cells is added to a connected set of the spanning tree, then one cell is chosen at random in each group to create an passageway northwards, it then fills the rest of the upwards and sideways edges with walls.

```
using MazeLibrary.Events;
using MazeLibrary.Helpers;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace MazeLibrary.Algorithms
{
    public class SidewinderAlgorithm : AlgorithmBase
    {
        //Constructor updates the value currently in Delay in AlgorithmBase, in case
the user has changed it between runs of the program
        public SidewinderAlgorithm(int delay)
        {
            _delay = delay;
        }

        private Random _random = new Random();

        public void Generate(Grid grid)
        {
            PrepareGridCells(grid);

            Point neighbourPoint;
            //activeGroup is a list of cells in the current group it will be adding to
            List<Cell> activeGroup = new List<Cell>();

            for (int y = 0; y < grid.Height; y++)
            {
                //Removes all cells from the current group
                activeGroup.Clear();

                for (int x = 0; x < grid.Width; x++)
                {
                    Cell currentCell = grid.GetCell(x, y);
                    activeGroup.Add(currentCell);

                    OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, activeGroup));

                    //Random boolean decides if the next run should be a new group or
a continuation of the current group
                    bool endCurrentRun = _random.Next(0, 2) == 0;
                    if (y > 0 && (x == grid.Width - 1 || endCurrentRun))
                    {
                        //Ends the current run and carves north
                        //Choose a random cell from the activeGroup and carve north
from it
                        //activeGroup needs to be restarted

                        //Puts a wall east
```

```
                        neighbourPoint = new Point(currentCell.Point.X + 1,
currentCell.Point.Y);
                        if (grid.ContainsPoint(neighbourPoint))
                        {
                            Cell neighbourCell = grid.GetCell(neighbourPoint);
                            grid.CreateWall(currentCell, neighbourCell,
Direction.East);
                            OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, currentCell, Direction.East));

                            if (_delay > 0)
                            {
                                Thread.Sleep(_delay);
                            }
                        }

                        int r = _random.Next(0, activeGroup.Count);
                        for (int i = 0; i < activeGroup.Count; i++)
                        {
                            Cell groupCell = activeGroup[i];

                            neighbourPoint = new Point(groupCell.Point.X,
groupCell.Point.Y - 1);
                            Cell neighbourCell = grid.GetCell(neighbourPoint);

                            //Only if i is equal to the one place for a passage does
it make it, so no loops can be made
                            if (i == r)
                            {
                                //Make a passage North
                                grid.CreatePassage(groupCell, neighbourCell,
Direction.North);
                                OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, currentCell, neighbourCell, Direction.North));
                            }
                            else
                            {
                                //Make a wall North
                                grid.CreateWall(groupCell, neighbourCell,
Direction.North);
                                OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, currentCell, Direction.North));
                            }

                            if (_delay > 0)
                            {
                                Thread.Sleep(_delay);
                            }
                        }

                        activeGroup.Clear();
                    }
                    else
                    {
                        //Carve East
                        //Add the cell to the activeGroup
                        //Carve east from the previous cell
                        neighbourPoint = new Point(x + 1, y);
                        if (grid.ContainsPoint(neighbourPoint))
                        {
                            Cell neighbourCell = grid.GetCell(neighbourPoint);
```

```
                            grid.CreatePassage(currentCell, neighbourCell,
Direction.East);
                            OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, currentCell, neighbourCell, Direction.East));
                        }
                        else
                        {
                            grid.CreateWall(currentCell, null, Direction.East);
                            OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, currentCell, Direction.East));
                        }
                        //activeGroup.Add(currentCell);

                        if (_delay > 0)
                        {
                            Thread.Sleep(_delay);
                        }
                    }
                }
            }
        }

        private void PrepareGridCells(Grid grid)
        {
            //Creates every cell in the grid and also sets all the border edges to be
walls
            for (int y = 0; y < grid.Height; y++)
            {
                for (int x = 0; x < grid.Width; x++)
                {
                    Cell currentCell = grid.CreateCell(x, y);

                    if (y == 0)
                    {
                        //Left border
                        grid.CreateWall(currentCell, null, Direction.North);
                        OnRaiseCellWallCreatedEvent(new CellWallCreatedEventArgs(grid,
currentCell, Direction.North));

                        if (_delay > 0)
                        {
                            Thread.Sleep(_delay);
                        }
                    }

                    if (y == grid.Height - 1)
                    {
                        //Right border
                        grid.CreateWall(currentCell, null, Direction.South);
                        OnRaiseCellWallCreatedEvent(new CellWallCreatedEventArgs(grid,
currentCell, Direction.South));

                        if (_delay > 0)
                        {
                            Thread.Sleep(_delay);
                        }
                    }

                    if (x == 0)
                    {
                        //Top border
                        grid.CreateWall(currentCell, null, Direction.West);
```

```
                        OnRaiseCellWallCreatedEvent(new CellWallCreatedEventArgs(grid,
currentCell, Direction.West));

                        if (_delay > 0)
                        {
                            Thread.Sleep(_delay);
                        }
                    }

                    if (x == grid.Width - 1)
                    {
                        //Bottom border
                        grid.CreateWall(currentCell, null, Direction.East);
                        OnRaiseCellWallCreatedEvent(new CellWallCreatedEventArgs(grid,
currentCell, Direction.East));

                        if (_delay > 0)
                        {
                            Thread.Sleep(_delay);
                        }
                    }
                }
            }
        }

        public List<Cell> Solve(Grid _grid)
        {
            throw new NotImplementedException();
        }
    }
}
```

SimpleRectangleAlgorithm

This algorithm is a simple algorithm that was implemented to test the base algorithm class.

```
using MazeLibrary.Events;
using MazeLibrary.Helpers;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace MazeLibrary.Algorithms
{
    public  class SimpleRectangleAlgorithm : AlgorithmBase
    {
        //Algorithm has no source and will not generate a maze. This algorithm is only
included
        //as it was the starting point to testing if maze generation would be possible
with the approach here:

        //Constructor updates the value currently in Delay in AlgorithmBase, in case
the user has changed it between runs of the program
        public SimpleRectangleAlgorithm(int delay)
        {
            _delay = delay;
        }

        public  void Generate(Grid grid)
        {
            for (int y = 0; y < grid.Height; y++)
            {
                for (int x = 0; x < grid.Width; x++)
                {
                    //Creates a cell at every point on the grid, after all the passes
through the loop
                    Cell currentCell = grid.CreateCell(x, y);
                    OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, new List<Cell>(new Cell[] { currentCell })));

                    //An array is used to store all the points around the currentCell,
ergo its neighbours
                    Point[] possiblePoints = new Point[] { new
Point(currentCell.Point.X, currentCell.Point.Y - 1), //North
                        new Point(currentCell.Point.X + 1, currentCell.Point.Y),
//East
                        new Point(currentCell.Point.X, currentCell.Point.Y + 1),
//South
                        new Point(currentCell.Point.X - 1, currentCell.Point.Y)};
//West

                    //Fills any border cell's edges, which lead off the grid, with
walls
                    for (int i = 0; i < DirectionHelper.Count; i++)
                    {
                        if (grid.ContainsPoint(possiblePoints[i]) == false)
                        {
                            grid.CreateWall(currentCell, null, (Direction)i);
                            OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, currentCell, (Direction)i));
```

```
                            if (_delay > 0)
                            {
                                Thread.Sleep(_delay);
                            }
                        }
                    }

                    //Only checks points north and east of the currentCell. It fills
all edges in the grid, bar border edges, with passages
                    if (grid.ContainsPoint(possiblePoints[0]))
                    {
                        grid.CreatePassage(currentCell,
grid.GetCell(possiblePoints[0]), Direction.North);
                        OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, currentCell, grid.GetCell(possiblePoints[0]),
Direction.North));
                    }
                    if (grid.ContainsPoint(possiblePoints[3]))
                    {
                        grid.CreatePassage(currentCell,
grid.GetCell(possiblePoints[3]), Direction.West);
                        OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, currentCell, grid.GetCell(possiblePoints[3]),
Direction.West));
                    }

                    //By setting edges to both walls and borders I have tested that
they both work as intended

                    if (_delay > 0)
                    {
                        Thread.Sleep(_delay);
                    }

                }
            }
        }

        public  List<Cell> Solve(Grid _grid)
        {
            throw new NotImplementedException();
        }
    }
}
```

WallFollowerAlgorithm

The WallFollowerAlgorithm has two different but related code implementations for solving mazes. The SolveLeft one continually priorities taking the left edge from every cell it visits to go next. If that cell has no left edge, it then tries forwards, right and finally the direction it came from. This eventually gets it to the end cell. The SolveRight one does the same but prioritising the right most paths instead.

```
using MazeLibrary.Events;
using MazeLibrary.Helpers;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
```

```
namespace MazeLibrary.Algorithms
{
    public class WallFollowerAlgorithm : AlgorithmBase
    {
        //Didn't use a source for the solving algorithms for this code, as its
procedure of working is easy to understand.
        // All it does is for the solveLeft it takes every opportunity to go left a
the next cell it gets to.
        // If it can't it tries straight on if it can't again it tries right and then,
finally, if wrong again it goes backwards.
        //In a straight hallway, therefore, it would try and go left, can't as a wall
is on that edge, tries going straight on (the same as left from that wall), it can so
it moves that way.
        // This process is the same for the solveRight algorithm just prioritising
right paths instead

        //Constructor updates the value currently in Delay in AlgorithmBase, in case
the user has changed it between runs of the program
        public WallFollowerAlgorithm(int delay)
        {
            _delay = delay;
        }

        public void SolveLeft(ref Grid grid, out List<Cell> activeCells, out
List<Cell> visitedCells)
        {
            activeCells = new List<Cell>();
            visitedCells = new List<Cell>();

            Cell startCell = grid.GetCell(0, 0);
            Cell endCell = grid.GetCell(grid.Width - 1, grid.Height - 1);

            Cell currentCell = startCell;
            activeCells.Add(currentCell);
            visitedCells.Add(currentCell);

            //Must assume that the entrance is from the north as I have favoured for
all my algorithms (These are the only two that are affected by it though)
            Direction cameFrom = Direction.North;

            while (currentCell != endCell)
            {
                //Function returns the left-most path at currentCell, where the
direction it came from is given by cameFrom
                Direction lefty = FindLeft(grid, currentCell, cameFrom);

                Point offset = DirectionHelper.GetOffset(lefty);
                Point point = new Point(currentCell.Point.X + offset.X,
currentCell.Point.Y + offset.Y);
                Cell neighbourCell = grid.GetCell(point);

                currentCell = neighbourCell;

                //If cell isn't in activeCells, is a new one so adds it in...
                if (!activeCells.Contains(currentCell))
                {
                    activeCells.Add(currentCell);
                }
                else
                {
```

```
                    //...else removes this cell as the program is trying to move
backwards from a dead end or visitedCells
                    activeCells.RemoveAt(activeCells.Count - 1);

                }

                //If not contained in visitedCells then add it in as now it has been
visited
                if (!visitedCells.Contains(currentCell))
                {
                    visitedCells.Add(currentCell);
                    //Update highlight path
                    OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, visitedCells));
                }

                //Update routePath
                OnRaiseRoutePathAddedNewCellEvent(new
RoutePathAddedNewCellEventArgs(grid, activeCells));

                if (_delay > 0)
                {
                    Thread.Sleep(_delay);
                }

                //cameFrom direction is now set as its opposite as before it was a
going to direction, so it in now reversed to be a came from direction again
                cameFrom = DirectionHelper.GetOpposite(lefty);
            }

        }

        private static Direction FindLeft(Grid grid, Cell currentCell, Direction
cameFrom)
        {
            //For checks on each edge of currentCell I check the 4 states of direction
            CellEdge testEdge;
            for (int i = 0; i < DirectionHelper.Count; i++)
            {
                //As going left is dependant on the direction you are facing or the
direction you came from, a check for cameFrom's direction is necessary
                if (cameFrom == Direction.North)
                {
                    //If cameFrom was north, the next cameFrom must be east as now it
must try and go right of the way it was facing
                    cameFrom = Direction.East;
                    testEdge = currentCell.GetEdge(cameFrom);
                    if (testEdge is CellPassage)
                    {
                        //If the testEdge was a cellPassage then it can go left, so
breaks out of the loop.
                        // If it isn't then below it will try the direction as if
cameFrom was east, i.e. as if it came from the wall just discovered and was trying to
go left from that position.
                        break;
                    }
                }
                if (cameFrom == Direction.East)
                {
                    //If cameFrom was east, the next cameFrom must be south as now it
must try and go right of the way it was facing
                    cameFrom = Direction.South;
```

```
                    testEdge = currentCell.GetEdge(cameFrom);
                    if (testEdge is CellPassage)
                    {
                        //If the testEdge was a cellPassage then it can go left, so
breaks out of the loop.
                        // If it isn't then below it will try the direction as if
cameFrom was south, i.e. as if it came from the wall just discovered and was trying to
go left from that position.
                        break;
                    }
                }
                if (cameFrom == Direction.South)
                {
                    //If cameFrom was south, the next cameFrom must be west as now it
must try and go right of the way it was facing
                    cameFrom = Direction.West;
                    testEdge = currentCell.GetEdge(cameFrom);
                    if (testEdge is CellPassage)
                    {
                        //If the testEdge was a cellPassage then it can go left, so
breaks out of the loop.
                        // If it isn't then below it will try the direction as if
cameFrom was west, i.e. as if it came from the wall just discovered and was trying to
go left from that position.
                        break;
                    }
                }
                if (cameFrom == Direction.West)
                {
                    //If cameFrom was west, the next cameFrom must be north as now it
must try and go right of the way it was facing
                    cameFrom = Direction.North;
                    testEdge = currentCell.GetEdge(cameFrom);
                    if (testEdge is CellPassage)
                    {
                        //If the testEdge was a cellPassage then it can go left, so
breaks out of the loop.
                        // If it isn't then below it will try the direction as if
cameFrom was north, i.e. as if it came from the wall just discovered and was trying to
go left from that position.
                        break;
                    }
                }
            }
            //At this time, the cameFrom direction is now equal to the direction the
next cell must have came from to get where it is now
            return cameFrom;
        }



        public void SolveRight(ref Grid grid, out List<Cell> activeCells, out
List<Cell> visitedCells)
        {
            activeCells = new List<Cell>();
          visitedCells = new List<Cell>();


            Cell startCell = grid.GetCell(0, 0);
            Cell endCell = grid.GetCell(grid.Width - 1, grid.Height - 1);

            Cell currentCell = startCell;
```

```
            activeCells.Add(currentCell);
            visitedCells.Add(currentCell);

            //Must assume that entrance from the north
            Direction cameFrom = Direction.North;

            while (currentCell != endCell)
            {
                //Function returns the right-most path at currentCell, where the
direction it came from is given by cameFrom
                Direction righty = FindRight(grid, currentCell, cameFrom);


                Point offset = DirectionHelper.GetOffset(righty);
                Point point = new Point(currentCell.Point.X + offset.X,
currentCell.Point.Y + offset.Y);
                Cell neighbourCell = grid.GetCell(point);


                currentCell = neighbourCell;

                //If cell isn't in activeCells, is a new one so adds it in...
                if (!activeCells.Contains(currentCell))
                {
                    activeCells.Add(currentCell);
                }
                else
                {
                    //...else removes this cell as the program is trying to move
backwards from a dead end or visitedCells
                    activeCells.RemoveAt(activeCells.Count - 1);

                }

                //If not contained in visitedCells then add it in as now it has been
visited
                if (!visitedCells.Contains(currentCell))
                {
                    visitedCells.Add(currentCell);
                    //Update highlight path
                    OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, visitedCells));
                }

                //Update routePath
                OnRaiseRoutePathAddedNewCellEvent(new
RoutePathAddedNewCellEventArgs(grid, activeCells));

                if (_delay > 0)
                {
                    Thread.Sleep(_delay);
                }

                //cameFrom direction is now set as its opposite as before it was a
going to direction, so it in now reversed to be a came from direction again
                cameFrom = DirectionHelper.GetOpposite(righty);
            }


        }
```

```
        private static Direction FindRight(Grid grid, Cell currentCell, Direction
cameFrom)
        {
            //For checks on each edge of currentCell, I check the 4 states of
direction
            CellEdge testEdge;

            for (int i = 0; i < DirectionHelper.Count; i++)
            {
                //As going left is dependant on the direction you are facing or the
direction you came from, a check for cameFrom's direction is necessary
                if (cameFrom == Direction.North)
                {
                    //If cameFrom was north, the next cameFrom must be west as now it
must try and go right of the way it was facing
                    cameFrom = Direction.West;
                    testEdge = currentCell.GetEdge(cameFrom);
                    if (testEdge is CellPassage)
                    {
                        //If the testEdge was a cellPassage then it can go right, so
breaks out of the loop.
                        // If it isn't then below it will try the direction as if
cameFrom was west, i.e. as if it came from the wall just discovered and was trying to
go right from that position.
                        break;
                    }
                }
                if (cameFrom == Direction.West)
                {
                    //If cameFrom was west, the next cameFrom must be south as now it
must try and go right of the way it was facing
                    cameFrom = Direction.South;
                    testEdge = currentCell.GetEdge(cameFrom);
                    if (testEdge is CellPassage)
                    {
                        //If the testEdge was a cellPassage then it can go right, so
breaks out of the loop.
                        // If it isn't then below it will try the direction as if
cameFrom was south, i.e. as if it came from the wall just discovered and was trying to
go right from that position.
                        break;
                    }
                }
                if (cameFrom == Direction.South)
                {
                    //If cameFrom was south, the next cameFrom must be east as now it
must try and go right of the way it was facing
                    cameFrom = Direction.East;
                    testEdge = currentCell.GetEdge(cameFrom);
                    if (testEdge is CellPassage)
                    {
                        //If the testEdge was a cellPassage then it can go right, so
breaks out of the loop.
                        // If it isn't then below it will try the direction as if
cameFrom was east, i.e. as if it came from the wall just discovered and was trying to
go right from that position.
                        break;
                    }
                }
                if (cameFrom == Direction.East)
                {
```

```
                    //If cameFrom was east, the next cameFrom must be north as now it
must try and go right of the way it was facing
                    cameFrom = Direction.North;
                    testEdge = currentCell.GetEdge(cameFrom);
                    if (testEdge is CellPassage)
                    {
                        //If the testEdge was a cellPassage then it can go right, so
breaks out of the loop.
                        // If it isn't then below it will try the direction as if
cameFrom was north, i.e. as if it came from the wall just discovered and was trying to
go right from that position.
                        break;
                    }
                }
            }

            //At this time, the cameFrom direction is now equal to the direction the
next cell must have camefrom to get where it is now
            return cameFrom;
        }
    }
}
```

WilsonsAlgorithm

The Wilson's Algorithm would only have maze generation code. It should move randomly from cell to cell creating edges between them. Once a completed route is found from a specified start cell to a specified end cell, it would be add it to the spanning tree, and the process would then be repeated, except that each time around, any point already in the tree is a valid ending cell so subsequent iterations complete much more quickly. I did not finish the implementation of this algorithm due to its requirement of storing a direction in every cell, which I will go into further detail in the Evaluation.

```
using MazeLibrary.Events;
using MazeLibrary.Helpers;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace MazeLibrary.Algorithms
{
    public class WilsonsAlgorithm : AlgorithmBase
    {
        static private Random _random = new Random();

        //Constructor updates the value currently in Delay in AlgorithmBase, in case
the user has changed it between runs of the program
        public WilsonsAlgorithm(int delay)
        {
            _delay = delay;
        }

        public void Generate(Grid grid)
        {
            throw new NotImplementedException();

            //List<Point> pointsToFind = new List<Point>();
            //pointsToFind.Add(new Point(0, 0));
```

```
            //List<Point> activePoints = new List<Point>();
            //List<Direction> directionList = new List<Direction>();

            //int gridSize = grid.Height * grid.Width;
            //do
            //{
            //    activePoints.Clear();
            //    directionList.Clear();

            //    Point point = GetNewStartCell(grid);
            //    Point currentPoint = point;
            //    activePoints.Add(currentPoint);
            //    //   OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, activePoints));

            //    while (!pointsToFind.Contains(currentPoint))
            //    {
            //        Direction direction = DirectionHelper.RandomDirection();
            //        Point offset = DirectionHelper.GetOffset(direction);
            //        point = new Point(currentPoint.X + offset.X, currentPoint.Y +
offset.Y);
            //        if (grid.ContainsPoint(point))
            //        {
            //            if (!pointsToFind.Contains(point) &&
!activePoints.Contains(point))
            //            {
            //                currentPoint = point;//unknown point
            //            }
            //            //else
            //            //{
            //            //    currentPoint = grid.GetCell(point);
            //            //}

            //            if (activePoints.Count <= 1 || !(currentPoint ==
activePoints[activePoints.Count - 2]))//! does not allow backtracking
            //            {

            //                if (activePoints.Contains(currentPoint))//hit itself
farther than one back
            //                {

            //                    while (activePoints[activePoints.Count - 1] !=
currentPoint)
            //                    {
            //                        activePoints.RemoveAt(activePoints.Count - 1);
            //                        directionList.RemoveAt(directionList.Count - 1);
            //                        //
OnRaiseHighlightPathAddedNewCellEvent(new HighlightPathAddedNewCellEventArgs(grid,
ConvertToCells(activePoints)));
            //                    }

            //                }
            //                else//new point
            //                {

            //                    activePoints.Add(currentPoint);
            //                    directionList.Add(direction);
            //                    //       OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, activePoints));
```

```
            //                    //  OnRaiseHighlightPathAddedNewCellEvent(new
HighlightPathAddedNewCellEventArgs(grid, activeCells));
            //                }

            //            }
            //            else
            //            {
            //                currentPoint = activePoints[activePoints.Count - 1];
            //            }
            //        }

            //        if (_delay > 0)
            //        {
            //            Thread.Sleep(_delay);
            //        }
            //        // Debug.WriteLine(currentPoint.Name);
            //    }

            //    for (int i = 0; i < activePoints.Count - 1; i++)
            //    {
            //        Cell c1 = grid.CreateCell(activePoints[i]);
            //        Cell c2 = grid.CreateCell(activePoints[i + 1]);
            //        Direction d = directionList[i];

            //        grid.CreatePassage(c1, c2, d);
            //        OnRaiseCellPassageCreatedEvent(new
CellPassageCreatedEventArgs(grid, c1, c2, d));

            //        for (int f = 0; f < DirectionHelper.Count; f++)
            //        {

            //            d = (Direction)i;
            //            CellEdge cellEdge = c1.GetEdge(d);
            //            Point offset = DirectionHelper.GetOffset(d);
            //            point = new Point(c1.Point.X + offset.X, c1.Point.Y +
offset.Y);
            //            Cell neighbourCell = grid.GetCell(point);

            //            if (neighbourCell != null && cellEdge == null)
            //            {
            //                grid.CreateWall(c1, grid.GetCell(point), d);
            //                OnRaiseCellWallCreatedEvent(new
CellWallCreatedEventArgs(grid, c1, d));
            //            }
            //        }


            //    }

            //    foreach (Point p in activePoints)
            //    {
            //        pointsToFind.Add(p);
            //    }
            //    pointsToFind.Distinct().ToList();

            //} while (pointsToFind.Count < gridSize);
            ////should now have list

            ////foreach point in activeCells
            ////   create cell, add to cellList
            ////foreach cell in cellList
            ////, sort all edges of cells in cellList
```

```
            //private List<Cell> ConvertToCells(List<Point> activePoints)
            //{
            //    List<Cell> activeCells = new List<Cell>();
            //        foreach (Point p in activePoints)
            //        {

            //        }
            //}


            //display_maze(grid)



            //private Point GetNewStartCell(Grid grid)
            //{
            //    for (int y = 0; y < grid.Height; y++)
            //    {
            //        for (int x = 0; x < grid.Width; x++)
            //        {
            //            if (grid.GetCell(x, y) == null)
            //            {
            //                return new Point(x, y);
            //            }
            //        }
            //    }
            //    return new Point(-1, -1);
            //}
        }

        public List<Cell> Solve(Grid grid)
        {
            throw new NotImplementedException();
        }
    }
}
```

Mazer Project

Main Windows and Dialogs

The main windows and dialogs of the Mazer application are described in the following sections. The source code for any WPF window is expressed in **XAML**. This is an **XML-based markup language** used to describe the user interface (UI) of a window or dialog. The Mazer WPF windows and dialogs rely on data binding in order to connect up the UI controls to objects in my View Model. This means that when properties in the View Model are updated the UI controls that are bound to those properties automatically update to reflect the changes. The data bindings go in the other direction as well. When a user changes the UI control states (e.g. they move the zoom slider up or down) the properties in the View Model automatically update.

Multi-Threading

The Mazer UI remains responsive through the use of **multi-threading** in both its main windows, the MazeGeneratorWindow.xaml and MazeSolverWindow.xaml.

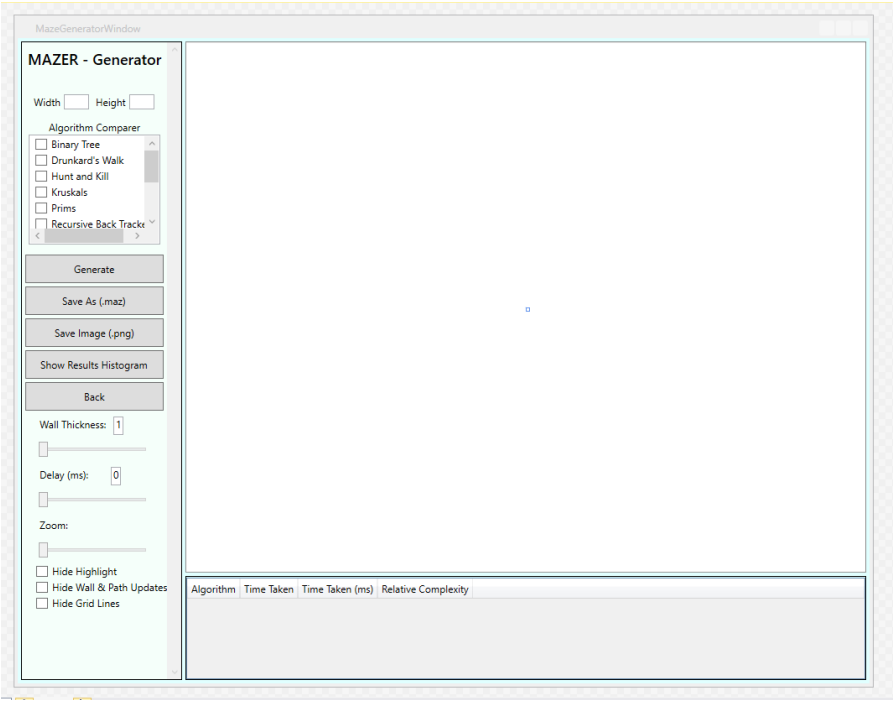
For example in the MazeGeneratorWindow.xaml the `btnGenerate_Click()` routine uses an `await Task.Run()` call to execute a block of code that calls the various selected maze algorithms. This `Task.Run()` ensures the block of code is started as a new task on a separate thread to the main user interface. The `await` causes `btnGenerate_Click()` to immediately return control back to the main window allowing it to remain responsive. This allows the user to carry on resizing the window, minimising it etc. as the algorithms execute in the background. When this algorithm task running on the extra thread finishes control immediately resumes execution at the statement after the `await` call. This is a good example of **multi-threading**.

As the maze generator is executing (on its own thread) it from time to time fires **events** along with **event arguments** that describe what it is doing. Any window that is hooked up to listen to these events can act upon them as they arrive and update the UI appropriately.

An identical technique is also followed in the `btnSolve_Click()` routine in the MazeSolverWindow window.

MazeGeneratorWindow

The generator window is displayed below. The UI elements and layout of this window closely follow the original design. The final implementation of the window does include a few extra features. These extra features were added to the UI making the application more graphically interesting to use. However, I am pleased with how closely I have been able to follow the original main design for this screen.



MazeGeneratorWindow.xaml

```
<Window x:Class="Mazer.MazeGeneratorWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:converters="clr-namespace:Mazer.Converters"
        Title="MazeGeneratorWindow" Height="800" Width="1024">
    <Window.Resources>
        <ResourceDictionary>
            <converters:HalfValueConverter x:Key="MyHalfValueConverter"/>
            <converters:RectConverter x:Key="MyRectConverter"/>
        </ResourceDictionary>
    </Window.Resources>

    <Grid Background="LightCyan">
        <Grid.Resources>
            <ResourceDictionary>
                <Style TargetType="Button">
                    <Setter Property="Margin" Value="4,4,4,0"/>
                    <Setter Property="Padding" Value="8"/>
                </Style>
            </ResourceDictionary>
        </Grid.Resources>

        <Grid.RowDefinitions>
            <RowDefinition Height="5*"/>
            <RowDefinition Height="1*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="200"/>
            <ColumnDefinition Width="1*"/>
        </Grid.ColumnDefinitions>

        <Border BorderBrush="Black" BorderThickness="1" Grid.RowSpan="2" Margin="4">
            <ScrollViewer >
                <StackPanel Background="MintCream" >
                    <TextBlock Text="MAZER - Generator" FontSize="18"
FontWeight="SemiBold" TextWrapping="WrapWithOverflow" HorizontalAlignment="Center"
Margin="0,8,0,4"/>
```

```
                    <TextBlock Text="{Binding AlgorithimLabel}"
HorizontalAlignment="Left" Margin="8,4,0,0" VerticalAlignment="Top" Width="164"
FontWeight="Bold" Height="14" Foreground="#FF0331FF" FontSize="8"/>
                    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center"
Margin="0,4,0,0">
                        <TextBlock Text="Width" Margin="4,0,0,0"
VerticalAlignment="Center"/>
                        <TextBox x:Name="txtbxMazeWidth" Width="30" Margin="4"
PreviewKeyDown="txtbxMazeWidth_PreviewKeyDown" Text="{Binding Width}"/>
                        <TextBlock Text="Height"  Margin="4,0,0,0"
VerticalAlignment="Center"/>
                        <TextBox x:Name="txtbxMazeHeight" Width="30" Margin="4"
PreviewKeyDown="txtbxMazeHeight_PreviewKeyDown" Text="{Binding Height}"/>
                    </StackPanel>

                    <StackPanel Margin="8">
                        <TextBlock Text="Algorithm Comparer"
HorizontalAlignment="Center" Margin="1"/>
                        <ListBox Height="131">
                            <CheckBox x:Name="BinaryTreeBox" Content="Binary Tree"/>
                            <CheckBox x:Name="DrunkardsWalkBox" Content="Drunkard's
Walk"/>
                            <CheckBox x:Name="HuntandKillBox" Content="Hunt and Kill"
/>
                            <CheckBox x:Name="KruskalsBox" Content="Kruskals"/>
                            <CheckBox x:Name="PrimsBox" Content="Prims" />
                            <CheckBox x:Name="RecursiveBackTrackerBox"
Content="Recursive Back Tracker" />
                            <CheckBox x:Name="RecursiveDivisionBox" Content="Recursive
Division"/>
                            <CheckBox x:Name="SidewinderBox" Content="Sidewinder"/>
                            <CheckBox x:Name="SimpleRectangleBox" Content="Simple
Rectangle"/>
                            <CheckBox x:Name="WilsonsAlgorithmBox" Content="Wilsons"
                                  />
                        </ListBox>
                    </StackPanel>
                    <Button x:Name="btnGenerate" Content="Generate"
Click="btnGenerate_Click"/>
                    <Button x:Name="btnSaveAsMaz" Content="Save As (.maz)"
Click="btnSaveGridAsMazFile_Click"/>
                    <Button x:Name="btnSaveImage" Content="Save Image (.png)"
Click="btnSaveImage_Click"/>
                    <Button x:Name="btnSaveResults" Content="Show Results Histogram"
Click="btnSaveResults_Click"/>
                    <Button x:Name="btnBack" Content="Back" Click="btnBack_Click"/>
                    <StackPanel>
                        <Grid>
                            <Label Content="Wall Thickness:" Margin="16,3,0,3"
Height="26"/>
                            <TextBox x:Name="txtWallThickness" Text="{Binding Value,
ElementName=sldWallThickness, Mode=TwoWay}" HorizontalAlignment="Center"
Margin="104,7,47,3" />
                        </Grid>
                        <Slider x:Name="sldWallThickness" VerticalAlignment="Center"
Margin="20,5,20,5" Minimum="1" Maximum="20" SmallChange="1" LargeChange="5"
TickFrequency="1" IsSnapToTickEnabled="True" Value="{Binding WallThickness}"/>
                    </StackPanel>
                    <StackPanel>
                        <Grid>
                            <Label Content="Delay (ms):" Margin="16,3,80,3"
Height="26"/>
```

```
                            <TextBox x:Name="txtDelay" Text="{Binding Value,
ElementName=sldDelay, Mode=TwoWay}" HorizontalAlignment="Center" Margin="100,7,50,3"/>
                        </Grid>
                        <Slider x:Name="sldDelay" VerticalAlignment="Center"
Margin="20,5,20,5" Minimum="0" Maximum="2000" SmallChange="10" LargeChange="50"
TickFrequency="1" IsSnapToTickEnabled="True"  Value="{Binding Delay}"/>
                    </StackPanel>
                    <Label Content="Zoom:" Margin="16,3,80,3"/>
                    <Slider x:Name="sldZoom" Minimum="1" Maximum="4" Value="{Binding
MouseWheel}" Margin="20,5,20,5" />
                    <CheckBox x:Name="hideHighlightBox" Content="Hide Highlight"
Margin="16,4,0,0"  IsChecked="{Binding HideHighlightSelected}"/>
                    <CheckBox  x:Name="hideWallAndPathUpdatesBox" Content="Hide Wall
&amp; Path Updates" Margin="16,4,0,0"  IsChecked="{Binding
HideWallAndPathUpdatesSelected}"/>
                    <CheckBox  x:Name="hideGridLinesBox" Content="Hide Grid Lines"
Margin="16,4,0,4"  IsChecked="{Binding HideGridLinesSelected}" />
                </StackPanel>
            </ScrollViewer>
        </Border>

        <Border x:Name="CanvasBorder" BorderBrush="Black" BorderThickness="1"
Grid.Column="1" Margin="0,4,4,4">
            <Border.Clip>
                <RectangleGeometry>
                    <RectangleGeometry.Rect>
                        <MultiBinding Converter="{StaticResource MyRectConverter}">
                            <Binding ElementName="CanvasBorder" Path="ActualWidth"/>
                            <Binding ElementName="CanvasBorder" Path="ActualHeight"/>
                        </MultiBinding>
                    </RectangleGeometry.Rect>
                </RectangleGeometry>
            </Border.Clip>

            <ContentPresenter RenderTransformOrigin="0.5,0.5" >
                <ContentPresenter.RenderTransform>
                    <ScaleTransform ScaleX="{Binding Path=Value, ElementName=sldZoom}"
ScaleY="{Binding Path=Value, ElementName=sldZoom}"/>
                </ContentPresenter.RenderTransform>
                <ContentPresenter.Content>
                    <Canvas x:Name="generatorCanvas" Background="White"
MouseWheel="generatorCanvas_MouseWheel">
                        <Path x:Name="GuidePath" Data="{Binding GuidePathData}"
StrokeEndLineCap="Round" StrokeStartLineCap="Round" StrokeThickness="1" Stroke="Red"
StrokeDashArray="1 1" SnapsToDevicePixels="True">
                            <Canvas.Left>
                                <MultiBinding Converter="{StaticResource
MyHalfValueConverter}">
                                    <Binding ElementName="generatorCanvas"
Path="ActualWidth"/>
                                    <Binding ElementName="GuidePath"
Path="ActualWidth"/>
                                </MultiBinding>
                            </Canvas.Left>
                            <Canvas.Top>
                                <MultiBinding Converter="{StaticResource
MyHalfValueConverter}">
                                    <Binding ElementName="generatorCanvas"
Path="ActualHeight"/>
                                    <Binding ElementName="GuidePath"
Path="ActualHeight"/>
                                </MultiBinding>
```

```
                            </Canvas.Top>
                        </Path>
                        <Path x:Name="HighlightPath" Data="{Binding
HighlightPathData}" Fill="Purple" Opacity="0.4" SnapsToDevicePixels="True">
                            <Canvas.Left>
                                <MultiBinding Converter="{StaticResource
MyHalfValueConverter}">
                                    <Binding ElementName="generatorCanvas"
Path="ActualWidth"/>
                                    <Binding ElementName="GuidePath"
Path="ActualWidth"/>
                                </MultiBinding>
                            </Canvas.Left>
                            <Canvas.Top>
                                <MultiBinding Converter="{StaticResource
MyHalfValueConverter}">
                                    <Binding ElementName="generatorCanvas"
Path="ActualHeight"/>
                                    <Binding ElementName="GuidePath"
Path="ActualHeight"/>
                                </MultiBinding>
                            </Canvas.Top>
                        </Path>
                        <Path x:Name="MazePath" Data="{Binding MazePathData}"
StrokeEndLineCap="Round" StrokeStartLineCap="Round" StrokeThickness="{Binding
WallThickness}" Stroke="Black" SnapsToDevicePixels="True">
                            <Canvas.Left>
                                <MultiBinding Converter="{StaticResource
MyHalfValueConverter}">
                                    <Binding ElementName="generatorCanvas"
Path="ActualWidth"/>
                                    <Binding ElementName="GuidePath"
Path="ActualWidth"/>
                                </MultiBinding>
                            </Canvas.Left>
                            <Canvas.Top>
                                <MultiBinding Converter="{StaticResource
MyHalfValueConverter}">
                                    <Binding ElementName="generatorCanvas"
Path="ActualHeight"/>
                                    <Binding ElementName="GuidePath"
Path="ActualHeight"/>
                                </MultiBinding>
                            </Canvas.Top>
                        </Path>
                    </Canvas>
                </ContentPresenter.Content>
            </ContentPresenter>
        </Border>
        <Border BorderBrush="Black" BorderThickness="1" Margin="0,0,4,4" Grid.Row="1"
Grid.Column="1">
            <DataGrid x:Name="dataGrid" ItemsSource="{Binding
AlgorithmMetricCollection}" AutoGenerateColumns="false" Height="121"
CanUserAddRows="False" CanUserDeleteRows="False" IsReadOnly="True"
MouseLeftButtonDown="Row_ClickedGenerator" CanUserSortColumns="False">
                <DataGrid.Columns>
                    <DataGridTextColumn  Header="Algorithm" Binding="{Binding
AlgorithmName}"  />
                    <DataGridTextColumn  Header="Time Taken" Binding="{Binding
TimeSpan, StringFormat={}{0:hh':'mm':'ss'.'ffff}}" />
                    <DataGridTextColumn  Header="Time Taken (ms)" Binding="{Binding
ElapsedMilliseconds}" />
```

```
                    <DataGridTextColumn  Header="Relative Complexity"
Binding="{Binding RelativeComplexity}" />
                </DataGrid.Columns>
            </DataGrid>
        </Border>
    </Grid>
</Window>
```

MazeGeneratorWindow.cs

The MazeGeneratorWindow.xaml page has a C# code behind file shown below. This is used to provide all the event handling and validation functionality for the UI controls and user inputs.

The `btnGenerate_Click()` routine uses **Try Catch Exception Handling** to ensure that an algorithm is always selected before any attempt to run a generator is made.

The `btnSaveGridAsMazFile_Click()` routine demonstrates the usage of the **Save File** dialog from standard controls provide by the Microsoft .NET Framework libraries.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Drawing;
using Mazer.Enums;
using Microsoft.CSharp;
using System.IO;
using Microsoft.Win32;
using MazeLibrary;
using System.Diagnostics;
using Mazer.Model;
using System.Collections.ObjectModel;
using MazeLibrary.Events;
using MazeLibrary.Algorithms;
using Mazer.Helpers;
using Mazer.ViewModels;
using System.Windows.Threading;

namespace Mazer
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MazeGeneratorWindow : Window
    {
        //_gridStorage stores all generated grids so the user can see them whereas
_grid stores only the current / latest one
        private MazeGeneratorWindowViewModel _mazeGeneratorWindowViewModel;
        private MazeLibrary.Grid _grid;
        private List<MazeLibrary.Grid> _gridStorage = new List<MazeLibrary.Grid>();

        public MazeGeneratorWindow()
        {
```

```
            InitializeComponent();

            //intialises a new MazeGeneratorWindowViewModel
            _mazeGeneratorWindowViewModel = new MazeGeneratorWindowViewModel();
            DataContext = _mazeGeneratorWindowViewModel;

            //Setting up buttons that require a grid maze to be inactive before a maze
has been created.
            btnSaveImage.IsEnabled = false;
            btnSaveAsMaz.IsEnabled = false;
            btnSaveResults.IsEnabled = false;

            //Setting default values for textboxes and sliders
            _mazeGeneratorWindowViewModel.Delay = 1;
            _mazeGeneratorWindowViewModel.Width = 34;
            _mazeGeneratorWindowViewModel.Height = 28;
            _mazeGeneratorWindowViewModel.WallSize = 25;
            _mazeGeneratorWindowViewModel.WallThickness = 2;

            //Unimplemented algorithm
            WilsonsAlgorithmBox.IsEnabled = false;

            //Aesthetically Maze looks better when grid lines aren't displayed
            hideGridLinesBox.IsChecked = true;
            _mazeGeneratorWindowViewModel.HideGridLinesSelected = true;
        }

        private void btnBack_Click(object sender, RoutedEventArgs e)
        {
            //Closes window and returns control to the startup screen
            Close();
        }

        private async void btnGenerate_Click(object sender, RoutedEventArgs e)
        {
            try
            {
                //Code validating input, by stopping a crash when non-usable values
are entered
                if (txtbxMazeHeight.Text == "" || txtbxMazeWidth.Text == "" ||
txtbxMazeHeight.Text == "0" || txtbxMazeWidth.Text == "0")
                {
                    MessageBox.Show("Both height and width inputs must be a number
greater than zero!");
                    return;
                }

                //delay gets the value from the view model
                int delay = _mazeGeneratorWindowViewModel.Delay;

                //User should not be able to edit the delay otherwise an unfair time
would be given for the generated maze
                sldDelay.IsEnabled = false;
                txtDelay.IsEnabled = false;
                btnGenerate.IsEnabled = false;

                btnSaveImage.IsEnabled = false;
                btnSaveAsMaz.IsEnabled = false;
                btnSaveResults.IsEnabled = false;

                hideHighlightBox.IsEnabled = false;
                hideGridLinesBox.IsEnabled = false;
```

```
                hideWallAndPathUpdatesBox.IsEnabled = false;

                //resets _gridStorage
                _gridStorage = new List<MazeLibrary.Grid>();

                //Clears view model of data grid data for new run
                _mazeGeneratorWindowViewModel.AlgorithmMetricCollection.Clear();

                //Code ensuring for most reasonable maze dimensions the maze will be
drawn within the bounds of the canvas
                _mazeGeneratorWindowViewModel.WallSize = 25;
                while (_mazeGeneratorWindowViewModel.WallSize *
_mazeGeneratorWindowViewModel.Width >= generatorCanvas.ActualWidth ||
                  _mazeGeneratorWindowViewModel.WallSize *
_mazeGeneratorWindowViewModel.Height >= generatorCanvas.ActualHeight)
                {
                    _mazeGeneratorWindowViewModel.WallSize--;
                    if (_mazeGeneratorWindowViewModel.WallSize <= 1) { break; }
                }

                //Creates new grid of specified dimensions
                _grid = new MazeLibrary.Grid("", _mazeGeneratorWindowViewModel.Width,
_mazeGeneratorWindowViewModel.Height);
                DisplayMaze(_grid);

                //Sets all timing data to null for the new run
                AlgorithmMetric binaryTreeAlgorithmAlgorithmMetric = null;
                AlgorithmMetric drunkardsWalkAlgorithmAlgorithmMetric = null;
                AlgorithmMetric huntAndKillAlgorithmAlgorithmMetric = null;
                AlgorithmMetric kruskalsAlgorithmAlgorithmMetric = null;
                AlgorithmMetric primsAlgorithmAlgorithmMetric = null;
                AlgorithmMetric recursiveBackTrackerAlgorithmAlgorithmMetric = null;
                AlgorithmMetric recursiveDivisionAlgorithmAlgorithmMetric = null;
                AlgorithmMetric sidewinderAlgorithmAlgorithmMetric = null;
                AlgorithmMetric simpleRectangleAlgorithmAlgorithmMetric = null;
                AlgorithmMetric wilsonsAlgorithmAlgorithmMetric = null;

                //Booleans store value in checkboxes so generation can't be confused
if the user clicks a
                //new algorithm box before it has finished making mazes
                bool doBinaryTreeAlgorithm = BinaryTreeBox.IsChecked.Value;
                bool doDrunkardsWalkAlgorithm = DrunkardsWalkBox.IsChecked.Value;
                bool doHuntAndKillAlgorithm = HuntandKillBox.IsChecked.Value;
                bool doKruskalsAlgorithm = KruskalsBox.IsChecked.Value;
                bool doPrimsAlgorithm = PrimsBox.IsChecked.Value;
                bool doRecursiveBackTrackerAlgorithm =
RecursiveBackTrackerBox.IsChecked.Value;
                bool doRecursiveDivisionAlgorithm =
RecursiveDivisionBox.IsChecked.Value;
                bool doSidewinderAlgorithm = SidewinderBox.IsChecked.Value;
                bool doSimpleRectangleAlgorithm = SimpleRectangleBox.IsChecked.Value;
                bool doWilsonsAlgorithm = WilsonsAlgorithmBox.IsChecked.Value;

                //Code ensuring that at least one algorithm has been selected to run,
if not throws the exception shown
                if (!doBinaryTreeAlgorithm && !doDrunkardsWalkAlgorithm &&
!doHuntAndKillAlgorithm && !doKruskalsAlgorithm && !doPrimsAlgorithm
                    && !doRecursiveBackTrackerAlgorithm &&
!doRecursiveDivisionAlgorithm && !doSidewinderAlgorithm && !doSimpleRectangleAlgorithm
&& !doWilsonsAlgorithm)
                {
                    throw new Exception("No algorithm selected");
```

```
                }

                //Simple method of hiding the guide path if that checkbox is checked
                if (_mazeGeneratorWindowViewModel.HideGridLinesSelected)
                {
                    GuidePath.Visibility = System.Windows.Visibility.Hidden;
                }
                else
                {
                    GuidePath.Visibility = System.Windows.Visibility.Visible;
                }

                //Allows the displaying of a maze to be on one thread while the
algorithms still create the maze on another.
                //(Algorithms, therefore, run in the background on one thread, firing
off events to update the UI on the main thread with new information)
                await Task.Run(() =>
                {
                    Stopwatch stopwatch = new Stopwatch();

                    //The guide for the maze is always the same, as every maze is made
with the same dimensions
                    DisplayGuide(_grid);

                    //All algorithm's if-checked statements follow the same procedure.
I will therefore only comment on the binary Tree algorithm

                    #region Binary Tree

                    if (doBinaryTreeAlgorithm)
                    {
                        //Clear all cells that are currently highlighted
                        //ClearHighlight();

                        //_grid is set to a new grid named this algorithm
                        _grid = new MazeLibrary.Grid("Grid: BinaryTree",
_mazeGeneratorWindowViewModel.Width, _mazeGeneratorWindowViewModel.Height);

                        //Updates a label with the currently running algorithm
                        _mazeGeneratorWindowViewModel.AlgorithimLabel =
string.Format("Generating ({0})", _grid.Name);

                        //Stopwatch is reset and started and BinaryTreeAlgorithm is
created with the required delay. Whenever a wall created or passage created event
                        //is called in the code it creates a new handle for it on
another thread. There it updates the UI canvas. After the maze is created on
                        //_grid the stopwatch is stopped
                        stopwatch.Reset();
                        stopwatch.Start();

                        //Initaliates a new BinaryTreeAlgorithm
                        BinaryTreeAlgorithm binaryTreeAlgorithm = new
BinaryTreeAlgorithm(delay);
                        if
(!_mazeGeneratorWindowViewModel.HideWallAndPathUpdatesSelected)
                        {
                            //Only if the hide wall and path updates checkbox is not
checked does it handle the event calls
                            binaryTreeAlgorithm.RaiseCellWallCreatedEvent +=
HandleCellWallCreatedEvent;
                            binaryTreeAlgorithm.RaiseCellPassageCreatedEvent +=
HandleCellPassageCreatedEvent;
```

```
                        }
                        //Only if the hide highlight updates checkbox is not checked
does it handle the event calls
                        if (!_mazeGeneratorWindowViewModel.HideHighlightSelected)
binaryTreeAlgorithm.RaiseHighlightPathAddedNewCellEvent += HandleHighlightCellEvent;

                        //Generates the maze on _grid
                        binaryTreeAlgorithm.Generate(_grid);
                        stopwatch.Stop();

                        //Add the grid to the storage. As all algorithms are ordered
the same as they appear in the data grid,
                        //the row selected in the data grid directly corresponds to
the index () in the _gridStorage list (Where both indexes start at zero)
                        _gridStorage.Add(_grid);

                        //Creates and stores all data recorded for use in the creation
of the histogram
                        binaryTreeAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "BinaryTree",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                            RelativeComplexity = GetComplexity(_grid),
                        };
                    }

                    #endregion

                    #region Drunkards Walk

                    if (doDrunkardsWalkAlgorithm)
                    {


                        _grid = new MazeLibrary.Grid("Grid: DrunkardsWalk",
_mazeGeneratorWindowViewModel.Width, _mazeGeneratorWindowViewModel.Height);
                        _mazeGeneratorWindowViewModel.AlgorithimLabel =
string.Format("Generating ({0})", _grid.Name);

                        stopwatch.Reset();
                        stopwatch.Start();
                        DrunkardsWalkAlgorithm drunkardsWalkAlgorithm = new
DrunkardsWalkAlgorithm(delay);
                        if
(!_mazeGeneratorWindowViewModel.HideWallAndPathUpdatesSelected)
                        {
                            drunkardsWalkAlgorithm.RaiseCellWallCreatedEvent +=
HandleCellWallCreatedEvent;
                            drunkardsWalkAlgorithm.RaiseCellPassageCreatedEvent +=
HandleCellPassageCreatedEvent;
                        }
                        if (!_mazeGeneratorWindowViewModel.HideHighlightSelected)
drunkardsWalkAlgorithm.RaiseHighlightPathAddedNewCellEvent +=
HandleHighlightCellEvent;
                        drunkardsWalkAlgorithm.Generate(_grid);
                        stopwatch.Stop();

                        _gridStorage.Add(_grid);

                        drunkardsWalkAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
```

```
                                AlgorithmName = "DrunkardsWalk",
                                TimeSpan = stopwatch.Elapsed,
                                ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                                RelativeComplexity = GetComplexity(_grid),
                            };
                        }

                        #endregion

                        #region Hunt and Kill

                        if (doHuntAndKillAlgorithm)
                        {


                            _grid = new MazeLibrary.Grid("Grid: HuntandKill",
_mazeGeneratorWindowViewModel.Width, _mazeGeneratorWindowViewModel.Height);
                            _mazeGeneratorWindowViewModel.AlgorithimLabel =
string.Format("Generating ({0})", _grid.Name);

                            stopwatch.Reset();
                            stopwatch.Start();
                            HuntAndKillAlgorithm huntAndKillAlgorithm = new
HuntAndKillAlgorithm(delay);
                            if
(!_mazeGeneratorWindowViewModel.HideWallAndPathUpdatesSelected)
                            {
                                huntAndKillAlgorithm.RaiseCellWallCreatedEvent +=
HandleCellWallCreatedEvent;
                                huntAndKillAlgorithm.RaiseCellPassageCreatedEvent +=
HandleCellPassageCreatedEvent;
                            }
                            if (!_mazeGeneratorWindowViewModel.HideHighlightSelected)
huntAndKillAlgorithm.RaiseHighlightPathAddedNewCellEvent += HandleHighlightCellEvent;
                            huntAndKillAlgorithm.Generate(_grid);
                            stopwatch.Stop();

                            _gridStorage.Add(_grid);

                            huntAndKillAlgorithmAlgorithmMetric = new AlgorithmMetric
                            {
                                AlgorithmName = "HuntandKill",
                                TimeSpan = stopwatch.Elapsed,
                                ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                                RelativeComplexity = GetComplexity(_grid),
                            };
                        }

                        #endregion

                        #region Kruskals

                        if (doKruskalsAlgorithm)
                        {


                            _grid = new MazeLibrary.Grid("Grid: Kruskals",
_mazeGeneratorWindowViewModel.Width, _mazeGeneratorWindowViewModel.Height);
                            _mazeGeneratorWindowViewModel.AlgorithimLabel =
string.Format("Generating ({0})", _grid.Name);

                            stopwatch.Reset();
```

```
                        stopwatch.Start();
                        KruskalsAlgorithm kruskalsAlgorithm = new
KruskalsAlgorithm(delay);
                        if
(!_mazeGeneratorWindowViewModel.HideWallAndPathUpdatesSelected)
                        {
                            kruskalsAlgorithm.RaiseCellWallCreatedEvent +=
HandleCellWallCreatedEvent;
                            kruskalsAlgorithm.RaiseCellPassageCreatedEvent +=
HandleCellPassageCreatedEvent;
                        }
                        if (!_mazeGeneratorWindowViewModel.HideHighlightSelected)
kruskalsAlgorithm.RaiseHighlightPathAddedNewCellEvent += HandleHighlightCellEvent;
                        kruskalsAlgorithm.Generate(_grid);
                        stopwatch.Stop();

                        _gridStorage.Add(_grid);

                        kruskalsAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "Kruskals",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                            RelativeComplexity = GetComplexity(_grid),
                        };
                    }

                    #endregion

                    #region Prims

                    if (doPrimsAlgorithm)
                    {


                        _grid = new MazeLibrary.Grid("Grid: Prims",
_mazeGeneratorWindowViewModel.Width, _mazeGeneratorWindowViewModel.Height);
                        _mazeGeneratorWindowViewModel.AlgorithimLabel =
string.Format("Generating ({0})", _grid.Name);

                        stopwatch.Reset();
                        stopwatch.Start();
                        PrimsAlgorithm primsAlgorithm = new PrimsAlgorithm(delay);
                        if
(!_mazeGeneratorWindowViewModel.HideWallAndPathUpdatesSelected)
                        {
                            primsAlgorithm.RaiseCellWallCreatedEvent +=
HandleCellWallCreatedEvent;
                            primsAlgorithm.RaiseCellPassageCreatedEvent +=
HandleCellPassageCreatedEvent;
                        }
                        if (!_mazeGeneratorWindowViewModel.HideHighlightSelected)
primsAlgorithm.RaiseHighlightPathAddedNewCellEvent += HandleHighlightCellEvent;
                        primsAlgorithm.Generate(_grid);
                        stopwatch.Stop();

                        _gridStorage.Add(_grid);

                        primsAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "Prims",
                            TimeSpan = stopwatch.Elapsed,
```

```
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                            RelativeComplexity = GetComplexity(_grid),
                        };
                    }

                    #endregion

                    #region Recursive Backtracker

                    if (doRecursiveBackTrackerAlgorithm)
                    {


                        _grid = new MazeLibrary.Grid("Grid: RecursiveBackTracker",
_mazeGeneratorWindowViewModel.Width, _mazeGeneratorWindowViewModel.Height);
                        _mazeGeneratorWindowViewModel.AlgorithimLabel =
string.Format("Generating ({0})", _grid.Name);

                        stopwatch.Reset();
                        stopwatch.Start();
                        RecursiveBackTrackerAlgorithm recursiveBackTrackerAlgorithm =
new RecursiveBackTrackerAlgorithm(delay);
                        if
(!_mazeGeneratorWindowViewModel.HideWallAndPathUpdatesSelected)
                        {
                            recursiveBackTrackerAlgorithm.RaiseCellWallCreatedEvent +=
HandleCellWallCreatedEvent;
                            recursiveBackTrackerAlgorithm.RaiseCellPassageCreatedEvent
+= HandleCellPassageCreatedEvent;
                        }
                        if (!_mazeGeneratorWindowViewModel.HideHighlightSelected)
recursiveBackTrackerAlgorithm.RaiseHighlightPathAddedNewCellEvent +=
HandleHighlightCellEvent;
                        recursiveBackTrackerAlgorithm.Generate(_grid);
                        stopwatch.Stop();

                        _gridStorage.Add(_grid);

                        recursiveBackTrackerAlgorithmAlgorithmMetric = new
AlgorithmMetric
                        {
                            AlgorithmName = "RecursiveBackTracker",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                            RelativeComplexity = GetComplexity(_grid),
                        };
                    }

                    #endregion

                    #region Recursive Division

                    if (doRecursiveDivisionAlgorithm)
                    {
                        ClearHighlight();

                        _grid = new MazeLibrary.Grid("Grid: RecursiveDivision",
_mazeGeneratorWindowViewModel.Width, _mazeGeneratorWindowViewModel.Height);
                        _mazeGeneratorWindowViewModel.AlgorithimLabel =
string.Format("Generating ({0})", _grid.Name);

                        stopwatch.Reset();
```

```
                        stopwatch.Start();
                        RecursiveDivisionAlgorithm recursiveDivisionAlgorithm = new
RecursiveDivisionAlgorithm(delay);
                        if
(!_mazeGeneratorWindowViewModel.HideWallAndPathUpdatesSelected)
                        {
                            recursiveDivisionAlgorithm.RaiseCellWallCreatedEvent +=
HandleCellWallCreatedEvent;
                            recursiveDivisionAlgorithm.RaiseCellPassageCreatedEvent +=
HandleCellPassageCreatedEvent;
                        }
                        if (!_mazeGeneratorWindowViewModel.HideHighlightSelected)
recursiveDivisionAlgorithm.RaiseHighlightPathAddedNewCellEvent +=
HandleHighlightCellEvent;
                        recursiveDivisionAlgorithm.Generate(_grid);
                        stopwatch.Stop();

                        _gridStorage.Add(_grid);

                        recursiveDivisionAlgorithmAlgorithmMetric = new
AlgorithmMetric
                        {
                            AlgorithmName = "RecursiveDivision",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                            RelativeComplexity = GetComplexity(_grid),
                        };
                    }

                    #endregion

                    #region Sidewinder

                    if (doSidewinderAlgorithm)
                    {


                        _grid = new MazeLibrary.Grid("Grid: Sidewinder",
_mazeGeneratorWindowViewModel.Width, _mazeGeneratorWindowViewModel.Height);
                        _mazeGeneratorWindowViewModel.AlgorithimLabel =
string.Format("Generating ({0})", _grid.Name);

                        stopwatch.Reset();
                        stopwatch.Start();
                        SidewinderAlgorithm sidewinderAlgorithm = new
SidewinderAlgorithm(delay);
                        if
(!_mazeGeneratorWindowViewModel.HideWallAndPathUpdatesSelected)
                        {
                            sidewinderAlgorithm.RaiseCellWallCreatedEvent +=
HandleCellWallCreatedEvent;
                            sidewinderAlgorithm.RaiseCellPassageCreatedEvent +=
HandleCellPassageCreatedEvent;
                        }
                        if (!_mazeGeneratorWindowViewModel.HideHighlightSelected)
sidewinderAlgorithm.RaiseHighlightPathAddedNewCellEvent += HandleHighlightCellEvent;
                        sidewinderAlgorithm.Generate(_grid);
                        stopwatch.Stop();

                        _gridStorage.Add(_grid);

                        sidewinderAlgorithmAlgorithmMetric = new AlgorithmMetric
```

```
                        {
                            AlgorithmName = "Sidewinder",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                            RelativeComplexity = GetComplexity(_grid),
                        };
                    }

                    #endregion

                    #region Simple Rectangle

                    if (doSimpleRectangleAlgorithm)
                    {


                        _grid = new MazeLibrary.Grid("Grid: SimpleRectangle",
_mazeGeneratorWindowViewModel.Width, _mazeGeneratorWindowViewModel.Height);
                        _mazeGeneratorWindowViewModel.AlgorithimLabel =
string.Format("Generating ({0})", _grid.Name);

                        stopwatch.Reset();
                        stopwatch.Start();
                        SimpleRectangleAlgorithm simpleRectangleAlgorithm = new
SimpleRectangleAlgorithm(delay);

                        if
(!_mazeGeneratorWindowViewModel.HideWallAndPathUpdatesSelected)
                        {
                            simpleRectangleAlgorithm.RaiseCellWallCreatedEvent +=
HandleCellWallCreatedEvent;
                            simpleRectangleAlgorithm.RaiseCellPassageCreatedEvent +=
HandleCellPassageCreatedEvent;
                        }

                        if (!_mazeGeneratorWindowViewModel.HideHighlightSelected)
simpleRectangleAlgorithm.RaiseHighlightPathAddedNewCellEvent +=
HandleHighlightCellEvent;
                        simpleRectangleAlgorithm.Generate(_grid);
                        stopwatch.Stop();

                        _gridStorage.Add(_grid);

                        simpleRectangleAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "SimpleRectangle",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                            RelativeComplexity = GetComplexity(_grid),
                        };
                    }

                    #endregion

                    #region Wilsons

                    if (doWilsonsAlgorithm)
                    {


                        _grid = new MazeLibrary.Grid("Grid: WilsonsAlgorithm",
_mazeGeneratorWindowViewModel.Width, _mazeGeneratorWindowViewModel.Height);
```

```
                        _mazeGeneratorWindowViewModel.AlgorithimLabel =
string.Format("Generating ({0})", _grid.Name);

                        stopwatch.Reset();
                        stopwatch.Start();
                        WilsonsAlgorithm wilsonsAlgorithm = new
WilsonsAlgorithm(delay);

                        if
(!_mazeGeneratorWindowViewModel.HideWallAndPathUpdatesSelected)
                        {
                            wilsonsAlgorithm.RaiseCellWallCreatedEvent +=
HandleCellWallCreatedEvent;
                            wilsonsAlgorithm.RaiseCellPassageCreatedEvent +=
HandleCellPassageCreatedEvent;
                        }

                        if (!_mazeGeneratorWindowViewModel.HideHighlightSelected)
wilsonsAlgorithm.RaiseHighlightPathAddedNewCellEvent += HandleHighlightCellEvent;
                        wilsonsAlgorithm.Generate(_grid);
                        stopwatch.Stop();

                        _gridStorage.Add(_grid);

                        wilsonsAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "WilsonsAlgorithm",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                            RelativeComplexity = GetComplexity(_grid),
                        };
                    }

                    #endregion

                    //As no maze is being generated now the label is set to empty
                    _mazeGeneratorWindowViewModel.AlgorithimLabel = "";

                    //Clear all cells that are currently highlighted
                    ClearHighlight();

                    //To ensure a maze is always displayed I force another displaying
of the latest grid here (As on the odd occasion if updates are slowing the program
down, but a new task is started then the old updates may just be forgotten about,
possibly leaving a broken maze).
                    DisplayMaze(_grid);

                });

                //Adds only the algorithm times from the algorithms that created a
maze to an AlgorithmMetricCollection as a storage for
                //all the algorithms used
                if (binaryTreeAlgorithmAlgorithmMetric != null)
                {

_mazeGeneratorWindowViewModel.AlgorithmMetricCollection.Add(binaryTreeAlgorithmAlgorit
hmMetric);
                }
                if (drunkardsWalkAlgorithmAlgorithmMetric != null)
                {
```

```
_mazeGeneratorWindowViewModel.AlgorithmMetricCollection.Add(drunkardsWalkAlgorithmAlgo
rithmMetric);
                }
                if (huntAndKillAlgorithmAlgorithmMetric != null)
                {

_mazeGeneratorWindowViewModel.AlgorithmMetricCollection.Add(huntAndKillAlgorithmAlgori
thmMetric);
                }
                if (kruskalsAlgorithmAlgorithmMetric != null)
                {

_mazeGeneratorWindowViewModel.AlgorithmMetricCollection.Add(kruskalsAlgorithmAlgorithm
Metric);
                }
                if (primsAlgorithmAlgorithmMetric != null)
                {

_mazeGeneratorWindowViewModel.AlgorithmMetricCollection.Add(primsAlgorithmAlgorithmMet
ric);
                }
                if (recursiveBackTrackerAlgorithmAlgorithmMetric != null)
                {

_mazeGeneratorWindowViewModel.AlgorithmMetricCollection.Add(recursiveBackTrackerAlgori
thmAlgorithmMetric);
                }
                if (recursiveDivisionAlgorithmAlgorithmMetric != null)
                {

_mazeGeneratorWindowViewModel.AlgorithmMetricCollection.Add(recursiveDivisionAlgorithm
AlgorithmMetric);
                }
                if (sidewinderAlgorithmAlgorithmMetric != null)
                {

_mazeGeneratorWindowViewModel.AlgorithmMetricCollection.Add(sidewinderAlgorithmAlgorit
hmMetric);
                }
                if (simpleRectangleAlgorithmAlgorithmMetric != null)
                {

_mazeGeneratorWindowViewModel.AlgorithmMetricCollection.Add(simpleRectangleAlgorithmAl
gorithmMetric);
                }
                if (wilsonsAlgorithmAlgorithmMetric != null)
                {

_mazeGeneratorWindowViewModel.AlgorithmMetricCollection.Add(wilsonsAlgorithmAlgorithmM
etric);
                }

                //As everything has finished and there are now accessable mazes all
sliders and buttons are now enabled
                sldDelay.IsEnabled = true;
                txtDelay.IsEnabled = true;
                btnGenerate.IsEnabled = true;

                btnSaveImage.IsEnabled = true;
                btnSaveAsMaz.IsEnabled = true;
                btnSaveResults.IsEnabled = true;
```

```
                hideHighlightBox.IsEnabled = true;
                hideGridLinesBox.IsEnabled = true;
                hideWallAndPathUpdatesBox.IsEnabled = true;
                ClearHighlight();
            }
            catch
            {
                //Throwing a error message if there is no selected algorithm to run.
Ensuring all controls are reset as well
                sldDelay.IsEnabled = true;
                txtDelay.IsEnabled = true;
                btnGenerate.IsEnabled = true;

                btnSaveImage.IsEnabled = false;
                btnSaveAsMaz.IsEnabled = false;
                btnSaveResults.IsEnabled = false;

                hideHighlightBox.IsEnabled = true;
                hideGridLinesBox.IsEnabled = true;
                hideWallAndPathUpdatesBox.IsEnabled = true;
                MessageBox.Show("Error: An algorithm must be selected to generate a
maze, please select one.", "Error Loading File", MessageBoxButton.OK,
MessageBoxImage.Error);
            }
        }

        private void ClearHighlight()
        {
            //Resets all highlighting
            _mazeGeneratorWindowViewModel.HighlightPathData = Geometry.Parse("");
        }

        void HandleCellPassageCreatedEvent(object sender, CellPassageCreatedEventArgs
e)
        {
            //Displays the current _grid in whatever state of completeness it is in
            DisplayMaze(e.Grid);
        }

        void HandleCellWallCreatedEvent(object sender, CellWallCreatedEventArgs e)
        {
            //Displays the current _grid in whatever state of completeness it is in
            DisplayMaze(e.Grid);
        }

        void HandleHighlightCellEvent(object sender,
HighlightPathAddedNewCellEventArgs e)
        {
            //Displays the highlight list of cells
            DisplayHighlight(e.Grid, e.HighlightCells);
        }

        private void DisplayGuide(MazeLibrary.Grid _grid)
        {
            //Runs this code at the same time as running the rest of the program
            Dispatcher.Invoke(DispatcherPriority.Normal, (Action)delegate
            {
                int width = _grid.Width;
                int height = _grid.Height;
                int wallSize = _mazeGeneratorWindowViewModel.WallSize;
```

```
                //Creates the guide path with the pre-created data
                string guidePath = PathHelper.GetGuidePath(width, height, wallSize);
                _mazeGeneratorWindowViewModel.GuidePathData =
Geometry.Parse(guidePath);
            });
        }

        private void DisplayMaze(MazeLibrary.Grid grid)
        {
            //Runs this code at the same time as running the rest of the program
            Dispatcher.Invoke(DispatcherPriority.Normal, (Action)delegate
            {
                int width = grid.Width;
                int height = grid.Height;
                int wallSize = _mazeGeneratorWindowViewModel.WallSize;

                //Creates the maze path with the pre-created data
                string mazePath = PathHelper.GetMazeGeneratorPath(grid, wallSize);
                _mazeGeneratorWindowViewModel.MazePathData = Geometry.Parse(mazePath);
            });
        }

        private void DisplayHighlight(MazeLibrary.Grid grid, List<Cell>
highlightCells)
        {
            //Runs this code at the same time as running the rest of the program
            Dispatcher.Invoke(DispatcherPriority.Normal, (Action)delegate
            {
                int width = _grid.Width;
                int height = _grid.Height;
                int wallSize = _mazeGeneratorWindowViewModel.WallSize;
                int wallThickness = _mazeGeneratorWindowViewModel.WallThickness;

                //Creates a highlight at each of the passed cells
                string highlightCellPath = PathHelper.GetHighlightCellPath(wallSize,
wallThickness, highlightCells);
                _mazeGeneratorWindowViewModel.HighlightPathData =
Geometry.Parse(highlightCellPath);
            });
        }

        private double GetComplexity(MazeLibrary.Grid _grid)
        {
            //Function returns complexity value for the passed in grid
            double onePassageWay = 0;
            double twoPassageWays = 0;
            double threePassageWays = 0;
            double fourPassageWays = 0;

            //The number of cells with only 1 passageway are added, as are those with
two, etc.
            for (int y = 0; y < _grid.Height; y++)
            {
                for (int x = 0; x < _grid.Width; x++)
                {
                    int passageCounter = 0;
                    Cell currentCell = _grid.GetCell(x, y);
                    for (int i = 0; i < 4; i++)
                    {
                        CellEdge cellEdge = currentCell.GetEdge((Direction)i);
                        if (cellEdge is CellPassage)
                        {
```

```
                            passageCounter++;
                        }
                    }
                    switch (passageCounter)
                    {
                        case 1:
                            onePassageWay++;
                            break;
                        case 2:
                            twoPassageWays++;
                            break;
                        case 3:
                            threePassageWays++;
                            break;
                        case 4:
                            fourPassageWays++;
                            break;
                    }
                }
            }

            //Complexity calculated as was in the Analysis
            double value = ((twoPassageWays + threePassageWays + fourPassageWays) /
onePassageWay);
            return Math.Round(value, 2, MidpointRounding.AwayFromZero);
        }

        private void txtbxMazeWidth_PreviewKeyDown(object sender, KeyEventArgs e)
        {
            //Handles key presses which are allowed in the textboxes. This is the form
of validating input I chose to implement
            e.Handled = true;

            switch (e.Key)
            {
                case Key.D0:
                case Key.D1:
                case Key.D2:
                case Key.D3:
                case Key.D4:
                case Key.D5:
                case Key.D6:
                case Key.D7:
                case Key.D8:
                case Key.D9:
                case Key.Back:
                case Key.Tab:
                case Key.Left:
                case Key.Right:
                case Key.Delete:
                    e.Handled = false;
                    break;
            }
        }

        private void txtbxMazeHeight_PreviewKeyDown(object sender, KeyEventArgs e)
        {
            //Handles key presses which are allowed in the textboxes. This is the form
of validating input I chose to implement
            e.Handled = true;

            switch (e.Key)
```

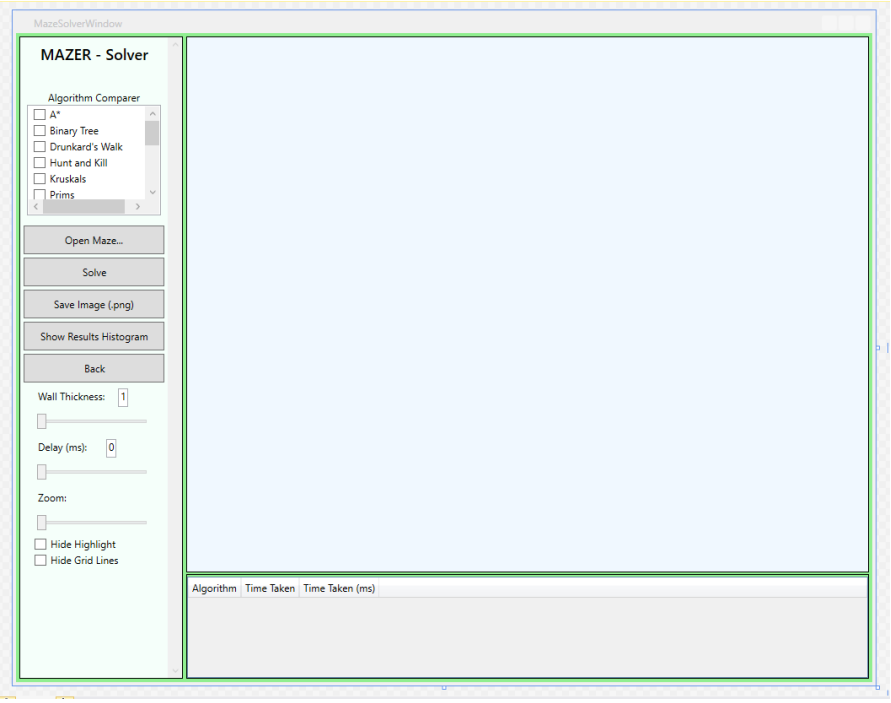
```
            {
                case Key.D0:
                case Key.D1:
                case Key.D2:
                case Key.D3:
                case Key.D4:
                case Key.D5:
                case Key.D6:
                case Key.D7:
                case Key.D8:
                case Key.D9:
                case Key.Back:
                case Key.Tab:
                case Key.Left:
                case Key.Right:
                case Key.Delete:
                    e.Handled = false;
                    break;
            }
        }

        private void btnSaveImage_Click(object sender, RoutedEventArgs e)
        {
            //Saves the currently displayed maze as an image in the SaveImageHelper
class
            SaveImageHelper.SavePngImage(generatorCanvas, "Generated");
        }

        private void btnSaveGridAsMazFile_Click(object sender, RoutedEventArgs e)
        {
            SaveFileDialog saveFileDialog = new SaveFileDialog();
            string applicationFolder = "Mazer Save Files";
            //Gets the path for the myDocuments path for whatever computer this
program is on.
            string myDocuments =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
            //combines these paths
            string applicationPath = System.IO.Path.Combine(myDocuments,
applicationFolder, "Generated Mazes (.maz)");
            //Sees if this path already exists
            if (!Directory.Exists(applicationPath))
            {
                Directory.CreateDirectory(applicationPath);
            }
            //All this provides the usual "Save As" function of any microsoft
application.
            saveFileDialog.Filter = "maz (*.maz)|*.maz|All Files (*.*)|*.*";
            saveFileDialog.FilterIndex = 0;
            saveFileDialog.Title = "Save Mazer text File";
            saveFileDialog.InitialDirectory = applicationPath;
            saveFileDialog.RestoreDirectory = true;
            saveFileDialog.CheckFileExists = false;
            saveFileDialog.CheckPathExists = true;
            saveFileDialog.DefaultExt = "maz";

            //dialogResult can be true, false or null (just in case the dialogue
returns null if it is closed, which would otherwise crash the program at this line)
            bool? dialogResult = saveFileDialog.ShowDialog();

            //If what the user enters is Save (OK) will save the file.
            if (dialogResult == true)
            {
```

```
                //Using is here to close the stream writer in case if forget to close
it as a good example of defensive coding
                using (StreamWriter writer = new StreamWriter(saveFileDialog.FileName,
true))
                {
                    //Writes the first line of the text file with the grid's name,
width and height. It seporates
                    //these with vertical bars so the reader in the solver window can
split on each part
                    writer.WriteLine(String.Format("{0}|{1}|{2}", _grid.Name,
_grid.Width, _grid.Height));

                    //Steps through every cell...
                    for (int y = 0; y < _grid.Height; y++)
                    {
                        for (int x = 0; x < _grid.Width; x++)
                        {
                            Cell currentCell = _grid.GetCell(x, y);


                            //CellEdge edgeNorth =
currentCell.GetEdge(Direction.North);
                            //CellEdge edgeEast = currentCell.GetEdge(Direction.East);
                            //CellEdge edgeSouth =
currentCell.GetEdge(Direction.South);
                            //CellEdge edgeWest = currentCell.GetEdge(Direction.West);

                            //...getting each edge and...
                            CellEdge[] currentCellEdges = new CellEdge[] {
currentCell.GetEdge(Direction.North),
                        currentCell.GetEdge(Direction.East),
                        currentCell.GetEdge(Direction.South),
                        currentCell.GetEdge(Direction.West)};

                            //...determines what edge type they are, writing the
correct string in to represent that spot
                            StringBuilder line = new StringBuilder();
                            foreach (CellEdge ce in currentCellEdges)
                            {
                                if (ce is CellPassage)
                                {
                                    writer.Write("P");
                                }
                                else if (ce is CellWall)
                                {
                                    writer.Write("W");
                                }
                                else
                                {
                                    writer.Write("-");
                                }
                            }

                            //Every four strings added starts a new line
                            writer.WriteLine();
                        }
                    }
                }
            }
        }

        private void generatorCanvas_MouseWheel(object sender, MouseWheelEventArgs e)
```

```
        {
            //Code for zooming into the canvas with the mousewheel
            _mazeGeneratorWindowViewModel.MouseWheel += (e.Delta / 100);
        }

        private void btnSaveResults_Click(object sender, RoutedEventArgs e)
        {
            //Adds each of the used algorithm's names and times to the algorithmMetric
            Dictionary<string, long> data = new Dictionary<string, long>();
            foreach (AlgorithmMetric algorithmMetric in
_mazeGeneratorWindowViewModel.AlgorithmMetricCollection)
            {
                data.Add(algorithmMetric.AlgorithmName,
algorithmMetric.ElapsedMilliseconds);
            }

            //Displays the results window with this data and the title of which window
it came from
            ResultsWindow resultsWindow = new ResultsWindow();
            resultsWindow.Owner = this;
            resultsWindow.ResultsWindowViewModel.Data = data;
            resultsWindow.ResultsWindowViewModel.SourceName = "Generator Results";
            resultsWindow.ShowDialog();
        }

        private void Row_ClickedGenerator(object sender, MouseButtonEventArgs e)
        {
            //Code for getting the correct index in _gridStorage corrasponding to the
index of the row clicked
            DataGridRow row =
(DataGridRow)ItemsControl.ContainerFromElement((DataGrid)sender,
(DependencyObject)e.OriginalSource);

            //If nothing is clicked do nothing
            if (row == null) return;

            //_grid gets the required maze grid
            _grid = _gridStorage[row.GetIndex()];

            //Displays the grid's name
            _mazeGeneratorWindowViewModel.AlgorithimLabel = string.Format("Generating
({0})", _grid.Name);

            //Displays the new grid
            DisplayMaze(_grid);

        }
    }
}
```

MazeSolverWindow

The solver window is displayed below. The UI elements and layout of this window closely follow the original design. The final implementation of the window does include a few extra features. These extra features were added to the UI making the application more graphically interesting to use. However, I am pleased with how closely I have been able to follow the original main design for this screen.



MazeSolverWindow.xaml

```
<Window x:Class="Mazer.MazeSolverWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:converters="clr-namespace:Mazer.Converters"
        Title="MazeSolverWindow" Height="800" Width="1024">
    <Window.Resources>
        <ResourceDictionary>
            <converters:HalfValueConverter x:Key="MyHalfValueConverter"/>
            <converters:RectConverter x:Key="MyRectConverter"/>
        </ResourceDictionary>
    </Window.Resources>

    <Grid Background="LightGreen">
        <Grid.Resources>
            <ResourceDictionary>
                <Style TargetType="Button">
                    <Setter Property="Margin" Value="4,4,4,0"/>
                    <Setter Property="Padding" Value="8"/>
                </Style>
            </ResourceDictionary>
        </Grid.Resources>

        <Grid.RowDefinitions>
            <RowDefinition Height="5*"/>
            <RowDefinition Height="1*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="200"/>
            <ColumnDefinition/>
```

```
        </Grid.ColumnDefinitions>

        <Border BorderBrush="Black" BorderThickness="1" Grid.RowSpan="2"
Margin="4,4,2,4">
            <ScrollViewer>
                <StackPanel Background="MintCream" >
                    <TextBlock Text="MAZER - Solver" FontSize="18"
FontWeight="SemiBold"  TextWrapping="WrapWithOverflow" HorizontalAlignment="Center"
Margin="0,8,0,4"/>
                    <TextBlock Text="{Binding AlgorithimLabel}"
HorizontalAlignment="Left" Margin="16,4,0,0" VerticalAlignment="Top" Width="164"
FontWeight="Bold" Height="14" Foreground="#FF0331FF" FontSize="10"/>
                    <StackPanel Margin="8">
                        <TextBlock Text="Algorithm Comparer"
HorizontalAlignment="Center" Margin="1"/>
                        <ListBox Height="131">
                            <CheckBox x:Name="AStarBox" Content="A*" />
                            <CheckBox x:Name="BinaryTreeBox" Content="Binary Tree" />
                            <CheckBox x:Name="DrunkardsWalkBox" Content="Drunkard's
Walk" />
                            <CheckBox x:Name="HuntandKillBox" Content="Hunt and Kill"
/>
                            <CheckBox x:Name="KruskalsBox" Content="Kruskals" />
                            <CheckBox x:Name="PrimsBox" Content="Prims"/>
                            <CheckBox x:Name="RecursiveBackTrackerBox"
Content="Recursive Back Tracker" />
                            <CheckBox x:Name="RecursiveDivisionBox" Content="Recursive
Division"/>
                            <CheckBox x:Name="SidewinderBox" Content="Sidewinder"/>
                            <CheckBox x:Name="SimpleRectangleBox" Content="Simple
Rectangle" />
                            <CheckBox x:Name="LeftWallFollowerBox"
Content="LeftWallFollowerBox" />
                            <CheckBox x:Name="RightWallFollowerBox"
Content="RightWallFollower" />
                            <CheckBox x:Name="WilsonsAlgorithmBox" Content="Wilsons"
/>
                        </ListBox>
                    </StackPanel>
                    <Button x:Name="btnOpenMaze" Content="Open Maze..."
Click="btnOpenMaze_Click"/>
                    <Button x:Name="btnSolve" Content="Solve" Click="btnSolve_Click"/>
                    <!--<Button x:Name="btnSaveAsMaz" Content="Save As (.maz)" />-->
                    <Button x:Name="btnSaveImage" Content="Save Image (.png)"
Click="btnSaveImage_Click"/>
                    <Button x:Name="btnSaveResults" Content="Show Results Histogram"
Click="btnSaveResults_Click"/>
                    <Button x:Name="btnBack" Content="Back" Click="btnBack_Click"/>
                    <StackPanel>
                        <Grid>
                            <Label Content="Wall Thickness:" Margin="16,3,0,3"
Height="26"/>
                            <TextBox x:Name="txtWallThickness" Text="{Binding Value,
ElementName=sldWallThickness, Mode=TwoWay}" HorizontalAlignment="Center"
Margin="111,7,41,3" />
                        </Grid>
                        <Slider x:Name="sldWallThickness" VerticalAlignment="Center"
Margin="20,5,20,5" Minimum="1" Maximum="20" SmallChange="1" LargeChange="5"
TickFrequency="1" IsSnapToTickEnabled="True" Value="{Binding WallThickness}"/>
                    </StackPanel>
                    <StackPanel>
                        <Grid>
```

```
                            <Label Content="Delay (ms):" Margin="16,3,80,3"
Height="26"/>
                            <TextBox x:Name="txtDelay" Text="{Binding Value,
ElementName=sldDelay, Mode=TwoWay}" HorizontalAlignment="Center" Margin="100,7,60,3"/>
                        </Grid>
                        <Slider x:Name="sldDelay" VerticalAlignment="Center"
Margin="20,5,20,5" Minimum="0" Maximum="2000" SmallChange="10" LargeChange="50"
TickFrequency="1" IsSnapToTickEnabled="True" Value="{Binding Delay}"/>
                    </StackPanel>
                    <Label Content="Zoom:" Margin="16,3,80,3"/>
                    <Slider x:Name="sldZoom" Minimum="1" Maximum="4" Value="{Binding
MouseWheel}" Margin="20,5,20,5"/>
                    <CheckBox x:Name="hideHighlightBox" Content="Hide Highlight"
Margin="16,4,0,0"  IsChecked="{Binding HideHighlightSelected}"/>
                    <CheckBox x:Name="hideGridLinesBox" Content="Hide Grid Lines"
Margin="16,4,0,0"  IsChecked="{Binding HideGridLinesSelected}"/>
                </StackPanel>
            </ScrollViewer>
        </Border>

        <Border x:Name="CanvasBorder" BorderBrush="Black" BorderThickness="1"
Grid.Column="1" Margin="2,4,4,2">
            <Border.Clip>
                <RectangleGeometry>
                    <RectangleGeometry.Rect>
                        <MultiBinding Converter="{StaticResource MyRectConverter}">
                            <Binding ElementName="CanvasBorder" Path="ActualWidth"/>
                            <Binding ElementName="CanvasBorder" Path="ActualHeight"/>
                        </MultiBinding>
                    </RectangleGeometry.Rect>
                </RectangleGeometry>
            </Border.Clip>

            <ContentPresenter RenderTransformOrigin="0.5,0.5">
                <ContentPresenter.RenderTransform>
                    <ScaleTransform ScaleX="{Binding Path=Value, ElementName=sldZoom}"
ScaleY="{Binding Path=Value, ElementName=sldZoom}"/>
                </ContentPresenter.RenderTransform>

                <ContentPresenter.Content>

                    <Canvas x:Name="solverCanvas" Background="AliceBlue"
ClipToBounds="True"  MouseWheel="SolverCanvas_MouseWheel">
                        <Path x:Name="GuidePath" Data="{Binding GuidePathData}"
StrokeEndLineCap="Round" StrokeStartLineCap="Round" StrokeThickness="1" Stroke="Red"
StrokeDashArray="4 4" SnapsToDevicePixels="True">
                            <Canvas.Left>
                                <MultiBinding Converter="{StaticResource
MyHalfValueConverter}">
                                    <Binding ElementName="solverCanvas"
Path="ActualWidth"/>
                                    <Binding ElementName="GuidePath"
Path="ActualWidth"/>
                                </MultiBinding>
                            </Canvas.Left>
                            <Canvas.Top>
                                <MultiBinding Converter="{StaticResource
MyHalfValueConverter}">
                                    <Binding ElementName="solverCanvas"
Path="ActualHeight"/>
                                    <Binding ElementName="GuidePath"
Path="ActualHeight"/>
```

```
                                </MultiBinding>
                            </Canvas.Top>
                        </Path>
                        <Path x:Name="HighlightPath" Data="{Binding
HighlightPathData}" Fill="Purple" Opacity="0.4" SnapsToDevicePixels="True">
                            <Canvas.Left>
                                <MultiBinding Converter="{StaticResource
MyHalfValueConverter}">
                                    <Binding ElementName="solverCanvas"
Path="ActualWidth"/>
                                    <Binding ElementName="GuidePath"
Path="ActualWidth"/>
                                </MultiBinding>
                            </Canvas.Left>
                            <Canvas.Top>
                                <MultiBinding Converter="{StaticResource
MyHalfValueConverter}">
                                    <Binding ElementName="solverCanvas"
Path="ActualHeight"/>
                                    <Binding ElementName="GuidePath"
Path="ActualHeight"/>
                                </MultiBinding>
                            </Canvas.Top>
                        </Path>
                        <Path x:Name="MazePath" Data="{Binding MazePathData}"
StrokeEndLineCap="Round" StrokeStartLineCap="Round" StrokeThickness="{Binding
WallThickness}" Stroke="Black" SnapsToDevicePixels="True">
                            <Canvas.Left>
                                <MultiBinding Converter="{StaticResource
MyHalfValueConverter}">
                                    <Binding ElementName="solverCanvas"
Path="ActualWidth"/>
                                    <Binding ElementName="GuidePath"
Path="ActualWidth"/>
                                </MultiBinding>
                            </Canvas.Left>
                            <Canvas.Top>
                                <MultiBinding Converter="{StaticResource
MyHalfValueConverter}">
                                    <Binding ElementName="solverCanvas"
Path="ActualHeight"/>
                                    <Binding ElementName="GuidePath"
Path="ActualHeight"/>
                                </MultiBinding>
                            </Canvas.Top>
                        </Path>
                        <Path x:Name="RoutePath" Data="{Binding RoutePathData}"
StrokeEndLineCap="Round" StrokeStartLineCap="Round" StrokeThickness="{Binding
RoutePathLineThickness}" Stroke="Green" SnapsToDevicePixels="True">
                            <Canvas.Left>
                                <MultiBinding Converter="{StaticResource
MyHalfValueConverter}">
                                    <Binding ElementName="solverCanvas"
Path="ActualWidth"/>
                                    <Binding ElementName="GuidePath"
Path="ActualWidth"/>
                                </MultiBinding>
                            </Canvas.Left>
                            <Canvas.Top>
                                <MultiBinding Converter="{StaticResource
MyHalfValueConverter}">
```

```
                                    <Binding ElementName="solverCanvas"
Path="ActualHeight"/>
                                    <Binding ElementName="GuidePath"
Path="ActualHeight"/>
                                </MultiBinding>
                            </Canvas.Top>
                        </Path>
                    </Canvas>
                </ContentPresenter.Content>
            </ContentPresenter>
        </Border>

        <Border BorderBrush="Black" BorderThickness="1" Grid.Row="1" Grid.Column="1"
Margin="2,0,4,4">
            <DataGrid x:Name="dataGrid" ItemsSource="{Binding
AlgorithmMetricCollection}" AutoGenerateColumns="False" Margin="0,2,0,0"
CanUserAddRows="False" CanUserDeleteRows="False" IsReadOnly="True"
MouseLeftButtonDown="Row_ClickedSolver" CanUserSortColumns="False">
                <DataGrid.Columns>
                    <DataGridTextColumn  Header="Algorithm" Binding="{Binding
AlgorithmName}"  />
                    <DataGridTextColumn  Header="Time Taken" Binding="{Binding
TimeSpan, StringFormat={}{0:hh':'mm':'ss'.'ffff}}" />
                    <DataGridTextColumn  Header="Time Taken (ms)" Binding="{Binding
ElapsedMilliseconds}" />
                </DataGrid.Columns>
            </DataGrid>
        </Border>
    </Grid>
</Window>
```

MazeSolverWindow.cs

The MazeSolverWindow.xaml page has a C# code behind file listed below. This is used to provide all the event handling and validation functionality for the UI controls and user inputs.

The `btnSolve_Click()` routine uses **Try Catch Exception Handling** to ensure that an algorithm is always selected before any attempt to find a solution is made.

The `btnOpenMaze_Click()` routine demonstrates the usage of the **File Open** dialog from standard controls provide by the Microsoft .NET Framework libraries. This is used to open a Maze (*.maz) text file which is the file format that the Mazer application saves its mazes as. This is exactly as was planned in my original design.

```
using MazeLibrary;
using MazeLibrary.Algorithms;
using MazeLibrary.Events;
using Mazer.Helpers;
using Mazer.Model;
using Mazer.ViewModels;
using Microsoft.Win32;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
```

```
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using System.Windows.Threading;

namespace Mazer
{
    /// <summary>
    /// Interaction logic for MazeSolverWindow.xaml
    /// </summary>
    public partial class MazeSolverWindow : Window
    {
        //_route stores the latest route and _grid stores it's corrasponding grid
        private MazeSolverWindowViewModel _mazeSolverWindowViewModel;
        private MazeLibrary.Grid _grid = null;
        private List<Cell> _route = null;
        private List<List<Cell>> _routeStorage = new List<List<Cell>>();
        private List<List<Cell>> _highlightStorage = new List<List<Cell>>();

        public MazeSolverWindow()
        {
            InitializeComponent();

            //intialises a new MazeSolverWindowViewModel
            _mazeSolverWindowViewModel = new MazeSolverWindowViewModel();
            DataContext = _mazeSolverWindowViewModel;

            //Setting up buttons that require a route through the maze to be inactive
before a route has been found.
            //btnSaveAsMaz.IsEnabled = false;
            btnSaveImage.IsEnabled = false;
            btnSaveResults.IsEnabled = false;
            btnSolve.IsEnabled = false;

            //Many algorithm boxes are inactive as they haven't been implemented yet
            BinaryTreeBox.IsEnabled = false;
            HuntandKillBox.IsEnabled = false;
            KruskalsBox.IsEnabled = false;
            PrimsBox.IsEnabled = false;
            RecursiveDivisionBox.IsEnabled = false;
            SidewinderBox.IsEnabled = false;
            SimpleRectangleBox.IsEnabled = false;
            WilsonsAlgorithmBox.IsEnabled = false;

            //Aesthetically Maze looks better when grid lines arn't displayed
            hideGridLinesBox.IsChecked = true;
            _mazeSolverWindowViewModel.HideGridLinesSelected = true;

            //Set up initial values for the ui window
            _mazeSolverWindowViewModel.Delay = 1;
            _mazeSolverWindowViewModel.WallThickness = 2;
            _mazeSolverWindowViewModel.WallSize = 25;
            _mazeSolverWindowViewModel.RoutePathLineThickness = 4;

        }

        private void btnBack_Click(object sender, RoutedEventArgs e)
        {
            //Closes window and returns control to the startup screen
            Close();
        }

        private async void btnSolve_Click(object sender, RoutedEventArgs e)
```

```
        {
            try
            {
                //Clears both route and highlight stores left over from any previous
runs of the program
                _routeStorage.Clear();
                _highlightStorage.Clear();

                //Clears previous algorithm and time data stored in the
AlgorithmMetricCollection list
                _mazeSolverWindowViewModel.AlgorithmMetricCollection.Clear();

                //Clears any leftover highlight from a previous run of the program
                ClearHighlight();

                //delay equals the Delay in the view model
                int delay = _mazeSolverWindowViewModel.Delay;

                //Setting buttons to be inactive while the program is busy solving a
maze
                sldDelay.IsEnabled = false;
                txtDelay.IsEnabled = false;
                btnSolve.IsEnabled = false;
                btnSaveImage.IsEnabled = false;
                btnSaveResults.IsEnabled = false;

                hideGridLinesBox.IsEnabled = false;
                hideHighlightBox.IsEnabled = false;

                //Budges the loaded maze so it will fit onto the canvas completely for
most reasonable mazes
                _mazeSolverWindowViewModel.WallSize = 25;
                _mazeSolverWindowViewModel.RoutePathLineThickness = 4;
                while (_mazeSolverWindowViewModel.WallSize * _grid.Width >=
solverCanvas.ActualWidth ||
                        _mazeSolverWindowViewModel.WallSize * _grid.Height >=
solverCanvas.ActualHeight)
                {
                    _mazeSolverWindowViewModel.WallSize--;

                    //Changes the RoutePathLineThickness so the line thickness will
fit inside most any wallSize on larger maze sizes
                    if (_mazeSolverWindowViewModel.WallSize == 15)
_mazeSolverWindowViewModel.RoutePathLineThickness = 3;
                    else if (_mazeSolverWindowViewModel.WallSize == 10)
_mazeSolverWindowViewModel.RoutePathLineThickness = 2;
                        else if (_mazeSolverWindowViewModel.WallSize == 5)
_mazeSolverWindowViewModel.RoutePathLineThickness = 1;

                    if (_mazeSolverWindowViewModel.WallSize <= 1)
                    {
                        break;
                    }
                }

                //Sets all timing data to null for the new run
                AlgorithmMetric aStarAlgorithmAlgorithmMetric = null;
                AlgorithmMetric binaryTreeAlgorithmAlgorithmMetric = null;
                AlgorithmMetric drunkardsWalkAlgorithmAlgorithmMetric = null;
                AlgorithmMetric huntAndKillAlgorithmAlgorithmMetric = null;
                AlgorithmMetric kruskalsAlgorithmAlgorithmMetric = null;
                AlgorithmMetric primsAlgorithmAlgorithmMetric = null;
```

```
                AlgorithmMetric recursiveBackTrackerAlgorithmAlgorithmMetric = null;
                AlgorithmMetric recursiveDivisionAlgorithmAlgorithmMetric = null;
                AlgorithmMetric sidewinderAlgorithmAlgorithmMetric = null;
                AlgorithmMetric simpleRectangleAlgorithmAlgorithmMetric = null;
                AlgorithmMetric LeftWallFollowerAlgorithmAlgorithmMetric = null;
                AlgorithmMetric RightWallFollowerAlgorithmAlgorithmMetric = null;
                AlgorithmMetric wilsonsAlgorithmAlgorithmMetric = null;

                //Booleans store value in checkboxes so solving can't be confused if
the user clicks a new algorithm
                //box before it has finished making mazes
                bool doAStarAlgorithm = AStarBox.IsChecked.Value;
                bool doBinaryTreeAlgorithm = BinaryTreeBox.IsChecked.Value;
                bool doDrunkardsWalkAlgorithm = DrunkardsWalkBox.IsChecked.Value;
                bool doHuntAndKillAlgorithm = HuntandKillBox.IsChecked.Value;
                bool doKruskalsAlgorithm = KruskalsBox.IsChecked.Value;
                bool doPrimsAlgorithm = PrimsBox.IsChecked.Value;
                bool doRecursiveBackTrackerAlgorithm =
RecursiveBackTrackerBox.IsChecked.Value;
                bool doRecursiveDivisionAlgorithm =
RecursiveDivisionBox.IsChecked.Value;
                bool doSidewinderAlgorithm = SidewinderBox.IsChecked.Value;
                bool doSimpleRectangleAlgorithm = SimpleRectangleBox.IsChecked.Value;
                bool doLeftWallFollowerAlgorithm =
LeftWallFollowerBox.IsChecked.Value;
                bool doRightWallFollowerAlgorithm =
RightWallFollowerBox.IsChecked.Value;
                bool doWilsonsAlgorithm = WilsonsAlgorithmBox.IsChecked.Value;

                //Ensures at least one algorithm was selected to run with, throws
exception if not
                if (!doAStarAlgorithm && !doBinaryTreeAlgorithm &&
!doDrunkardsWalkAlgorithm && !doHuntAndKillAlgorithm && !doKruskalsAlgorithm &&
!doPrimsAlgorithm
                     && !doRecursiveBackTrackerAlgorithm &&
!doRecursiveDivisionAlgorithm && !doSidewinderAlgorithm && !doSimpleRectangleAlgorithm
&&
                     !doLeftWallFollowerAlgorithm && !doRightWallFollowerAlgorithm &&
!doWilsonsAlgorithm)
                {
                    throw new Exception("No algorithm selected");
                }

                //These cells are the ones in the algorithm that will be highlighted
                List<Cell> visitedCells = new List<Cell>();

                //Hides gridlines if checkbox is checked
                if (_mazeSolverWindowViewModel.HideGridLinesSelected)
                {
                    GuidePath.Visibility = System.Windows.Visibility.Hidden;
                }
                else
                {
                    GuidePath.Visibility = System.Windows.Visibility.Visible;
                }

                //Allows the displaying of a maze to be on one thread while the
algorithms still create the maze on another.
                //(Algorithms, therefore, run in the background on one thread, firing
off events to update the UI on the main thread with new information)
                await Task.Run(() =>
                {
```

```
                    //The guide for the maze is always the same, as every maze is made
with the same dimensions as is the maze so that is displayed as well

                    DisplayGuide(_grid);
                    DisplayMaze(_grid);

                    Stopwatch stopwatch = new Stopwatch();

                    //All algorithm's if-checked statements follow the same procedure.
I will therefore only comment on the A* algorithm
                    #region A*

                    if (doAStarAlgorithm)
                    {
                        //Starts a new route and displays it, therefore reseting any
previous route
                        _route = new List<Cell>();
                        DisplayRoute(_grid, _route);
                        _mazeSolverWindowViewModel.AlgorithimLabel =
string.Format("Solving (A*)");
                        //Stopwatch is reset and started and AStarAlgorithm is created
with the required delay. Whenever a routeAddingNewCell event
                        //is called in the code it creates a new handle for it on
another thread. There it updates the ui canvas. After the _route has
                        //been made the stopwatch is stopped
                        stopwatch.Reset();
                        stopwatch.Start();
                        AStarAlgorithm aStar = new AStarAlgorithm(delay);
                        aStar.RaiseRoutePathAddedNewCellEvent +=
HandleRoutePathAddingNewCellEvent;
                        if (!_mazeSolverWindowViewModel.HideHighlightSelected)
aStar.RaiseHighlightPathAddedNewCellEvent += HandleHighlightCellEvent;
                        //_route = aStar.Solve(_grid);
                        aStar.Solve(ref _grid, out _route, out visitedCells);
                        stopwatch.Stop();

                        // _routeAndHighlightStorage.Add(_route, visitedCells);
                        _routeStorage.Add(_route);
                        _highlightStorage.Add(visitedCells);

                        //Creates and stores all data recorded for use in the creation
of the histogram
                        aStarAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "A*",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                        };
                    }

                    #endregion

                    #region Binary Tree

                    if (doBinaryTreeAlgorithm)
                    {
                        _route = new List<Cell>();
                        DisplayRoute(_grid, _route);

                        _mazeSolverWindowViewModel.AlgorithimLabel =
string.Format("Solving (Binary Tree)");
```

```
                        stopwatch.Reset();
                        stopwatch.Start();
                        BinaryTreeAlgorithm binaryTreeAlgorithm = new
BinaryTreeAlgorithm(delay);
                        binaryTreeAlgorithm.RaiseRoutePathAddedNewCellEvent +=
HandleRoutePathAddingNewCellEvent;
                        _route = binaryTreeAlgorithm.Solve(_grid);
                        stopwatch.Stop();

                        ////_routeStorage.Add(_route);

                        binaryTreeAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "BinaryTree",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                        };
                    }

                    #endregion

                    #region Drunkards Walk

                    if (doDrunkardsWalkAlgorithm)
                    {
                        _route = new List<Cell>();
                        DisplayRoute(_grid, _route);

                        _mazeSolverWindowViewModel.AlgorithimLabel =
string.Format("Solving (Drunkard's Walk)");

                        stopwatch.Reset();
                        stopwatch.Start();
                        DrunkardsWalkAlgorithm drunkardsWalkAlgorithm = new
DrunkardsWalkAlgorithm(delay);
                        drunkardsWalkAlgorithm.RaiseRoutePathAddedNewCellEvent +=
HandleRoutePathAddingNewCellEvent;
                        if (!_mazeSolverWindowViewModel.HideHighlightSelected)
drunkardsWalkAlgorithm.RaiseHighlightPathAddedNewCellEvent +=
HandleHighlightCellEvent;
                        // _route = drunkardsWalkAlgorithm.Solve(_grid);
                        drunkardsWalkAlgorithm.Solve(ref _grid, out _route, out
visitedCells);
                        stopwatch.Stop();

                        _routeStorage.Add(_route);
                        _highlightStorage.Add(visitedCells);

                        drunkardsWalkAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "DrunkardsWalk",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                        };
                    }

                    #endregion

                    #region Hunt and Kill

                    if (doHuntAndKillAlgorithm)
                    {
```

```
                        _route = new List<Cell>();
                        DisplayRoute(_grid, _route);
                        _mazeSolverWindowViewModel.AlgorithimLabel =
string.Format("Solving (Hunt & Kill)");

                        stopwatch.Reset();
                        stopwatch.Start();
                        HuntAndKillAlgorithm huntAndKillAlgorithm = new
HuntAndKillAlgorithm(delay);
                        huntAndKillAlgorithm.RaiseRoutePathAddedNewCellEvent +=
HandleRoutePathAddingNewCellEvent;
                        _route = huntAndKillAlgorithm.Solve(_grid);
                        stopwatch.Stop();

                        //_routeStorage.Add(_route);

                        huntAndKillAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "HuntandKill",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                        };
                    }

                    #endregion

                    #region Kruskals

                    if (doKruskalsAlgorithm)
                    {
                        _route = new List<Cell>();
                        DisplayRoute(_grid, _route);
                        _mazeSolverWindowViewModel.AlgorithimLabel =
string.Format("Solving (Krustals)");

                        stopwatch.Reset();
                        stopwatch.Start();
                        KruskalsAlgorithm kruskalsAlgorithm = new
KruskalsAlgorithm(delay);
                        kruskalsAlgorithm.RaiseRoutePathAddedNewCellEvent +=
HandleRoutePathAddingNewCellEvent;
                        _route = kruskalsAlgorithm.Solve(_grid);
                        stopwatch.Stop();

                        //////_routeStorage.Add(_route);

                        kruskalsAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "Kruskals",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                        };
                    }

                    #endregion

                    #region Prims

                    if (doPrimsAlgorithm)
                    {
                        _route = new List<Cell>();
                        DisplayRoute(_grid, _route);
```

```
                        _mazeSolverWindowViewModel.AlgorithimLabel =
string.Format("Solving (Prims)");

                        stopwatch.Reset();
                        stopwatch.Start();
                        PrimsAlgorithm primsAlgorithm = new PrimsAlgorithm(delay);
                        primsAlgorithm.RaiseRoutePathAddedNewCellEvent +=
HandleRoutePathAddingNewCellEvent;
                        _route = primsAlgorithm.Solve(_grid);
                        stopwatch.Stop();

                        ////_routeStorage.Add(_route);

                        primsAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "Prims",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                        };
                    }

                    #endregion

                    #region Recursive Backtracker

                    if (doRecursiveBackTrackerAlgorithm)
                    {
                        _route = new List<Cell>();
                        DisplayRoute(_grid, _route);
                        _mazeSolverWindowViewModel.AlgorithimLabel =
string.Format("Solving (Recursive Backtracker)");

                        stopwatch.Reset();
                        stopwatch.Start();
                        RecursiveBackTrackerAlgorithm recursiveBackTrackerAlgorithm =
new RecursiveBackTrackerAlgorithm(delay);
                        recursiveBackTrackerAlgorithm.RaiseRoutePathAddedNewCellEvent
+= HandleRoutePathAddingNewCellEvent;
                        if (!_mazeSolverWindowViewModel.HideHighlightSelected)
recursiveBackTrackerAlgorithm.RaiseHighlightPathAddedNewCellEvent +=
HandleHighlightCellEvent;
                        //  _route = recursiveBackTrackerAlgorithm.Solve(_grid);
                        recursiveBackTrackerAlgorithm.Solve(ref _grid, out _route, out
visitedCells);
                        //RecursiveBackTrackerAlgorithm.Solve(_grid);
                        stopwatch.Stop();

                        _routeStorage.Add(_route);
                        _highlightStorage.Add(visitedCells);

                        recursiveBackTrackerAlgorithmAlgorithmMetric = new
AlgorithmMetric
                        {
                            AlgorithmName = "RecursiveBackTracker",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                        };
                    }

                    #endregion

                    #region Recursive Division
```

```
                    if (doRecursiveDivisionAlgorithm)
                    {
                        _route = new List<Cell>();
                        DisplayRoute(_grid, _route);
                        _mazeSolverWindowViewModel.AlgorithimLabel =
string.Format("Solving (Recursive Division)");

                        stopwatch.Reset();
                        stopwatch.Start();
                        RecursiveDivisionAlgorithm recursiveDivisionAlgorithm = new
RecursiveDivisionAlgorithm(delay);
                        recursiveDivisionAlgorithm.RaiseRoutePathAddedNewCellEvent +=
HandleRoutePathAddingNewCellEvent;
                        _route = recursiveDivisionAlgorithm.Solve(_grid);
                        stopwatch.Stop();

                        //_routeStorage.Add(_route);

                        recursiveDivisionAlgorithmAlgorithmMetric = new
AlgorithmMetric
                        {
                            AlgorithmName = "RecursiveDivision",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                        };
                    }

                    #endregion

                    #region Sidewinder

                    if (doSidewinderAlgorithm)
                    {
                        _route = new List<Cell>();
                        DisplayRoute(_grid, _route);
                        _mazeSolverWindowViewModel.AlgorithimLabel =
string.Format("Solving (Sidewinder)");

                        stopwatch.Reset();
                        stopwatch.Start();
                        SidewinderAlgorithm sidewinderAlgorithm = new
SidewinderAlgorithm(delay);
                        sidewinderAlgorithm.RaiseRoutePathAddedNewCellEvent +=
HandleRoutePathAddingNewCellEvent;
                        _route = sidewinderAlgorithm.Solve(_grid);
                        stopwatch.Stop();

                        //_routeStorage.Add(_route);

                        sidewinderAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "Sidewinder",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                        };
                    }

                    #endregion

                    #region LeftWallFollower
```

```
                    if (doLeftWallFollowerAlgorithm)
                    {
                        _route = new List<Cell>();
                        DisplayRoute(_grid, _route);
                        _mazeSolverWindowViewModel.AlgorithimLabel =
string.Format("Solving (Left Wall Follower)");
                        stopwatch.Reset();
                        stopwatch.Start();
                        WallFollowerAlgorithm wallFollowerAlgorithm = new
WallFollowerAlgorithm(delay);
                        wallFollowerAlgorithm.RaiseRoutePathAddedNewCellEvent +=
HandleRoutePathAddingNewCellEvent;
                        if (!_mazeSolverWindowViewModel.HideHighlightSelected)
wallFollowerAlgorithm.RaiseHighlightPathAddedNewCellEvent += HandleHighlightCellEvent;
                        // _route = wallFollowerAlgorithm.SolveLeft2(_grid);
                        wallFollowerAlgorithm.SolveLeft(ref _grid, out _route, out
visitedCells);
                        stopwatch.Stop();

                        _routeStorage.Add(_route);
                        _highlightStorage.Add(visitedCells);

                        LeftWallFollowerAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "LeftWallFollower",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                        };
                    }

                    #endregion

                    #region RightWallFollower

                    if (doRightWallFollowerAlgorithm)
                    {
                        _route = new List<Cell>();
                        DisplayRoute(_grid, _route);
                        _mazeSolverWindowViewModel.AlgorithimLabel =
string.Format("Solving (Right Wall Follower)");
                        stopwatch.Reset();
                        stopwatch.Start();
                        WallFollowerAlgorithm wallFollowerAlgorithm = new
WallFollowerAlgorithm(delay);
                        wallFollowerAlgorithm.RaiseRoutePathAddedNewCellEvent +=
HandleRoutePathAddingNewCellEvent;
                        if (!_mazeSolverWindowViewModel.HideHighlightSelected)
wallFollowerAlgorithm.RaiseHighlightPathAddedNewCellEvent += HandleHighlightCellEvent;
                        //_route = wallFollowerAlgorithm.SolveRight3(_grid);
                        wallFollowerAlgorithm.SolveRight(ref _grid, out _route, out
visitedCells);
                        stopwatch.Stop();

                        _routeStorage.Add(_route);
                        _highlightStorage.Add(visitedCells);

                        RightWallFollowerAlgorithmAlgorithmMetric = new
AlgorithmMetric
                        {
                            AlgorithmName = "RightWallFollower",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
```

```
                        };
                    }

                    #endregion

                    #region Simple Rectangle

                    if (doSimpleRectangleAlgorithm)
                    {
                        _route = new List<Cell>();
                        DisplayRoute(_grid, _route);
                        _mazeSolverWindowViewModel.AlgorithimLabel =
string.Format("Solving (Simple Rectangle)");

                        stopwatch.Reset();
                        stopwatch.Start();
                        SimpleRectangleAlgorithm simpleRectangleAlgorithm = new
SimpleRectangleAlgorithm(delay);
                        simpleRectangleAlgorithm.RaiseRoutePathAddedNewCellEvent +=
HandleRoutePathAddingNewCellEvent;
                        _route = simpleRectangleAlgorithm.Solve(_grid);
                        stopwatch.Stop();

                        //_routeStorage.Add(_route);

                        simpleRectangleAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "SimpleRectangle",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                        };
                    }

                    #endregion

                    #region Wilsons

                    if (doWilsonsAlgorithm)
                    {
                        _route = new List<Cell>();
                        DisplayRoute(_grid, _route);
                        _mazeSolverWindowViewModel.AlgorithimLabel =
string.Format("Solving (Wilsons)");

                        stopwatch.Reset();
                        stopwatch.Start();
                        WilsonsAlgorithm wilsonsAlgorithm = new
WilsonsAlgorithm(delay);
                        wilsonsAlgorithm.RaiseRoutePathAddedNewCellEvent +=
HandleRoutePathAddingNewCellEvent;
                        _route = wilsonsAlgorithm.Solve(_grid);
                        stopwatch.Stop();

                        //_routeStorage.Add(_route);

                        wilsonsAlgorithmAlgorithmMetric = new AlgorithmMetric
                        {
                            AlgorithmName = "WilsonsAlgorithm",
                            TimeSpan = stopwatch.Elapsed,
                            ElapsedMilliseconds = stopwatch.ElapsedMilliseconds,
                        };
                    }
```

```
                    #endregion

                    _mazeSolverWindowViewModel.AlgorithimLabel = "";


                    //To ensure a route is always displayed I force another displaying
of the latest grid here
                    DisplayRoute(_grid, _route);
                });

                //Adds only the algorithm times from the algorithms that created a
maze to an AlgorithmMetricCollection as a storage for all the algorithms used
                if (aStarAlgorithmAlgorithmMetric != null)
                {

_mazeSolverWindowViewModel.AlgorithmMetricCollection.Add(aStarAlgorithmAlgorithmMetric
);
                }
                if (binaryTreeAlgorithmAlgorithmMetric != null)
                {

_mazeSolverWindowViewModel.AlgorithmMetricCollection.Add(binaryTreeAlgorithmAlgorithmM
etric);
                }
                if (drunkardsWalkAlgorithmAlgorithmMetric != null)
                {

_mazeSolverWindowViewModel.AlgorithmMetricCollection.Add(drunkardsWalkAlgorithmAlgorit
hmMetric);
                }
                if (huntAndKillAlgorithmAlgorithmMetric != null)
                {

_mazeSolverWindowViewModel.AlgorithmMetricCollection.Add(huntAndKillAlgorithmAlgorithm
Metric);
                }
                if (kruskalsAlgorithmAlgorithmMetric != null)
                {

_mazeSolverWindowViewModel.AlgorithmMetricCollection.Add(kruskalsAlgorithmAlgorithmMet
ric);
                }
                if (primsAlgorithmAlgorithmMetric != null)
                {

_mazeSolverWindowViewModel.AlgorithmMetricCollection.Add(primsAlgorithmAlgorithmMetric
);
                }
                if (recursiveBackTrackerAlgorithmAlgorithmMetric != null)
                {

_mazeSolverWindowViewModel.AlgorithmMetricCollection.Add(recursiveBackTrackerAlgorithm
AlgorithmMetric);
                }
                if (recursiveDivisionAlgorithmAlgorithmMetric != null)
                {

_mazeSolverWindowViewModel.AlgorithmMetricCollection.Add(recursiveDivisionAlgorithmAlg
orithmMetric);
                }
                if (sidewinderAlgorithmAlgorithmMetric != null)
```

```
                {
_mazeSolverWindowViewModel.AlgorithmMetricCollection.Add(sidewinderAlgorithmAlgorithmM
etric);
                }
                if (simpleRectangleAlgorithmAlgorithmMetric != null)
                {
_mazeSolverWindowViewModel.AlgorithmMetricCollection.Add(simpleRectangleAlgorithmAlgor
ithmMetric);
                }
                if (LeftWallFollowerAlgorithmAlgorithmMetric != null)
                {
_mazeSolverWindowViewModel.AlgorithmMetricCollection.Add(LeftWallFollowerAlgorithmAlgo
rithmMetric);
                }
                if (RightWallFollowerAlgorithmAlgorithmMetric != null)
                {
_mazeSolverWindowViewModel.AlgorithmMetricCollection.Add(RightWallFollowerAlgorithmAlg
orithmMetric);
                }
                if (wilsonsAlgorithmAlgorithmMetric != null)
                {
_mazeSolverWindowViewModel.AlgorithmMetricCollection.Add(wilsonsAlgorithmAlgorithmMetr
ic);
                }

                //As everything has finished and there are now accessable routes all
sliders and buttons are now enabled
                sldDelay.IsEnabled = true;
                txtDelay.IsEnabled = true;
                btnSolve.IsEnabled = true;
                btnSaveImage.IsEnabled = true;
                //btnSaveAsMaz.IsEnabled = true;
                btnSaveResults.IsEnabled = true;

                hideGridLinesBox.IsEnabled = true;
                hideHighlightBox.IsEnabled = true;
            }
            catch
            {
                //Throwing a error message if there is no selected algorithm to run.
Ensuring all controls are reset as well
                sldDelay.IsEnabled = true;
                txtDelay.IsEnabled = true;
                btnSolve.IsEnabled = true;

                btnSaveImage.IsEnabled = false;
                btnSaveResults.IsEnabled = false;

                hideHighlightBox.IsEnabled = true;
                hideGridLinesBox.IsEnabled = true;
                MessageBox.Show("Error: An algorithm must be selected to solve a maze,
please select one.", "Error Loading File", MessageBoxButton.OK,
MessageBoxImage.Error);
            }
        }

        private void ClearHighlight()
```

```
        {
            //Clears all highlight data
            _mazeSolverWindowViewModel.HighlightPathData = Geometry.Parse("");
        }

        void HandleRoutePathAddingNewCellEvent(object sender,
RoutePathAddedNewCellEventArgs e)
        {
            //Displays the current _Route in whatever state of completeness it is in
            DisplayRoute(e.Grid, e.RouteCells);
        }

        void HandleHighlightCellEvent(object sender,
HighlightPathAddedNewCellEventArgs e)
        {
            //Displays the highlight list of cells
            DisplayHighlight(e.Grid, e.HighlightCells);
        }

        private void DisplayGuide(MazeLibrary.Grid _grid)
        {
            //Runs this code at the same time as running the rest of the program
            Dispatcher.Invoke(DispatcherPriority.Normal, (Action)delegate
            {
                int width = _grid.Width;
                int height = _grid.Height;
                int wallSize = _mazeSolverWindowViewModel.WallSize;

                //Creates the guide path with the pre-created data
                string guidePath = PathHelper.GetGuidePath(width, height, wallSize);
                _mazeSolverWindowViewModel.GuidePathData = Geometry.Parse(guidePath);
            });
        }

        private void DisplayMaze(MazeLibrary.Grid grid)
        {
            //Runs this code at the same time as running the rest of the program
            Dispatcher.Invoke(DispatcherPriority.Normal, (Action)delegate
            {
                int width = grid.Width;
                int height = grid.Height;
                int wallSize = _mazeSolverWindowViewModel.WallSize;

                //Creates the maze path with the pre-created data and the current grid
                string mazePath = PathHelper.GetMazeGeneratorPath(grid, wallSize);
                _mazeSolverWindowViewModel.MazePathData = Geometry.Parse(mazePath);
            });
        }

        private void DisplayRoute(MazeLibrary.Grid grid, List<Cell> route)
        {
            //Runs this code at the same time as running the rest of the program
            Dispatcher.Invoke(DispatcherPriority.Normal, (Action)delegate
            {
                int width = grid.Width;
                int height = grid.Height;
                int wallSize = _mazeSolverWindowViewModel.WallSize;

                //Creates the route path with the pre-created data and the current
route
                string routePath = PathHelper.GetMazeSolverPath(grid,
_mazeSolverWindowViewModel.WallSize, route);
```

```
                _mazeSolverWindowViewModel.RoutePathData = Geometry.Parse(routePath);
            });
        }

        private void DisplayHighlight(MazeLibrary.Grid grid, List<Cell>
highlightCells)
        {
            //Runs this code at the same time as running the rest of the program
            Dispatcher.Invoke(DispatcherPriority.Normal, (Action)delegate
            {
                int width = _grid.Width;
                int height = _grid.Height;
                int wallSize = _mazeSolverWindowViewModel.WallSize;
                int wallThickness = _mazeSolverWindowViewModel.WallThickness;

                //Creates a highlight at each of the passed cells
                string highlightCellPath = PathHelper.GetHighlightCellPath(wallSize,
wallThickness, highlightCells);
                _mazeSolverWindowViewModel.HighlightPathData =
Geometry.Parse(highlightCellPath);
            });
        }

        private void btnSaveImage_Click(object sender, RoutedEventArgs e)
        {
            //Saves the currently displayed maze as an image in the SaveImageHelper
class
            SaveImageHelper.SavePngImage(solverCanvas, "Solved");
        }

        private void btnOpenMaze_Click(object sender, RoutedEventArgs e)
        {
            //try is used for Exception Handling, in case the maze file is corrupted
            try
            {
                //Clears residue highlight left from an algorithm's run
                _mazeSolverWindowViewModel.AlgorithmMetricCollection.Clear();

                OpenFileDialog openFileDialog = new OpenFileDialog();
                string applicationFolder = "Mazer Save Files";
                //Gets the path for the myDocuments path for whatever computer this
program is on.
                string myDocuments =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
                //combines these paths
                string applicationPath = System.IO.Path.Combine(myDocuments,
applicationFolder, "Generated Mazes (.maz)");
                //Sees if this path already exists
                if (!Directory.Exists(applicationPath))//?needed?
                {
                    Directory.CreateDirectory(applicationPath);
                }
                //All this provides the usual "Save As" function of any microsoft
application.
                openFileDialog.Filter = "maz (*.maz)|*.maz|All Files (*.*)|*.*";
                openFileDialog.FilterIndex = 0;
                openFileDialog.Title = "Open Mazer text File";
                openFileDialog.InitialDirectory = applicationPath;
                openFileDialog.RestoreDirectory = true;
                openFileDialog.CheckFileExists = false;
                openFileDialog.CheckPathExists = true;
                openFileDialog.DefaultExt = "maz";
```

```
                //dialogResult can be true, false or null (just in case)
                bool? dialogResult = openFileDialog.ShowDialog();

                //If what the user enters is Open (OK) will open the file.
                if (dialogResult == true)
                {
                    //_grid is collected using the filename from the function
GetGridFromFile
                    string filename = openFileDialog.FileName;

                    //Function returning the grid once it has been recreated from the
file
                    _grid = MazeLibrary.Grid.GetGridFromFile(filename);

                    //Sets dimensions of new grid
                    _mazeSolverWindowViewModel.Width = _grid.Width;
                    _mazeSolverWindowViewModel.Height = _grid.Height;

                    //Budges the loaded maze so it will fit onto the canvas completely
for most reasonable mazes
                    _mazeSolverWindowViewModel.WallSize = 25;
                    while (_mazeSolverWindowViewModel.WallSize * _grid.Width >=
solverCanvas.ActualWidth ||
                           _mazeSolverWindowViewModel.WallSize * _grid.Height >=
solverCanvas.ActualHeight)
                    {
                        _mazeSolverWindowViewModel.WallSize--;
                        if (_mazeSolverWindowViewModel.WallSize <= 1) { break; }
                    }


                    //Checks if checkbox is checked or not, if so set guide path to be
invisible
                    if (_mazeSolverWindowViewModel.HideGridLinesSelected)
                    {
                        GuidePath.Visibility = System.Windows.Visibility.Hidden;
                    }
                    else
                    {
                        GuidePath.Visibility = System.Windows.Visibility.Visible;
                    }

                    //Builds the guidePath and the mazePath using the dimensions and
the new grid for the maze path
                    string guidePath = PathHelper.GetGuidePath(_grid.Width,
_grid.Height, _mazeSolverWindowViewModel.WallSize);
                    _mazeSolverWindowViewModel.GuidePathData =
Geometry.Parse(guidePath);
                    string mazePath = PathHelper.GetMazeGeneratorPath(_grid,
_mazeSolverWindowViewModel.WallSize);
                    _mazeSolverWindowViewModel.MazePathData =
Geometry.Parse(mazePath);

                    //Reset route path data
                    _mazeSolverWindowViewModel.RoutePathData = Geometry.Parse("");

                    //Re-enable buttons that can now be run now that there  is a route
path
                    btnSolve.IsEnabled = true;
                    btnSaveImage.IsEnabled = false;
                    //btnSaveAsMaz.IsEnabled = false;
```

```
                    btnSaveResults.IsEnabled = false;
                    //Ensures no residual highlight is left from the last maze solving
                    ClearHighlight();
                }
            }
            catch
            {
                //Throwing a error message if the maze cannot be loaded. This stops
the loading of a maze, but doesn't crash the program
                StringBuilder sb = new StringBuilder();
                sb.AppendLine("Error: The maze you are trying to load may be
corrupted.");
                sb.AppendLine("");
                sb.AppendLine("Ensure the file has is structure as is shown:");
                sb.AppendLine("");
                sb.AppendLine("MazeName |Width|Height");
                sb.AppendLine("PWPW");
                sb.AppendLine("PPPW");
                sb.AppendLine("PWWW");
                sb.AppendLine("WWPP");
                sb.AppendLine("PPPW");
                sb.AppendLine("...");
                sb.AppendLine("");
                sb.AppendLine("Also ensure that the number of 4 character lines is
equal to the Width * the Height and that every 4 character line has at least one
character as a 'P'.");
                MessageBox.Show(sb.ToString(), "Error Loading File",
MessageBoxButton.OK, MessageBoxImage.Error);
            }
        }

        private void SolverCanvas_MouseWheel(object sender, MouseWheelEventArgs e)
        {
            //Code for zooming into the canvas with the mousewheel
            _mazeSolverWindowViewModel.MouseWheel += (e.Delta / 100);
        }

        private void btnSaveResults_Click(object sender, RoutedEventArgs e)
        {
            //Adds each of the used algorithm's names and times to the algorithmMetric
            Dictionary<string, long> data = new Dictionary<string, long>();
            foreach (AlgorithmMetric algorithmMetric in
_mazeSolverWindowViewModel.AlgorithmMetricCollection)
            {
                data.Add(algorithmMetric.AlgorithmName,
algorithmMetric.ElapsedMilliseconds);
            }

            //Displays the results window with this data and the title of which window
it came from
            ResultsWindow resultsWindow = new ResultsWindow();
            resultsWindow.Owner = this;
            resultsWindow.ResultsWindowViewModel.Data = data;
            resultsWindow.ResultsWindowViewModel.SourceName = "Solver Results";
            resultsWindow.ShowDialog();
        }

        private void Row_ClickedSolver(object sender, MouseButtonEventArgs e)
        {
            //Merp
```

```
            DataGridRow row =
(DataGridRow)ItemsControl.ContainerFromElement((DataGrid)sender,
(DependencyObject)e.OriginalSource);
            if (row == null) return;

            //Set the current route to the one at the desired index in _routeStorage
and the highlight to that index as well
            int index = row.GetIndex();
            DisplayRoute(_grid, _routeStorage[index]);
            DisplayHighlight(_grid, _highlightStorage[index]);
        }
    }
}
```

MazeStartUpWindow

The startup window is displayed below. The UI elements and layout of this window closely follow the original design.

MazeStartUpWindow.xaml

```
<Window x:Class="Mazer.MazeStartUpWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MazeStartUpWindow" Height="288" Width="231">
    <Grid>
        <TextBlock Text="MAZER" FontSize="18" HorizontalAlignment="Center"
VerticalAlignment="Top" Margin="0,12" FontWeight="Bold"/>

        <StackPanel Margin="50,60,50,20" VerticalAlignment="Center">
            <Button x:Name="btnGenerateMaze" Content="Generate Maze" Margin="4"
Click="btnGenerateMaze_Click" />
            <Button x:Name="btnSolveMaze" Content="Solve Maze" Margin="4"
Click="btnSolveMaze_Click"/>
            <Button x:Name="btnQuit" Content="Quit" Margin="4" Click="btnQuit_Click"/>
        </StackPanel>
    </Grid>
</Window>
```

MazeStartUpWindow.cs

The MazeStartUpWindow.xaml page has a C# code behind file listed below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Mazer
{
    /// <summary>
    /// Interaction logic for MazeStartUpWindow.xaml
    /// </summary>
    public partial class MazeStartUpWindow : Window
    {
        //Simply gives click events for the three buttons on this window. One for
opening the generate window,
        //one for opening the solve window and one for closing the current window
        public MazeStartUpWindow()
        {
            InitializeComponent();
        }

        private void btnQuit_Click(object sender, RoutedEventArgs e)
        {
            Close();
        }

        private void btnGenerateMaze_Click(object sender, RoutedEventArgs e)
        {
            MazeGeneratorWindow mazeGeneratorWindow = new MazeGeneratorWindow();
            mazeGeneratorWindow.ShowDialog();
        }

        private void btnSolveMaze_Click(object sender, RoutedEventArgs e)
        {
            MazeSolverWindow mazeSolverWindow = new MazeSolverWindow();
            mazeSolverWindow.ShowDialog();
        }
    }
}
```

View Models

The View Model classes are used for the data binding between the controls in the UI interface and the other parts of the program. The two-way data binding is defined in the XAML of the windows and ensures that the state of the various UI controls stays in step with properties in the view model and vice verse. For example, this ensures that if the wall thickness is equal to the value 4 in the view model then it is also equal to 4 in the MazeGeneratorWindow and wherever else a data binding has been set up.

MazeGeneratorWindowViewModel

This view model class is used specifically for data binding in the MazeGeneratorWindow.

Property	Description
`Width`	The width of the current grid, which is universal across grids.
`Height`	The height of the current grid, which is universal across grids.
`WallSize`	The wall size which is the distance from one wall to another.
`WallThickness`	The thickness of the walls.
`Delay`	The delay of each iteration to impose in an algorithm as the maze is generating.
`GuidePathData`	The geometric path for the guide path, which is underneath the maze path when it is visible.
`HighlightPathData`	The geometric path for the highlight path, which shows what cells the generator is currently thinking about.
`MazePathData`	The geometric path for the maze path, which is the actual maze walls.
`MouseWheel`	The value for the mouse wheel's rotation used to zoom in on the canvas.
`HideHighlightSelected`	A boolean indicating if the highlight path should be visible.
`HideWallAndPathUpdatesSelected`	A boolean indicating if the wall and path updates should be visible as the maze is generated.
`HideGridLinesSelected`	A boolean indicating if the background grid lines should be visible.
`AlgorithimLabel`	The name of what algorithm is currently.

```
using Mazer.Model;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Media;

namespace Mazer.ViewModels
{
    public class MazeGeneratorWindowViewModel : INotifyPropertyChanged
```

```
    {
        //INotifyPropertyChanged is inherited so all the properties in this ViewModel
can be automatically updated when they change,
        //e.g. algorithm checkbox is ticked then unticked

        public event PropertyChangedEventHandler PropertyChanged;

        private int _width;
        private int _height;
        private int _wallSize;
        private int _wallThickness;
        private int _delay;
        private Geometry _guidePathData;
        private Geometry _highlightPathData;
        private Geometry _mazePathData;
        private int _mouseWheel;
        private bool _hideHighlightSelected;
        private bool _hideWallAndPathUpdatesSelected;
        private bool _hideGridLinesSelected;
        private string _algorithimLabel;

        public MazeGeneratorWindowViewModel()
        {
            _algorithmMetricCollection = new ObservableCollection<AlgorithmMetric>();
        }

        //Property for building the results histogram
        private ObservableCollection<AlgorithmMetric> _algorithmMetricCollection;

        public ObservableCollection<AlgorithmMetric> AlgorithmMetricCollection
        {
            get { return _algorithmMetricCollection; }
            set
            {
                if (_algorithmMetricCollection != value)
                {
                    _algorithmMetricCollection = value;
                    NotifyPropertyChanged("AlgorithmMetricCollection");
                }
            }
        }

        //Property for getter and automatic setter (when value changed) for mouse
wheel (as alternative to zoom slider)
        public int MouseWheel
        {
            get { return _mouseWheel; }
            set
            {
                if (_mouseWheel != value)
                {
                    _mouseWheel = value;
                    NotifyPropertyChanged("MouseWheel");
                }
            }
        }

        //Properties for getters and automatic setters (when value changed) for width,
height, wallSize, wallThickness, delay,
        // guidePathData, highLightPathData, mazePathData, hideHighlightSelected,
hideWallAndPathUpdatesSelected, HideGridLinesSelected and AlgorithimLabel
        public int Width
```

```
        {
            get { return _width; }
            set
            {
                if (_width != value)
                {
                    _width = value;
                    NotifyPropertyChanged("Width");
                }
            }
        }

        public int Height
        {
            get { return _height; }
            set
            {
                if (_height != value)
                {
                    _height = value;
                    NotifyPropertyChanged("Height");
                }
            }
        }

        public int WallSize
        {
            get { return _wallSize; }
            set
            {
                if (_wallSize != value)
                {
                    _wallSize = value;
                    NotifyPropertyChanged("WallSize");
                }
            }
        }

        public int WallThickness
        {
            get { return _wallThickness; }
            set
            {
                if (_wallThickness != value)
                {
                    _wallThickness = value;
                    NotifyPropertyChanged("WallThickness");
                }
            }
        }

        public int Delay
        {
            get { return _delay; }
            set
            {
                if (_delay != value)
                {
                    _delay = value;
                    NotifyPropertyChanged("Delay");
                }
            }
```

```
        }

        public Geometry GuidePathData
        {
            get { return _guidePathData; }
            set
            {
                if (_guidePathData != value)
                {
                    _guidePathData = value;
                    NotifyPropertyChanged("GuidePathData");
                }
            }
        }

        public Geometry HighlightPathData
        {
            get { return _highlightPathData; }
            set
            {
                if (_highlightPathData != value)
                {
                    _highlightPathData = value;
                    NotifyPropertyChanged("HighlightPathData");
                }
            }
        }

        public Geometry MazePathData
        {
            get { return _mazePathData; }
            set
            {
                if (_mazePathData != value)
                {
                    _mazePathData = value;
                    NotifyPropertyChanged("MazePathData");
                }
            }
        }

        public bool HideHighlightSelected
        {
            get { return _hideHighlightSelected; }
            set
            {
                if (_hideHighlightSelected != value)
                {
                    _hideHighlightSelected = value;
                    NotifyPropertyChanged("HideHighlightSelected");
                }
            }
        }

        public bool HideWallAndPathUpdatesSelected
        {
            get { return _hideWallAndPathUpdatesSelected; }
            set
            {
                if (_hideWallAndPathUpdatesSelected != value)
                {
                    _hideWallAndPathUpdatesSelected = value;
```

```
                NotifyPropertyChanged("HideWallAndPathUpdatesSelected");
            }
        }
    }

    public bool HideGridLinesSelected
    {
        get { return _hideGridLinesSelected; }
        set
        {
            if (_hideGridLinesSelected != value)
            {
                _hideGridLinesSelected = value;
                NotifyPropertyChanged("HideGridLinesSelected");
            }
        }
    }

    public string AlgorithimLabel
    {
        get { return _algorithimLabel; }
        set
        {
            if (_algorithimLabel != value)
            {
                _algorithimLabel = value;
                NotifyPropertyChanged("AlgorithimLabel");
            }
        }
    }

    //Procedure tells the specific property variable to update it's stored value
as it has been changed
    private void NotifyPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }

    }
}
```

MazeSolverWindowViewModel

This view model class is used specifically for data binding in the MazeSolverWindowViewModel.

Property	Description
Width	The width of the current grid.
Weight	The height of the current grid.
WallSize	The wall size which is the distance from one wall to another.
WallThickness	The thickness of the walls.
Delay	The delay that will be applied to each iteration in and algorithm when solving a maze.
GuidePathData	The geometric path for the guide path. This is underneath the maze path when visible.
HighlightPathData	The geometric path for the highlight path, which shows what the solver is currently thinking about.
MazePathData	The geometric path for the maze path, which is the actual maze.
RoutePathData	The geometric path for the route path, which is the route from the start to the current cell. Once solved it is the route from the start to the end.
MouseWheel	The value for the mouse wheel's rotation used to zoom in on the canvas.
HideHighlightSelected	A boolean indicating if the highlight path should be visible.
HideWallAndPathUpdatesSelected	A boolean indicating if the wall and path updates should be visible as the maze is solved.
HideGridLinesSelected	A boolean indicating if the background grid lines should be visible.
AlgorithimLabel	The name of what algorithm is running.
RoutePathLineThickness	The value for how thick the route path line is, it is changed to be smaller if the wall size is small.

```
using Mazer.Model;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Media;

namespace Mazer.ViewModels
{
    public class MazeSolverWindowViewModel : INotifyPropertyChanged
    {
```

```
        //INotifyPropertyChanged is inherited so all the properties in this ViewModel
can be automatically updated when they change,
        //e.g. algorithm checkbox is ticked then unticked

        public event PropertyChangedEventHandler PropertyChanged;

        private int _width;
        private int _height;
        private int _wallSize;
        private int _wallThickness;
        private int _delay;
        private Geometry _guidePathData;
        private Geometry _highlightPathData;
        private Geometry _mazePathData;
        private Geometry _routePathData;
        private int _mouseWheel;
        private bool _hideHighlightSelected;
        private bool _hideGridLinesSelected;
        private string _algorithimLabel;
        private int _routePathLineThickness;

        public MazeSolverWindowViewModel()
        {
            _algorithmMetricCollection = new ObservableCollection<AlgorithmMetric>();
        }

        //Property for building the results histogram
        private ObservableCollection<AlgorithmMetric> _algorithmMetricCollection;

        public ObservableCollection<AlgorithmMetric> AlgorithmMetricCollection
        {
            get { return _algorithmMetricCollection; }
            set
            {
                if (_algorithmMetricCollection != value)
                {
                    _algorithmMetricCollection = value;
                    NotifyPropertyChanged("AlgorithmMetricCollection");
                }
            }
        }

        //Property for getter and automatic setter (when value changed) for mouse
wheel (as alternative to zoom slider)
        public int MouseWheel
        {
            get { return _mouseWheel; }
            set
            {
                if (_mouseWheel != value)
                {
                    _mouseWheel = value;
                    NotifyPropertyChanged("MouseWheel");
                }
            }
        }

        //Properties for getters and automatic setters (when value changed) for width,
height, wallSize, wallThickness, delay, guidePathData,
        // highLightPathData, mazePathData, RoutePathData, hideHighlightSelected,
HideGridLinesSelected, AlgorithimLabel and RoutePathLineThickness
        public int Width
```

```
        {
            get { return _width; }
            set
            {
                if (_width != value)
                {
                    _width = value;
                    NotifyPropertyChanged("Width");
                }
            }
        }

        public int Height
        {
            get { return _height; }
            set
            {
                if (_height != value)
                {
                    _height = value;
                    NotifyPropertyChanged("Height");
                }
            }
        }

        public int WallSize
        {
            get { return _wallSize; }
            set
            {
                if (_wallSize != value)
                {
                    _wallSize = value;
                    NotifyPropertyChanged("WallSize");
                }
            }
        }

        public int WallThickness
        {
            get { return _wallThickness; }
            set
            {
                if (_wallThickness != value)
                {
                    _wallThickness = value;
                    NotifyPropertyChanged("WallThickness");
                }
            }
        }

        public int Delay
        {
            get { return _delay; }
            set
            {
                if (_delay != value)
                {
                    _delay = value;
                    NotifyPropertyChanged("Delay");
                }
            }
```

```
        }

        public Geometry GuidePathData
        {
            get { return _guidePathData; }
            set
            {
                if (_guidePathData != value)
                {
                    _guidePathData = value;
                    NotifyPropertyChanged("GuidePathData");
                }
            }
        }


        public Geometry HighlightPathData
        {
            get { return _highlightPathData; }
            set
            {
                if (_highlightPathData != value)
                {
                    _highlightPathData = value;
                    NotifyPropertyChanged("HighlightPathData");
                }
            }
        }

        public Geometry MazePathData
        {
            get { return _mazePathData; }
            set
            {
                if (_mazePathData != value)
                {
                    _mazePathData = value;
                    NotifyPropertyChanged("MazePathData");
                }
            }
        }

        public Geometry RoutePathData
        {
            get { return _routePathData; }
            set
            {
                if (_routePathData != value)
                {
                    _routePathData = value;
                    NotifyPropertyChanged("RoutePathData");
                }
            }
        }

        public bool HideHighlightSelected
        {
            get { return _hideHighlightSelected; }
            set
            {
                if (_hideHighlightSelected != value)
                {
```

```
                _hideHighlightSelected = value;
                NotifyPropertyChanged("HideHighlightSelected");
            }
        }
    }

    public bool HideGridLinesSelected
    {
        get { return _hideGridLinesSelected; }
        set
        {
            if (_hideGridLinesSelected != value)
            {
                _hideGridLinesSelected = value;
                NotifyPropertyChanged("HideGridLinesSelected");
            }
        }
    }

    public string AlgorithimLabel
    {
        get { return _algorithimLabel; }
        set
        {
            if (_algorithimLabel != value)
            {
                _algorithimLabel = value;
                NotifyPropertyChanged("AlgorithimLabel");
            }
        }
    }

    public int RoutePathLineThickness
    {
        get { return _routePathLineThickness; }
        set
        {
            if (_routePathLineThickness != value)
            {
                _routePathLineThickness = value;
                NotifyPropertyChanged("RoutePathLineThickness");
            }
        }
    }

    //Procedure tells the specific property variable to update its stored value as
it has been changed
    private void NotifyPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
    }
}
```

ResultsWindowViewModel

This view model class is used specifically for data binding in the ResultsWindowViewModel.

Property	Description
Data	A dictionary of names again elapsed times. The key is the name of an algorithm and the value is the elapsed time (in milliseconds) that the algorithm took to complete.
SourceName	The name of the source of the results.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mazer.ViewModels
{
    public class ResultsWindowViewModel : INotifyPropertyChanged
    {
        //INotifyPropertyChanged is inherited so all the properties in this ViewModel
can be automatically updated when they change,
        //e.g. a new time has been recorded for an algorithm

        public event PropertyChangedEventHandler PropertyChanged;

        private Dictionary<string, long> _data;
        private string _sourceName;

        //Properties for getters and automatic setters (when value changed) for Data
and SourceName
        public Dictionary<string, long> Data
        {
            get { return _data; }
            set
            {
                if (_data != value)
                {
                    _data = value;
                    NotifyPropertyChanged("Data");
                }
            }
        }

        public string SourceName
        {
            get { return _sourceName; }
            set
            {
                if (_sourceName != value)
                {
                    _sourceName = value;
                    NotifyPropertyChanged("SourceName");
                }
            }
        }
```

```
        //Procedure tells the specific property variable to update its stored value as
it has been changed
        private void NotifyPropertyChanged(string propertyName)
        {
            PropertyChangedEventHandler handler = PropertyChanged;
            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(propertyName));
            }
        }
    }
}
```

Model

This model code is used classes that are used by the Mazer application. They are directly related to the UI or the View Model but are used or referenced by them.

AlgorithmMetric

This class is used for holding the metrics about an algorithm as it executed. This result information is required by the Results Data Grid in the UI.

Property	Description	Results Data Grid Column Number
AlgorithmName	Holds the algorithms name.	1
TimeSpan	Holds the time for the algorithm.	2
ElapsedMilliseconds	Holds the timespan converted to milliseconds.	3
RelativeComplexity	Holds the relative complexity of the algorithm.	4

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mazer.Model
{
    public class AlgorithmMetric : INotifyPropertyChanged
    {
        //Class for results histogram window. It requires time an algorithm name and a
complexity.
        //INotifyPropertyChanged is inherited so these properties will update if they
are ever changed in the running of the program
        public event PropertyChangedEventHandler PropertyChanged;

        private TimeSpan _timeSpan = new TimeSpan();
        private string _algorithmName;
        private long _elapsedMilliseconds;
        private double _relativeComplexity;

        //Properties for getters and automatic setters (when value changed) for
timeSpan, algorithmName, elapsedMilliseconds and relativeComplexity
        public TimeSpan TimeSpan
        {
            get { return _timeSpan; }
            set
            {
                if (_timeSpan != value)
                {
                    _timeSpan = value;
                    NotifyPropertyChanged("TimeSpan");
                }
            }
        }

        public string AlgorithmName
        {
            get { return _algorithmName; }
            set
            {
```

```
                if (_algorithmName != value)
                {
                    _algorithmName = value;
                    NotifyPropertyChanged("AlgorithmName");
                }
            }
        }

        public long ElapsedMilliseconds
        {
            get { return _elapsedMilliseconds; }
            set
            {
                if (_elapsedMilliseconds != value)
                {
                    _elapsedMilliseconds = value;
                    NotifyPropertyChanged("ElapsedMilliseconds");
                }
            }
        }

        public double RelativeComplexity
        {
            get { return _relativeComplexity; }
            set
            {
                if (_relativeComplexity != value)
                {
                    _relativeComplexity = value;
                    NotifyPropertyChanged("RelativeComplexity");
                }
            }
        }

        private void NotifyPropertyChanged(string propertyName)
        {
            PropertyChangedEventHandler handler = PropertyChanged;
            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(propertyName));
            }
        }
    }
}
```

Helpers

These helper classes are used to help other parts of the program such as the creation of a path to be converted into a XAML geometry in the UI.

PathHelper

This class is used to create a guide path geometry based on a grid dimensions and wall size. It also helps to convert a maze grid into a geometry for the maze path. It can convert a list of Cells into the highlight and route paths. All these paths are constructed out of strings using a special XAML geometry path syntax. These strings can then be parsed into true XAML geometry objects that are then draw in the UI through data binding.

```
using MazeLibrary;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mazer.Helpers
{
    public static class PathHelper
    {
        //M = "Move or translate", V = "Vertical translation", H = "Horizontal
translation", L = "Line or both vertical and horizontal translations"
        public static string GetGuidePath(int gridWidth, int gridHeight, int wallSize)
        {
            //Creates the grid of dashed red lines over which walls and passageways
are placed
            StringBuilder sb = new StringBuilder();
            for (int x = 0; x <= gridWidth; x++)
            {
                //Vertical Line is started from the x position in the grid X-wallsize
to get it to the correct place in terms of the rest of the grid
                sb.AppendFormat("M{0},{1}V{2}", x * wallSize, 0, gridHeight *
wallSize);
            }

            for (int y = 0; y <= gridHeight; y++)
            {
                //Horizontal Line is started from the y position in the grid Y-
wallsize to get it to the correct place in terms of the rest of the grid
                sb.AppendFormat("M{0},{1}H{2}", 0, y * wallSize, gridWidth *
wallSize);
            }
            return sb.ToString();
        }

        internal static string GetMazeGeneratorPath(MazeLibrary.Grid grid, int
wallSize)
        {
            //Starts forming the maze grid in string geometries where placement is
decided on by x and y and distance to the next cell is wallsize, i.e. next point is px
and py away
            StringBuilder sb = new StringBuilder();
            for (int x = 0; x < grid.Width; x++)
            {
                for (int y = 0; y < grid.Height; y++)
                {
                    MazeLibrary.Cell cell = grid.GetCell(x, y);
                    int px = x * wallSize;
                    int py = y * wallSize;
```

```
                    //If statement required as a direction for a null cell, or a null
edge, cannot be displayed
                    if (cell != null)
                    {

                        //North Wall
                        CellEdge northWall = cell.GetEdge(Direction.North);
                        if (northWall is CellWall)
                        {
                            sb.AppendFormat("M{0},{1}H{2}", px, py, px + wallSize);
                        }

                        //East Wall
                        CellEdge eastWall = cell.GetEdge(Direction.East);
                        if (eastWall is CellWall)
                        {
                            sb.AppendFormat("M{0},{1}V{2}", px + wallSize, py, py +
wallSize);
                        }

                        //South Wall
                        CellEdge southWall = cell.GetEdge(Direction.South);
                        if (southWall is CellWall)
                        {
                            sb.AppendFormat("M{0},{1}H{2}", px, py + wallSize, px +
wallSize);
                        }

                        //West Wall
                        CellEdge westWall = cell.GetEdge(Direction.West);
                        if (westWall is CellWall)
                        {
                            sb.AppendFormat("M{0},{1}V{2}", px, py, py + wallSize);
                        }
                    }
                }
            }
            return sb.ToString();
        }

        internal static string GetMazeSolverPath(MazeLibrary.Grid grid, int wallSize,
List<Cell> route)
        {
            //Converts the routePath into a string geometry
            StringBuilder sb = new StringBuilder();

            for (int i = 0; i < route.Count - 1; i++)
            {

                Cell currentCell = route[i];
                Cell neighbourCell = route[i + 1];

                //currentCell's position
                int currentX = (currentCell.Point.X * wallSize) + wallSize / 2;
                int currentY = (currentCell.Point.Y * wallSize) + wallSize / 2;

                //Distance to neighbourCells position
                int offsetX = (neighbourCell.Point.X - currentCell.Point.X) *
wallSize;
                int offsetY = (neighbourCell.Point.Y - currentCell.Point.Y) *
wallSize;
```

```
                //Addition of the two getting the next position
                int nextX = currentX + offsetX;
                int nextY = currentY + offsetY;

                //Creating this data in a geometry
                string statement = string.Format("M{0},{1}L{2},{3}", currentX,
currentY, nextX, nextY);
                sb.AppendFormat(statement);

            }
            return sb.ToString();
        }


        internal static string GetHighlightCellPath(int wallSize, int wallThickness,
List<Cell> highlightCells)
        {
            StringBuilder sb = new StringBuilder();

            foreach (Cell cell in highlightCells)
            {
                //offset to next cell calculated
                int xOffset = cell.Point.X * wallSize;
                int yOffset = cell.Point.Y * wallSize;
                sb.AppendFormat("M {0},{1} V{2} H{3} V{1} Z",
                    xOffset,
                    yOffset,
                    yOffset + wallSize,
                    xOffset + wallSize);
            }

            return sb.ToString();
        }
    }
}
```

SaveImageHelper

This class is used to convert a canvas into an image file and then to save it to disk. The image file format supported is the Portable Network Graphics (png) format.

```
using Microsoft.Win32;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Media;
using System.Windows.Media.Imaging;

namespace Mazer.Helpers
{
    public static class SaveImageHelper
    {
        public static void SavePngImage(System.Windows.Controls.Canvas canvas, string
folderType)
        {
            SaveFileDialog saveFileDialog = new SaveFileDialog();
            string applicationFolder = "Mazer Save Files";
```

```
            //Gets the path for the myDocuments path for whatever computer this
program is on.
            string myDocuments =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
            //combines these paths
            string applicationPath = System.IO.Path.Combine(myDocuments,
applicationFolder, string.Format("{0} Mazer Images", folderType));
            //Sees if this path already exists
            if (!Directory.Exists(applicationPath))
            {
                Directory.CreateDirectory(applicationPath);
            }
            //All this provides the usual "Save As" function of any common microsoft
application.
            saveFileDialog.Filter = "png (*.png)|*.png|All Files (*.*)|*.*";
            saveFileDialog.FilterIndex = 0;
            saveFileDialog.Title = "Save Mazer File";
            saveFileDialog.InitialDirectory = applicationPath;
            saveFileDialog.RestoreDirectory = true;
            saveFileDialog.CheckFileExists = false;
            saveFileDialog.CheckPathExists = true;
            saveFileDialog.DefaultExt = "gol";

            //dialogResult can be true, false or null (just in case)
            bool? dialogResult = saveFileDialog.ShowDialog();

            //If what the user enters is Save (OK) will save the file.
            if (dialogResult == true)
            {
                string saveFilePath = saveFileDialog.FileName;

                // Save current canvas transform
                Transform transform = canvas.LayoutTransform;
                // reset current transform (in case it is scaled or rotated)
                canvas.LayoutTransform = null;

                // Get the size of canvas
                Size size = new Size(canvas.ActualWidth, canvas.ActualHeight);
                // Measure and arrange the canvas
                // VERY IMPORTANT
                canvas.Measure(size);
                canvas.Arrange(new Rect(size));

                // Create a render bitmap and push the canvas to it
                RenderTargetBitmap renderBitmap =
                  new RenderTargetBitmap(
                    (int)size.Width,
                    (int)size.Height,
                    96d,
                    96d,
                    PixelFormats.Pbgra32);
                renderBitmap.Render(canvas);

                string currentPath = System.IO.Path.Combine(myDocuments,
applicationFolder, string.Format("{0} Mazer Images", folderType), saveFilePath);

                // Create a file stream for saving image
                using (FileStream outStream = new FileStream(currentPath,
FileMode.Create))
                {
                    // Use png encoder for our data
                    PngBitmapEncoder encoder = new PngBitmapEncoder();
```

```
                    // push the rendered bitmap to it
                    encoder.Frames.Add(BitmapFrame.Create(renderBitmap));
                    // save the data to the stream
                    encoder.Save(outStream);
                }

                // Restore previously saved layout
                canvas.LayoutTransform = transform;
            }
        }
    }
}
```

Enums

The enums hold all the enumerable types that I use in the Mazer application.

MazeAlgorithm

This enumerable type is used to hold all the algorithm names. This provides clarity in the source code when referencing algorithms.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mazer.Enums
{
    //Enum class so I can reference a keyword instead of a number when implementing
the algorithms
    public enum MazeAlgorithm
    {
        AStar,
        BinaryTree,
        DrunkardsWalk,
        HuntAndKill,
        Kruskals,
        Prims,
        RecursiveBackTracker,
        RecursiveDivision,
        Sidewinder,
        SimpleRectangle,
        WallFollower,
        Wilsons,
    }
}
```

Converters

Converters are used by the XAML data binding code in the UI to convert values of one type to values of a different type or to change one input value proportionally, no matter what value goes in, to produce a new valid value.

HalfValueConverter

This class is used to centre the gridlines to be shown in the centre of the canvas. All other geometries such as the maze path geometry use this to also centre themselves enabling them all to line up when zoom is being used.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Data;

namespace Mazer.Converters
{
    public class HalfValueConverter : IMultiValueConverter
    {
        //The class here is used to always centre the gridline in the centre of the
canvas.
        //In the WPF code, by giving it the canvas' width or height and the grid's
width or height it calculates
        //the intent required to move the grid to the centre of the screen, no matter
how the window is moved or resized
        public object Convert(object[] values, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
        {
            //Only centers it if the values are not null, there are two of them and
they are set to any value
            //other than the default value for the DependencyProperty
            if (values != null && values.Length == 2 && values[0] !=
DependencyProperty.UnsetValue && values[1] != DependencyProperty.UnsetValue)
            {
                double canvasLength = (double)values[0];
                double mazeLength = (double)values[1];

                //Here it returns the new position indent, calculated the same for the
horizontal or vertical plane
                return (canvasLength - mazeLength) / 2;
            }

            //returns the default value for the DependencyProperty should the above if
statement be false,
            //I will then see there is a problem and can correct it
            return DependencyProperty.UnsetValue;
        }

        //ConvertBack function required for the use of IMultiValueConverter. There is
no need to use it in this
        //program though so it will be unused
        public object[] ConvertBack(object value, Type[] targetTypes, object
parameter, System.Globalization.CultureInfo culture)
        {
            throw new NotImplementedException();
        }
    }
}
```

RectConverter

This converter is used to change the size of the canvas into a new rectangle which is returned. This is used to allow the canvas to be stretched underneath the other controls on the window. When zoom is in use on the canvas the outside border of the canvas disappears, this is due to it being pushed out from the centre of the canvas. This, however, works well as there is now a visual cue to see if you are currently zoomed in or not.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Data;

namespace Mazer.Converters
{
    public class RectConverter : IMultiValueConverter
    {
        //The class here is used to force correct clipping to the canvas when it is
being zoomed,
        //if this rectangle for the border wasn't being returned on every change to
the canvas size,
        //it would stretch the canvas over the other controls in the window
        public object Convert(object[] values, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
        {
            //Only returns a rectangle if the values are not null, there are two of
them and they are set to any value
            //other than the default value for the DependencyProperty
            if (values != null && values.Length == 2 && values[0] !=
DependencyProperty.UnsetValue && values[1] != DependencyProperty.UnsetValue)
            {
                double width = (double)values[0];
                double height = (double)values[1];
                //Returns the new dimensions
                return new Rect(0, 0, width, height);
            }

            //If the above failed, returns the default value for the
DependencyProperty so
            //I will see there is a problem and then I will resolve it
            return DependencyProperty.UnsetValue;
        }

        //Again a ConvertBack function required for the use of IMultiValueConverter.
There is no need to use it in this
        //program though so it will be unused
        public object[] ConvertBack(object value, Type[] targetTypes, object
parameter, System.Globalization.CultureInfo culture)
        {
            throw new NotImplementedException();
        }
    }
}
```

MazeLibraryUnitTests

This section details the code used in all of the unit tests written for the Testing section and follows the planned Test Strategy that I described in my Design. You will notice that each test is small and tests only one feature of the solution. In addition, the actual test name itself describes what is actually being tested. I have done this in order to follow the best practices and guidelines for unit testing that I have read about during my research.

The reason why the unit testing approach was adopted is explained in the Testing section that follows.

AlgorithmUnitTests

These unit tests are used to test individual algorithms.

AStarAlgorithmUnitTest

```
using MazeLibrary;
using MazeLibrary.Algorithms;
using MazeLibraryUnitTests.Helpers;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.IO;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class AStarAlgorithmUnitTest : AlgorithmTestBase
    {
        private AStarAlgorithm _aStarAlgorithm = new AStarAlgorithm(0);

        [TestMethod]
        public void CheckRouteEqualsActualRouteGrid1()
        {
            Grid grid = GridFactory.GetGrid1();

            List<Cell> route = new List<Cell>();
            List<Cell> visitedCells = new List<Cell>();
            _aStarAlgorithm.Solve(ref grid, out route, out visitedCells);
            route.Reverse();

            CheckAllInRouteAreSameGrid1(route);
        }

        [TestMethod]
        public void CheckRouteEqualsActualRouteGrid2()
        {
            Grid grid = GridFactory.GetGrid2();

            List<Cell> route = new List<Cell>();
            List<Cell> visitedCells = new List<Cell>();
            _aStarAlgorithm.Solve(ref grid, out route, out visitedCells);
            route.Reverse();

            CheckAllInRouteAreSameGrid2(route);
        }
    }
}
```

BinaryTreeAlgorithmUnitTest

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
```

```
using MazeLibrary.Algorithms;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class BinaryTreeAlgorithmUnitTest : AlgorithmTestBase
    {
        private BinaryTreeAlgorithm _binaryTreeAlgorithm = new BinaryTreeAlgorithm(0);

        [TestMethod]
        public void GridHasCorrectNumberOfCells()
        {
            _binaryTreeAlgorithm.Generate(_smallGrid);
            Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);
        }

        [TestMethod]
        public void AllCellsExist()
        {
            _binaryTreeAlgorithm.Generate(_smallGrid);
            CheckAllCellsExist(_smallGrid);
        }

        [TestMethod]
        public void AllEdgesAreFullyIntialized()
        {
            _binaryTreeAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveFullyInitializedEdges(_smallGrid);
        }

        [TestMethod]
        public void AllBordersCreated()
        {
            _binaryTreeAlgorithm.Generate(_smallGrid);
            CheckAllBordersExist(_smallGrid);
        }

        [TestMethod]
        public void AllCellsHaveAtLeastOnePassageWay()
        {
            _binaryTreeAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);
        }
    }
}
```

DrunkardsWalkAlgorithmUnitTest

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary.Algorithms;
using MazeLibrary;
using MazeLibraryUnitTests.Helpers;
using System.Collections.Generic;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class DrunkardsWalkAlgorithmUnitTest : AlgorithmTestBase
    {
        private DrunkardsWalkAlgorithm _drunkardsWalkAlgorithm = new
DrunkardsWalkAlgorithm(0);
```

```
        [TestMethod]
        public void CheckRouteEqualsActualRouteGrid1()
        {
            Grid grid = GridFactory.GetGrid1();

            List<Cell> route = new List<Cell>();
            List<Cell> visitedCells = new List<Cell>();
            _drunkardsWalkAlgorithm.Solve(ref grid, out route, out visitedCells);


            CheckAllInRouteAreSameGrid1(route);
        }

        [TestMethod]
        public void CheckRouteEqualsActualRouteGrid2()
        {
            Grid grid = GridFactory.GetGrid2();

            List<Cell> route = new List<Cell>();
            List<Cell> visitedCells = new List<Cell>();
            _drunkardsWalkAlgorithm.Solve(ref grid, out route, out visitedCells);


            CheckAllInRouteAreSameGrid2(route);
        }

        [TestMethod]
        public void GridHasCorrectNumberOfCells()
        {
            _drunkardsWalkAlgorithm.Generate(_smallGrid);
            Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);
        }

        [TestMethod]
        public void AllCellsExist()
        {
            _drunkardsWalkAlgorithm.Generate(_smallGrid);
            CheckAllCellsExist(_smallGrid);
        }

        [TestMethod]
        public void AllEdgesAreFullyIntialized()
        {
            _drunkardsWalkAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveFullyInitializedEdges(_smallGrid);
        }

        [TestMethod]
        public void AllBordersCreated()
        {
            _drunkardsWalkAlgorithm.Generate(_smallGrid);
            CheckAllBordersExist(_smallGrid);
        }

        [TestMethod]
        public void AllCellsHaveAtLeastOnePassageWay()
        {
            _drunkardsWalkAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);
        }
    }
```

```
}
```

HuntAndKillAlgorithmUnitTest

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary.Algorithms;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class HuntAndKillAlgorithmUnitTest : AlgorithmTestBase
    {
        private HuntAndKillAlgorithm _huntAndKillAlgorithm = new
HuntAndKillAlgorithm(0);

        [TestMethod]
        public void GridHasCorrectNumberOfCells()
        {
            _huntAndKillAlgorithm.Generate(_smallGrid);
            Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);
        }

        [TestMethod]
        public void AllCellsExist()
        {
            _huntAndKillAlgorithm.Generate(_smallGrid);
            CheckAllCellsExist(_smallGrid);
        }

        [TestMethod]
        public void AllEdgesAreFullyIntialized()
        {
            _huntAndKillAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveFullyInitializedEdges(_smallGrid);
        }

        [TestMethod]
        public void AllBordersCreated()
        {
            _huntAndKillAlgorithm.Generate(_smallGrid);
            CheckAllBordersExist(_smallGrid);
        }

        [TestMethod]
        public void AllCellsHaveAtLeastOnePassageWay()
        {
            _huntAndKillAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);
        }
    }
}
```

KruskalsAlgorithmUnitTest

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary.Algorithms;

namespace MazeLibraryUnitTests
{
    [TestClass]
```

```
    public class KruskalsAlgorithmUnitTest : AlgorithmTestBase
    {
        private KruskalsAlgorithm _kruskalsAlgorithm = new KruskalsAlgorithm(0);

        [TestMethod]
        public void GridHasCorrectNumberOfCells()
        {
            _kruskalsAlgorithm.Generate(_smallGrid);
            Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);
        }

        [TestMethod]
        public void AllCellsExist()
        {
            _kruskalsAlgorithm.Generate(_smallGrid);
            CheckAllCellsExist(_smallGrid);
        }

        [TestMethod]
        public void AllEdgesAreFullyIntialized()
        {
            _kruskalsAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveFullyInitializedEdges(_smallGrid);
        }

        [TestMethod]
        public void AllBordersCreated()
        {
            _kruskalsAlgorithm.Generate(_smallGrid);
            CheckAllBordersExist(_smallGrid);
        }

        [TestMethod]
        public void AllCellsHaveAtLeastOnePassageWay()
        {
            _kruskalsAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);
        }
    }
}
```

PrimsAlgorithmUnitTest

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary.Algorithms;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class PrimsAlgorithmUnitTest : AlgorithmTestBase
    {
        private PrimsAlgorithm _primsAlgorithm = new PrimsAlgorithm(0);

        [TestMethod]
        public void GridHasCorrectNumberOfCells()
        {
            _primsAlgorithm.Generate(_smallGrid);
            Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);
        }

        [TestMethod]
```

```
        public void AllCellsExist()
        {
            _primsAlgorithm.Generate(_smallGrid);
            CheckAllCellsExist(_smallGrid);
        }

        [TestMethod]
        public void AllEdgesAreFullyIntialized()
        {
            _primsAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveFullyInitializedEdges(_smallGrid);
        }

        [TestMethod]
        public void AllBordersCreated()
        {
            _primsAlgorithm.Generate(_smallGrid);
            CheckAllBordersExist(_smallGrid);
        }

        [TestMethod]
        public void AllCellsHaveAtLeastOnePassageWay()
        {
            _primsAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);
        }
    }
}
```

RecursiveBackTrackerAlgorithmUnitTest

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary.Algorithms;
using MazeLibrary;
using MazeLibraryUnitTests.Helpers;
using System.Collections.Generic;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class RecursiveBackTrackerAlgorithmUnitTest : AlgorithmTestBase
    {
        private RecursiveBackTrackerAlgorithm _recursiveBackTrackerAlgorithm = new
RecursiveBackTrackerAlgorithm(0);

        [TestMethod]
        public void CheckRouteEqualsActualRouteGrid1()
        {
            Grid grid = GridFactory.GetGrid1();

            List<Cell> route = new List<Cell>();
            List<Cell> visitedCells = new List<Cell>();
            _recursiveBackTrackerAlgorithm.Solve(ref grid, out route, out
visitedCells);


            CheckAllInRouteAreSameGrid1(route);
        }

        [TestMethod]
        public void CheckRouteEqualsActualRouteGrid2()
```

```
        {
            Grid grid = GridFactory.GetGrid2();

            List<Cell> route = new List<Cell>();
            List<Cell> visitedCells = new List<Cell>();
            _recursiveBackTrackerAlgorithm.Solve(ref grid, out route, out
visitedCells);


            CheckAllInRouteAreSameGrid2(route);
        }

        [TestMethod]
        public void GridHasCorrectNumberOfCells()
        {
            _recursiveBackTrackerAlgorithm.Generate(_smallGrid);
            Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);
        }

        [TestMethod]
        public void AllCellsExist()
        {
            _recursiveBackTrackerAlgorithm.Generate(_smallGrid);
            CheckAllCellsExist(_smallGrid);
        }

        [TestMethod]
        public void AllEdgesAreFullyIntialized()
        {
            _recursiveBackTrackerAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveFullyInitializedEdges(_smallGrid);
        }

        [TestMethod]
        public void AllBordersCreated()
        {
            _recursiveBackTrackerAlgorithm.Generate(_smallGrid);
            CheckAllBordersExist(_smallGrid);
        }

        [TestMethod]
        public void AllCellsHaveAtLeastOnePassageWay()
        {
            _recursiveBackTrackerAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);
        }
    }
}
```

RecursiveDivisionAlgorithmUnitTest

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary.Algorithms;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class RecursiveDivisionAlgorithmUnitTest : AlgorithmTestBase
    {
        private RecursiveDivisionAlgorithm _recursiveDivisionAlgorithm = new
RecursiveDivisionAlgorithm(0);
```

```
        [TestMethod]
        public void GridHasCorrectNumberOfCells()
        {
            _recursiveDivisionAlgorithm.Generate(_smallGrid);
            Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);
        }

        [TestMethod]
        public void AllCellsExist()
        {
            _recursiveDivisionAlgorithm.Generate(_smallGrid);
            CheckAllCellsExist(_smallGrid);
        }

        [TestMethod]
        public void AllEdgesAreFullyIntialized()
        {
            _recursiveDivisionAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveFullyInitializedEdges(_smallGrid);
        }

        [TestMethod]
        public void AllBordersCreated()
        {
            _recursiveDivisionAlgorithm.Generate(_smallGrid);
            CheckAllBordersExist(_smallGrid);
        }

        [TestMethod]
        public void AllCellsHaveAtLeastOnePassageWay()
        {
            _recursiveDivisionAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);
        }
    }
}
```

SidewinderAlgorithmUnitTest

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary.Algorithms;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class SidewinderAlgorithmUnitTest : AlgorithmTestBase
    {
        private SidewinderAlgorithm _sidewinderAlgorithm = new SidewinderAlgorithm(0);

        [TestMethod]
        public void GridHasCorrectNumberOfCells()
        {
            _sidewinderAlgorithm.Generate(_smallGrid);
            Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);
        }

        [TestMethod]
        public void AllCellsExist()
        {
            _sidewinderAlgorithm.Generate(_smallGrid);
```

```
            CheckAllCellsExist(_smallGrid);
        }

        [TestMethod]
        public void AllEdgesAreFullyIntialized()
        {
            _sidewinderAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveFullyInitializedEdges(_smallGrid);
        }

        [TestMethod]
        public void AllBordersCreated()
        {
            _sidewinderAlgorithm.Generate(_smallGrid);
            CheckAllBordersExist(_smallGrid);
        }

        [TestMethod]
        public void AllCellsHaveAtLeastOnePassageWay()
        {
            _sidewinderAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);
        }
    }
}
```

SimpleRectangleAlgorithmUnitTest

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary.Algorithms;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class SimpleRectangleAlgorithmUnitTest : AlgorithmTestBase
    {
        private SimpleRectangleAlgorithm _simpleRectangleAlgorithm = new
SimpleRectangleAlgorithm(0);

        [TestMethod]
        public void GridHasCorrectNumberOfCells()
        {
            _simpleRectangleAlgorithm.Generate(_smallGrid);
            Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);
        }

        [TestMethod]
        public void AllCellsExist()
        {
            _simpleRectangleAlgorithm.Generate(_smallGrid);
            CheckAllCellsExist(_smallGrid);
        }

        [TestMethod]
        public void AllEdgesAreFullyIntialized()
        {
            _simpleRectangleAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveFullyInitializedEdges(_smallGrid);
        }

        [TestMethod]
```

```
        public void AllBordersCreated()
        {
            _simpleRectangleAlgorithm.Generate(_smallGrid);
            CheckAllBordersExist(_smallGrid);
        }

        [TestMethod]
        public void AllCellsHaveAtLeastOnePassageWay()
        {
            _simpleRectangleAlgorithm.Generate(_smallGrid);
            CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);
        }
    }
}
```

WallFollowerAlgorithmSolveLeftUnitTest

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary.Algorithms;
using MazeLibrary;
using System.Collections.Generic;
using MazeLibraryUnitTests.Helpers;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class WallFollowerAlgorithmSolveLeftUnitTest : AlgorithmTestBase
    {
        private WallFollowerAlgorithm _wallFollowerAlgorithm = new
WallFollowerAlgorithm(0);

        [TestMethod]
        public void CheckRouteEqualsActualRouteGrid1()
        {
            Grid grid = GridFactory.GetGrid1();

            List<Cell> route = new List<Cell>();
            List<Cell> visitedCells = new List<Cell>();
            _wallFollowerAlgorithm.SolveLeft(ref grid, out route, out visitedCells);


            CheckAllInRouteAreSameGrid1(route);
        }

        [TestMethod]
        public void CheckRouteEqualsActualRouteGrid2()
        {
            Grid grid = GridFactory.GetGrid2();

            List<Cell> route = new List<Cell>();
            List<Cell> visitedCells = new List<Cell>();
            _wallFollowerAlgorithm.SolveLeft(ref grid, out route, out visitedCells);


            CheckAllInRouteAreSameGrid2(route);
        }
    }
}
```

WallFollowerAlgorithmSolveRightUnitTest

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary.Algorithms;
using MazeLibrary;
using System.Collections.Generic;
using MazeLibraryUnitTests.Helpers;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class WallFollowerAlgorithmSolveRightUnitTest : AlgorithmTestBase
    {
        private WallFollowerAlgorithm _wallFollowerAlgorithm = new
WallFollowerAlgorithm(0);

        [TestMethod]
        public void CheckRouteEqualsActualRouteGrid1()
        {
            Grid grid = GridFactory.GetGrid1();

            List<Cell> route = new List<Cell>();
            List<Cell> visitedCells = new List<Cell>();
            _wallFollowerAlgorithm.SolveRight(ref grid, out route, out visitedCells);


            CheckAllInRouteAreSameGrid1(route);
        }

        [TestMethod]
        public void CheckRouteEqualsActualRouteGrid2()
        {
            Grid grid = GridFactory.GetGrid2();

            List<Cell> route = new List<Cell>();
            List<Cell> visitedCells = new List<Cell>();
            _wallFollowerAlgorithm.SolveRight(ref grid, out route, out visitedCells);


            CheckAllInRouteAreSameGrid2(route);
        }
    }
}
```

WilsonsAlgorithmUnitTest

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary.Algorithms;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class WilsonsAlgorithmUnitTest : AlgorithmTestBase
    {
        private WilsonsAlgorithm _wilsonsAlgorithm = new WilsonsAlgorithm(0);
    }
}
```

CoreUnitTests

These unit tests are used to test the classes in the Core of the Mazer program.

CellEdgeUnitTest

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary;
using System.Drawing;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class CellEdgeUnitTest : AlgorithmTestBase
    {
        [TestMethod]
        public void CellEdgeHasCorrectDirection()
        {
            Cell cell = _smallGrid.CreateCell(0, 0);
            Cell neighbourCell;
            CellEdge cellEdge;

            _smallGrid.CreateWall(cell, null, Direction.North);
            cellEdge = cell.GetEdge(Direction.North);
            Assert.IsTrue(cellEdge.Direction == Direction.North);

            neighbourCell = _smallGrid.CreateCell(1, 0);
            _smallGrid.CreatePassage(cell, neighbourCell, Direction.East);
            cellEdge = cell.GetEdge(Direction.East);
            Assert.IsTrue(cellEdge.Direction == Direction.East);

            neighbourCell = _smallGrid.CreateCell(0, 1);
            _smallGrid.CreateWall(cell, neighbourCell, Direction.South);
            cellEdge = cell.GetEdge(Direction.South);
            Assert.IsTrue(cellEdge.Direction == Direction.South);
        }

        [TestMethod]
        public void CellEdgeHasCorrectStartCell()
        {
            Cell cell = _smallGrid.CreateCell(0, 0);
            Cell neighbourCell;
            CellEdge cellEdge;

            _smallGrid.CreateWall(cell, null, Direction.North);
            cellEdge = cell.GetEdge(Direction.North);
            Assert.IsTrue(cellEdge.StartCell == cell);

            neighbourCell = _smallGrid.CreateCell(1, 0);
            _smallGrid.CreatePassage(cell, neighbourCell, Direction.East);
            cellEdge = cell.GetEdge(Direction.East);
            Assert.IsTrue(cellEdge.StartCell == cell);

            neighbourCell = _smallGrid.CreateCell(0, 1);
            _smallGrid.CreateWall(cell, neighbourCell, Direction.South);
            cellEdge = cell.GetEdge(Direction.South);
            Assert.IsTrue(cellEdge.StartCell == cell);

        }

        [TestMethod]
        public void CellEdgeHasCorrectEndCell()
        {
```

```
            Cell cell = _smallGrid.CreateCell(0, 0);
            Cell neighbourCell;
            CellEdge cellEdge;

            _smallGrid.CreateWall(cell, null, Direction.North);
            cellEdge = cell.GetEdge(Direction.North);
            Assert.IsTrue(cellEdge.EndCell == null);

            neighbourCell = _smallGrid.CreateCell(1, 0);
            _smallGrid.CreatePassage(cell, neighbourCell, Direction.East);
            cellEdge = cell.GetEdge(Direction.East);
            Assert.IsTrue(cellEdge.EndCell == neighbourCell);

            neighbourCell = _smallGrid.CreateCell(0, 1);
            _smallGrid.CreateWall(cell, neighbourCell, Direction.South);
            cellEdge = cell.GetEdge(Direction.South);
            Assert.IsTrue(cellEdge.EndCell == neighbourCell);
        }
    }
}
```

CellUnitTests

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary;
using System.Drawing;
using MazeLibrary.Helpers;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class CellUnitTests : AlgorithmTestBase
    {
        [TestMethod]
        public void CellNameIsCorrect()
        {
            Cell cell = new Cell("Cell_New", 0, 0);
            Assert.AreSame("Cell_New", cell.Name);
        }

        [TestMethod]
        public void CellIntialisedEdgeCountIsCorrect()
        {
            Cell cell = _tinyGrid.CreateCell(0, 0);

            Assert.IsTrue(cell.InitialisedEdgeCount == 0);

            for (int i = 0; i < DirectionHelper.Count; i++)
            {
                Assert.IsTrue(cell.InitialisedEdgeCount == i);
                _tinyGrid.CreateWall(cell, null, (Direction)i);
                Assert.IsTrue(cell.InitialisedEdgeCount == i + 1);
            }
        }

        [TestMethod]
        public void CellIsFullyIntialisedIsCorrect()
        {
            Cell cell = _tinyGrid.CreateCell(0, 0);

            for (int i = 0; i < DirectionHelper.Count; i++)
```

```
            {
                _tinyGrid.CreateWall(cell, null, (Direction)i);
            }
            Assert.IsTrue(cell.IsFullyInitialized);
        }

        [TestMethod]
        public void CellGetEdgeReturnsCorrectEdge()
        {
            _tinyGrid.CreateCell(0, 0);
            Cell cell = _tinyGrid.GetCell(0, 0);

            _tinyGrid.CreateWall(cell, null, Direction.North);

            Assert.IsTrue(cell.GetEdge(Direction.North) is CellWall);
        }

        [TestMethod]
        public void CellSetsEdgeCorrectly()
        {
            _smallGrid.CreateCell(0, 0);
            Cell cell = _smallGrid.GetCell(0, 0);
            CellPassage cp = new CellPassage();

            cell.SetEdge(Direction.North, cp);

            Assert.IsTrue(cell.GetEdge(Direction.North) is CellPassage);
        }

        [TestMethod]
        public void CellGetsRandomUninitializedDirectionCorrectly()
        {
            _smallGrid.CreateCell(0, 0);
            Cell cell = _smallGrid.GetCell(0, 0);

            for (int i = 0; i < DirectionHelper.Count; i++)
            {
                Direction direction = cell.GetRandomUninitializedDirection();
                Assert.IsTrue(cell.GetEdge(direction) == null);
                _smallGrid.CreateWall(cell, null, (Direction)i);
            }
        }
    }
}
```

DirectionHelperUnitTests

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary;
using System.Drawing;
using MazeLibrary.Helpers;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class DirectionHelperUnitTests : AlgorithmTestBase
    {
        [TestMethod]
        public void DirectionHelperCountReturnsIntegerFour()
        {
            Assert.AreEqual(4, DirectionHelper.Count);
```

```
        }

        [TestMethod]
        public void DirectionHelperReturnsCorrectOffset()
        {
            //The startCell is at: (0, 0)
            Point offset;
            Point point;

            offset = DirectionHelper.GetOffset(Direction.North);
            point = new Point(offset.X, offset.Y);
            Assert.IsTrue(new Point(0, -1) == point);

            offset = DirectionHelper.GetOffset(Direction.East);
            point = new Point(offset.X, offset.Y);
            Assert.IsTrue(new Point(1, 0) == point);

            offset = DirectionHelper.GetOffset(Direction.South);
            point = new Point(offset.X, offset.Y);
            Assert.IsTrue(new Point(0, 1) == point);

            offset = DirectionHelper.GetOffset(Direction.West);
            point = new Point(offset.X, offset.Y);
            Assert.IsTrue(new Point(-1, 0) == point);
        }

        [TestMethod]
        public void DirectionHelperReturnsOppositeDirection()
        {
            Assert.IsTrue(Direction.South ==
DirectionHelper.GetOpposite(Direction.North));
            Assert.IsTrue(Direction.North ==
DirectionHelper.GetOpposite(Direction.South));
            Assert.IsTrue(Direction.East ==
DirectionHelper.GetOpposite(Direction.West));
            Assert.IsTrue(Direction.West ==
DirectionHelper.GetOpposite(Direction.East));
        }
    }
}
```

GridUnitTests

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MazeLibrary;
using System.Drawing;

namespace MazeLibraryUnitTests
{
    [TestClass]
    public class GridUnitTests : AlgorithmTestBase
    {
        [TestMethod]
        public void AllGridsHaveCorrectNames()
        {
            Assert.AreSame("tinyGrid", _tinyGrid.Name);
            Assert.AreSame("smallGrid", _smallGrid.Name);
            Assert.AreSame("mediumGrid", _mediumGrid.Name);
            Assert.AreSame("largeGrid", _largeGrid.Name);
            Assert.AreSame("veryLargeGrid", _veryLargeGrid.Name);
            Assert.AreSame("extremelyLargeGrid", _extremelyLargeGrid.Name);
```

```
        }

        [TestMethod]
        public void AllGridsHaveCorrectWidths()
        {
            Assert.IsTrue(1 == _tinyGrid.Width);
            Assert.IsTrue(5 == _smallGrid.Width);
            Assert.IsTrue(11 == _mediumGrid.Width);
            Assert.IsTrue(31 == _largeGrid.Width);
            Assert.IsTrue(102 == _veryLargeGrid.Width);
            Assert.IsTrue(210 == _extremelyLargeGrid.Width);
        }

        [TestMethod]
        public void AllGridsHaveCorrectHeights()
        {
            Assert.IsTrue(1 == _tinyGrid.Height);
            Assert.IsTrue(6 == _smallGrid.Height);
            Assert.IsTrue(13 == _mediumGrid.Height);
            Assert.IsTrue(54 == _largeGrid.Height);
            Assert.IsTrue(74 == _veryLargeGrid.Height);
            Assert.IsTrue(190 == _extremelyLargeGrid.Height);
        }

        [TestMethod]
        public void GridCreatesNonNullCell()
        {
            Cell cell = _smallGrid.CreateCell(0, 0);
            Assert.IsNotNull(cell);
        }

        [TestMethod]
        public void GridContainsOnGridPoint()
        {
            int x = 0;
            int y = 0;

            Assert.IsTrue(_smallGrid.ContainsPoint(x, y));
        }

        [TestMethod]
        public void GridContainsOffGridPoint()
        {
            int x = -1;
            int y = -1;

            Assert.IsFalse(_smallGrid.ContainsPoint(x, y));
        }

        [TestMethod]
        public void GridCreatesPassage()
        {
            Cell cell1 = _smallGrid.CreateCell(0, 0);
            Cell cell2 = _smallGrid.CreateCell(1, 0);

            _smallGrid.CreatePassage(cell1, cell2, Direction.East);

            Assert.IsTrue(cell1.GetEdge(Direction.East) is CellPassage);
            Assert.IsTrue(cell2.GetEdge(Direction.West) is CellPassage);

            Assert.IsFalse(cell1.GetEdge(Direction.East) is CellWall);
            Assert.IsFalse(cell2.GetEdge(Direction.West) is CellWall);
```

```
        }

        [TestMethod]
        public void GridCreatesWall()
        {
            Cell cell1 = _smallGrid.CreateCell(0, 0);
            Cell cell2 = _smallGrid.CreateCell(1, 0);

            _smallGrid.CreateWall(cell1, cell2, Direction.East);

            Assert.IsTrue(cell1.GetEdge(Direction.East) is CellWall);
            Assert.IsTrue(cell2.GetEdge(Direction.West) is CellWall);

            Assert.IsFalse(cell1.GetEdge(Direction.East) is CellPassage);
            Assert.IsFalse(cell2.GetEdge(Direction.West) is CellPassage);
        }

        [TestMethod]
        public void GridGetsNonNullCell()
        {
            _smallGrid.CreateCell(0, 0);

            Cell cell = _smallGrid.GetCell(0,0);
            Assert.IsNotNull(cell);
        }
    }
}
```

Helpers

These various helper classes all provide test variables that have known values and can be checked against in the unit testing of the Algorithm and Core parts of the Mazer application.

AlgorithmTestBase

A class for supplying standard known grids that can be used for comparing against the results of running the maze generation and solver algorithm tests.

```
using MazeLibrary;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Drawing;
using MazeLibraryUnitTests.Helpers;
using MazeLibrary.Helpers;

namespace MazeLibraryUnitTests
{
    public class AlgorithmTestBase
    {
        protected Grid _tinyGrid = new Grid("tinyGrid", 1, 1);
        protected Grid _smallGrid = new Grid("smallGrid", 5, 6);
        protected Grid _mediumGrid = new Grid("mediumGrid", 11, 13);
        protected Grid _largeGrid = new Grid("largeGrid", 31, 54);
        protected Grid _veryLargeGrid = new Grid("veryLargeGrid", 102, 74);
        protected Grid _extremelyLargeGrid = new Grid("extremelyLargeGrid", 210, 190);

        public void CheckAllInRouteAreSameGrid1(List<Cell> route)
        {
            List<Point> routePoints = new List<Point>();
            foreach (Cell c in route)
            {
                routePoints.Add(new Point(c.Point.X, c.Point.Y));
            }

            List<Point> actualRoutePoints = GridFactory.ActualRouteGrid1();

            for (int i = 0; i < route.Count - 1; i++)
            {
                Assert.IsTrue(actualRoutePoints[i] == routePoints[i]);
            }
        }

        public void CheckAllInRouteAreSameGrid2(List<Cell> route)
        {
            List<Point> routePoints = new List<Point>();
            foreach (Cell c in route)
            {
                routePoints.Add(new Point(c.Point.X, c.Point.Y));
            }

            List<Point> actualRoutePoints = GridFactory.ActualRouteGrid2();

            for (int i = 0; i < route.Count - 1; i++)
            {
                Assert.IsTrue(actualRoutePoints[i] == routePoints[i]);
            }
        }
```

```
        public void CheckAllCellsHaveAtLeastOnePassageWay(Grid g)
        {
            for (int y = 0; y < g.Height; y++)
            {
                for (int x = 0; x < g.Width; x++)
                {
                    Cell cell = g.GetCell(x, y);
                    int count = 0;
                    for (int i = 0; i < DirectionHelper.Count; i++)
                    {
                        CellEdge cellEdge = cell.GetEdge((Direction)i);
                        if (cellEdge is CellPassage) count++;
                    }
                    Assert.IsTrue(count >= 1);
                }
            }
        }

        public void CheckAllCellsExist(Grid g)
        {
            for (int y = 0; y < g.Height; y++)
            {
                for (int x = 0; x < g.Width; x++)
                {
                    Assert.IsNotNull(g.GetCell(x, y));
                }
            }
        }

        public void CheckAllCellsHaveFullyInitializedEdges(Grid g)
        {
            for (int y = 0; y < g.Height; y++)
            {
                for (int x = 0; x < g.Width; x++)
                {
                    Cell cell = g.GetCell(x, y);
                    bool b = cell.IsFullyInitialized;
                    Assert.IsTrue(cell.IsFullyInitialized == true);
                }
            }
        }

        public void CheckAllBordersExist(Grid g)
        {
            for (int y = 0; y < g.Height; y++)
            {
                for (int x = 0; x < g.Width; x++)
                {
                    Cell currentCell = g.GetCell(x, y);
                    if (y == 0)
                    {
                        CellEdge ce = g.GetCell(x, y).GetEdge(Direction.North);
                        Assert.IsTrue(g.GetCell(x, y).GetEdge(Direction.North) is
CellWall);
                    }

                    if (y == g.Height - 1)
                    {
                        Assert.IsTrue(g.GetCell(x, y).GetEdge(Direction.South) is
CellWall);
                    }
```

```
                    if (x == 0)
                    {
                        Assert.IsTrue(g.GetCell(x, y).GetEdge(Direction.West) is
CellWall);
                    }

                    if (x == g.Width - 1)
                    {
                        Assert.IsTrue(g.GetCell(x, y).GetEdge(Direction.East) is
CellWall);
                    }
                }
            }
        }
    }
}
```

GridFactory

A class for comparing solved maze routes against actual known routes through a maze.

```
using MazeLibrary;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;

namespace MazeLibraryUnitTests.Helpers
{
    public static class GridFactory
    {

        public static Grid GetGrid1()
        {
            string filePath =
Path.Combine(Path.Combine(AppDomain.CurrentDomain.BaseDirectory, @"Data Files\Test
Maze 17X13 (RecursiveBacktracker).maz"));
            Grid grid = MazeLibrary.Grid.GetGridFromFile(filePath);
            return grid;
        }

        public static List<Point> ActualRouteGrid1()
        {
            List<Point> actualRoute = new List<Point>(new Point[]
            {
                new Point(0,0), new Point(1,0), new Point(2,0), new Point(2,1), new
Point(1,1), new Point(0,1), new Point(0,2), new Point(1,2), new Point(1,3),
                new Point(2,3), new Point(2,4), new Point(3,4), new Point(4,4), new
Point(4,5), new Point(4,6), new Point(5,6), new Point(6,6), new Point(7,6),
                new Point(7,7), new Point(7,8), new Point(6,8), new Point(6,7), new
Point(5,7), new Point(4,7), new Point(3,7), new Point(3,8), new Point(2,8),
                new Point(1,8), new Point(1,7), new Point(0,7), new Point(0,8), new
Point(0,9), new Point(0,10), new Point(0,11), new Point(0,12), new Point(1,12),
                new Point(1,11), new Point(2,11), new Point(3,11), new Point(3,12),
new Point(4,12), new Point(5,12), new Point(6,12), new Point(7,12), new Point(8,12),
                new Point(9,12), new Point(9,11), new Point(10,11), new Point(11,11),
new Point(12,11), new Point(12,10), new Point(12,9), new Point(13,9),
                new Point(13,10), new Point(14,10), new Point(14,11), new
Point(15,11), new Point(15,10), new Point(16,10), new Point(16,11), new Point(16,12)
```

```
            });

            return actualRoute;
        }

        public static Grid GetGrid2()
        {
            string filePath =
Path.Combine(Path.Combine(AppDomain.CurrentDomain.BaseDirectory, @"Data Files\Test
Maze 25X20 (RecursiveDivision).maz"));
            Grid grid = MazeLibrary.Grid.GetGridFromFile(filePath);
            return grid;
        }

        public static List<Point> ActualRouteGrid2()
        {
            List<Point> actualRoute = new List<Point>(new Point[]
            {
               new Point(0,0), new Point(0,1), new Point(0,2), new Point(0,3), new
Point(1,3), new Point(1,2), new Point(2,2), new Point(3,2), new Point(3,3),
               new Point(4,3), new Point(4,2), new Point(5,2), new Point(6,2), new
Point(6,3), new Point(6,4), new Point(7,4), new Point(7,5), new Point(8,5),
               new Point(8,6), new Point(8,7), new Point(9,7), new Point(9,8), new
Point(10,8), new Point(11,8), new Point(11,7), new Point(12,7), new Point(13,7),
               new Point(13,6), new Point(13,5), new Point(13,4), new Point(14,4), new
Point(14,5), new Point(15,5), new Point(15,4), new Point(16,4), new Point(17,4),
               new Point(17,3), new Point(18,3), new Point(19,3), new Point(19,4), new
Point(20,4), new Point(20,3), new Point(21,3), new Point(21,4), new Point(21,5),
               new Point(20,5), new Point(20,6), new Point(19,6), new Point(19,7), new
Point(20,7), new Point(21,7), new Point(21,8), new Point(22,8), new Point(22,7),
               new Point(23,7), new Point(23,8), new Point(24,8), new Point(24,9), new
Point(23,9), new Point(23,10), new Point(24,10), new Point(24,11),
               new Point(24,12), new Point(24,13), new Point(24,14), new Point(24,15),
new Point(23,15), new Point(23,16), new Point(24,16), new Point(24,17),
               new Point(24,18), new Point(24,19),
            });

            return actualRoute;
        }
    }
}
```

Test Data Files

The following listings are the two data files I used when testing the Mazer application.

Test Maze 17X13 (RecursiveBacktracker) [.maz]

```
Grid: RecursiveBackTracker|17|13
WPWW
WPWP
WWPP
WPPW
WWPP
WPPW
WPWP
WPWP
WPPP
WPWP
WPWP
WWPP
WPPW
WPWP
WWPP
WPWW
WWPP
WPPW
WPWP
PWPP
PWPW
PWPW
PWPW
WPPW
WWPP
PWPW
WPPW
WWPP
PPPW
PWWP
WPWW
PPWP
WWPP
PWPW
PPPW
WWPP
PWWW
PWPW
PWPW
PPWW
PWWP
PWWW
PPPW
PWWP
PWPW
PWWW
WPPW
WWPP
WPWW
PPWP
PWWP
PWPW
PPWW
WWPP
PWWW
PPWW
WWPP
WPPW
WWPP
PPWW
WWPP
PWPW
WPPW
PWWP
PWPW
WPPW
WPWP
WWPP
PWPW
WWPW
```

```
PPWW
WPWP
WWPP
PPWW
PWWP
PPWW
WWPP
PWPW
PPWW
PPWP
WWWP
PPWW
PPWP
WWWP
PWPW
PPWW
PPWP
WWPP
WWPW
PPPW
WPWP
WPWP
WWWP
PWPW
PPWW
WWPP
WWPW
WPPW
WPWP
WPWP
WPWP
PWPP
WPWW
WPWP
PPPP
PWWP
PPWW
WPWP
WPWP
WWPP
PWPW
WWPW
PPWW
PWPP
PPWW
WPWP
WWPP
WWPW
PWPW
WPPW
WWPP
PWWW
WPPW
WPWP
WPWP
WWPP
PWPW
PWPW
PPWW
WWPP
PWPW
WPPW
WWPP
PWPW
PWPW
PWPW
PWPW
PPWW
WPWP
PWWP
WWPW
WPPW
PPWP
PWWP
PWPW
WPPW
PWWP
```

```
PPPW
PWWP
PWWW
PWPW
PPWW
PWWP
PWPW
WPPW
WPPP
WPWP
PWWP
PWPW
WPPW
WPWP
PWWP
PPPW
WWPP
PWPW
WPPW
WWPP
PPWW
WPWP
WWPP
PWPW
PWWW
PPWW
WPWP
WPPP
PWWP
PWPW
WPPW
WPWP
PWWP
PPWW
PWWP
PWPW
PPWW
WWPP
WPPW
PWPP
PWPW
WPPW
WPWP
WWPP
PWWW
WPPW
PWWP
PPWW
WWWP
WPPW
WPWP
WPWP
PWWP
WWPW
PPWW
PWWP
PWPW
PPWW
PPWP
WWWP
PPWW
WPWP
PPWP
WPWP
WPWP
WPWP
PWWP
WPWW
WPWP
WPWP
PPWP
WPWP
WPWP
PWWP
```

Test Maze 25X20 (RecursiveDivision) [.maz]

```
Grid: RecursiveDivision|25|20
WWPW
WWPW
WWPW
WPWW
WPWP
WPWP
WWPP
WPPW
WWPP
WPPW
WWPP
WWPW
WPPW
WPPP
WPWP
WPPP
WWPP
WWPW
WPWW
WPPP
WWPP
WPPW
WWWP
WPPW
WWPP
PPPW
PPWP
PWWP
WPWW
WWPP
WPPW
PWWP
PWPW
PPPW
PWWP
PPPW
PPPP
PWWP
PWPW
WPPW
PWPP
PWPW
PPPW
WPPP
PWPP
PPPW
PWPP
WWPW
PWPW
PWPW
PWPW
WPPW
WPWP
WWPP
PPPW
PPPP
WWPP
PWWW
PWWW
WPWW
PWWP
PPWW
WWWP
PWWW
PWWW
PWWW
PPWW
PWWP
PWWW
PWWW
PWPW
PPWW
PPWP
PWWP
```

```
PWWW
PPPW
PWPP
WWPW
PPPW
PWWP
PWWW
PWPW
WWPW
WPWW
WPWP
WPWP
WPWP
WPWP
WWPP
WPWW
WPWP
WPWP
WPPP
WPPP
WWPP
PPPW
WWPP
WWPW
WPPW
WWPP
PWWW
PPWW
PWPP
PPWW
WWWP
WPWW
PPPP
PWPP
WPWW
WPWP
WWPP
WWPW
WPPW
PPPP
WWPP
WPPW
WPPP
PWPP
PWWW
PPWW
PWWP
PPPW
PWWP
PWPW
PWWW
WWPW
WWPW
PWPW
WPWW
WPWP
WPPP
PWWP
PPPW
WWPP
WPPW
PWPP
PPWW
PWWP
PWPW
PPPW
PWWP
PWWW
PWWW
WWPW
WPWW
WPPP
PWWP
WWPW
PPPW
WWWP
PPWW
```

```
PWPP
PWPW
WPPW
WPWP
PPWP
WWWP
PWPW
PPPW
PWWP
PWWW
WWPW
WWPW
PWPW
PWPW
WPPW
WWPP
WWPW
PPPW
WPPP
PWWP
WWPW
PWPW
PPWW
WWPP
WPPW
PPWP
PWWP
PPPW
WWPP
WPPW
WWWP
PWPW
PPWW
WWPP
WWPW
PPPW
PPWP
PWWP
PWPW
PWWW
PPPW
PWPP
PWPW
PPPW
WPPP
PWPP
PPPW
WWPP
PWPW
PPWW
WPPP
WWWP
PWPW
PWWW
PWPW
WWPW
PWPW
WPPW
PPPP
PPPP
PWPP
WWPW
WWPW
PPWW
WPPP
PWWP
PWPW
PWWW
PWWW
PWWW
PPPW
PWWP
PPWW
PWPP
WPPW
PWWP
WWPW
```

```
PPWW
WPWP
PPPP
PWWP
PWWW
PWWW
PWWW
PWWW
PPWW
PPWP
PWWP
WPWW
PPWP
WWWP
PWWW
WPWW
WPWP
WPWP
PPPP
WWWP
WPPW
PWWP
PWPW
WPPW
PWPP
WPWW
WPWP
PPPP
WWWP
WPPW
WWPP
WPPW
WPPP
WWPP
WWPW
WPWW
WWPP
WPPW
WWPP
WWPW
WPWW
WPPP
WPWP
PWPP
WWPW
PPPW
WWPP
PPPW
PWWP
PWWW
WPPW
WPPP
PWWP
WWPW
PWPW
PWWW
PWPW
PWWW
PWPW
PPPW
WPPP
PPPP
PWWP
PPPW
PWWP
WWPW
PWPW
WWPW
PWPW
PWPW
PWWW
PWPW
PPWW
WPPP
WWWP
PWPW
PWPW
```

```
WPPW
PWPP
PPWW
WPWP
PPWP
WWWP
PPWW
PWPP
PWWW
PWWW
WPPW
PWPP
WWPW
PPWW
PWWP
PPWW
PPWP
PWPP
WWPW
PWPW
WWPW
PPPW
WWPP
PWWW
PPWW
PWPP
PWWW
WWPW
WWPW
WWPW
WPWW
WPPP
PWPP
WPPW
WWPP
PWWW
PPPW
PWWP
WWPW
WPWW
WWPP
WPWW
PWPP
PPWW
PWPP
PWPW
PWPW
PWWW
WPWW
WPWP
PPWP
WWPP
PPPW
PWWP
PWPW
WPPW
PWPP
PWPW
PWPW
PWWW
WPWW
PWPP
WWPW
PPPW
WPPP
PPPP
WWPP
PWPW
WWPW
PWPW
PPWW
PPWP
WWPP
WPWW
WPPP
WPWP
PWWP
```

```
PPWW
WPWP
PPPP
PWWP
PWWW
PWWW
PPPW
WWPP
WWPW
PPPW
PWPP
PWWW
PWWW
PWPW
PPWW
PWWP
PPPW
PWWP
WPWW
WPWP
PWPP
WPPW
PWPP
WWPW
WWPW
WPPW
WWWP
PPPW
WPWP
WWWP
WWPW
PWPW
PWWW
PPWW
PWPP
PWWW
WPPW
WPPP
PWPP
WWPW
WWPW
PPPW
WWPP
WPPW
WPPP
PWWP
PWPW
PPWW
PWWP
PWPW
PPWW
WPPP
PPWP
WPPP
WPPP
PWPP
PPWW
WWPP
WPWW
PPWP
WWPP
PWPW
PWPW
PPPW
PPWP
PWPP
PWWW
PWPW
PWPW
PWPW
WWPW
PPPW
WPPP
WWPP
PWPW
WWPW
PWPW
```

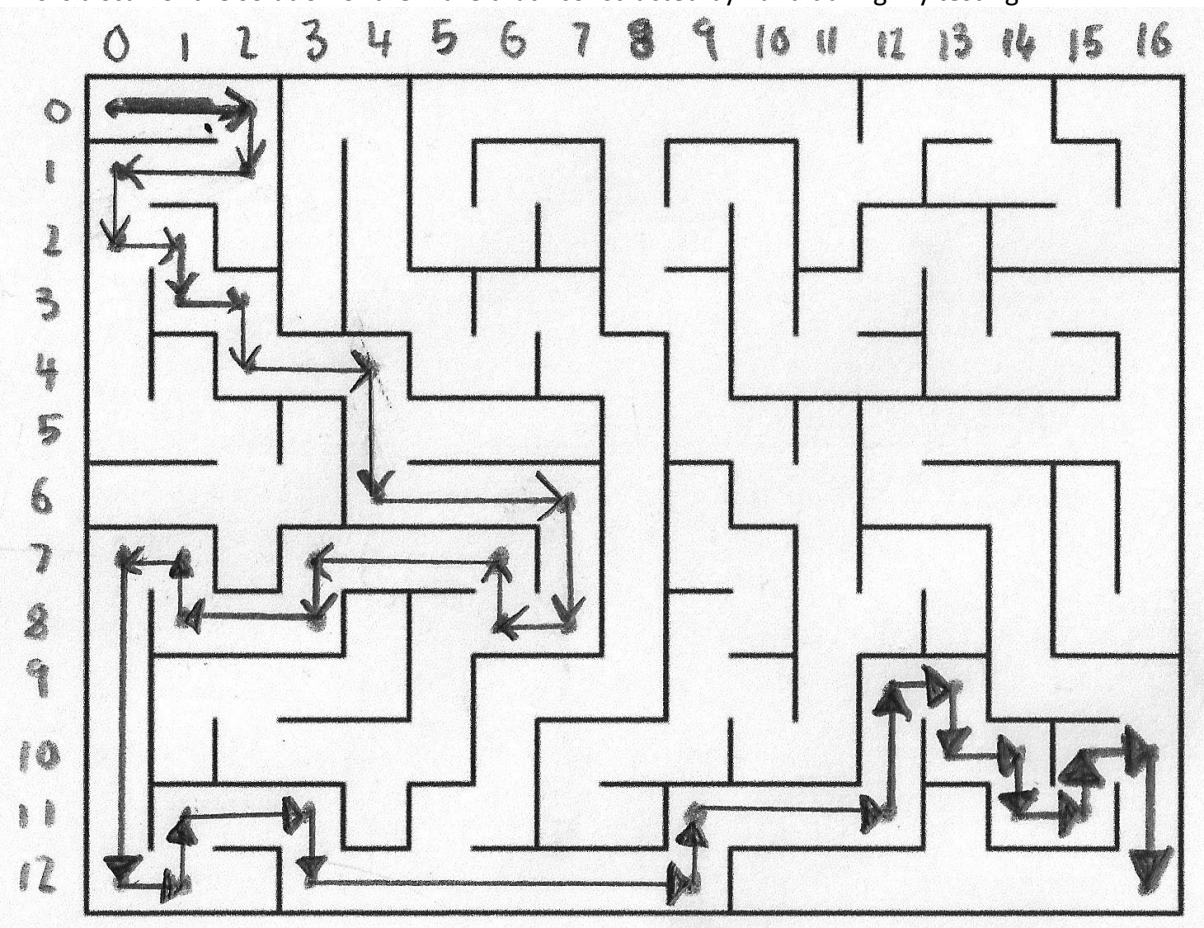
```
WPPW
PWPP
PWPW
PWPW
WPPW
PWPP
WPWW
WPWP
PWPP
PWWW
PWWW
PPPW
WWWP
PWWW
WPPW
PWPP
PWWW
PPWW
PWWP
PWWW
PWWW
PPWW
PWWP
PPWW
PWWP
PWWW
PWWW
PWWW
PPWW
PWWP
PWWW
WPWW
WPWP
PPWP
WPWP
WPWP
PPWP
WPWP
WWWP
PWWW
PWWW
```

Test Images

The following images are the mazes that the are described in the Data Files section above. Each has the solution included that I have calculated manually as part of my testing.

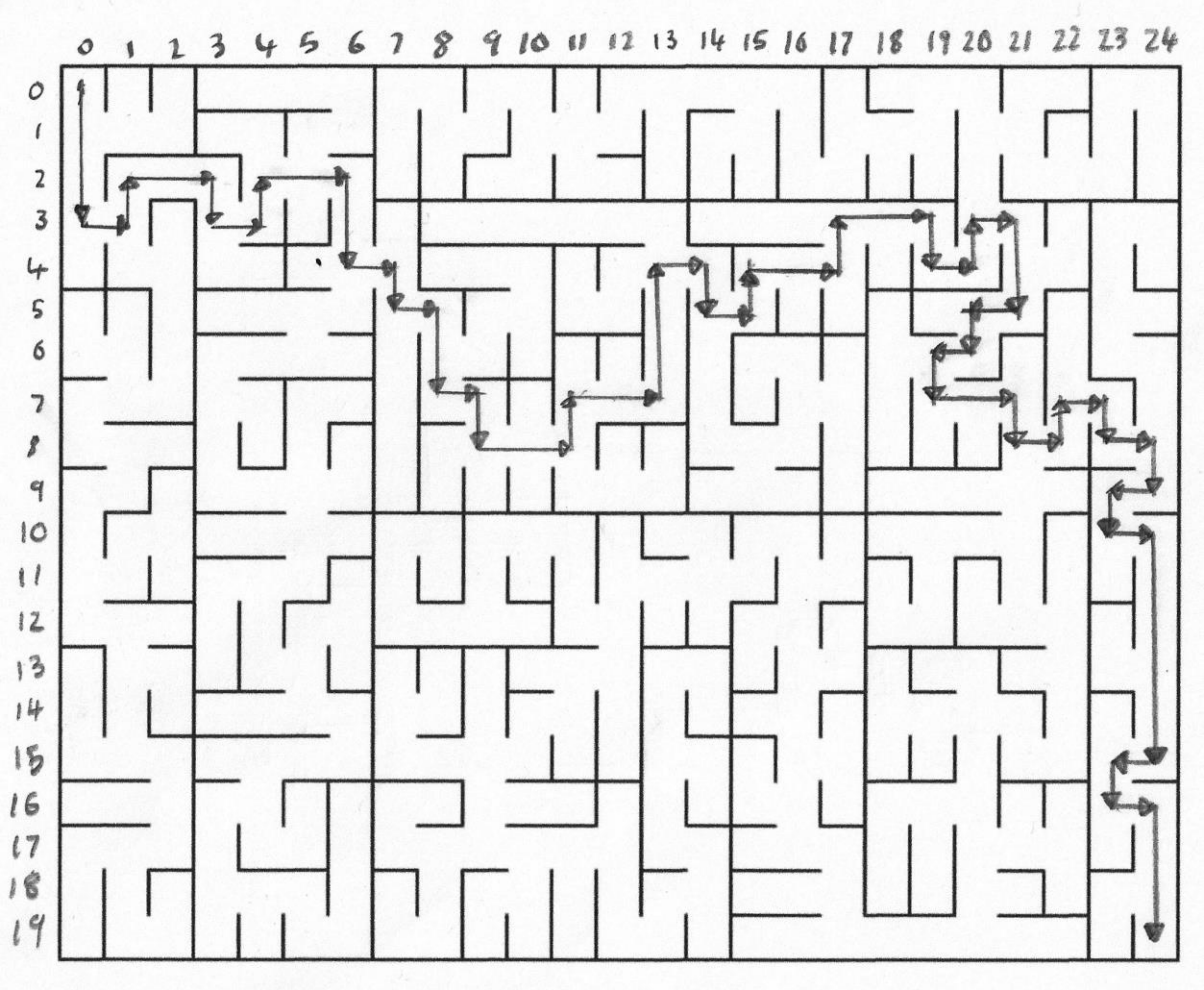
Test Maze 17X13 (RecursiveBacktracker)

This is a scan of the solution of the maze that I constructed by hand during my testing.



Test Maze 25X20 (RecursiveDivision)

This is a scan of the solution of the maze that I constructed by hand during my testing.



Testing

The following sections will describe the testing strategy that has been followed along with the evidence of the testing that has been carried out.

Test strategy

To test the Mazer application, data must be entered and certain options must be chosen before the results can be compared against expected results. Regardless of how the tests are done any differences between the expected and actual test results must be corrected in the code and the testing then repeated. This process will be followed until no known errors are found giving a high level of confidence of quality that has been reached in the application implementation.

My test strategy is going to be split into two sections. Unit Testing and Manual Testing.

Unit Testing

Certain parts of the application are suitable for testing using a Unit Testing approach. These parts are mainly the classes in the Core of the Mazer application. For example, in the case of a grid being made with a non-expected size, a test checking maze sizes would fail and changes to the code could then be made. All unexpected results from the unit tests will be remedied and then re-tested to ensure the issue has been solved. All unit tests will be re-run until all the tests pass with success.

Manual Testing

The user interface parts of the application will be tested manually. These tests will be on the UI interface which includes buttons, text boxes and the data grid. These parts of the application will be tested manually where the user follows a set procedure with an expected result. Screenshots will be taken and kept to show that the tests have passed.

Core Unit Testing

The following sections describe the unit testing carried on the classes in the Core of the Mazer application.

AlgorithmTestBase

The following table describes the unit test code use for testing the AlgorithmTestBase class.

Test Description	Test Code

Creates several grids of different sizes and names.
These are each used as known actual grids in the tests below.
All algorithm tests will use the small grid.
	`protected Grid _tinyGrid = new Grid("tinyGrid", 1, 1);`
	`protected Grid _smallGrid = new Grid("smallGrid", 5, 6);`
	`protected Grid _mediumGrid = new Grid("mediumGrid", 11, 13);`
	`protected Grid _largeGrid = new Grid("largeGrid", 31, 54);`
	`protected Grid _veryLargeGrid = new Grid("veryLargeGrid", 102, 74);`
	`protected Grid _extremelyLargeGrid = new Grid("extremelyLargeGrid", 210, 190);`

This code checks that every cell in a grid have has at least one passageway.
 	
Should one not have a passageway then there must be no way to get to the cell meaning it is a failed maze.
	`public void CheckAllCellsHaveAtLeastOnePassageWay(Grid g)`
	`{`
	`  for (int y = 0; y < g.Height; y++)`
	`  {`
	`  for (int x = 0; x < g.Width; x++)`
	`  {`
	`  Cell cell = g.GetCell(x, y);`
	`  int count = 0;`
	`  for (int i = 0; i < DirectionHelper.Count; i++)`
	`  {`
	`  CellEdge cellEdge = cell.GetEdge((Direction)i);`
	`  if (cellEdge is CellPassage) count++;`
	`  }`
	`  Assert.IsTrue(count >= 1);`
	`  }`
	`  }`
	`}`

Test Description	Test Code

This checks every cell to check that it even exists, to begin with.
 	
If a cell doesn't exist, then the solver algorithms will not be able to use it so the maze will fail.
	`public void CheckAllCellsExist(Grid g)`
	`{`
	`  for (int y = 0; y < g.Height; y++)`
	`  {`
	`  for (int x = 0; x < g.Width; x++)`
	`  {`
	`  Assert.IsNotNull(g.GetCell(x, y));`
	`  }`
	`  }`
	`}`

Code here checks that every cell has a fully initialised edge count.
 	
Should a cell be found where it does not have fully initialized edge count, then the solver algorithms will not be able to use it so the maze will fail.
	`public void CheckAllCellsHaveFullyInitializedEdges(Grid g)`
	`{`
	`  for (int y = 0; y < g.Height; y++)`
	`  {`
	`  for (int x = 0; x < g.Width; x++)`
	`  {`
	`  Cell cell = g.GetCell(x, y);`
	`  bool b = cell.IsFullyInitialized;`
	`  Assert.IsTrue(cell.IsFullyInitialized == true);`
	`  }`
	`  }`
	`}`

Test Description	Test Code
This code checks that every border cell's edge is stored as a cell wall in the appropriate direction. If one or more is not a solver algorithm would crash if it tried to move through the hole, creating a failed maze.	

```
public void CheckAllBordersExist(Grid g)
{
    for (int y = 0; y < g.Height; y++)
    {
        for (int x = 0; x < g.Width; x++)
        {
            Cell currentCell = g.GetCell(x, y);
            if (y == 0)
            {
                CellEdge ce = g.GetCell(x, y).GetEdge(Direction.North);
                Assert.IsTrue(g.GetCell(x, y).GetEdge(Direction.North) is CellWall);
            }

            if (y == g.Height - 1)
            {
                Assert.IsTrue(g.GetCell(x, y).GetEdge(Direction.South) is CellWall);
            }

            if (x == 0)
            {
                Assert.IsTrue(g.GetCell(x, y).GetEdge(Direction.West) is CellWall);
            }

            if (x == g.Width - 1)
            {
                Assert.IsTrue(g.GetCell(x, y).GetEdge(Direction.East) is CellWall);
            }
        }
    }
}
```

Test Description	Test Code

This code checks that every cell's points in an algorithms solved route is the same as the actual route's points.
 	
If they are not then the test will fail.
 	
This is run for the "Test Maze 17X13 (RecursiveBacktracker)", image shown below as figure 1.
	`public void CheckAllInRouteAreSameGrid1(List<Cell> route)`
	`{`
	`  List<Point> routePoints = new List<Point>();`
	`  foreach (Cell c in route)`
	`  {`
	`  routePoints.Add(new Point(c.Point.X, c.Point.Y));`
	`  }`
	
	`  List<Point> actualRoutePoints = GridFactory.ActualRouteGrid1();`
	
	`  for (int i = 0; i < route.Count - 1; i++)`
	`  {`
	`  Assert.IsTrue(actualRoutePoints[i] == routePoints[i]);`
	`  }`
	`}`

This code checks that every cell's points in an algorithms solved route is the same as the actual route's points.
 	
If they are not then the test will fail. This is run for the "Test Maze 25X20 (RecursiveDivision)", image shown below as figure 2.
	`public void CheckAllInRouteAreSameGrid2(List<Cell> route)`
	`{`
	`  List<Point> routePoints = new List<Point>();`
	`  foreach (Cell c in route)`
	`  {`
	`  routePoints.Add(new Point(c.Point.X, c.Point.Y));`
	`  }`
	
	`  List<Point> actualRoutePoints = GridFactory.ActualRouteGrid2();`
	
	`  for (int i = 0; i < route.Count - 1; i++)`
	`  {`
	`  Assert.IsTrue(actualRoutePoints[i] == routePoints[i]);`
	`  }`
	`}`

Grid Tests

The following unit tests all test the features of the Grid class.

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

1a	Testing all the grids to ensure they all are named correctly.	`[TestMethod]`
		`public void AllGridsHaveCorrectNames()`
		`{`
		`  Assert.AreSame("tinyGrid", _tinyGrid.Name);`
		`  Assert.AreSame("smallGrid", _smallGrid.Name);`
		`  Assert.AreSame("mediumGrid", _mediumGrid.Name);`
		`  Assert.AreSame("largeGrid", _largeGrid.Name);`
		`  Assert.AreSame("veryLargeGrid", _veryLargeGrid.Name);`
		`  Assert.AreSame("extremelyLargeGrid", _extremelyLargeGrid.Name);`
		`}`
			GridUnitTests (9)
			AllGridsHaveCorrectHeights
			AllGridsHaveCorrectNames
			AllGridsHaveCorrectWidths
			GridContainsOffGridPoint
			GridContainsOnGridPoint
			GridCreatesNonNullCell
			GridCreatesPassage
			GridCreatesWall
			GridGetsNonNullCell

1b	Testing all the grids to ensure they all have the correct widths.	`[TestMethod]`
		`public void AllGridsHaveCorrectWidths()`
		`{`
		`  Assert.IsTrue(1 == _tinyGrid.Width);`
		`  Assert.IsTrue(5 == _smallGrid.Width);`
		`  Assert.IsTrue(11 == _mediumGrid.Width);`
		`  Assert.IsTrue(31 == _largeGrid.Width);`
		`  Assert.IsTrue(102 == _veryLargeGrid.Width);`
		`  Assert.IsTrue(210 == _extremelyLargeGrid.Width);`
		`}`	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

1c	Testing all the grids to ensure they all have the correct Heights.	`[TestMethod]`
		`public void AllGridsHaveCorrectHeights()`
		`{`
		`  Assert.IsTrue(1 == _tinyGrid.Height);`
		`  Assert.IsTrue(6 == _smallGrid.Height);`
		`  Assert.IsTrue(13 == _mediumGrid.Height);`
		`  Assert.IsTrue(54 == _largeGrid.Height);`
		`  Assert.IsTrue(74 == _veryLargeGrid.Height);`
		`  Assert.IsTrue(190 == _extremelyLargeGrid.Height);`
		`}`	

1d	Testing all the grids to ensure they all have a non-null cell at position (0, 0) after a cell is created at position (0, 0)	`[TestMethod]`
		`public void GridCreatesNonNullCell()`
		`{`
		`  Cell cell = _smallGrid.CreateCell(0, 0);`
		`  Assert.IsNotNull(cell);`
		`}`	

1e	Testing all the grids to ensure they all have a return true when checking that (0, 0) is within their bounds.	`[TestMethod]`
		`public void GridContainsOnGridPoint()`
		`{`
		`  int x = 0;`
		`  int y = 0;`
		
		`  Assert.IsTrue(_smallGrid.ContainsPoint(x, y));`
		`}`	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

1f	Testing all the grids to ensure they all have a return false when checking that (-1, -1) is within their bounds.	[TestMethod]
		public void GridContainsOffGridPoint()
		{
		int x = -1;
		int y = -1;
		
		Assert.IsFalse(_smallGrid.ContainsPoint(x, y));
		}	

1g	Testing all grids to ensure that when a passage is created between two cells within its bounds, the cellEdge now is a CellPassage, and is defiantly not a CellWall.	[TestMethod]
		public void GridCreatesPassage()
		{
		Cell cell1 = _smallGrid.CreateCell(0, 0);
		Cell cell2 = _smallGrid.CreateCell(1, 0);
		
		_smallGrid.CreatePassage(cell1, cell2, Direction.East);
		
		Assert.IsTrue(cell1.GetEdge(Direction.East) is CellPassage);
		Assert.IsTrue(cell2.GetEdge(Direction.West) is CellPassage);
		
		Assert.IsFalse(cell1.GetEdge(Direction.East) is CellWall);
		Assert.IsFalse(cell2.GetEdge(Direction.West) is CellWall);
		}	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

1h	Testing all grids to ensure that when a wall is created between two cells within its bounds, the cellEdge now is a CellWall, and is defiantly not a CellPassage.	`[TestMethod]`
		`public void GridCreatesWall()`
		`{`
		`  Cell cell1 = _smallGrid.CreateCell(0, 0);`
		`  Cell cell2 = _smallGrid.CreateCell(1, 0);`
		
		`  _smallGrid.CreateWall(cell1, cell2, Direction.East);`
		
		`  Assert.IsTrue(cell1.GetEdge(Direction.East) is CellWall);`
		`  Assert.IsTrue(cell2.GetEdge(Direction.West) is CellWall);`
		
		`  Assert.IsFalse(cell1.GetEdge(Direction.East) is CellPassage);`
		`  Assert.IsFalse(cell2.GetEdge(Direction.West) is CellPassage);`
		`}`	

1i	Test ensuring that when a cell is created at a position the grid recognises that the cell is no longer null.	`[TestMethod]`
		`public void GridGetsNonNullCell()`
		`{`
		`  _smallGrid.CreateCell(0, 0);`
		
		`  Cell cell = _smallGrid.GetCell(0,0);`
		`  Assert.IsNotNull(cell);`
		`}`	

Cell Tests

The following unit tests all test the features of the Cell class.

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

2a	Test checking that a cell if created with a name, has the same string name attached to it.	`[TestMethod]`
		`public void CellNameIsCorrect()`
		`{`
		`  Cell cell = new Cell("Cell_New", 0, 0);`
		`  Assert.AreSame("Cell_New", cell.Name);`
		
		`}`
			CellUnitTests (6)
			CellGetEdgeReturnsCorrectEdge
			CellGetsRandomUninitializedDirectionCorrectly
			CellIntialisedEdgeCountIsCorrect
			CellisFullyIntialisedIsCorrect
			CellNameIsCorrect
			CellSetsEdgeCorrectly

2b	Test checking that before and after every time a cell wall is created from a specific cell, it's edge count is correct. i.e. at zero walls edge count is 0, at one wall edge count is 1 and so on.	`[TestMethod]`
		`public void CellIntialisedEdgeCountIsCorrect()`
		`{`
		`  Cell cell = _tinyGrid.CreateCell(0, 0);`
		
		`  Assert.IsTrue(cell.InitialisedEdgeCount == 0);`
		
		`  for (int i = 0; i < DirectionHelper.Count; i++)`
		`  {`
		`  Assert.IsTrue(cell.InitialisedEdgeCount == i);`
		`  _tinyGrid.CreateWall(cell, null, (Direction)i);`
		`  Assert.IsTrue(cell.InitialisedEdgeCount == i + 1);`
		`  }`
		`}`	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

2c	Test checks that when a created cell has four edges it is now considered fully initialised.	`[TestMethod]`
		`public void CellisFullyIntialisedIsCorrect()`
		`{`
		`  Cell cell = _tinyGrid.CreateCell(0, 0);`
		
		`  for (int i = 0; i < DirectionHelper.Count; i++)`
		`  {`
		`  _tinyGrid.CreateWall(cell, null, (Direction)i);`
		`  }`
		`  Assert.IsTrue(cell.IsFullyInitialized);`
		`}`	

2d	Test checks that the getEdge function on a cell returns the correct edge correspondin g to a direction from that cell.	`[TestMethod]`
		`public void CellGetEdgeReturnsCorrectEdge()`
		`{`
		`  _tinyGrid.CreateCell(0, 0);`
		`  Cell cell = _tinyGrid.GetCell(0, 0);`
		
		`  _tinyGrid.CreateWall(cell, null, Direction.North);`
		
		`  Assert.IsTrue(cell.GetEdge(Direction.North) is CellWall);`
		`}`	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

2e	Test checks that when an edge is set, the cell records this information.	[TestMethod]
		public void CellSetsEdgeCorrectly()
		{
		_smallGrid.CreateCell(0, 0);
		Cell cell = _smallGrid.GetCell(0, 0);
		CellPassage cp = new CellPassage();
		
		cell.SetEdge(Direction.North, cp);
		
		Assert.IsTrue(cell.GetEdge(Direction.North) is CellPassage);
		
		}	

2f	Test checks that when an initialised direction is returned for a given cell it is always a null edge. To make sure it doesn't return the same one over and over, I set the returned one as a cell wall.	[TestMethod]
		public void CellGetsRandomUninitializedDirectionCorrectly()
		{
		_smallGrid.CreateCell(0, 0);
		Cell cell = _smallGrid.GetCell(0, 0);
		
		for (int i = 0; i < DirectionHelper.Count; i++)
		{
		Direction direction = cell.GetRandomUninitializedDirection();
		Assert.IsTrue(cell.GetEdge(direction) == null);
		_smallGrid.CreateWall(cell, null, (Direction)i);
		}
		}	

Cell Edge Tests

The following unit tests all test the features of the CellEdge class.

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

3a	Tests whether an edge has its corresponding direction correctly. I test this with all the types of cell edge including passageway, wall and wall to a null end cell.	`[TestMethod]`
		`public void CellEdgeHasCorrectDirection()`
		`{`
		`  Cell cell = _smallGrid.CreateCell(0, 0);`
		`  Cell neighbourCell;`
		`  CellEdge cellEdge;`
		
		`  _smallGrid.CreateWall(cell, null, Direction.North);`
		`  cellEdge = cell.GetEdge(Direction.North);`
		`  Assert.IsTrue(cellEdge.Direction == Direction.North);`
		
		`  neighbourCell = _smallGrid.CreateCell(1, 0);`
		`  _smallGrid.CreatePassage(cell, neighbourCell, Direction.East);`
		`  cellEdge = cell.GetEdge(Direction.East);`
		`  Assert.IsTrue(cellEdge.Direction == Direction.East);`
		
		`  neighbourCell = _smallGrid.CreateCell(0, 1);`
		`  _smallGrid.CreateWall(cell, neighbourCell, Direction.South);`
		`  cellEdge = cell.GetEdge(Direction.South);`
		`  Assert.IsTrue(cellEdge.Direction == Direction.South);`
		`}`
			◢ **CellEdgeUnitTest** (3)
			✓ CellEdgeHasCorrectDirection
			✓ CellEdgeHasCorrectEndCell
			✓ CellEdgeHasCorrectStartCell

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

3b	Tests whether an edge stores its corresponding start cell correctly. I test this with all the types of cell edge including passageway, wall and wall to a null end cell.	`[TestMethod]`
		`public void CellEdgeHasCorrectStartCell()`
		`{`
		`  Cell cell = _smallGrid.CreateCell(0, 0);`
		`  Cell neighbourCell;`
		`  CellEdge cellEdge;`
		
		`  _smallGrid.CreateWall(cell, null, Direction.North);`
		`  cellEdge = cell.GetEdge(Direction.North);`
		`  Assert.IsTrue(cellEdge.StartCell == cell);`
		
		`  neighbourCell = _smallGrid.CreateCell(1, 0);`
		`  _smallGrid.CreatePassage(cell, neighbourCell, Direction.East);`
		`  cellEdge = cell.GetEdge(Direction.East);`
		`  Assert.IsTrue(cellEdge.StartCell == cell);`
		
		`  neighbourCell = _smallGrid.CreateCell(0, 1);`
		`  _smallGrid.CreateWall(cell, neighbourCell, Direction.South);`
		`  cellEdge = cell.GetEdge(Direction.South);`
		`  Assert.IsTrue(cellEdge.StartCell == cell);`
		
		`}`	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

3c	Tests whether an edge stores its corresponding end cell correctly. I test this with all the types of cell edge including passageway, wall and wall to a null end cell.	`[TestMethod]`
		`public void CellEdgeHasCorrectEndCell()`
		`{`
		`  Cell cell = _smallGrid.CreateCell(0, 0);`
		`  Cell neighbourCell;`
		`  CellEdge cellEdge;`
		
		`  _smallGrid.CreateWall(cell, null, Direction.North);`
		`  cellEdge = cell.GetEdge(Direction.North);`
		`  Assert.IsTrue(cellEdge.EndCell == null);`
		
		`  neighbourCell = _smallGrid.CreateCell(1, 0);`
		`  _smallGrid.CreatePassage(cell, neighbourCell, Direction.East);`
		`  cellEdge = cell.GetEdge(Direction.East);`
		`  Assert.IsTrue(cellEdge.EndCell == neighbourCell);`
		
		`  neighbourCell = _smallGrid.CreateCell(0, 1);`
		`  _smallGrid.CreateWall(cell, neighbourCell, Direction.South);`
		`  cellEdge = cell.GetEdge(Direction.South);`
		`  Assert.IsTrue(cellEdge.EndCell == neighbourCell);`
		`}`	

Direction Helper Tests

The following unit tests all test the features of the various helper classes.

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

4a	Test confirming that DirectionHelper.Count is a constant of 4.	`[TestMethod]`
		`public void DirectionHelperCountReturnsIntegerFour()`
		`{`
		`  Assert.AreEqual(4, DirectionHelper.Count);`
		`}`
			◢ **DirectionHelperUnitTests** (3)
			✓ DirectionHelperCountReturnsIntegerFour
			✓ DirectionHelperReturnsCorrectOffset
			✓ DirectionHelperReturnsOppositeDirection

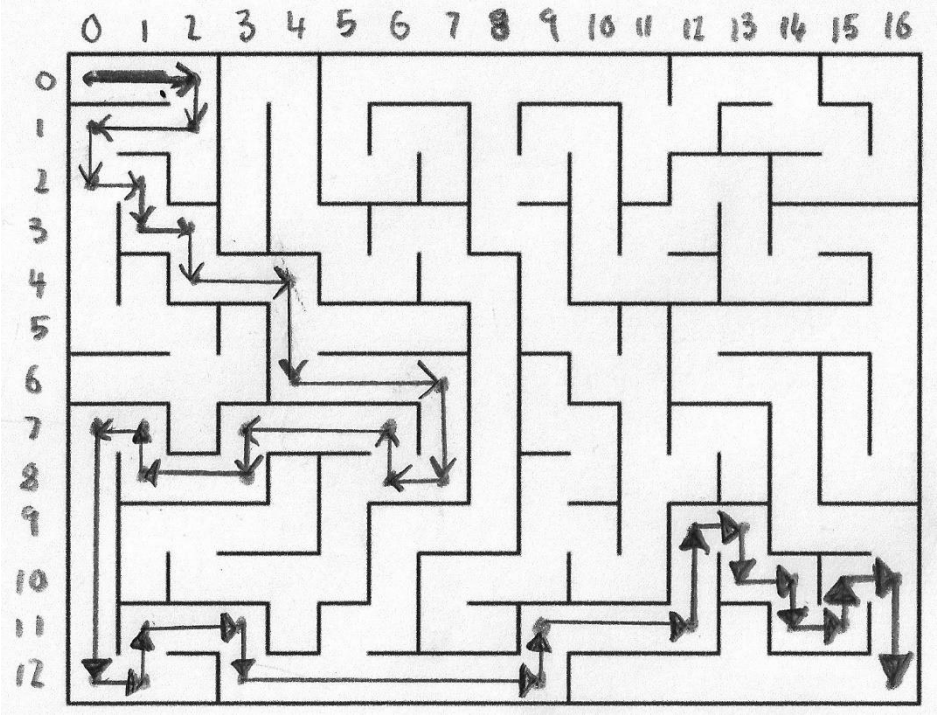
4b	Test confirming that for a cell at (0, 0) if you went east from it you would arrive at the point (1, 0). This basic test is carried out for all directions.	`[TestMethod]`
		`public void DirectionHelperReturnsCorrectOffset()`
		`{`
		`  //The startCell is at: (0, 0)`
		`  Point offset;`
		`  Point point;`
		
		`  offset = DirectionHelper.GetOffset(Direction.North);`
		`  point = new Point(offset.X, offset.Y);`
		`  Assert.IsTrue(new Point(0, -1) == point);`
		
		`  offset = DirectionHelper.GetOffset(Direction.East);`
		`  point = new Point(offset.X, offset.Y);`
		`  Assert.IsTrue(new Point(1, 0) == point);`
		
		`  offset = DirectionHelper.GetOffset(Direction.South);`
		`  point = new Point(offset.X, offset.Y);`
		`  Assert.IsTrue(new Point(0, 1) == point);`
		
		`  offset = DirectionHelper.GetOffset(Direction.West);`
		`  point = new Point(offset.X, offset.Y);`
		`  Assert.IsTrue(new Point(-1, 0) == point);`
		`}`	

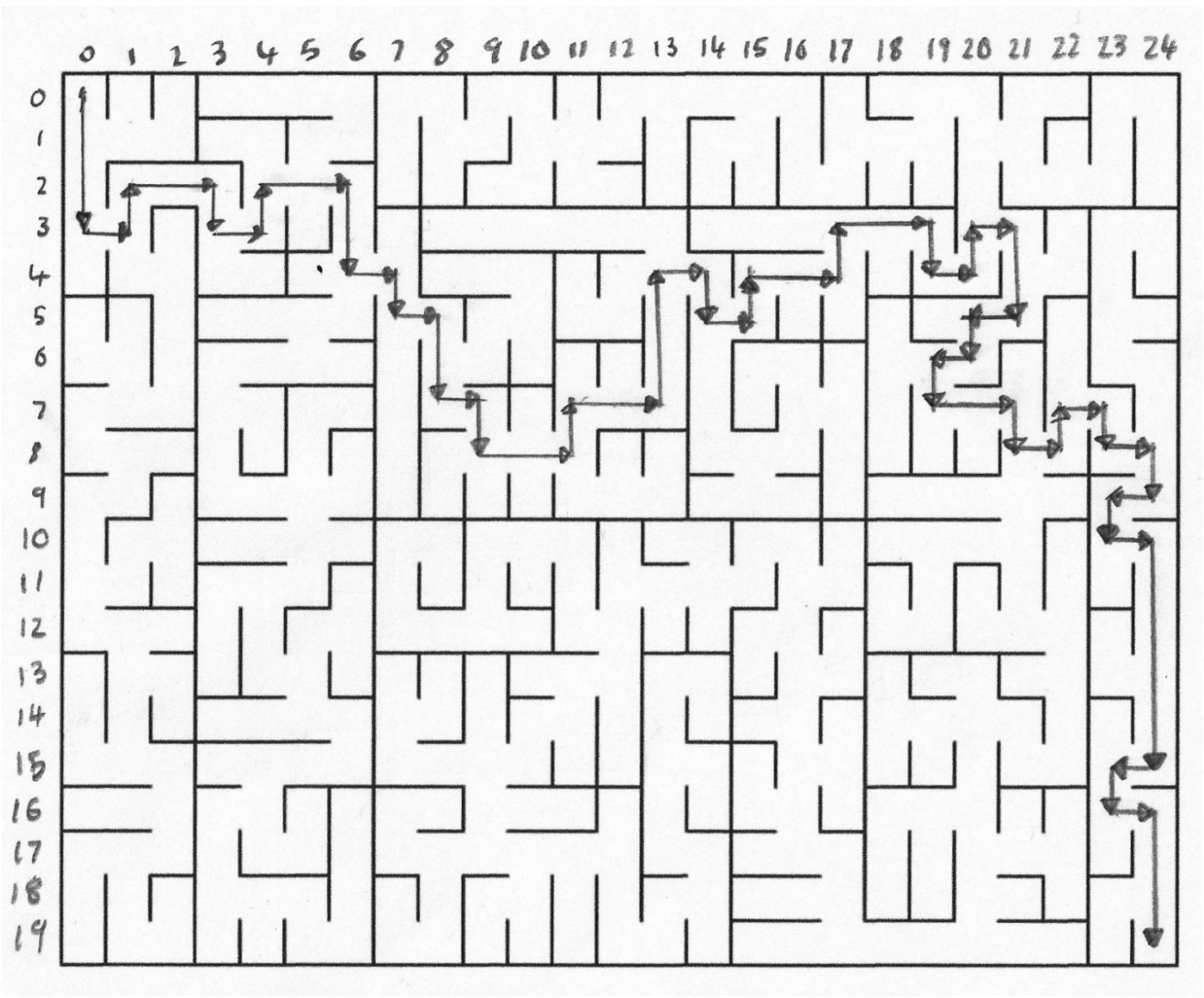
Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

4c	Test confirming that for any direction the opposite of it is correct.	`[TestMethod]`
		`public void DirectionHelperReturnsOppositeDirection()`
		`{`
		`  Assert.IsTrue(Direction.South == DirectionHelper.GetOpposite(Direction.North));`
		`  Assert.IsTrue(Direction.North == DirectionHelper.GetOpposite(Direction.South));`
		`  Assert.IsTrue(Direction.East == DirectionHelper.GetOpposite(Direction.West));`
		`  Assert.IsTrue(Direction.West == DirectionHelper.GetOpposite(Direction.East));`
		`}`	

Figures

The following table gives images of the mazes and their solutions used by the unit test in the previous sections.

Title	Image
Figure 1	

Title	Image
Figure 2	

Algorithm Unit Testing

The following section the unit test and their results used for testing the various algorithms.

A Star

Solver

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

5ai	Test checking that after loading in and solving "Test Maze 17X13 (RecursiveBacktracker)", (Figure 1), the route is the same as the actual route for the maze.	`[TestMethod]`
		`public void CheckRouteEqualsActualRouteGrid1()`
		`{`
		`  Grid grid = GridFactory.GetGrid1();`
		
		`  List<Cell> route = new List<Cell>();`
		`  List<Cell> visitedCells = new List<Cell>();`
		`  _aStarAlgorithm.Solve(ref grid, out route, out visitedCells);`
		`  route.Reverse();`
		
		`  CheckAllInRouteAreSameGrid1(route);`
		`}`
			◢ **AStarAlgorithmUnitTest** (2)
			✓ CheckRouteEqualsActualRouteGrid1
			✓ CheckRouteEqualsActualRouteGrid2

5aii	Test checking that after loading in and solving "Test Maze 25X20 (RecursiveDivision)", (Figure 2), the route is the same as the actual route for the maze.	`[TestMethod]`
		`public void CheckRouteEqualsActualRouteGrid2()`
		`{`
		`  Grid grid = GridFactory.GetGrid2();`
		
		`  List<Cell> route = new List<Cell>();`
		`  List<Cell> visitedCells = new List<Cell>();`
		`  _aStarAlgorithm.Solve(ref grid, out route, out visitedCells);`
		`  route.Reverse();`
		
		`  CheckAllInRouteAreSameGrid2(route);`
		`}`	

Binary Tree

Generator

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

5bi	Test checking the generated maze has the correct number of cells in total.	`[TestMethod]`
		`public void GridHasCorrectNumberOfCells()`
		`{`
		`  _binaryTreeAlgorithm.Generate(_smallGrid);`
		`  Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);`
		`}`
			BinaryTreeAlgorithmUnitTest (5)
			AllBordersCreated
			AllCellsExist
			AllCellsHaveAtLeastOnePassageWay
			AllEdgesAreFullyIntialized
			GridHasCorrectNumberOfCells

5bii	Test checking that every cell in the generated maze isn't null.	`[TestMethod]`
		`public void AllCellsExist()`
		`{`
		`  _binaryTreeAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsExist(_smallGrid);`
		`}`	

5biii	Test checking that all edges in the generated maze are fully initialised, so are not null.	`[TestMethod]`
		`public void AllEdgesAreFullyIntialized()`
		`{`
		`  _binaryTreeAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveFullyInitializedEdges(_smallGrid);`
		`}`	

5biv	Test checking that all the borders in the generated maze are cell walls.	`[TestMethod]`
		`public void AllBordersCreated()`
		`{`
		`  _binaryTreeAlgorithm.Generate(_smallGrid);`
		`  CheckAllBordersExist(_smallGrid);`
		`}`	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

5bv	Test checking that every cell in the generated maze has at least one passageway.	`[TestMethod]`
		`public void AllCellsHaveAtLeastOnePassageWay()`
		`{`
		`  _binaryTreeAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);`
		`}`	

Drunkard's Walk

Generator

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

6i	Test checking the generated maze has the correct number of cells in total.	`[TestMethod]`
		`public void GridHasCorrectNumberOfCells()`
		`{`
		`  _drunkardsWalkAlgorithm.Generate(_smallGrid);`
		`  Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);`
		`}`
			DrunkardsWalkAlgorithmUnitTest (7)
			AllBordersCreated
			AllCellsExist
			AllCellsHaveAtLeastOnePassageWay
			AllEdgesAreFullyIntialized
			GridHasCorrectNumberOfCells

6ii	Test checking that every cell in the generated maze isn't null.	`[TestMethod]`
		`public void AllCellsExist()`
		`{`
		`  _drunkardsWalkAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsExist(_smallGrid);`
		`}`	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

6iii	Test checking that all edges in the generated maze are fully initialised, so are not null.	`[TestMethod]`
		`public void AllEdgesAreFullyIntialized()`
		`{`
		`  _drunkardsWalkAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveFullyInitializedEdges(_smallGrid);`
		`}`	

6iv	Test checking that all the borders in the generated maze are cell walls.	`[TestMethod]`
		`public void AllBordersCreated()`
		`{`
		`  _drunkardsWalkAlgorithm.Generate(_smallGrid);`
		`  CheckAllBordersExist(_smallGrid);`
		`}`	

6v	Test checking that every cell in the generated maze has at least one passageway.	`[TestMethod]`
		`public void AllCellsHaveAtLeastOnePassageWay()`
		`{`
		`  _drunkardsWalkAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);`
		`}`	

Solver

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

7i	Test checking that after loading in and solving “Test Maze 17X13 (RecursiveBacktracker)”, (Figure 1), the route is the same as the actual route for the maze.	`[TestMethod]`
		`public void CheckRouteEqualsActualRouteGrid1()`
		`{`
		`  Grid grid = GridFactory.GetGrid1();`
		
		`  List<Cell> route = new List<Cell>();`
		`  List<Cell> visitedCells = new List<Cell>();`
		`  _drunkardsWalkAlgorithm.Solve(ref grid, out route, out visitedCell`
		
		`  CheckAllInRouteAreSameGrid1(route);`
		`}`
			DrunkardsWalkAlgorithmUnitTest (7)
			CheckRouteEqualsActualRouteGrid1
			CheckRouteEqualsActualRouteGrid2

7ii	Test checking that after loading in and solving “Test Maze 25X20 (RecursiveDivision)”, (Figure 2), the route is the same as the actual route for the maze.	`[TestMethod]`
		`public void CheckRouteEqualsActualRouteGrid2()`
		`{`
		`  Grid grid = GridFactory.GetGrid2();`
		
		`  List<Cell> route = new List<Cell>();`
		`  List<Cell> visitedCells = new List<Cell>();`
		`  _drunkardsWalkAlgorithm.Solve(ref grid, out route, out visitedCell`
		
		`  CheckAllInRouteAreSameGrid2(route);`
		`}`	

Hunt & Kill

Generator

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

8i	Test checking the generated maze has the correct number of cells in total.	`[TestMethod]`
		`public void GridHasCorrectNumberOfCells()`
		`{`
		`  _huntAndKillAlgorithm.Generate(_smallGrid);`
		`  Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);`
		`}`
			HuntAndKillAlgorithmUnitTest (5)
			AllBordersCreated
			AllCellsExist
			AllCellsHaveAtLeastOnePassageWay
			AllEdgesAreFullyIntialized
			GridHasCorrectNumberOfCells

8ii	Test checking that every cell in the generated maze isn't null.	`[TestMethod]`
		`public void AllCellsExist()`
		`{`
		`  _huntAndKillAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsExist(_smallGrid);`
		`}`	

8iii	Test checking that all edges in the generated maze are fully initialised, so are not null.	`[TestMethod]`
		`public void AllEdgesAreFullyIntialized()`
		`{`
		`  _huntAndKillAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveFullyInitializedEdges(_smallGrid);`
		`}`	

8iv	Test checking that all the borders in the generated maze are cell walls.	`[TestMethod]`
		`public void AllBordersCreated()`
		`{`
		`  _huntAndKillAlgorithm.Generate(_smallGrid);`
		`  CheckAllBordersExist(_smallGrid);`
		`}`	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

8v	Test checking that every cell in the generated maze has at least one passageway.	`[TestMethod]`
		`public void AllCellsHaveAtLeastOnePassageWay()`
		`{`
		`  _huntAndKillAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);`
		`}`	

Krustal's

Generator

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

9i	Test checking the generated maze has the correct number of cells in total.	`[TestMethod]`
		`public void GridHasCorrectNumberOfCells()`
		`{`
		`  _kruskalsAlgorithm.Generate(_smallGrid);`
		`  Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);`
		`}`
			KruskalsAlgorithmUnitTest (5)
			AllBordersCreated
			AllCellsExist
			AllCellsHaveAtLeastOnePassageWay
			AllEdgesAreFullyIntialized
			GridHasCorrectNumberOfCells

9ii	Test checking that every cell in the generated maze isn't null.	`[TestMethod]`
		`public void AllCellsExist()`
		`{`
		`  _kruskalsAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsExist(_smallGrid);`
		`}`	

9iii	Test checking that all edges in the generated maze are fully initialised, so are not null.	`[TestMethod]`
		`public void AllEdgesAreFullyIntialized()`
		`{`
		`  _kruskalsAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveFullyInitializedEdges(_smallGrid);`
		`}`	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

9iv	Test checking that all the borders in the generated maze are cell walls.	`[TestMethod]`
		`public void AllBordersCreated()`
		`{`
		`  _kruskalsAlgorithm.Generate(_smallGrid);`
		`  CheckAllBordersExist(_smallGrid);`
		`}`	

9v	Test checking that every cell in the generated maze has at least one passageway.	`[TestMethod]`
		`public void AllCellsHaveAtLeastOnePassageWay()`
		`{`
		`  _kruskalsAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);`
		`}`	

Prims

Generator

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

10i	Test checking the generated maze has the correct number of cells in total.	`[TestMethod]`
		`public void GridHasCorrectNumberOfCells()`
		`{`
		`  _primsAlgorithm.Generate(_smallGrid);`
		`  Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);`
		`}`	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

10ii	Test checking that every cell in the generated maze isn't null.	`[TestMethod]`
		`public void AllCellsExist()`
		`{`
		`  _primsAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsExist(_smallGrid);`
		`}`
			PrimsAlgorithmUnitTest (5)
			AllBordersCreated
			AllCellsExist
			AllCellsHaveAtLeastOnePassageWay
			AllEdgesAreFullyIntialized
			GridHasCorrectNumberOfCells

10iii	Test checking that all edges in the generated maze are fully initialised, so are not null.	`[TestMethod]`
		`public void AllEdgesAreFullyIntialized()`
		`{`
		`  _primsAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveFullyInitializedEdges(_smallGrid);`
		`}`	

10iv	Test checking that all the borders in the generated maze are cell walls.	`[TestMethod]`
		`public void AllBordersCreated()`
		`{`
		`  _primsAlgorithm.Generate(_smallGrid);`
		`  CheckAllBordersExist(_smallGrid);`
		`}`	

10v	Test checking that every cell in the generated maze has at least one passageway.	`[TestMethod]`
		`public void AllCellsHaveAtLeastOnePassageWay()`
		`{`
		`  _primsAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);`
		`}`	

Recursive Backtracker

Generator

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

11i	Test checking the generated maze has the correct number of cells in total.	`[TestMethod]`
		`public void GridHasCorrectNumberOfCells()`
		`{`
		`  _recursiveBackTrackerAlgorithm.Generate(_smallGrid);`
		`  Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);`
		`}`
			RecursiveBackTrackerAlgorithmUnitTest (7)
			AllBordersCreated
			AllCellsExist
			AllCellsHaveAtLeastOnePassageWay
			AllEdgesAreFullyIntialized
			GridHasCorrectNumberOfCells

11ii	Test checking that every cell in the generated maze isn't null.	`[TestMethod]`
		`public void AllCellsExist()`
		`{`
		`  _recursiveBackTrackerAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsExist(_smallGrid);`
		`}`	

11iii	Test checking that all edges in the generated maze are fully initialised, so are not null.	`[TestMethod]`
		`public void AllEdgesAreFullyIntialized()`
		`{`
		`  _recursiveBackTrackerAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveFullyInitializedEdges(_smallGrid);`
		`}`	

11iv	Test checking that all the borders in the generated maze are cell walls.	`[TestMethod]`
		`public void AllBordersCreated()`
		`{`
		`  _recursiveBackTrackerAlgorithm.Generate(_smallGrid);`
		`  CheckAllBordersExist(_smallGrid);`
		`}`	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

11v	Test checking that every cell in the generated maze has at least one passageway.	`[TestMethod]`
		`public void AllCellsHaveAtLeastOnePassageWay()`
		`{`
		`  _recursiveBackTrackerAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);`
		`}`	

Solver

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

12i	Test checking that after loading in and solving "Test Maze 17X13 (RecursiveBacktracker)", (Figure 1), the route is the same as the actual route for the maze.	`[TestMethod]`
		`public void CheckRouteEqualsActualRouteGrid1()`
		`{`
		`  Grid grid = GridFactory.GetGrid1();`
		
		`  List<Cell> route = new List<Cell>();`
		`  List<Cell> visitedCells = new List<Cell>();`
		`  _recursiveBackTrackerAlgorithm.Solve(ref grid, out route, out visit`
		
		`  CheckAllInRouteAreSameGrid1(route);`
		`}`
			RecursiveBackTrackerAlgorithmUnitTes
			CheckRouteEqualsActualRouteGrid1
			CheckRouteEqualsActualRouteGrid2

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

12ii	Test checking that after loading in and solving “Test Maze 25X20 (RecursiveDivision)”, (Figure 2), the route is the same as the actual route for the maze.	[TestMethod]
		public void CheckRouteEqualsActualRouteGrid2()
		{
		Grid grid = GridFactory.GetGrid2();
		
		List<Cell> route = new List<Cell>();
		List<Cell> visitedCells = new List<Cell>();
		_recursiveBackTrackerAlgorithm.Solve(ref grid, out route, out visit
		
		CheckAllInRouteAreSameGrid2(route);
		}	

Recursive Division

Generator

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

13i	Test checking the generated maze has the correct number of cells in total.	`[TestMethod]`
		`public void GridHasCorrectNumberOfCells()`
		`{`
		`  _recursiveDivisionAlgorithm.Generate(_smallGrid);`
		`  Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);`
		`}`
			◢ **RecursiveDivisionAlgorithmUnitTest** (5)
			✗ AllEdgesAreFullyIntialized
			✓ AllBordersCreated
			✓ AllCellsExist
			✓ AllCellsHaveAtLeastOnePassageWay
			✓ GridHasCorrectNumberOfCells

13ii	Test checking that every cell in the generated maze isn't null.	`[TestMethod]`
		`public void AllCellsExist()`
		`{`
		`  _recursiveDivisionAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsExist(_smallGrid);`
		`}`	

13iii	Test checking that all edges in the generated maze are fully initialised, so are not null.	`[TestMethod]`
		`public void AllEdgesAreFullyIntialized()`
		`{`
		`  _recursiveDivisionAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveFullyInitializedEdges(_smallGrid);`
		`}`	

13iv	Test checking that all the borders in the generated maze are cell walls.	`[TestMethod]`
		`public void AllBordersCreated()`
		`{`
		`  _recursiveDivisionAlgorithm.Generate(_smallGrid);`
		`  CheckAllBordersExist(_smallGrid);`
		`}`	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

13v	Test checking that every cell in the generated maze has at least one passageway.	`[TestMethod]`
		`public void AllCellsHaveAtLeastOnePassageWay()`
		`{`
		`  _recursiveDivisionAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);`
		`}`	

Failed Tests

The following table describes one of the failed test that I found during unit testing.

Test Id	Failed Test	Error Thrown	Cause of Error

13iii	AllEdgesAreFullyIntialized	`}`
		`}`
		`public void CheckAllCellsExist(Grid g)`
		`{`
		`  for (int y = 0; y < g.Height; y++)`
		`  {`
		`  for (int x = 0; x < g.Width; x++)`
		`  {`
		`  Assert.IsNotNull(g.GetCell(x, y));`
		`  }`
		`  }`
		`}`
		`public void CheckAllCellsHaveFullyInitializedEdges(Grid g)`
		`{`
		`  for (int y = 0; y < g.Height; y++)`
		`  {`
		`  for (int x = 0; x < g.Width; x++)`
		`  {`
		`  Cell cell = g.GetCell(x, y);`
		`  bool b = cell.IsFullyInitialized;`
		`  Assert.IsTrue(cell.IsFullyInitialized == true);`
		`  }`
		`  }`
		`}`
		`public void CheckAllBordersExist(Grid g)`
		
		AssertFailedException was unhandled by user code
		An exception of type 'Microsoft.VisualStudio.TestTools.UnitTesting.AssertFailedException' occurred in Microsoft.VisualStudio.QualityTools.UnitTestFramework.dll but was not handled in user code
		Additional information: Assert.IsTrue failed.
		Troubleshooting tips:
		Get general help for exceptions.
		Search for more Help Online...
		Exception settings:
		☑ Break when this exception type is user-unhandled
		Actions:
		View Detail...
		Enable editing
		Copy exception detail to the clipboard
		Open exception settings
			The error occurred for some of the cells generated in the maze. Their cellEdge count was not exactly equal to 4.
			
			The reason for this is that this algorithm occasionally sets a cellEdge to one type of edge then, later on, replaces it with a new type of edge.
			
			The cell.IsFullyIntialized function depends on an

Test Id	Failed Test	Error Thrown	Cause of Error
			edge count that is incremented each time an edge is set which causes this test to fail.

Failed Test Corrections

The following table describes the fix that I implemented to fix the previous error.

Test Id	Broken Class	Broken Code	Corrected Code

13iii	Cell	`//Method for replacing whatever value was in that element of the _edges array with a new cellEdge and a direct`
		`//_initialisedEdgeCount stores the number of edges that a cell has, it should never go above 4`
		`public void SetEdge(Direction direction, CellEdge cellEdge)`
		`{`
		`_initialisedEdgeCount += 1;`
		`_edges[(int)direction] = cellEdge;`
		`}`
			`//Method for replacing whatever value was in that element of the _edges array with a new cellEdge and a direc`
			`//_initialisedEdgeCount stores the number of edges that a cell has, it should never go above 4`
			`public void SetEdge(Direction direction, CellEdge cellEdge)`
			`{`
			`//Only increase the initialised edge count if the edge that is being set is currently null. If`
			`//it is not null then we are changing an edge that was already initialised so the count does not`
			`//need to be increased.`
			`if (_edges[(int)direction] == null)`
			`{`
			`_initialisedEdgeCount += 1;`
			`}`
			`_edges[(int)direction] = cellEdge;`
			`}`

Re-testing Corrected Tests

The following table describes the results from re-running my previous failed unit test.

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

13iii	Test checking that all edges in the generated maze are fully initialised, so are not null. Corrected code now ensures that only if an edge was previously null, it adds to	`[TestMethod]`
		`public void AllEdgesAreFullyIntialized()`
		`{`
		`_recursiveDivisionAlgorithm.Generate(_smallGrid);`
		`CheckAllCellsHaveFullyInitializedEdges(_smallGrid);`
		`}`
			RecursiveDivisionAlgorithmUnitTest (5)
			AllEdgesAreFullyIntialized

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed
	the edge count, otherwise the count stays the same.		

Sidewinder

Generator

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

14i	Test checking the generated maze has the correct number of cells in total.	`[TestMethod]`
		`public void GridHasCorrectNumberOfCells()`
		`{`
		`  _sidewinderAlgorithm.Generate(_smallGrid);`
		`  Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);`
		`}`
			SidewinderAlgorithmUnitTest (5)
			✗ AllEdgesAreFullyIntialized
			✓ AllBordersCreated
			✓ AllCellsExist
			✓ AllCellsHaveAtLeastOnePassageWay
			✓ GridHasCorrectNumberOfCells

14ii	Test checking that every cell in the generated maze isn't null.	`[TestMethod]`
		`public void AllCellsExist()`
		`{`
		`  _sidewinderAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsExist(_smallGrid);`
		`}`	

14iii	Test checking that all edges in the generated maze are fully initialised, so are not null.	`[TestMethod]`
		`public void AllEdgesAreFullyIntialized()`
		`{`
		`  _sidewinderAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveFullyInitializedEdges(_smallGrid);`
		`}`	

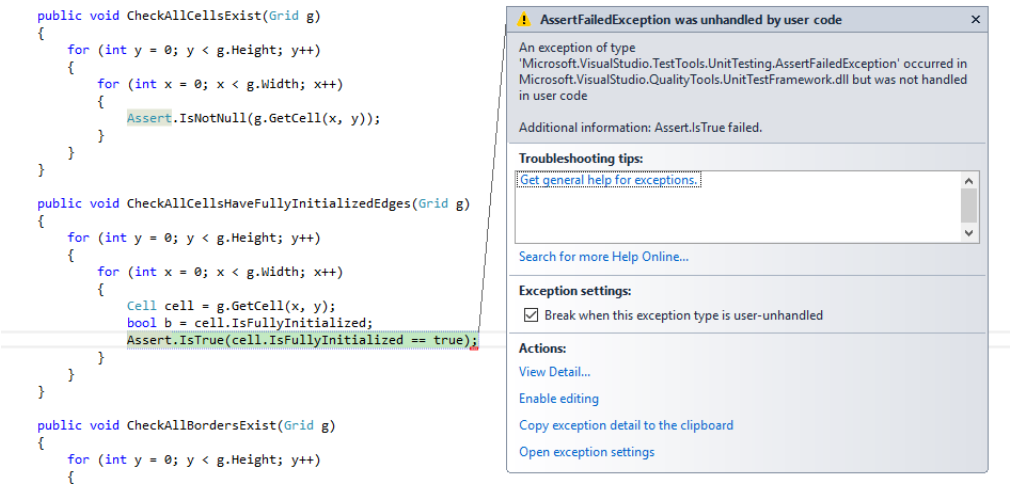
14iv	Test checking that all the borders in the generated maze are cell walls.	`[TestMethod]`
		`public void AllBordersCreated()`
		`{`
		`  _sidewinderAlgorithm.Generate(_smallGrid);`
		`  CheckAllBordersExist(_smallGrid);`
		`}`	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

14v	Test checking that every cell in the generated maze has at least one passageway.	`[TestMethod]`
		`public void AllCellsHaveAtLeastOnePassageWay()`
		`{`
		`  _sidewinderAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);`
		`}`	

Failed Tests

The following table describes one of the failed test that I found during unit testing.

Test Id	Failed Test Name	Error Thrown	Cause of Error
14iii	AllEdgesAreFullyIntialized		The error occurred for some of the cells generated in the maze. Their cellEdge count was not exactly equal to 4. The reason for this is that this algorithm occasionally sets a cellEdge to one type of edge then, later on, replaces it with a new type of edge. The cell.IsFullyIntialized function depends on an edge count that is incremented each time an

Test Id	Failed Test Name	Error Thrown	Cause of Error
			edge is set which causes this test to fail.

Failed Test Corrections

The following table describes the fix that I implemented to fix the previous error.

Test Id	Broken Class	Broken Code	Corrected Code

13iii	Cell	`//Method for replacing whatever value was in that element of the _edges array with a new cellEdge and a direction.`
		`//_initialisedEdgeCount stores the number of edges that a cell has, it should never go above 4`
		`public void SetEdge(Direction direction, CellEdge cellEdge)`
		`{`
		`_initialisedEdgeCount += 1;`
		`_edges[(int)direction] = cellEdge;`
		`}`
			`//Method for replacing whatever value was in that element of the _edges array with a new cellEdge and a direction.`
			`//_initialisedEdgeCount stores the number of edges that a cell has, it should never go above 4`
			`public void SetEdge(Direction direction, CellEdge cellEdge)`
			`{`
			`//Only increase the initialised edge count if the edge that is being set is currently null. If`
			`//it is not null then we are changing an edge that was already initialised so the count does not`
			`//need to be increased.`
			`if (_edges[(int)direction] == null)`
			`{`
			`_initialisedEdgeCount += 1;`
			`}`
			`_edges[(int)direction] = cellEdge;`
			`}`

Re-testing Corrected Tests

The following table describes the results from re-running my previous failed unit test.

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

13iii	Test checking that all edges in the generated maze are fully initialised, so are not null. Corrected code now ensures that only if an edge was previously null, it adds to the edge count, otherwise the count stays the same.	`[TestMethod]`
		`public void AllEdgesAreFullyIntialized()`
		`{`
		`_sidewinderAlgorithm.Generate(_smallGrid);`
		`CheckAllCellsHaveFullyInitializedEdges(_smallGrid);`
		`}`
			◢ **SidewinderAlgorithmUnitTest** (5)
			✔ AllEdgesAreFullyIntialized

Simple Rectangle (Initial Test)

Generator

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

15i	Test checking the generated maze has the correct number of cells in total.	`[TestMethod]`
		`public void GridHasCorrectNumberOfCells()`
		`{`
		`  _simpleRectangleAlgorithm.Generate(_smallGrid);`
		`  Assert.AreEqual(30, _smallGrid.Width * _smallGrid.Height);`
		`}`
			◢ **SimpleRectangleAlgorithmUnitTest** (5)
			✔ AllBordersCreated
			✔ AllCellsExist
			✔ AllCellsHaveAtLeastOnePassageWay
			✔ AllEdgesAreFullyIntialized
			✔ GridHasCorrectNumberOfCells

15ii	Test checking that every cell in the generated maze isn't null.	`[TestMethod]`
		`public void AllCellsExist()`
		`{`
		`  _simpleRectangleAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsExist(_smallGrid);`
		`}`	

15iii	Test checking that all edges in the generated maze are fully initialised, so are not null.	`[TestMethod]`
		`public void AllEdgesAreFullyIntialized()`
		`{`
		`  _simpleRectangleAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveFullyInitializedEdges(_smallGrid);`
		`}`	

15iv	Test checking that all the borders in the generated maze are cell walls.	`[TestMethod]`
		`public void AllBordersCreated()`
		`{`
		`  _simpleRectangleAlgorithm.Generate(_smallGrid);`
		`  CheckAllBordersExist(_smallGrid);`
		`}`	

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

15v	Test checking that every cell in the generated maze has at least one passageway.	`[TestMethod]`
		`public void AllCellsHaveAtLeastOnePassageWay()`
		`{`
		`  _simpleRectangleAlgorithm.Generate(_smallGrid);`
		`  CheckAllCellsHaveAtLeastOnePassageWay(_smallGrid);`
		`}`	

Wall Follower (Left)

Solver

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

16i	Test checking that after loading in and solving "Test Maze 17X13 (RecursiveBacktracker)", (Figure 1), the route is the same as the actual route for the maze.	`[TestMethod]`
		`public void CheckRouteEqualsActualRouteGrid1()`
		`{`
		`  Grid grid = GridFactory.GetGrid1();`
		
		`  List<Cell> route = new List<Cell>();`
		`  List<Cell> visitedCells = new List<Cell>();`
		`  _wallFollowerAlgorithm.SolveLeft(ref grid, out route, out visitedCells);`
		
		`  CheckAllInRouteAreSameGrid1(route);`
		`}`
			WallFollowerAlgorithmSolveLeftUnitTest (2)
			CheckRouteEqualsActualRouteGrid1
			CheckRouteEqualsActualRouteGrid2

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

16ii	Test checking that after loading in and solving “Test Maze 25X20 (RecursiveDivision)”, (Figure 2), the route is the same as the actual route for the maze.	
		[TestMethod]
		public void CheckRouteEqualsActualRouteGrid2()
		{
		Grid grid = GridFactory.GetGrid2();
		
		List<Cell> route = new List<Cell>();
		List<Cell> visitedCells = new List<Cell>();
		_wallFollowerAlgorithm.SolveLeft(ref grid, out route, out visitedCells);
		
		CheckAllInRouteAreSameGrid2(route);
		}	

Wall Follower (Right)

Solver

Test Id	What is Being Tested	Unit Test Code	Test Passed or Failed

17i	Test checking that after loading in and solving “Test Maze 17X13 (RecursiveBacktracker)”, (Figure 1), the route is the same as the actual route for the maze.	`[TestMethod]`
		`public void CheckRouteEqualsActualRouteGrid1()`
		`{`
		`Grid grid = GridFactory.GetGrid1();`
		
		`List<Cell> route = new List<Cell>();`
		`List<Cell> visitedCells = new List<Cell>();`
		`_wallFollowerAlgorithm.SolveRight(ref grid, out route, out visitedCells);`
		
		`CheckAllInRouteAreSameGrid1(route);`
		`}`
			WallFollowerAlgorithmSolveRightUnitTest (2)
			CheckRouteEqualsActualRouteGrid1
			CheckRouteEqualsActualRouteGrid2

17ii	Test checking that after loading in and solving “Test Maze 25X20 (RecursiveDivision)”, (Figure 2), the route is the same as the actual route for the maze.	`[TestMethod]`
		`public void CheckRouteEqualsActualRouteGrid2()`
		`{`
		`Grid grid = GridFactory.GetGrid2();`
		
		`List<Cell> route = new List<Cell>();`
		`List<Cell> visitedCells = new List<Cell>();`
		`_wallFollowerAlgorithm.SolveRight(ref grid, out route, out visitedCells);`
		
		`CheckAllInRouteAreSameGrid2(route);`
		`}`	

UI Manual Testing

Button Testing

Window Initialisation

This series of tests are done on the controls within the windows. The tests are ensuring that controls are initialized to active or inactive correctly.

Start-up Window

Test id	Element	Expected Initialised Result	Initialised Result	Success?
18i	Start-up Window: “Generate Maze” button	Active	Active	Success
18ii	Start-up Window: “Solve Maze” button	Active	Active	Success
18iii	Start-up Window: “Quit” button	Active	Active	Success

Generate Window

Test id	Element	Expected Initialised Result	Initialised Result	Expected Result when “Generate” Clicked	Result when “Generate” Clicked	Success?
19i	Generate Window: “txtbxMazeWidth” textbox	Active	Active	Active	Active	Success
19ii	Generate Window: “txtbxMazeHeight” textbox	Active	Active	Active	Active	Success
19iii	Generate Window: “`list box`” `ListBox`	Active	Active	Active	Active	Success
19iv	Generate window: “Generate” Button	Active	Active	Inactive	Inactive	Success
19v	Generate Window: “Save As (.maz)” button	Inactive	Inactive	Active	Active	Success

Test id	Element	Expected Initialised Result	Initialised Result	Expected Result when “Generate” Clicked	Result when “Generate” Clicked	Success?
19vi	Generate Window: “Save Image (.png)” button	Inactive	Inactive	Active	Active	Success
19vii	Generate Window: “Show Results Histogram” button	Inactive	Inactive	Active	Active	Success
19viii	Generate Window: “Back” button	Active	Active	Active	Active	Success
19ix	Generate Window: “sldWallThickness” slider and textbox	Active	Active	Active	Active	Success
19x	Generate Window: “sldDelay” slider and textbox	Active	Active	Active	Active	Success
19xi	Generate Window: “sldZoom” slider and text box	Active	Active	Active	Active	Success
19xii	Generate Window: “Hide Highlight” checkbox	Active	Active	Inactive	Inactive	Success
19xiii	Generate Window: “Hide Wall and Path Updates” checkbox	Active	Active	Inactive	Inactive	Success
19xiv	Generate Window: “Hide Highlight” checkbox	Active	Active	Inactive	Inactive	Success
19xv	Generate Window: “`dataGrid`” Data Grid	Active	Active	Active	Active	Success

Solver Window

Test id	Element	Expected Initialised Result	Initialised Result	Expected Result when "Generate" Clicked	Result when "Generate" Clicked	Success?
20i	"`list box`" `ListBox`	Active	Active	Active	Active	Success
20ii	"Open Maze" Button	Active	Active	Inactive	Inactive	Success
20iii	"Solver" Button	Active	Active	Inactive	Inactive	Success
20iv	"Save Image (.png)" button	Inactive	Inactive	Active	Active	Success
20v	"Show Results Histogram" button	Inactive	Inactive	Active	Active	Success
20vi	"Back" button	Active	Active	Active	Active	Success
20vii	"sldWallThickness" slider and textbox	Active	Active	Active	Active	Success
20viii	"sldDelay" slider and text box	Active	Active	Inactive	Inactive	Success
20ix	sldZoom" slider and text box	Active	Active	Active	Active	Success
20x	Hide Highlight" checkbox	Active	Active	Inactive	Inactive	Success
20xi	Hide Highlight" checkbox	Active	Active	Inactive	Inactive	Success
20xii	Generate Window: "`dataGrid`" Data Grid	Active	Active	Active	Active	Success

UI Canvas Manual Testing

All UI manual tests shall be carried out using a 32 by 32 size maze.

Generation

Algorithms on Canvas

Note: The Recursive Division algorithm doesn't use a highlight because there is no usage of a current cell in that algorithms code.

Algorithm	Expected Maze Generation and Resulting Shape	Hide Highlight works?	Hide Grid Lines Works?	Hide Wall and Passage Updates Works?	Maze Generation and Resulting Shape correct?	Evidence
Binary Tree	• Generate one row completed at a time going down. • First row empty of walls other than borders. • First column empty of walls other than borders. • A general pattern of passages moving South-East.	Yes	Yes	Yes	Successful	Figure 3
Drunkard's Walk	• Generate from one position as position moves around grid. • Maze consist of many short hallways with occasional longer ones is rather random so is harder to place how mazes might look.	Yes	Yes	Yes	Successful	Figure 4
Hunt & Kill	• Drunkard's Walk's generation but where position to continue from is changed to an uncompleted cell closest to the top and left when the current run gets stuck.	Yes	Yes	Yes	Successful	Figure 5
Krustal's	• Generation creates two cells and links them (with an edge type) then creates two more and links them and so on.	Yes	Yes	Yes	Successful	Figure 6
Prims	• Maze generated starts spreading out like a virus from one initial point. • This point is where all passages lead to.	Yes	Yes	Yes	Successful	Figure 7

Algorithm	Expected Maze Generation and Resulting Shape	Hide Highlight works?	Hide Grid Lines Works?	Hide Wall and Passage Updates Works?	Maze Generation and Resulting Shape correct?	Evidence
Recursive Backtracker	• Hunt & Kill but every time it gets stuck, it backtracks until it finds a new spot to continue from. • Lots of long winding hallways.	Yes	Yes	Yes	Successful	Figure 8
Recursive Division	• Splits the grid recursively laying down walls with one passage. • Lots of both short and long passages but with a lot of splits in the path.	Yes	Yes	Yes	Successful	Figure 9
Sidewinder	• Binary tree, but the first column is used and groups of cells are joined horizontally with one of the group having a vertical passage.	Yes	Yes	Yes	Successful	Figure 10
Simple Rectangle	• Box with only the borders as walls.	Yes	Yes	Yes	Successful	Figure 11

Events, Extras and Zoom on Canvas

Prims algorithm (chosen randomly) will be tested in regards to the checkboxes; hide grid lines, hide highlight and hide wall and passage updates. Ad-hoc testing will be carried out on the other algorithms although this will not be recorded in this document. As all checks for the tests occur before the algorithms are generated it would be unexpected for a problem to occur due to the actual algorithm that is used in this test.

Algorithm Chosen at Random	All Checkboxes Off	Grid Lines Checkbox On	Grid Lines and Wall & Path Updates Checkboxes On	Highlight and Grid Lines Checkboxes On	Highlight and Wall & Path Updates Checkboxes On	All Checkboxes On	Zoom Working	All Successful
Prims	Figure 12	Figure 13	Figure 14	Figure 15	Figure 16	Figure 17	Figure 18	Successful

Algorithm Clicked Displays Corresponding Maze

Algorithm Clicked	Maze Loaded	Successful?
Binary Tree	Binary Tree's Maze	Successful
Drunkard's Walk	Drunkard's Walk's Maze	Successful
Hunt & Kill	Hunt & Kill's Maze	Successful
Krustal's	Krustal's Maze	Successful
Prims	Prims' Maze	Successful – Figure 19
Recursive Backtracker	Recursive Backtracker's Maze	Successful
Recursive Division	Recursive Division's Maze	Successful – Figure 20
Sidewinder	Sidewinder's Maze	Successful
Simple Rectangle	Simple Rectangle's Maze	Successful – Figure 21

Solving

Algorithms on Canvas

Algorithm	Expected Solving Method	Resulting Maze Shape	Evidence
A*	• Acts like prim's generation, spreading out like a virus. • Always prefers exploring paths closer to the end point, ignoring walls.	Successful	Figure 22
Drunkard's Walk	• Acts as Drunkard's Walks generation, moving randomly until the end is found.	Successful	Figure 23
Recursive Backtracker	• Acts as Recursive Backtracker's generation backtracks whenever a dead-end is found. • Will never re-enter a path leading to a dead-end.	Successful	Figure 24

Algorithm	Expected Solving Method	Resulting Maze Shape	Evidence
Wall Follower (Left)	• Follows the left wall until the end is found.	Successful	Figure 25
Wall Follower (Right)	• Follows the right wall until the end is found.	Successful	Figure 26

Events, Extras and Zoom on Canvas

Three algorithms will be checked for these tests as there is no need to do anymore. All checks for the tests occur before the algorithms are solved and so should not be affected by the actual algorithm selected for the test.

Algorithm Chosen at Random	All Checkboxes Off	Grid Lines Checkbox On	Highlight Checkboxes On	All Checkboxes On	Zoom Working	Successful?
A Star	Figure 27	Figure 28	Figure 29	Figure 30	Figure 31	Successful

Algorithm Clicked Displays Corresponding Maze

Algorithm Clicked	Maze Loaded	Successful?
A*	A*'s Maze	Successful – Figure 32
Drunkard's Walk	Drunkard's Walk's Maze	Successful
Recursive Backtracker	Recursive Backtracker's Maze	Successful – Figure 33
Wall Follower (Left)	Wall Follower (Left)'s Maze	Successful – Figure 34
Wall Follower (Right)	Wall Follower (Right)'s Maze	Successful

Figures

The following table gives images of the mazes and their solutions created during the manual testing in the previous sections.

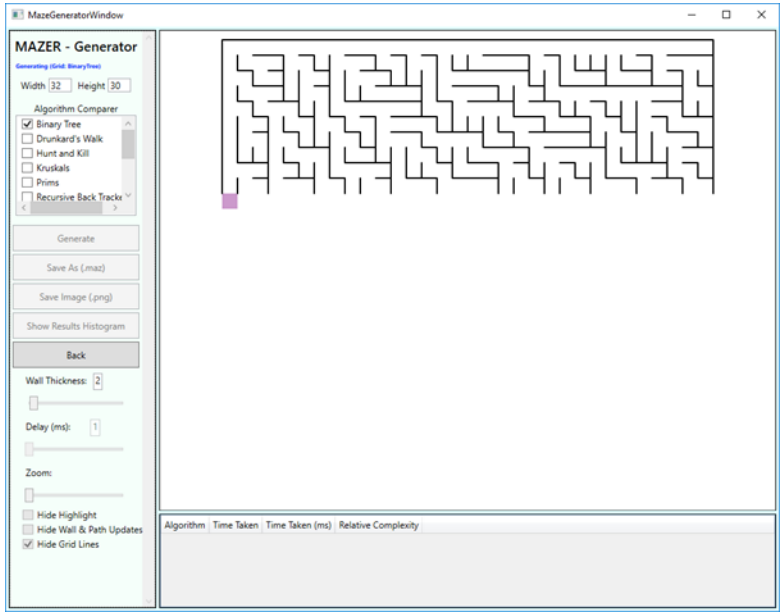
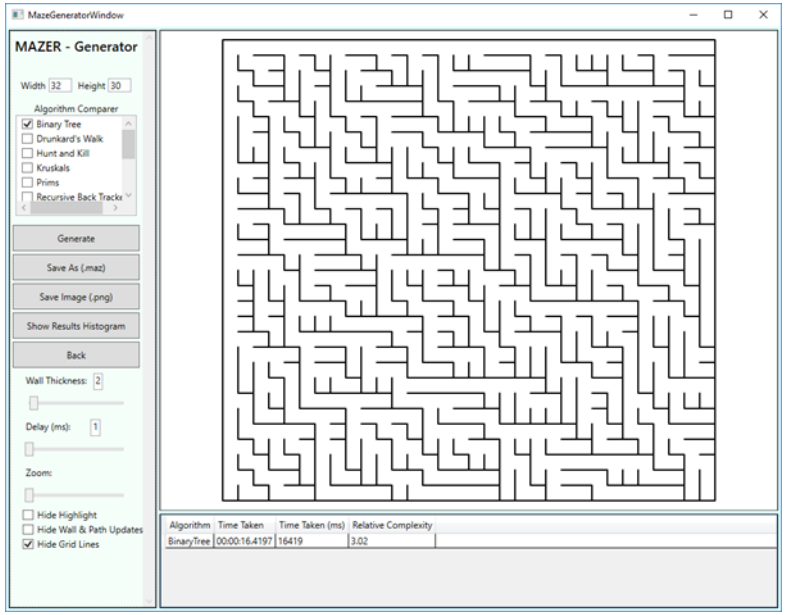
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
Generation: Algorithms on Canvas	Figure 3		

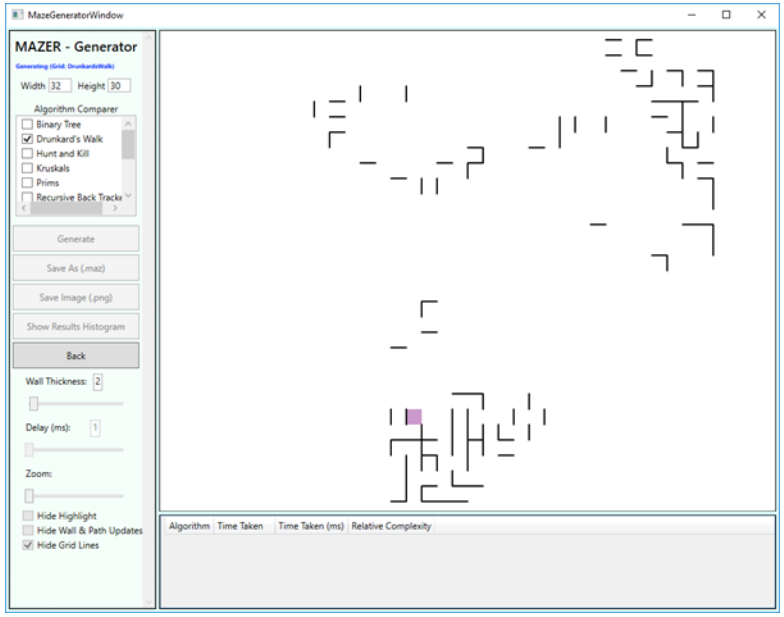
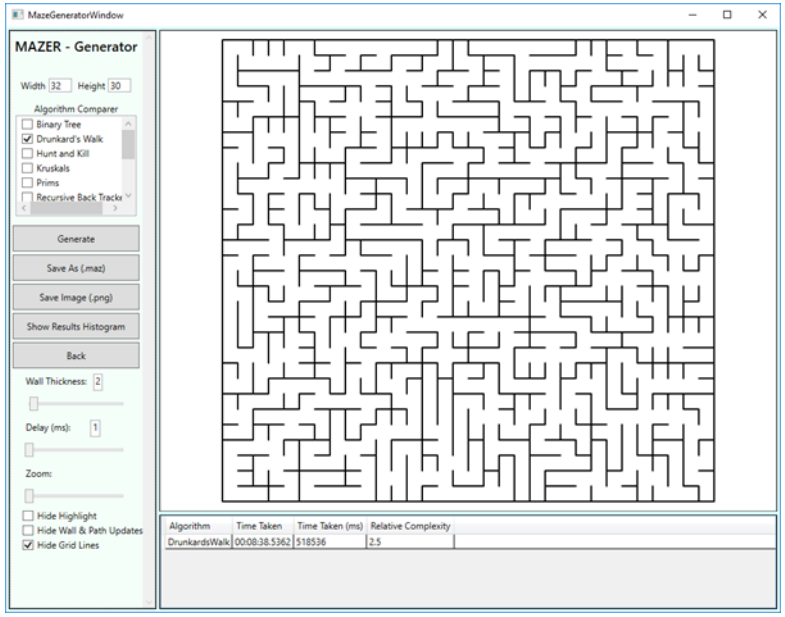
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 4		

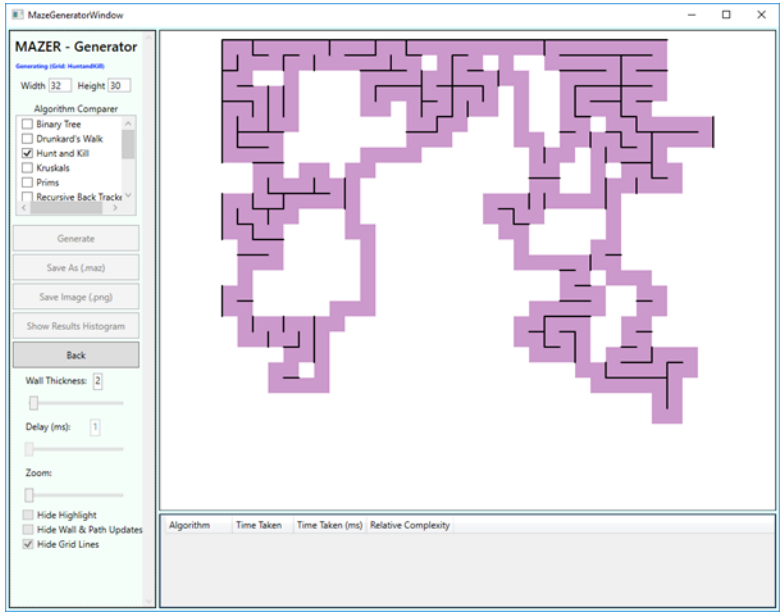
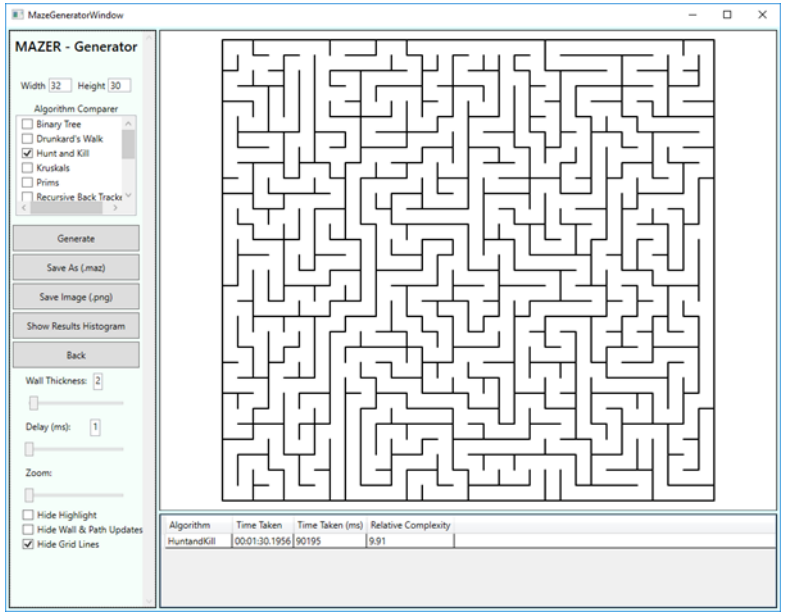
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 5		

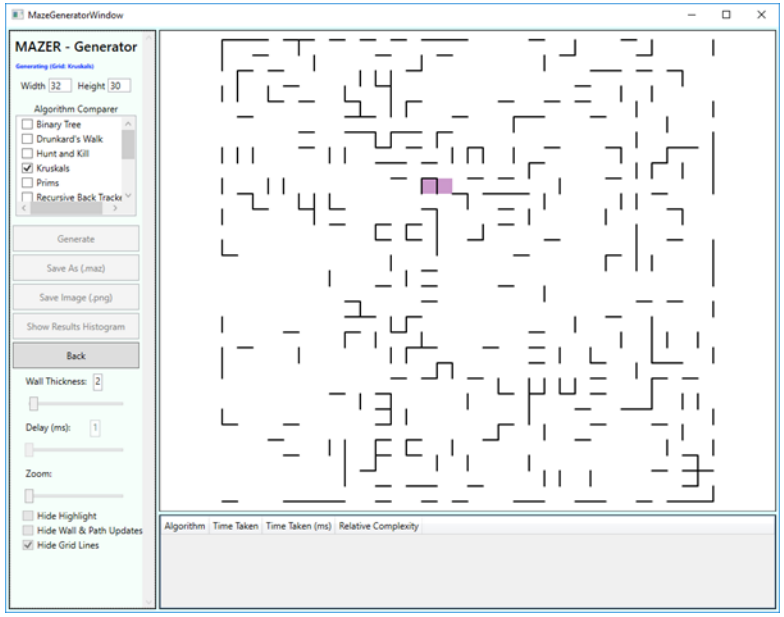
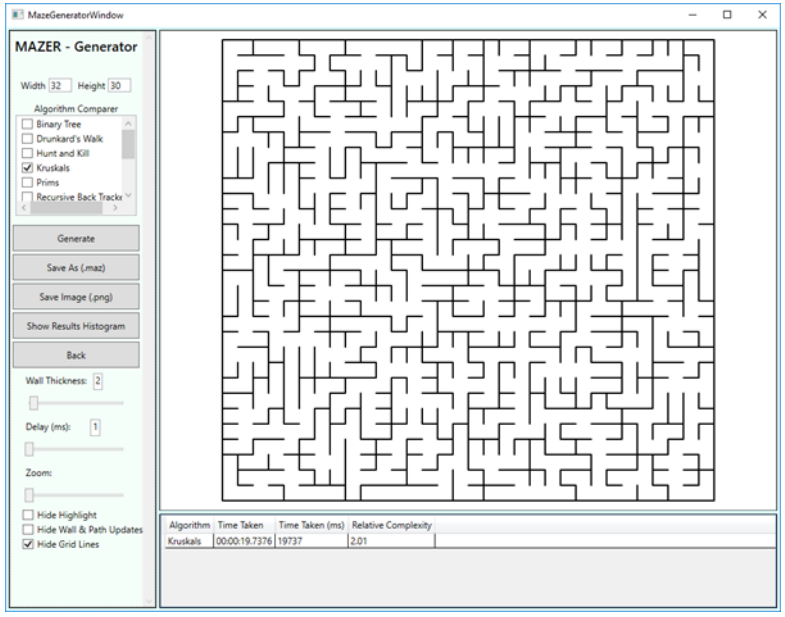
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 6		

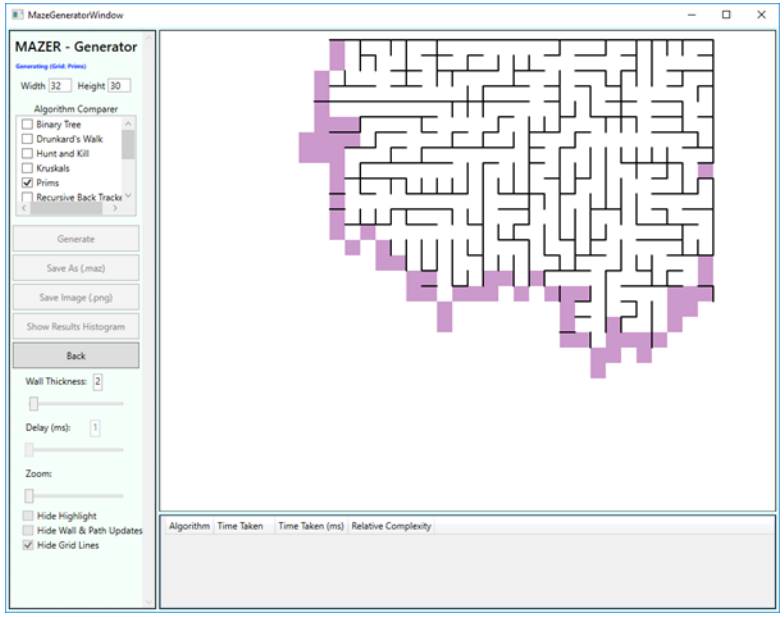
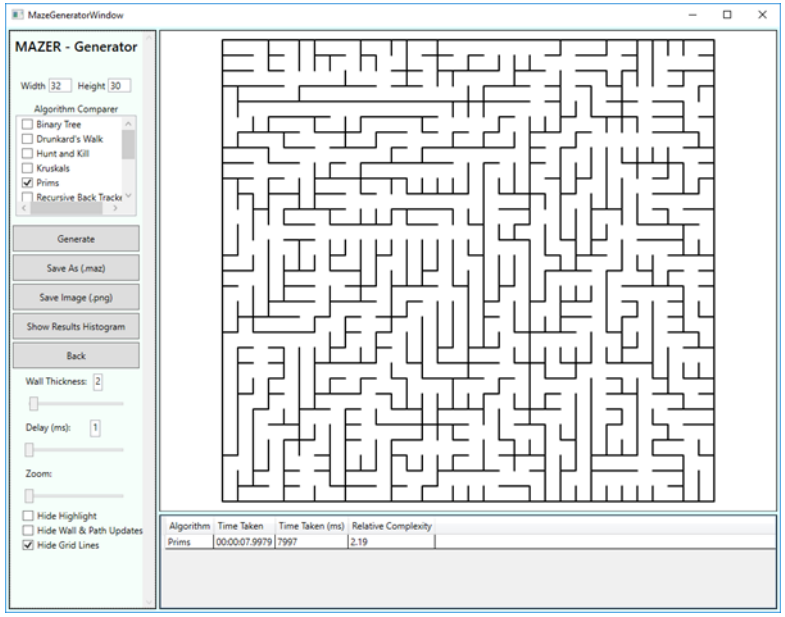
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 7		

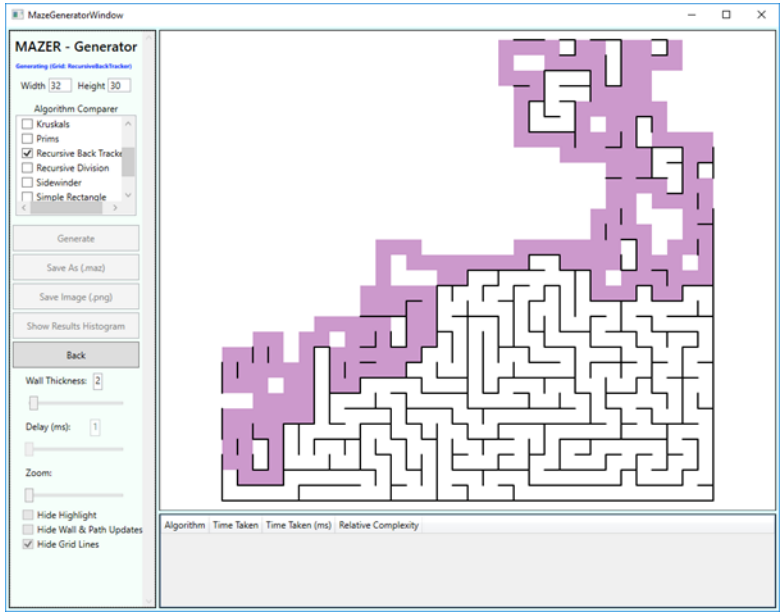
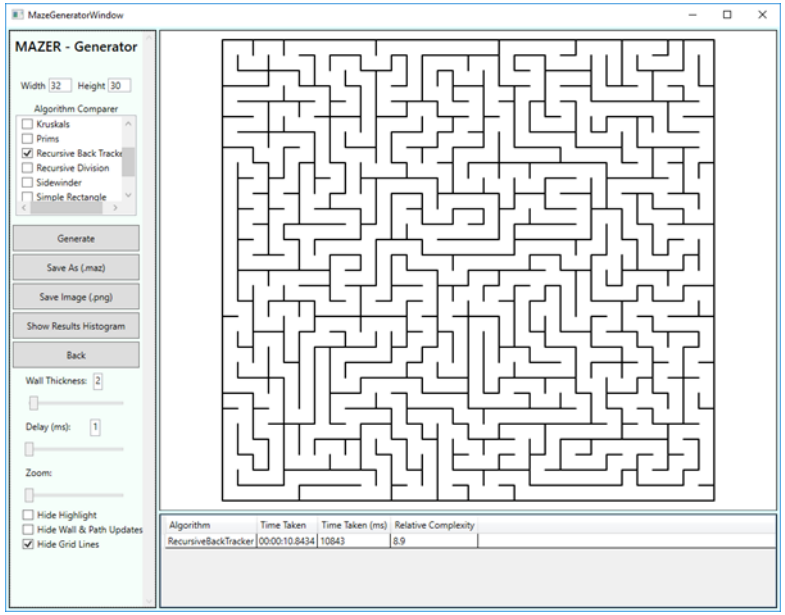
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 8		

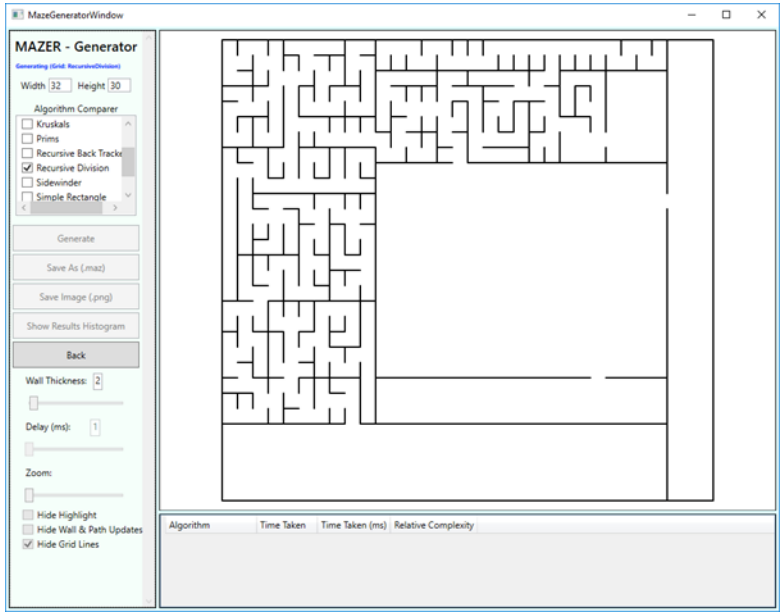
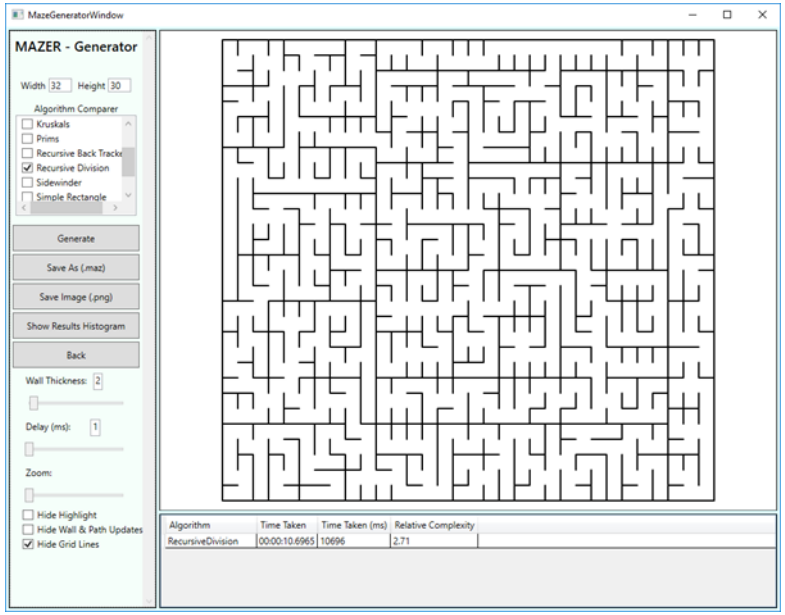
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 9		

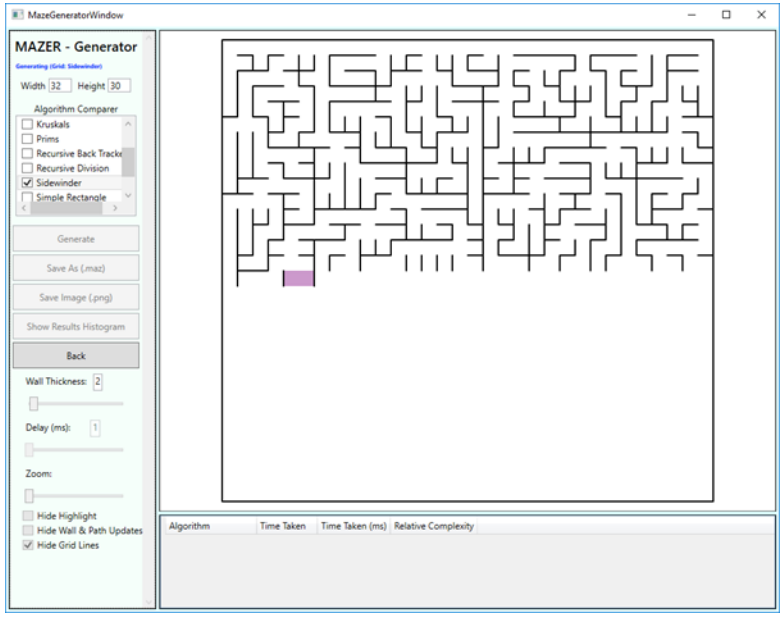
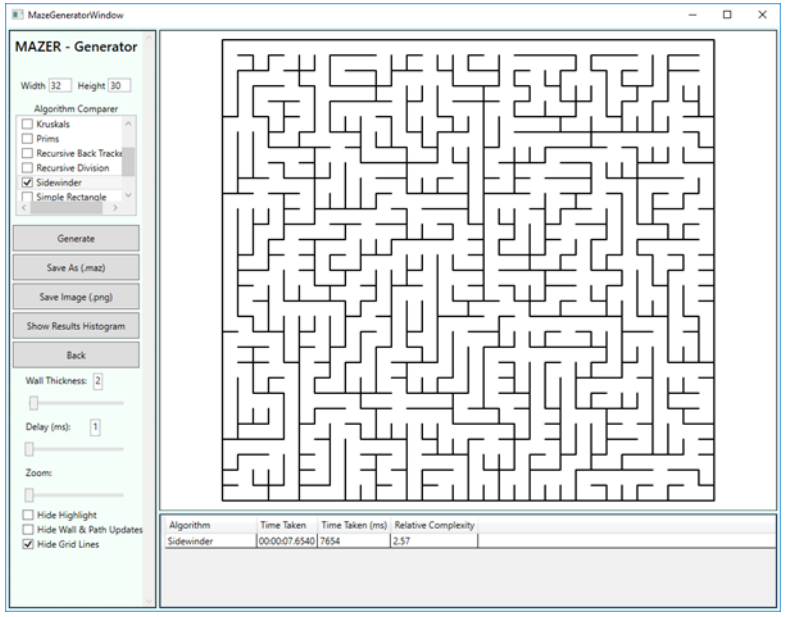
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 10		

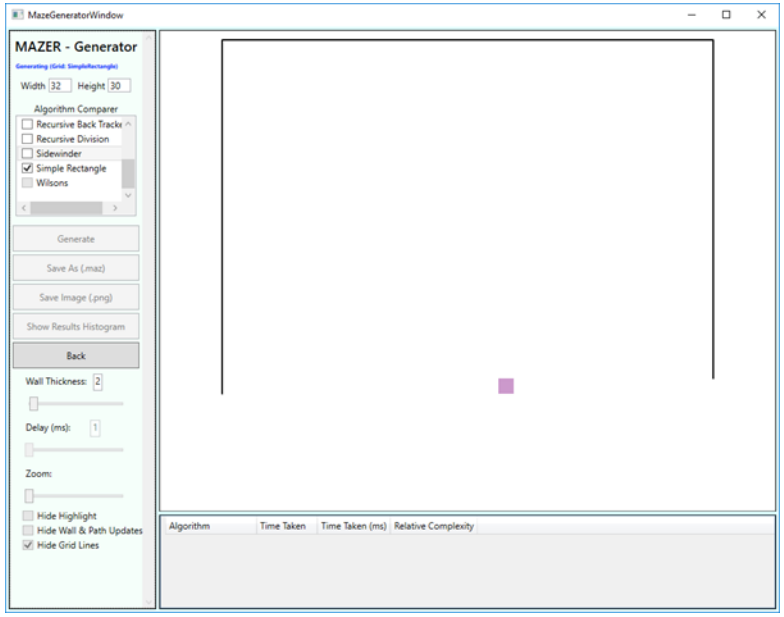
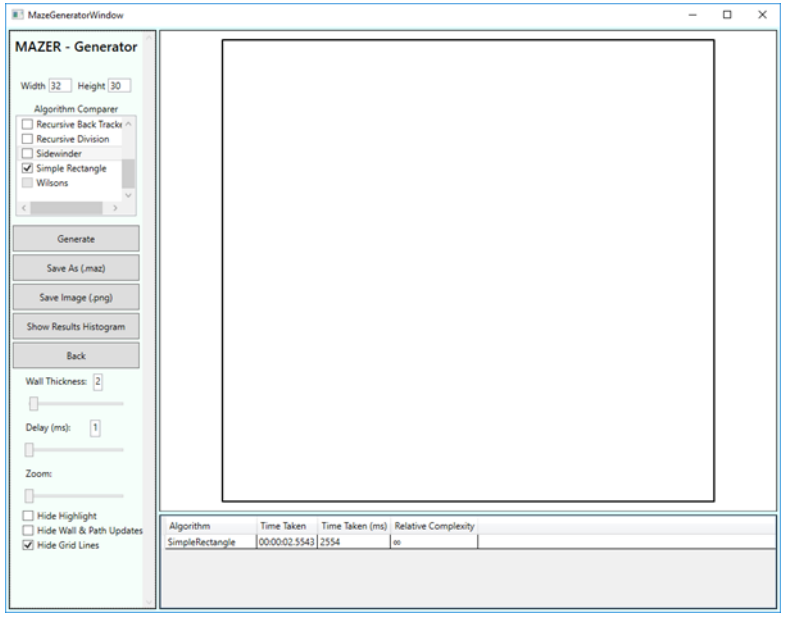
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 11		

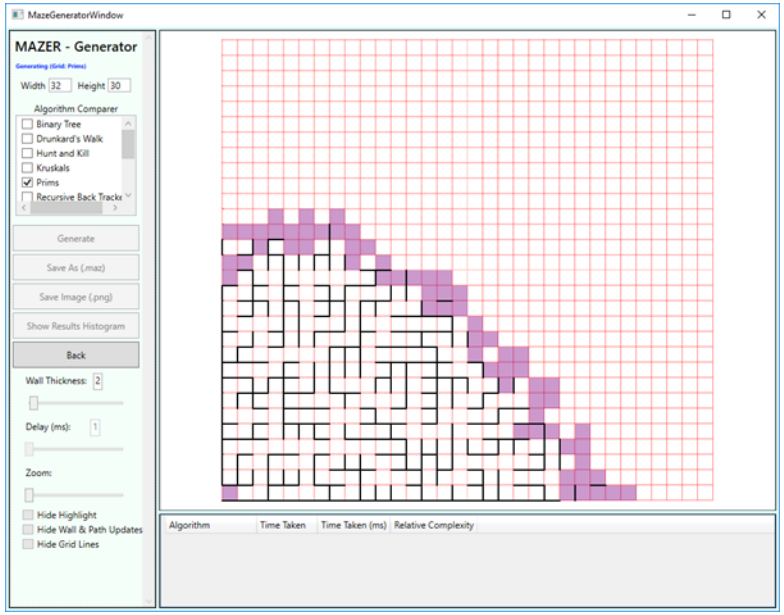
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
Generation: *Events, Extras and Zoom on Canvas*	Figure 12		N/A
		All Off	

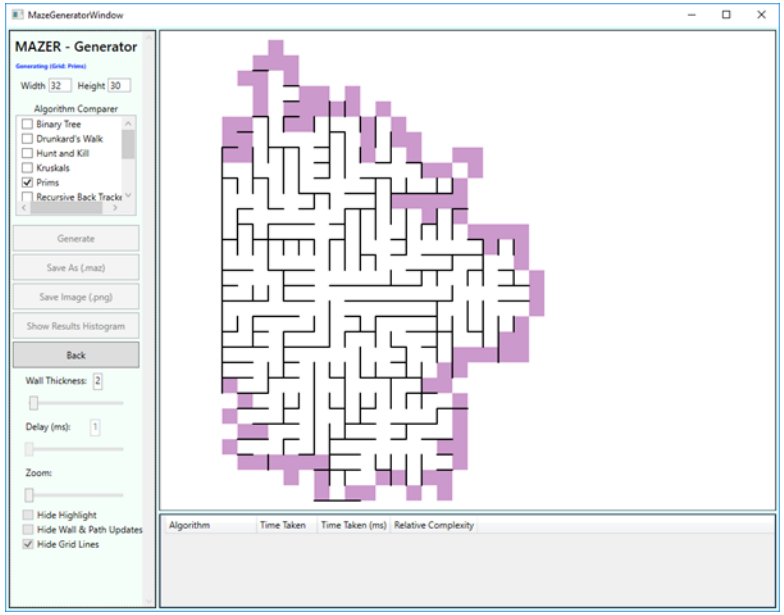
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 13		N/A
		Only Hide GridLines On	

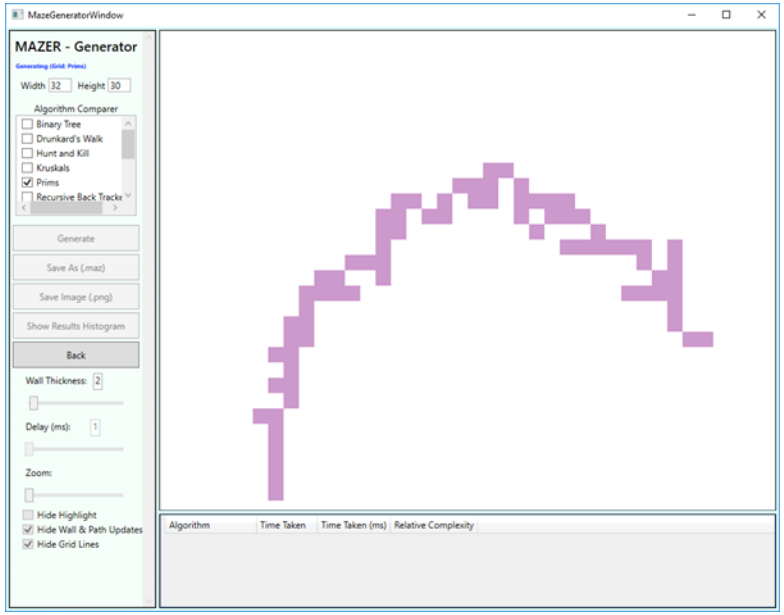
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 14		
		Hide Grid Lines and Hide Wall and Path Updates On	

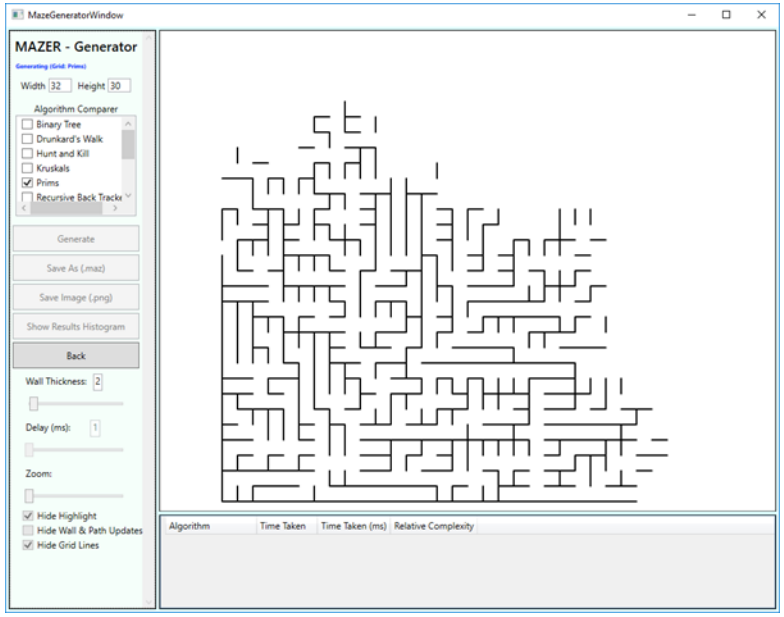
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 15		
		Hide Highlight and Hide Grid Lines On	

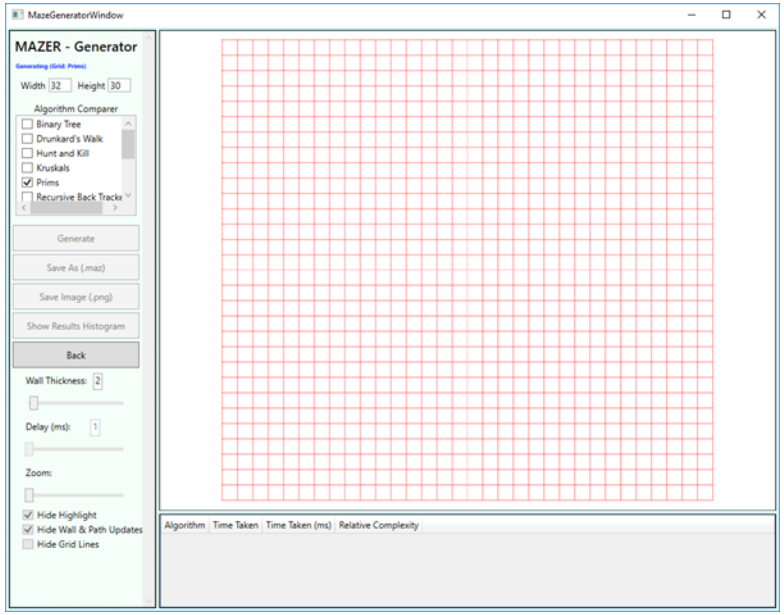
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 16		
		Hide Highlight and Hide Wall and Path Updates On	

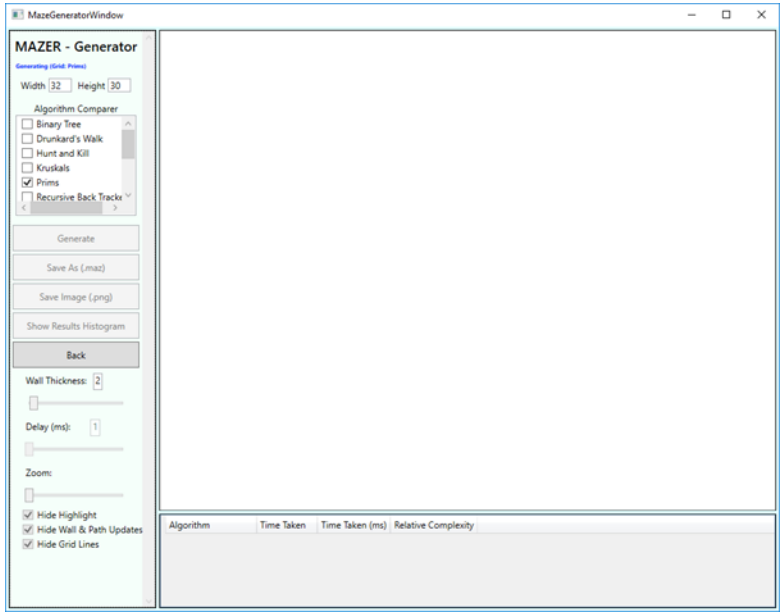
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 17		
		All On (While Generating)	

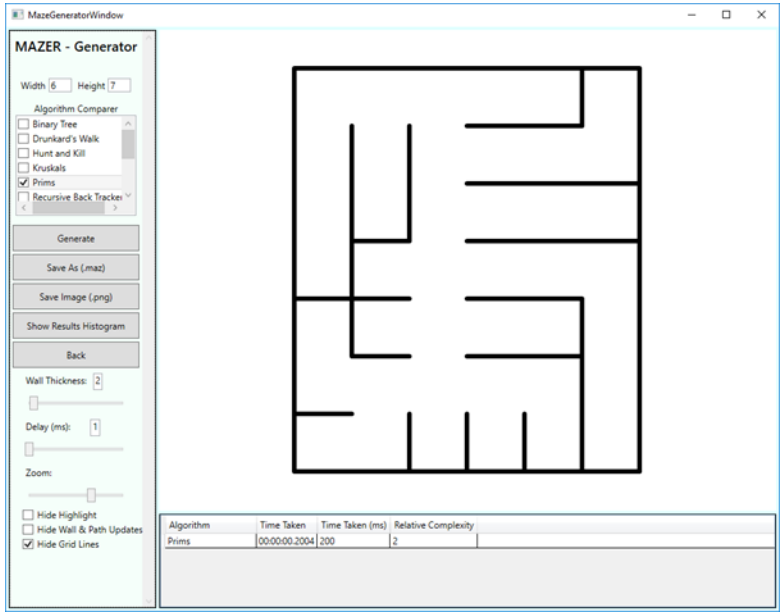
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 18		
		Zoom Works	

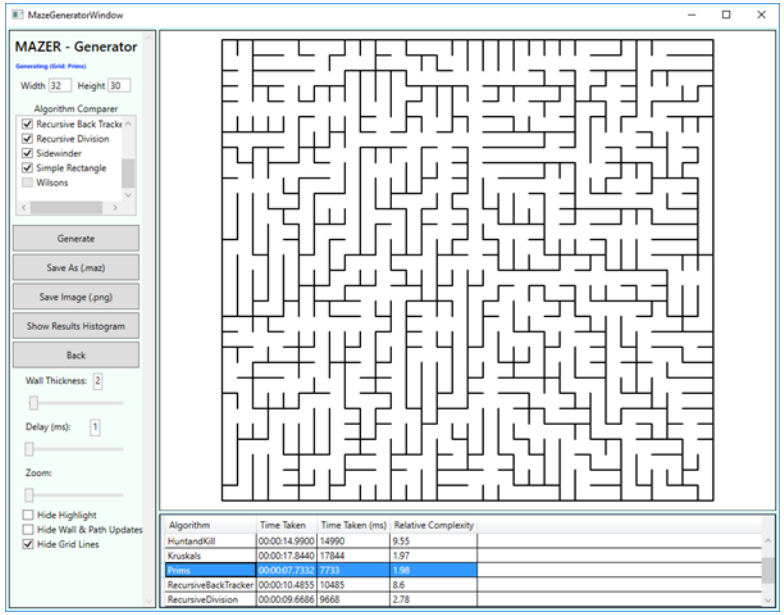
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
Generation: Algorithm Clicked Displays Corresponding Maze	Figure 19		

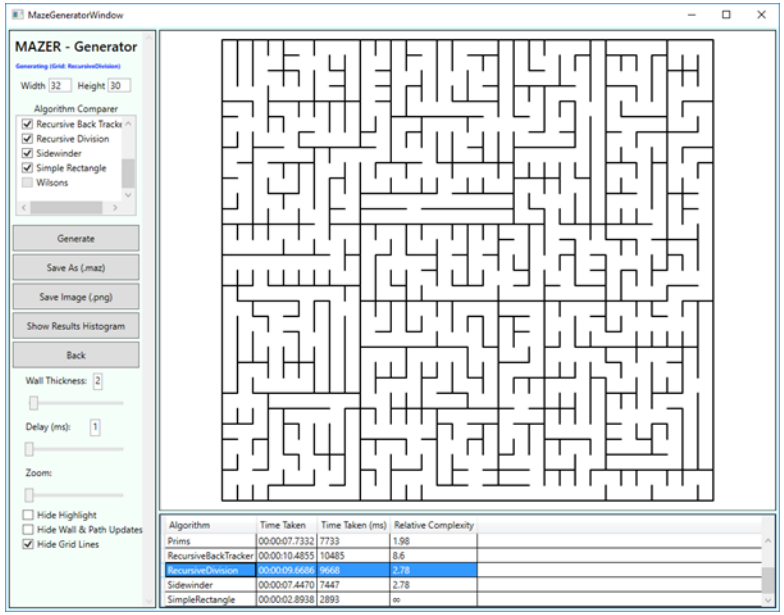
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 20		

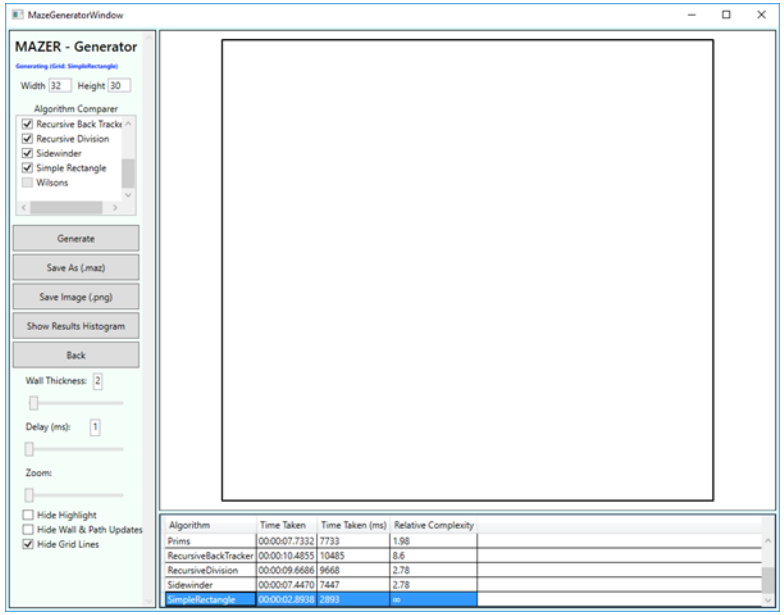
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 21		

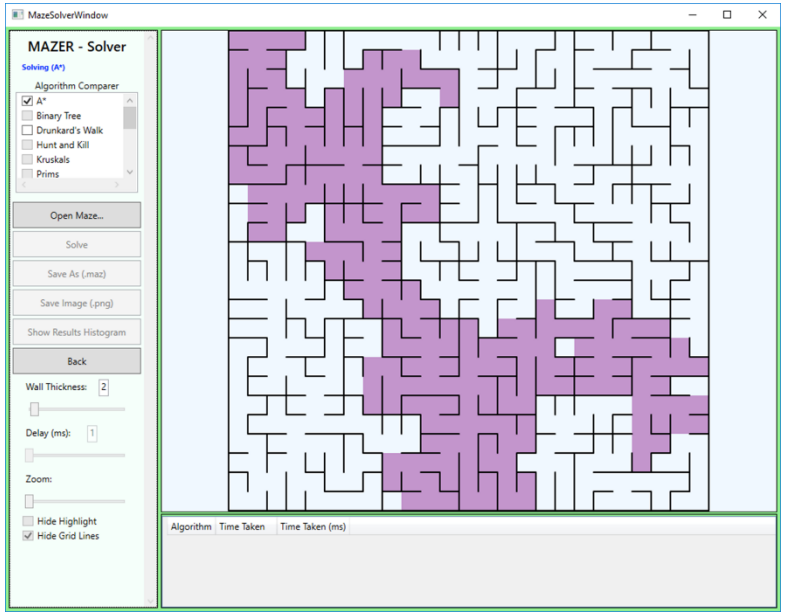
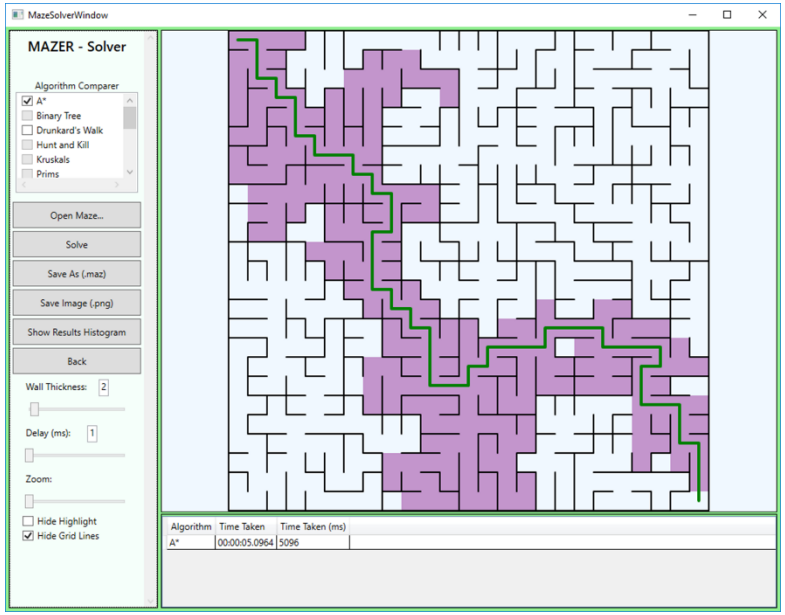
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
Solver: Algorithms on Canvas	Figure 22		

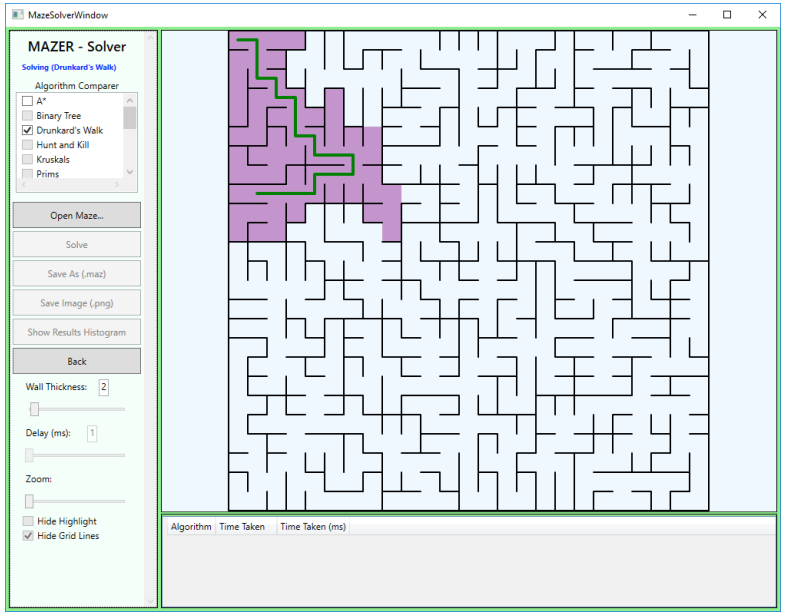
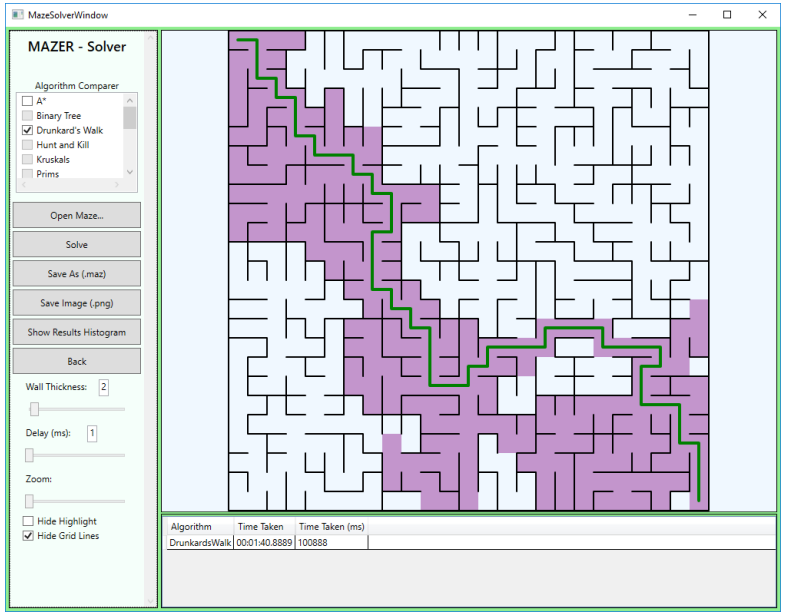
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 23		

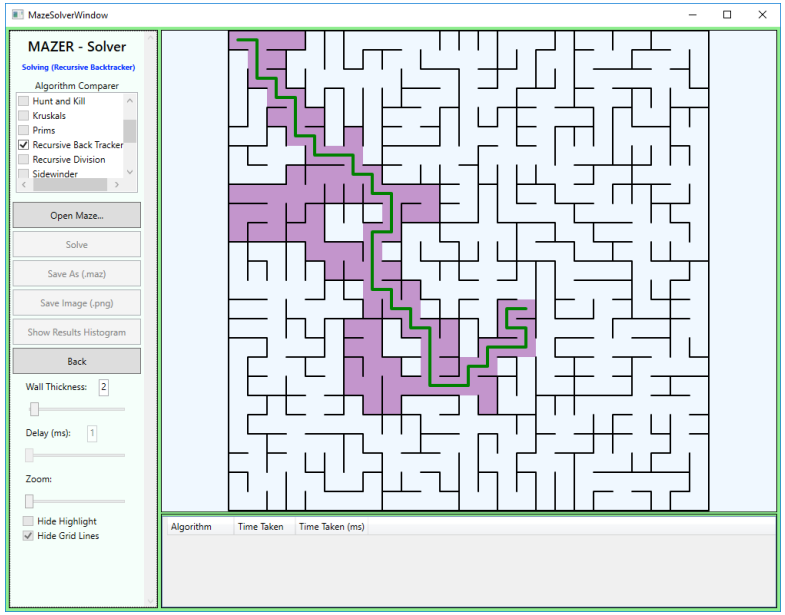
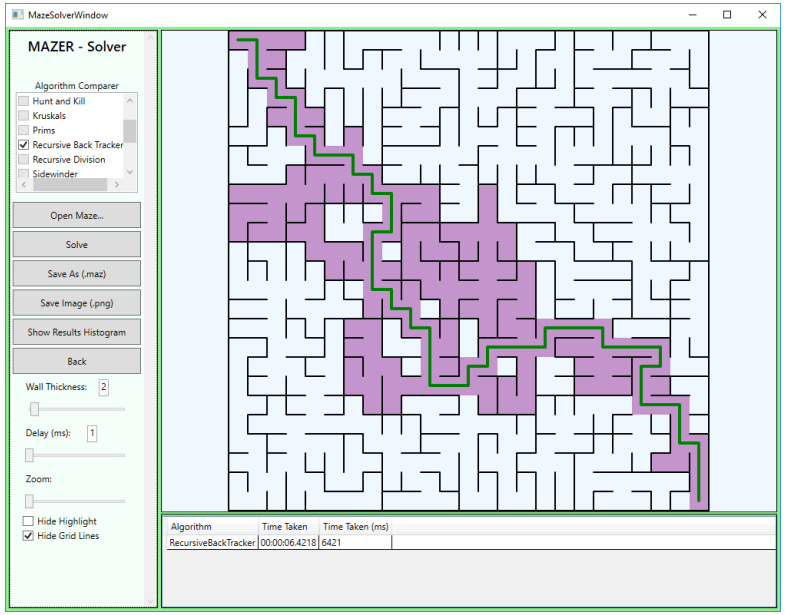
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 24		

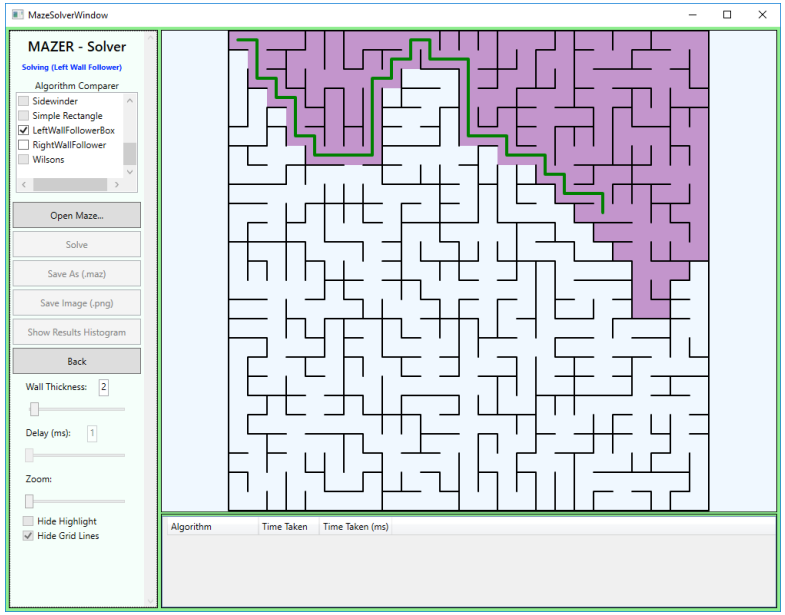
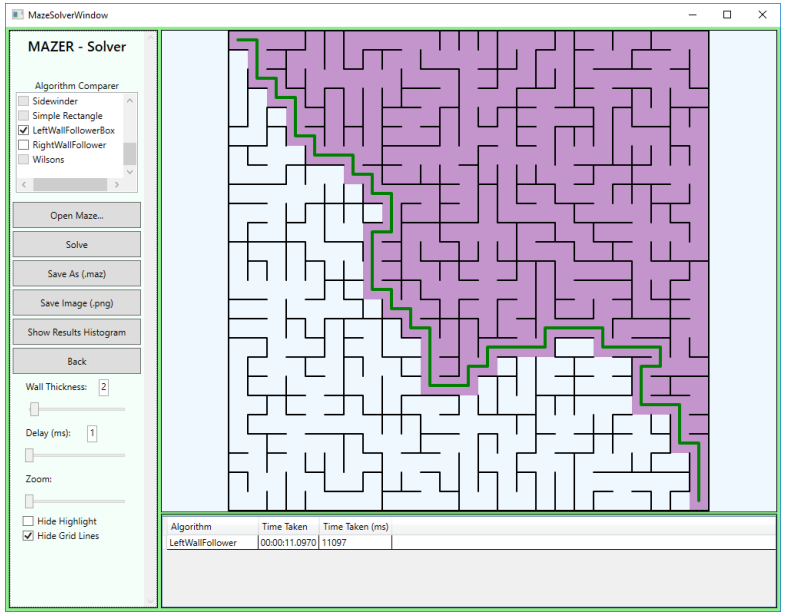
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 25		

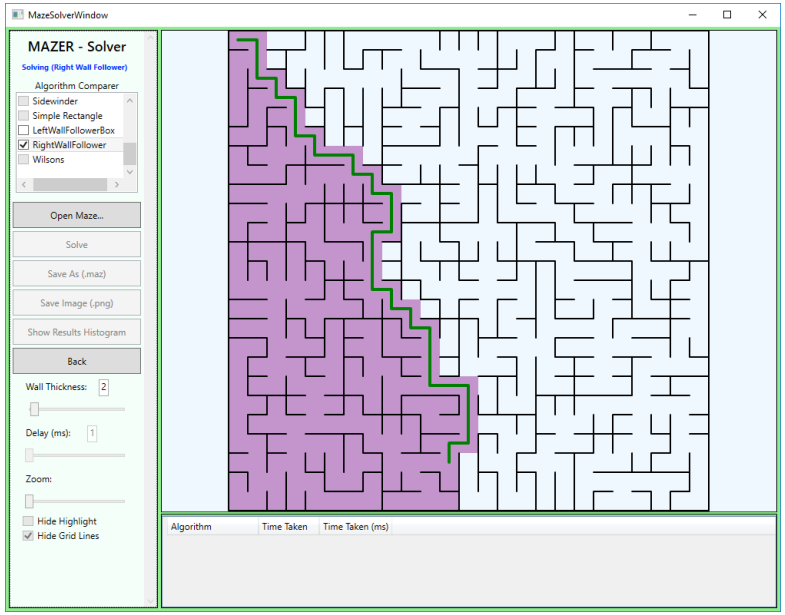
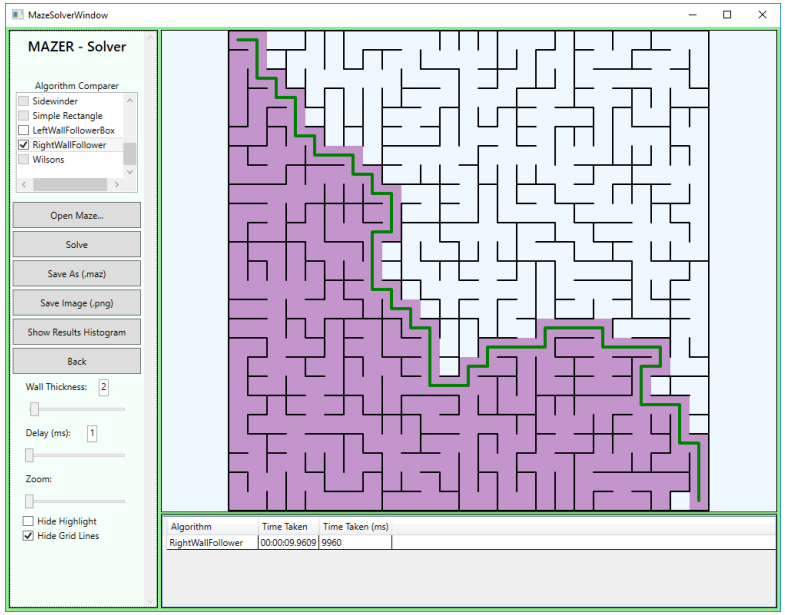
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 26		

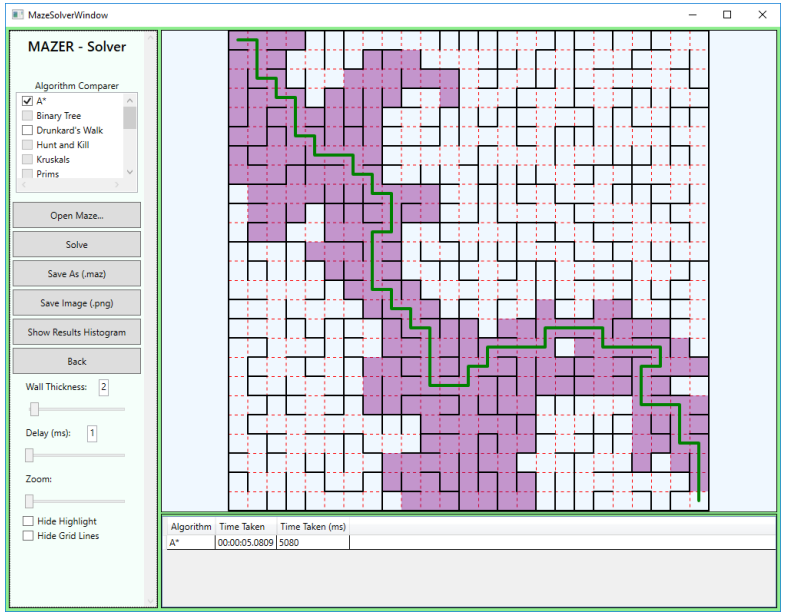
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
Solver: *Events, Extras and Zoom on Canvas*	Figure 27		

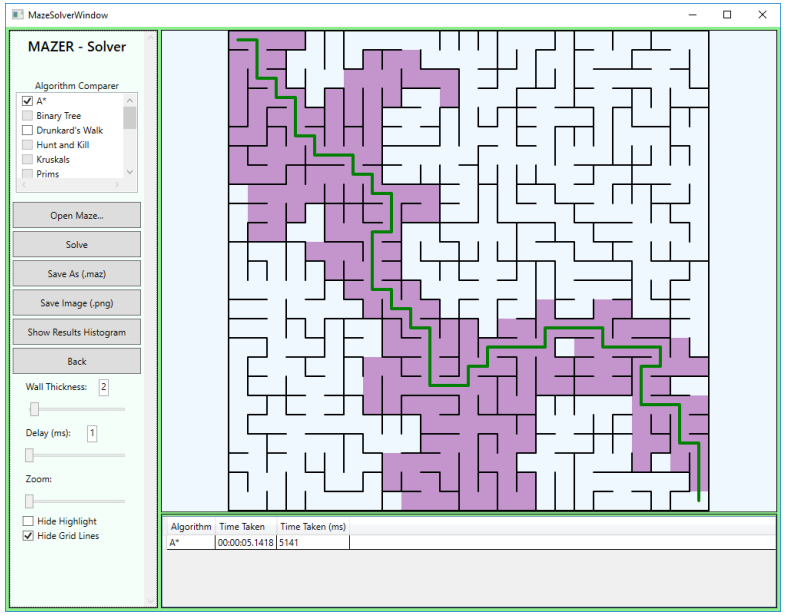
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 28		

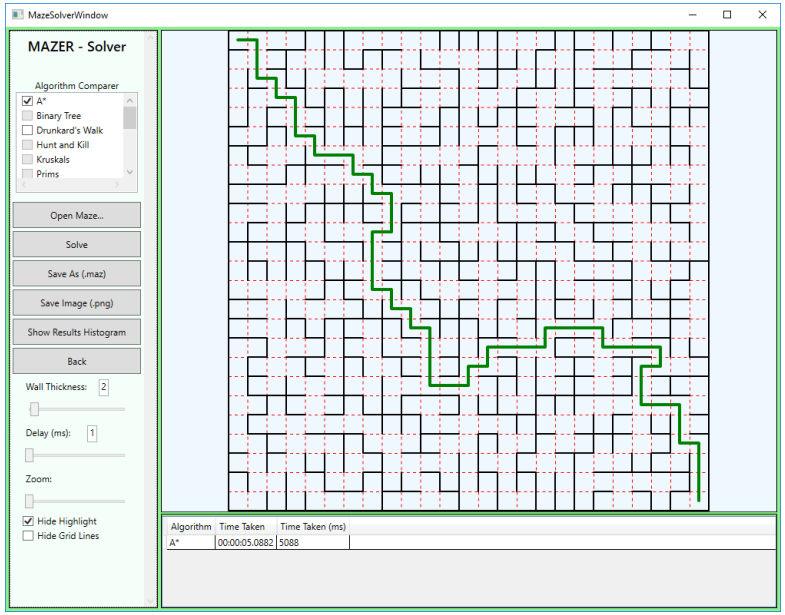
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 29		

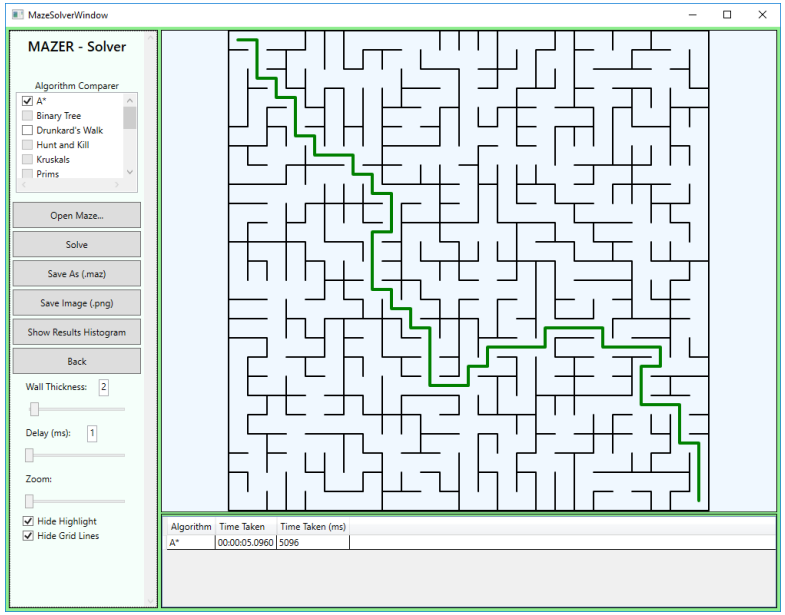
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 30		

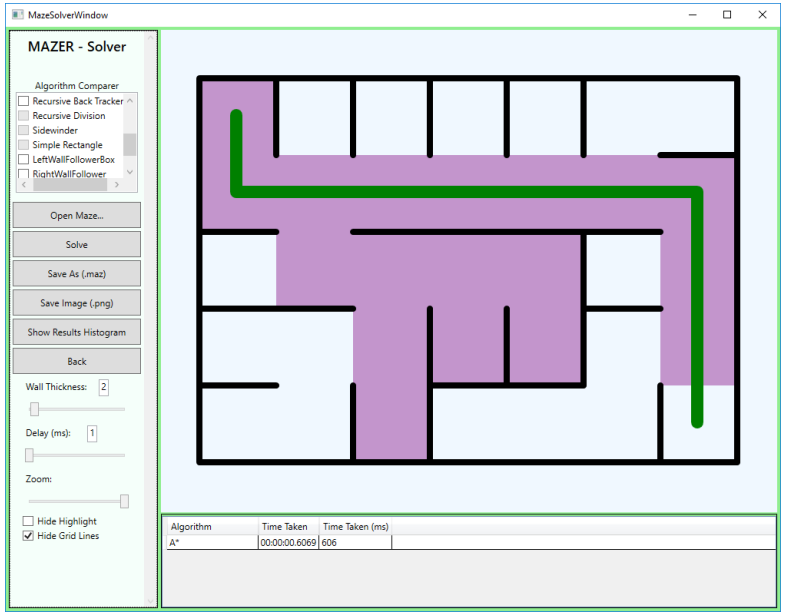
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
Solver: Algorithm Clicked Displays Corresponding Maze	Figure 31		

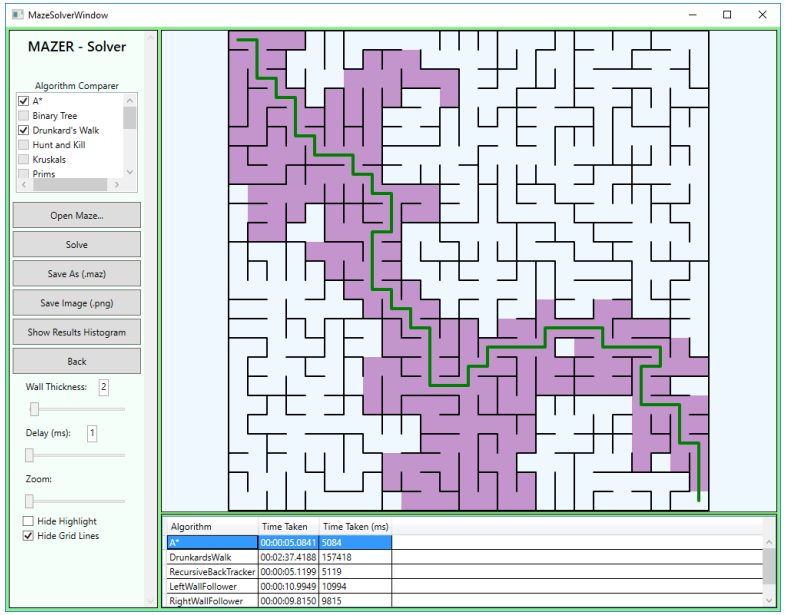
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 32		

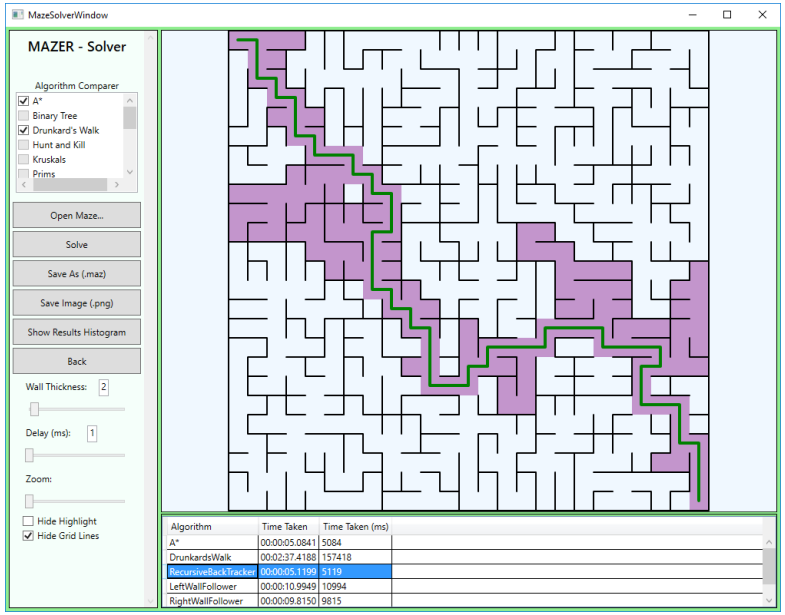
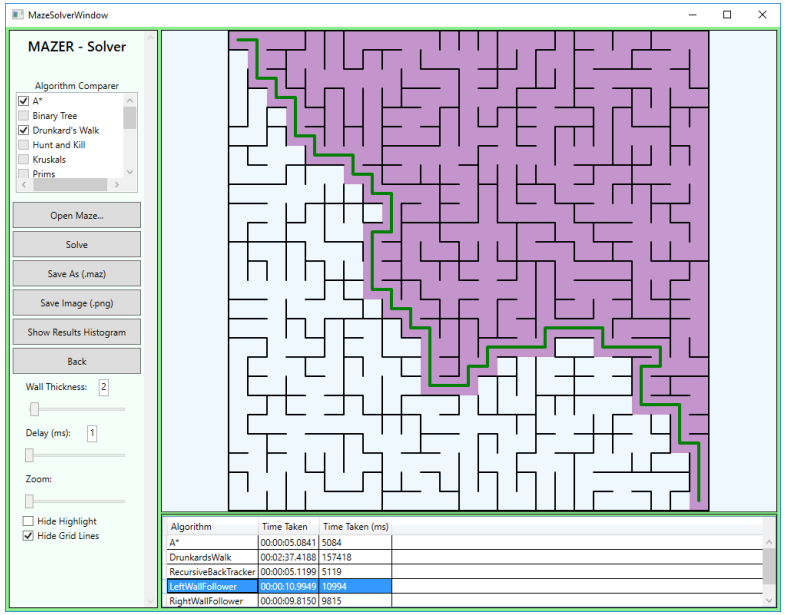
Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 33		

Table Reference	Figure	While Generating / Solving	Once Generated / Solved
	Figure 34		

Other Manual UI Tests

Generating

Button	Functions correctly	Evidence	Successful
Save As (.Maz)	Yes	Figure 34	Success
Save Image (.png)	Yes	Figure 34	Success
Show Results Histogram	Yes	Figure 34	Success

Solving

Button	Functions correctly	Evidence	Successful
Save Image (.png)	Yes	Figure 34	Success
Show Results Histogram	Yes	Figure 34	Success

Figures

The following table gives images of the mazes and their solutions created during the manual UI testing in the previous sections.

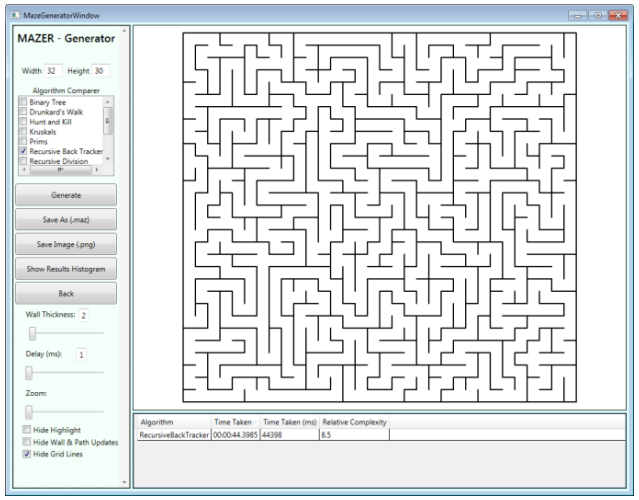
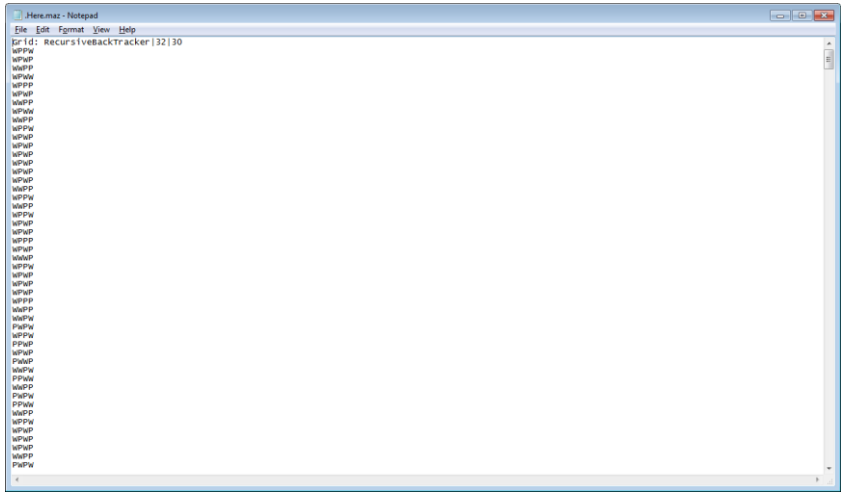
Figure	Evidence
Figure 35	 

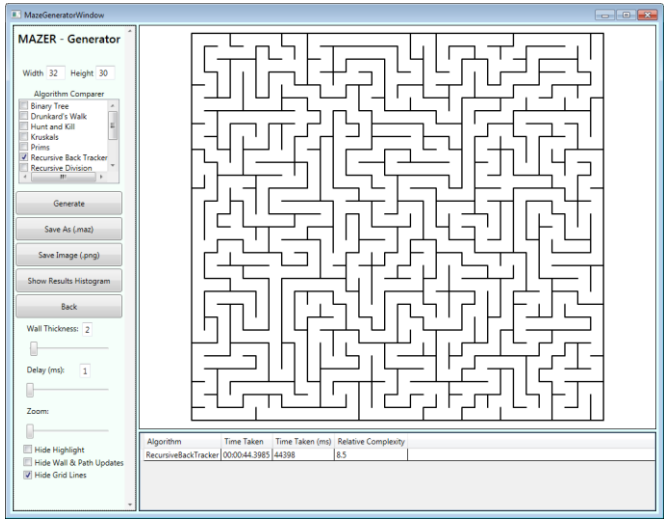
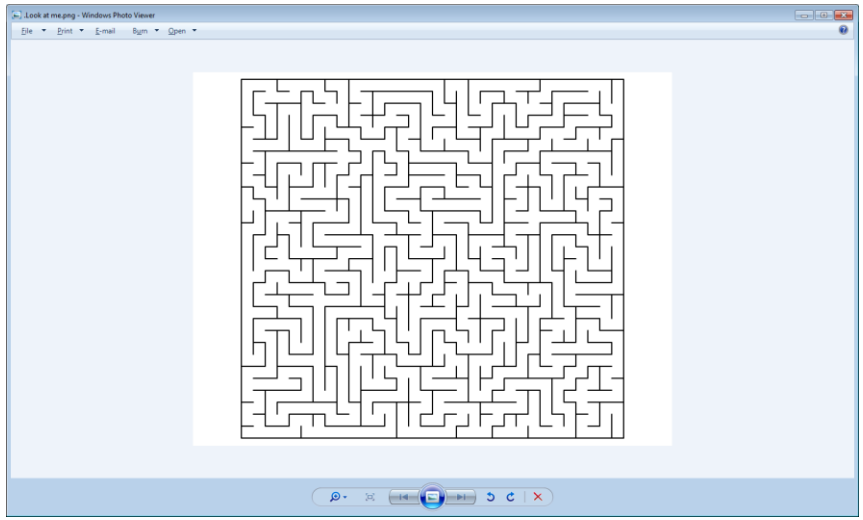
Figure	Evidence
Figure 36	 

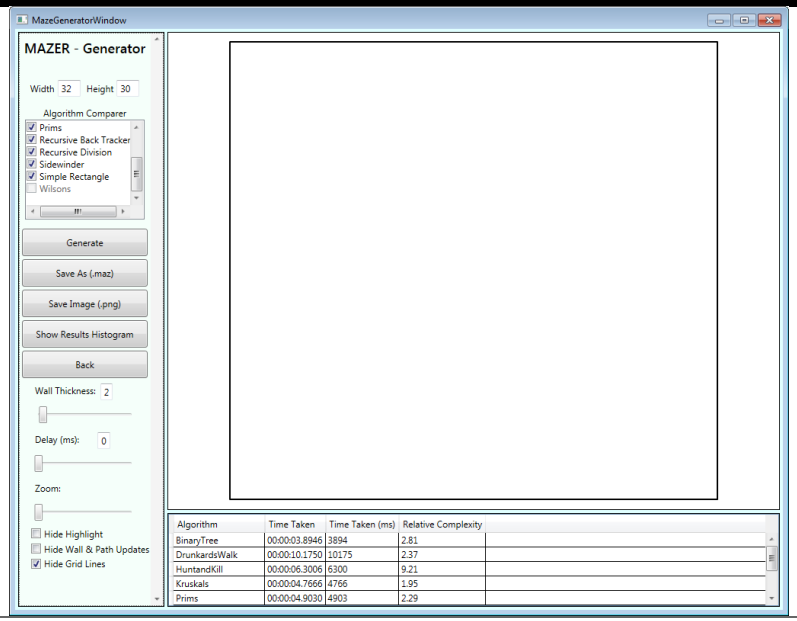
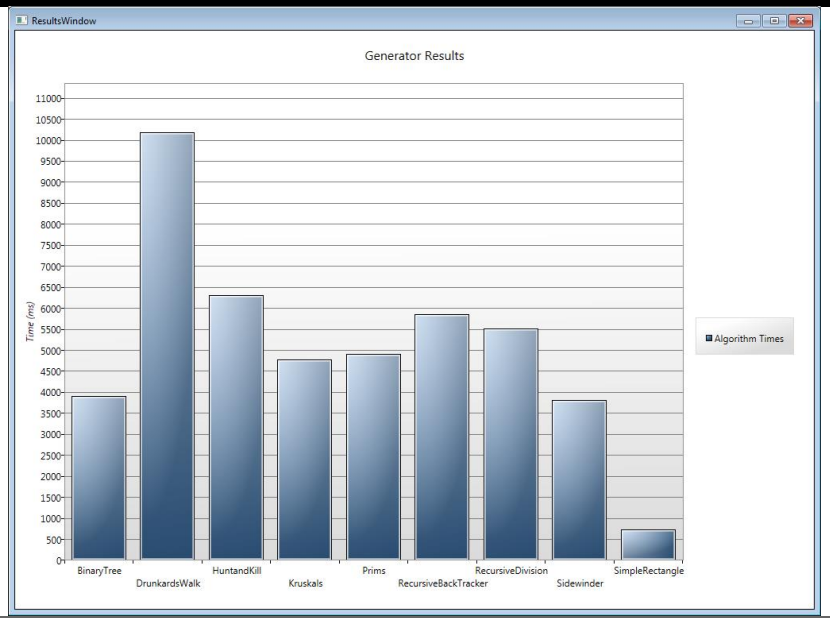
Figure	Evidence
	 

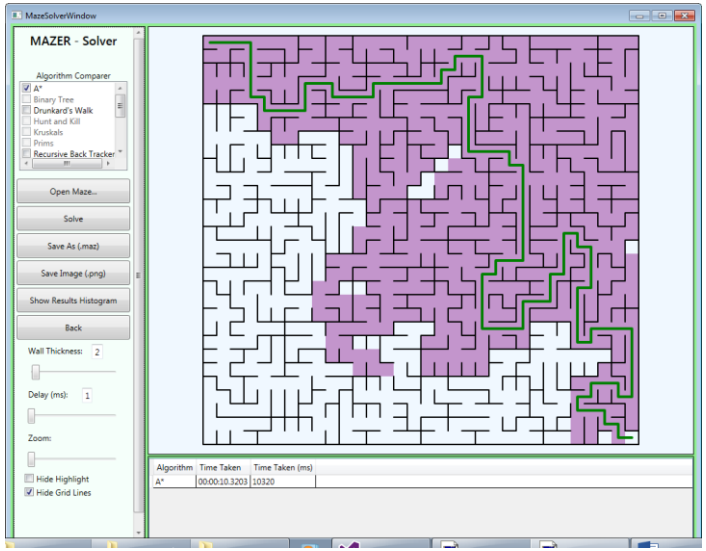
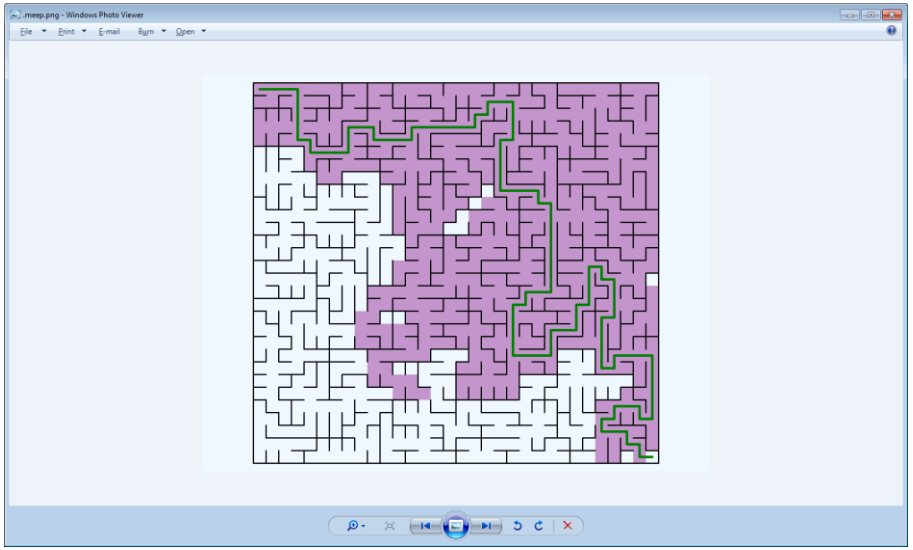
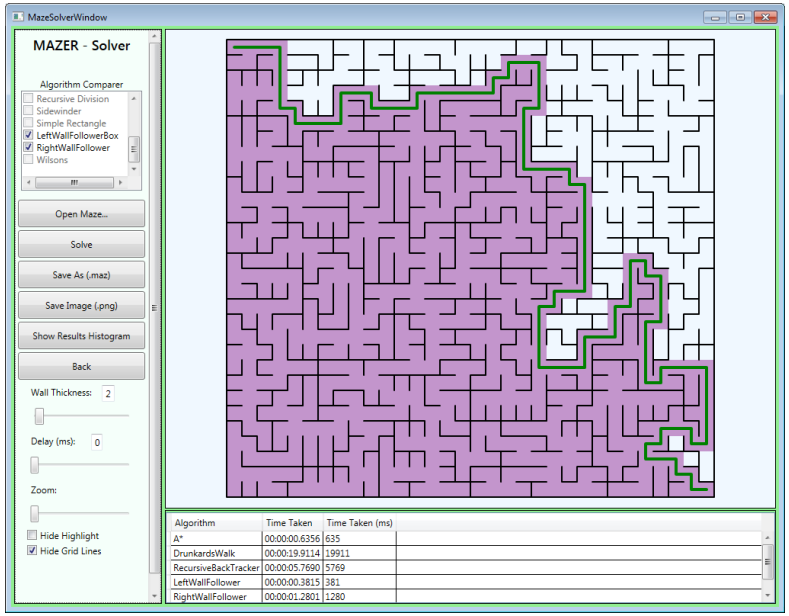
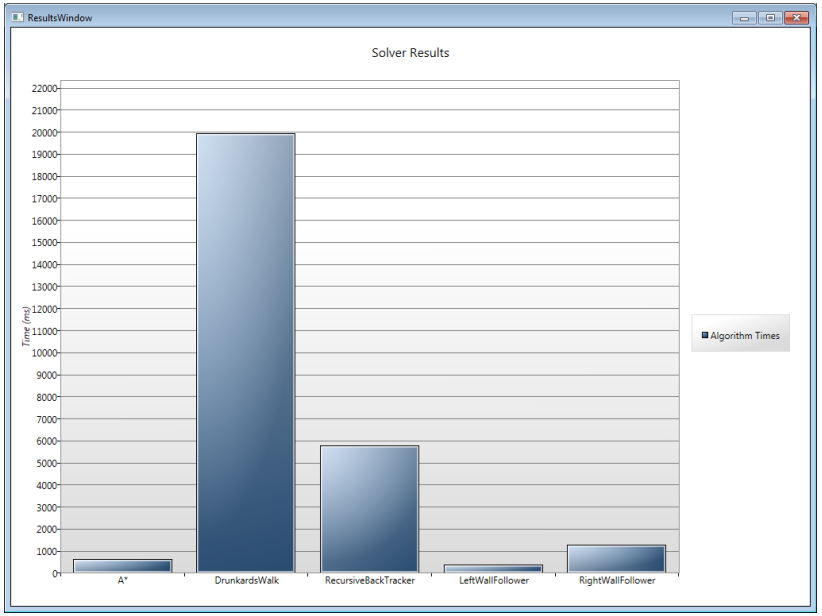
Figure	Evidence
	 

Figure	Evidence
	 

Evaluation

In this section, I will address what has been successful with my program, such as the ways that it has met or expanded beyond the objectives set in the Analysis. I will also gauge any improvements that could take the solution further and also reason if the solution as a whole meets the function I originally designed it for.

Meeting Objectives

The following section provides an assessment of the final application against the original objectives that were produced in the analysis phase of this project.

Objective Number	Description	Success or Failure	Justification and Discussion
1	Creation of a user interface, which includes the creation of a start-up menu, a maze creation dialog window and a maze solver dialog window. Should include a simple way to select algorithms as well as any other required buttons or UI elements needed by the various windows.	Success	The creation of the user interface was completed in WPF and had all the required buttons to run successfully. Solution also has a user-friendly method of selecting one or many algorithms to use and displays information clearly. The solution has several extra features not mentioned in the original objectives, but are still worthwhile for the benefit of the user. Extra features include a zooming capability for smaller mazes, a slider for changing wall widths and checkboxes that can change how mazes and paths are displayed.
2	Inputs into the user interface must be validated to ensure smooth running of the solution. Inputs such as the entering of maze size, for example, must take only integer values and should also be validated for appropriate maximum and minimum values.	Success	Inputs such as maze sizes are validated by allowing only certain inputs for the text boxes, such as the numbers or delete keys being pressed. Opening the maze is validated if a maze cannot be loaded it aborts the action. **Try Catch Exception handling** is used.
3	All algorithms written must produce undirected tree graph for generation, and produce a clear route from a designated start to a designated end for solving.	Success	All algorithms produce the required structure when generating, and by running tests I have found that they all produce a path from a start (0, 0) to the end (MazeWidth, MazeHeight).
4	The generation of a maze should be shown on screen, perhaps updating after each new addition to the maze. This should be clear and flow smoothly without screen flicker or pausing background processes.	Success	By checking specific checkboxes the mazes can be displayed updating after each new action is completed when the highlight path changes or all at once. This is achieved through the use of **multi-threading** and **event handling**.

Objective Number	Description	Success or Failure	Justification and Discussion
5	All algorithms must not require extensive time to generate or solve mazes. This only applies when an algorithm takes more than a minute for a small grid size (e.g. 10 by 10).	Success	All algorithms run effectively not requiring an extreme time for a trivial problem. Drunkard's Walk does occasionally take a long time to run, but this is more a fault of how the algorithm works rather than its implementation in the solution. Code makes extensive use of **object-oriented** techniques such as **Polymorphism**, **Inheritance** and **Encapsulation** to remain maintainable and fast.
6	All algorithms should, within reason, be fair. Realistically some algorithms may apply methods speeding up processes before or after algorithms create or solve a maze, where others don't, placing them at an unfair advantage. This should be avoided where possible.	Mostly Success	In most cases, this objective is successfully met. All algorithms are timed the same way ensuring a fair test in that regard. The only cause for worry I will address in the Suggested Improvements section later.
7	The program must calculate the time taken to create and solve the mazes. It should be capable of creating a histogram (either directly in the application or possibly in graph in Microsoft's Excel) which can be viewed in the application itself.	Success	All results are output in a table format and can be loaded into a graph on a popup user dialog. The times taken for each algorithm are displayed in a histogram directly in the application.
8	The solution must be able to save mazes generated as a bitmap and also be able to open these maze bitmap files in the application.	Success	All mazes and solved mazes can be saved as an Image File in png format or into a Text File as a .MAZ file. Solved mazes are saved by the application but these serve no purpose as the solution is not read back in. I will address this issue in the Suggested Improvements section later.
9	The solution should display a complexity value for the mazes that it creates.	Success	The application displays its calculated complexity value for any maze that it creates.

The Results

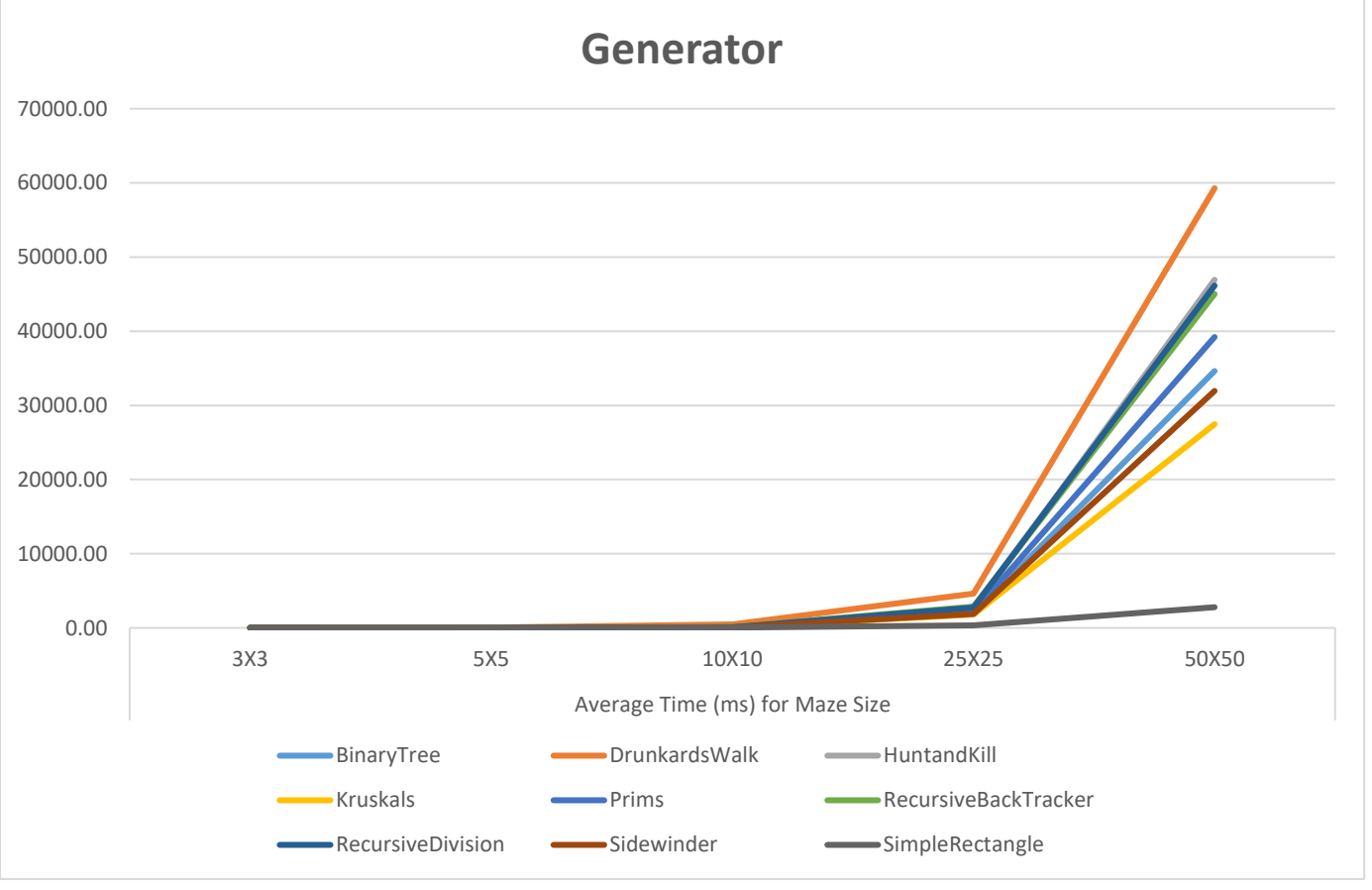
Results were gathered on each algorithm after they were executed. The graphed results have also been produced in the sections below along with a discussion of the meaning.

Generator

I ran each generation algorithm creating mazes of the following sizes and recorded the times taken. This was repeated 3 times for each in order to get an average time for each algorithm.

	Average Time (ms) for Maze Size				
Algorithm Name	3X3	5X5	10X10	25X25	50X50
BinaryTree	9.67	21.67	97.67	1841.67	34637.00
DrunkardsWalk	23.00	61.67	457.67	4601.33	59279.33
HuntandKill	5.67	10.33	94.33	2475.00	46940.33
Kruskals	5.67	10.67	76.00	1803.00	27463.67
Prims	5.00	21.00	73.67	2136.67	39217.00
RecursiveBackTracker	7.33	24.33	123.67	2877.33	45018.33
RecursiveDivision	4.67	11.00	87.67	2746.67	46148.33
Sidewinder	4.67	12.33	75.67	1864.00	31951.00
SimpleRectangle	3.33	7.33	26.33	333.33	2788.00

As is shown in this graph, the most efficient algorithm for generating mazes was Kruskals. This is followed by Sidewinder, Binary Tree and Prims. Surprisingly the Hunt and Kill, Recursive Backtracker and Recursive Division algorithms were much slower than these. The Drunkard's Walk was the slowest algorithm.



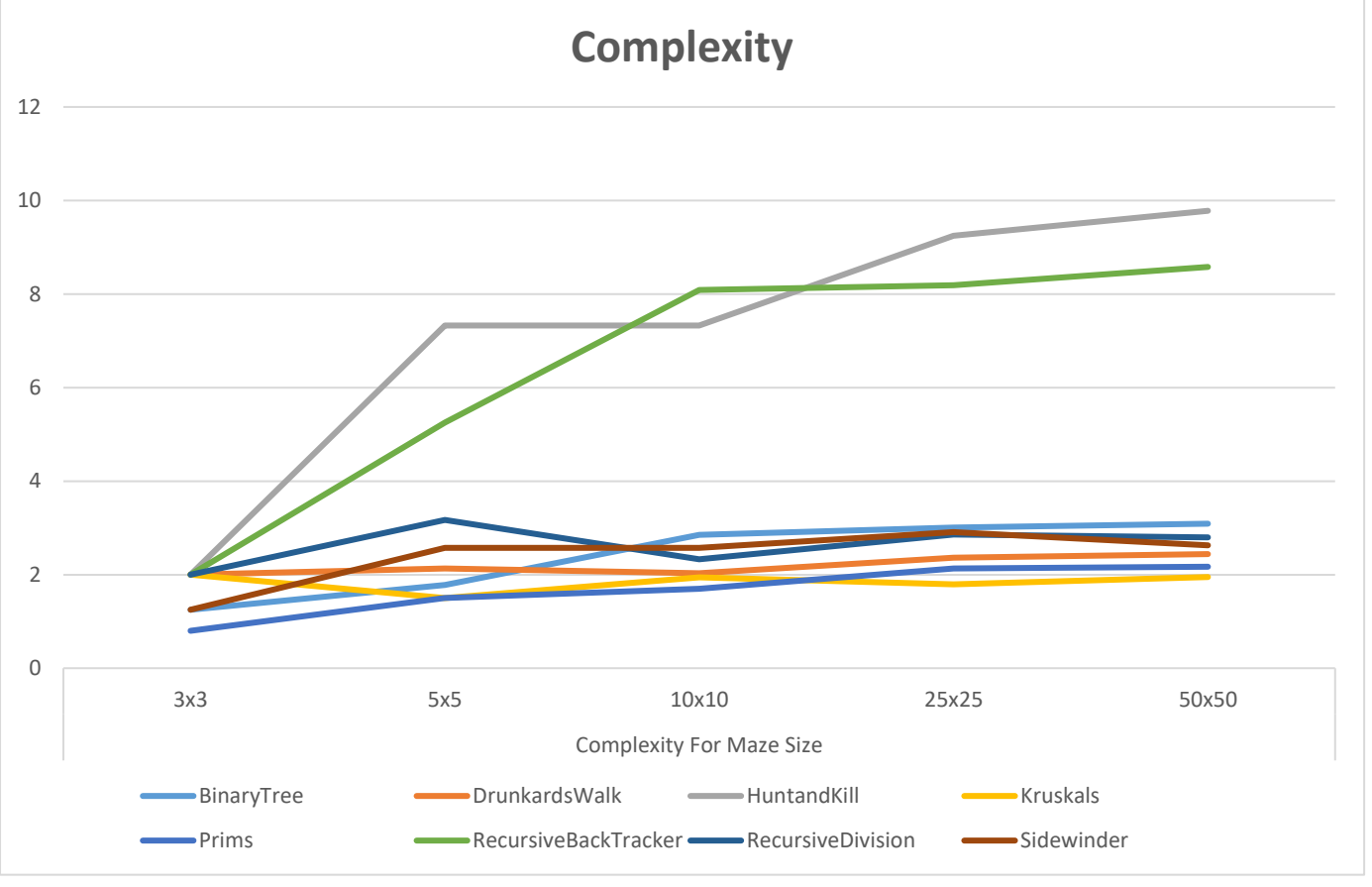
These results show that for algorithms creating small mazes the results are very similar. However, for the generation of very large mazes the less complex algorithms, such as Binary Tree and Kruskals, take a far shorter time. Clearly, the Drunkards Walk comes out as the slowest due to it being far too dependent on a random factor.

Complexity

I ran each generation algorithm creating mazes of the following sizes and recorded the times taken. This was repeated 3 times for each in order to get an average complexity value for each algorithm.

	Complexity				
Algorithm Name	3x3	5x5	10x10	25x25	50x50
BinaryTree	1.25	1.78	2.85	3.01	3.09
DrunkardsWalk	2	2.13	2.03	2.36	2.44
HuntandKill	2	7.33	7.33	9.25	9.78
Kruskals	2	1.5	1.94	1.79	1.95
Prims	0.8	1.5	1.7	2.13	2.17
RecursiveBackTracker	2	5.25	8.09	8.19	8.58
RecursiveDivision	2	3.17	2.33	2.86	2.8
Sidewinder	1.25	2.57	2.57	2.91	2.63
SimpleRectangle	N/A	N/A	N/A	N/A	N/A

In these results, the most effective algorithms for creating complex mazes was both the Hunt and Kill and the Recursive Backtracker algorithms. All other algorithms fell far behind, but with the Recursive Division algorithm coming in as the third most complex.



In comparison with the Generation times, the Complexity Values appear to show the opposite outcome. The slowest algorithms for generation, bar the Drunkards Walk, have produced the most complex mazes with the fastest algorithms producing the least complex mazes.

This might suggest an inverse correlation between time to create a maze and the complexity of the finished result.

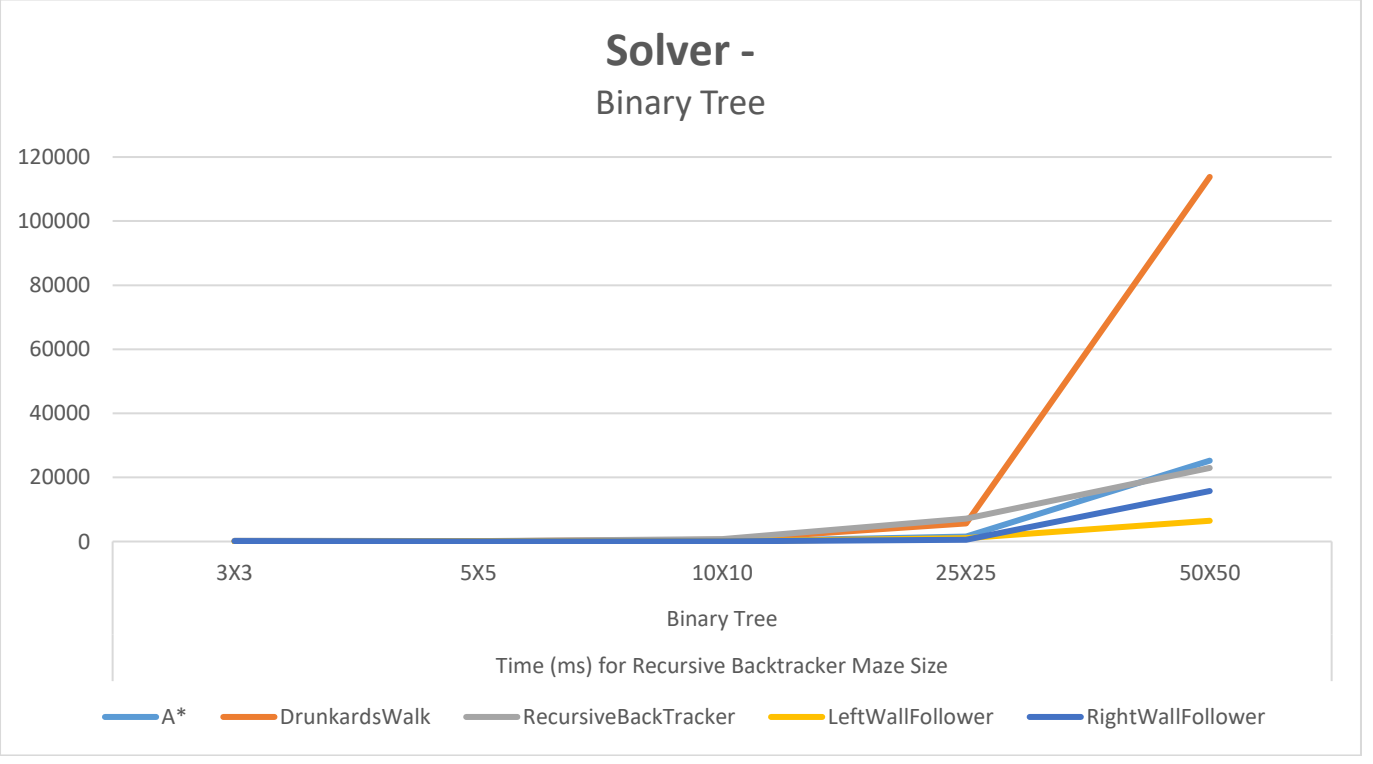
Solver

For solving I ran each solver on mazes created from three of the generation algorithms mazes. This allows the solvers to be measured against mazes of varying complexity. Each solver algorithm was run 3 times for each maze size in order to get an average time for each algorithm.

Binary Tree Mazes

	Time (ms) for Recursive Backtracker Maze Size Binary Tree				
Algorithm Name	3X3	5X5	10X10	25X25	50X50
A*	6	14	92	1573	25232
DrunkardsWalk	117	84	572	5644	113792
RecursiveBackTracker	241	129	783	7182	22963
LeftWallFollower	3	14	63	1099	6474
RightWallFollower	126	5	67	529	15760

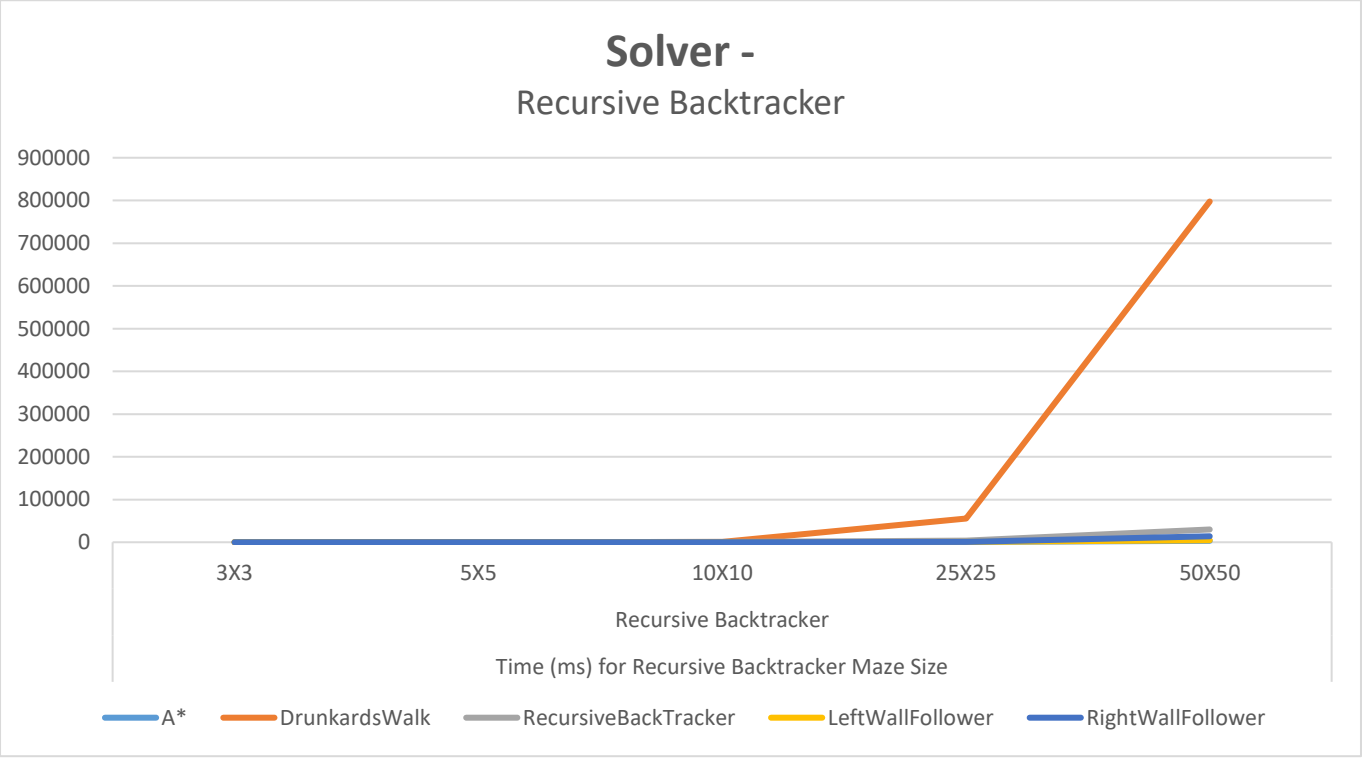
As is shown the most efficient solvers were the Wall Follower Algorithms. These were followed by the A*, the Recursive Backtracker and finally the Drunkard's Walk.



Recursive Backtracker Mazes

	Time (ms) for Recursive Backtracker Maze Size				
	Recursive Backtracker				
Algorithm	3X3	5X5	10X10	25X25	50X50
A*	1	9	43	890	3264
DrunkardsWalk	9	14	433	55270	797599
RecursiveBackTracker	51	153	821	4129	29860
LeftWallFollower	2	4	32	748	4774
RightWallFollower	2	5	47	748	13987

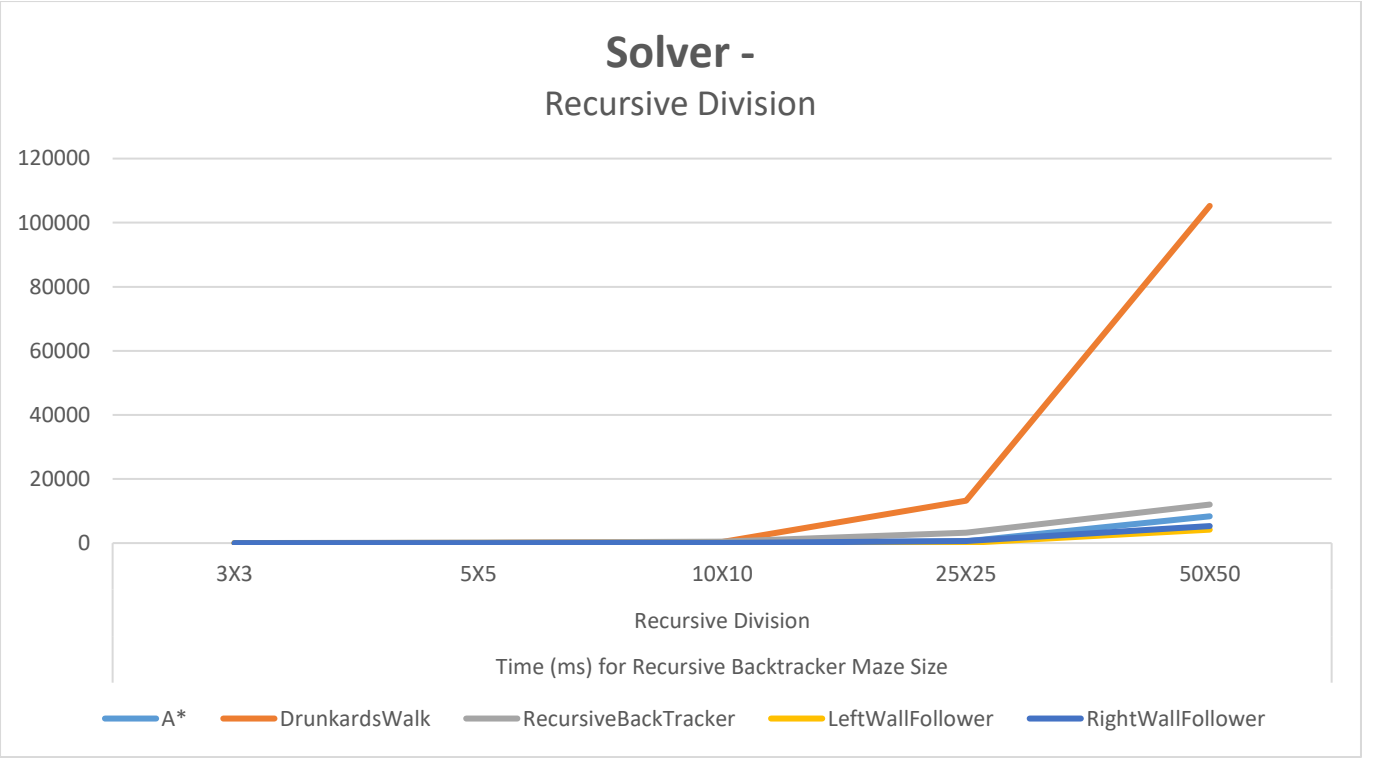
In this case, all algorithms were very efficient in solving the mazes but the most efficient was the A* followed closely by the Wall Followers and then the Backtracker. Again the Drunkard's Walk was the slowest.



Recursive Division Mazes

Algorithm	Time (ms) for Recursive Backtracker Maze Size Recursive Division 3X3	5X5	10X10	25X25	50X50
A*	2	4	28	502	8359
DrunkardsWalk	1	29	391	13257	105233
RecursiveBackTracker	10	113	418	3284	12027
LeftWallFollower	3	7	31	77	4194
RightWallFollower	3	3	36	685	5305

The Drunkard's Walk again took the longest time. The Recursive backtracker the second longest, the A*, and then the Wall Followers being fastest.



Suggested Improvements

To further the effectiveness of the Mazer application there are a few areas that could have been extended and developed further.

One such area is the use of the Save Maze button for the Solver Window. This was described in the Design. The reason I removed this button from the Solver was that there was no clear reason why a user should be able to load an already created and solved maze back into the application. I felt that this would add nothing to the solution. I could have instead created a new window which would allow multiple previously saved mazes in to be loaded and compared. Using such a feature, I could have saved the time to solve the maze in the .MAZ file. This would then be a base value for comparison with the next group of selected algorithms to run. This process would have opened up the opportunity to compare already solved maze times with ones that have not been done yet, gaining further insight on how efficient the maze's solver algorithms are.

Another area that could have been improved is the assurance that every algorithm was being run fairly compared to any other. An example of this is in the creation of borders on the generation algorithms, some algorithms create all the borders at the beginning such as with the Sidewinder algorithm whereas others create it as they come to the border like with any of the Drunkard's Walk based algorithms. I haven't been able to test whether this difference actually gives an unfair advantage to one set over the other and have generally decided to ignore for now. Given more time I would remove this difference making all algorithms completely comparable.

The last improvement that I can foresee is that of adding more algorithms to be run for both the generating and the solving of mazes. One example of this is the addition of Wilson's algorithm to the generation of mazes which I believed would be possible during the Analyse phase. Since working on it, however, I discovered that the algorithms expected structure wasn't in anyway easy to replicate in my chosen solution, as can be seen by the commented-out code under the Wilsons Generation title in the Implementation. The reason for the failure in the implementation of the algorithm was due, in part, to it requiring the storage of a direction for where the generator is moving next within the cell it is currently at. This allows a path to be stored for later retrieval. The classes in my Core would need extending a little in order to handle this. Wilson's was unable to be completed because of this.

Conclusion

I have found that the most efficient algorithm for generating complex mazes was the Recursive Backtracker. The others while fast, produced inferior mazes that were far more trivial to solve.

The Most efficient algorithms for solving mazes, as my results showed, were the Wall Follower algorithms. The A* algorithm has also proven to be very efficient. The A* algorithms performance suggests that it is the most efficient algorithm in solving mazes and probably graphs in general outside this project.

It is also worth considering that one of the wall follower algorithms would always perform well since if the left gets a very long path producing a long time, the right would get a far shorter path producing a small time on the same maze.

In conclusion, my produced solution has shown which algorithms answer the investigation I started. The solution has met all the targets put to it and has fully implemented most if not all of the desired features as set out in the Analysis.

It has allowed me to come to a decision regarding what I think of the is most efficient algorithm for generating and solving mazes. These are the Recursive Backtracker for maze creation and A* algorithm for solving.