Liverpool John Moores University
School of Computer Science and Mathematics

6200COMP Project

Final Year Dissertation

submitted by

Luke Curran

958598

Computer Science

Solving Procedurally Generated 3D Mazes

Supervised by

Reino Niskanen

Submitted on

19 April 2024

Abstract

This dissertation aim is to study and analyse the process of procedurally generating and solving 3D mazes through this I can help determine which algorithms are most compatible with one another. This topic has become more relevant because of its uses in modern technology like GPS and self-driving cars. There is a consensus that some algorithms work better than other but this is under different circumstances I want to be able to definitively say that "this" parameter will lead to "this" algorithm being the most efficient. Throughout this dissertation I have done background research and literature reviews. In the next chapter is the requirements analysis and methodologies in which I discuss how I plan on completing the project and the artefact and hardware and software requirements needed to complete the project and artefact. After that I went over the design of the artefact which you'll find things like flowcharts and UML class diagrams and use case diagrams as well as pseudocode. Then my implementation chapter where I show the artefact and how that artefact works and the output of it and then the results chapter. My findings are that it depends on what you want. What I mean by this is that some maze generation algorithms excel in making bigger mazes than others. Others make more complex or interesting looking mazes that and for the solving in general from worst to best in terms of time and space it would go Trémaux, backtracking solver, shortest path, and then random mouse but again this is if you care about time and space because although Trémaux might be fastest it won't always find the best path possible whereas shortest path will. Then for the mazes it goes from best to worst in terms of time and space to solve it goes binary tree, backtracking generator, Ellers then Wilsons.

ACKNOWLEDGEMENT

I would like to thank Dr Reino Niskanen, my supervisor for this dissertation, for his valuable comments, suggestions, and time he spent helping me with this work. I would also like to thank my family for moral support.

Table of Contents

Table of Figures

Introduction

Mazes have been around for a very long time. Mazes are believed to originate from the labyrinth from ancient Crete and then further developed in Venice. Now their main purpose is mostly for fun and entertainment purposes. As the education level has been rising worldwide, mazes have been used as fun educational resources. This has also led to mazes being explored in a much more mathematical way. Because of this exploration, mazes have been applied to real life problems like GPS, self-driving cars, and urban planning. This new need for mazes in a lot of technology has led to a need for fast and easy maze generation and solving this problem has been accomplished by automating the process using computer algorithms and simulations. That is why this topic is so relevant and needs to be discussed as without these algorithms many pieces of everyday technology would not exist. That is why I am discussing and explaining this topic in this dissertation.

The aim of this dissertation is to summarize several maze generation and solving algorithms and to evaluate the time and space each solving algorithm takes to solve each procedurally generated maze. The first part of this paper will present the concept of mazes, the history of mazes, there uses and the categorization of mazes. This will be followed by maze generation and solving techniques, graph theory and more basic concepts relevant to mazes such as trees and spanning trees. This will be followed by the artefact design which will explain how the algorithms work and will also predefine how I intend my artefact to end up being this will be done through UML diagrams, flowcharts, and pseudocode. Then my implementation chapter that will show the process I went through to make the artefact and explain the end result. Next will be my evaluation in which I will evaluate the results I will collect from my artefact. Finally finishing with an overall evaluation of the project along with possible future work for the project.

Solving both problems can be done by humans with no problem except for the fact that if you were to do this for a maze that has the dimensions of 100 x 100 x 3 it would take a long time to make the maze and to solve it. That is why this problem is best solved using computers as it will take computers a fraction of the time it would take a human to make and solve the mazes and it does also get rid of human error as a human might double back on themselves when solving a computer wouldn’t do that.

Introduction into Mazes

In this chapter I will talk about the concept of mazes, the history of mazes their real life uses and the categorization of mazes. This will be followed by maze generation and solving techniques, graph theory and more basic concepts relevant to mazes such as trees and spanning trees.

Maze concepts

Mazes are a type of tour puzzle. Tour puzzle is a type of puzzle where a player uses a token to traverse around a board. There are differences in tour puzzles but with a maze the only rule is that you need to start in one location and find your way to a the designated "end" location. Other rules may be that the player needs to go to a specific location, or they might be able to use one or more tokens to solve it. Mazes can be different in terms of size, shape, and difficulty. This will be explained more in the coming chapters.

The History of Mazes

Many people believe that mazes originate from ancient Crete from the labyrinth. Whilst some others say that there is evidence that the ancient Egyptians invented the maze so there is no absolute origin for the maze but there are some possible starting points. Labyrinths and mazes are different from one another. A labyrinth is curved passages, forming a unicursal, or one-way path from the outside toward the centre whereas a maze has dead end and junctions meaning there are multiple paths to take (Matthews, 1970 #2). People do have theories in why mazes were built in the past some believe that there are certain spiritual and religious reasons for why mazes were built in the past as many have been found near religious sights and burial grounds. But now mazes are largely used for entertainment and fun. This will be expanded upon in the next section.

Current applications of mazes and solving them

In more modern time mazes have been used as physical attractions for example life size mazes that people have to traverse themselves can be found around the world during festival and other events a good example. The world's largest maze was built in Dixon, California that measured over 40.5 acres. They have also found their way into pop culture into films like The Shinning (1980) and Labyrinth (1986). They are also used for fun educational exercises to help kids think logically. Now that technology has been constantly improving and so has education people have now found a technological uses for mazes. Maze solving can now be used to solve some more modern problems for example mazes help to solve problems with GPS as the goal of GPS is to get someone to a location in the shortest amount of time possible and by travelling shorter distances. Mazes are also used in the video game industry a lot as games need random interiors to a building this can be procedurally generated using maze generation algorithms. Other uses include urban planning, art, VR and AR games and autonomous vehicles etc.

Categorizing mazes

Mazes have been categorized by many people in many ways but one way that has been well cited and quoted is the work done by Walter D Pullen (Pullen, 2022 #3). According to him there are seven categories that a maze will fall into:

1. Dimension
2. Hyper-dimension
3. Topology
4. Tessellation
5. Routing
6. Texture
7. Focus

These categories will usually have major influences on a couple of factors such as time to solve, difficulty, fun, enjoyment. Obviously, these factors are only influential on human solvers and these factors can be different depending on the person solving the maze. These factors probably won't affect a computer-based solver in solving the maze they just might affect things like solve time and space.

Dimension

Although the idea of dimensions is quite simple it is best to define it to understand it properly. The oxford dictionary defines dimensions as "measurable or spatial extent of any kind, as length, breadth, thickness, area, volume; measurement, measure, magnitude, size" {Oxford, 2023 #4).This dissertation will focus on 3D mazes but one way to imagine 3D mazes is by "stacking" 2D mazes on top of each other whilst also allowing the movement of the player or solver to go up or down in the mazes.

As 3D mazes are really just 2D mazes on top of one another the main problem I will face in this is figuring out how to stack the multiple mazes on top of each other. There is also something called weave mazes or 2.5D mazes. This is where passages of the maze overlap each other allowing the player to quickly get to another part of the maze a good real-life example of this is to imagine a bridge that will connect one portion of the maze to another. Weave mazes never allow the change of dimension when in one cell but only when the move along the entire passage does the dimension technically change.

You can have higher dimension mazes (4D or higher), but it is quite difficult to fully explain but a way to describe is 3D mazes with "portals" to travel through the 4th dimension. But as 3D mazes are the focus this is not important (Pullen, 2022 #3).

Hyperdimension

Hyperdimension refers to the dimension of the object you move through the maze not the maze itself. There are 3 types of hyper dimensions non-hyper maze, hyper maze, and hyper hyper maze. A non-hyper maze can be found in pretty much all mazes even those with higher dimensions or special rules this is because non-hyper mazes happen when you use something that moves from point to point and the path behind forms a single line and the choices made at each point are easily countable. When talking about hyper mazes the object you move through the maze with is a line which when you move it through the maze it leaves behind a plane which is a 2-dimensional object. Hyper hyper mazes happen you move through a 4D, and the object increases in dimension again leaving behind a solid object. This dissertation will focus on non-hyper mazes. Hyper mazes and hyper hyper mazes are not relevant in this topic. Hyperdimensional maze can be seen in figure 1.

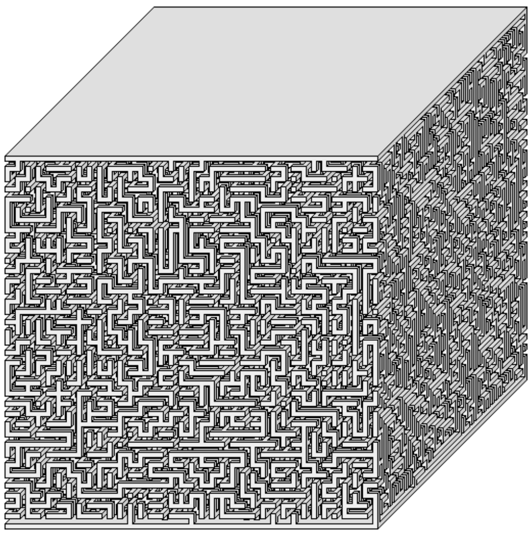


Figure 1 example of a hypermaze.

Topology

Topology refers to the geometry of the space the maze as a whole exists in. There are two types of topologies these are normal and planair. Normal topology refers to a maze that exists in Euclidean space (Britannica, 2009 #5). Planair refers to any maze with an abnormal topology. For example, interconnected maze on the surface of a cube or a moebius strip. As there is no need for me to do planair topology I will implement a normal topology in my artefact.

Tessellation

Tessellation refers to the geometry of the individual cells that make up the maze. The types of tessellations are:

- Orthogonal- This is the standard for most mazes. The cells in this maze are made up of square and rectangles meaning that each cell has the chance to have four neighbouring cells and four walls and passage intersect at right angles.
- Delta- This is composed of interlocking triangles and each cell can have up to three passages connected to it.
- Sigma- This is composed of interlocking hexagons and each cell can have up to six passages connected to it.
- Theta- These mazes are composed of concentric circles of passages; the start or end can be in the middle and then the other is on the outer edge of the maze. The cells usually have four passage connections, but they can have even more if there a large number of cells in the outer rings of the maze.
- Upsilon- Upsilon mazes are made up of octagons and squares. Each cell may have up to eight or four passages.

- Zeta- Zeta mazes are on rectangular grids but unlike other tessellations 45-degree angle passage are allowed.
- Crack- A crack maze is a maze with no consistent tessellation meaning that has walls or passages at random angles.
- Fractal- A fractal maze is a maze that is made up of different mazes that are then put together to make on big maze.

As tessellation will not be of much importance to me my maze will most likely follow an orthogonal tessellation. As this is the most basic and most used tessellation. But using other tessellations could produce different results.

Routing

Routing refers to the type of passage a maze has. This can sometimes help humans to determine the method of creation of the maze as some of the passages are a lot more distinctive than others. The types of routing are:

- Perfect- a perfect maze is one where there is only one passage that is the solution to the maze. It also does not have any loops or closed circuits and it doesn't have any inaccessible areas.
- Braid- braid mazes have no dead ends instead of this the passage will coil around each other and go back into each other this will make the solver go in circles. Walter D Pullen claims that if you have a braid maze the same size as a perfect maze the braid maze will be more difficult (Pullen, 2022 #3).
- Unicursal- a unicursal maze does not have any junctions and is one continuous passage. This is also known as a labyrinth. I have discussed this earlier in the chapter.
- Sparseness- A sparse maze happens when not all the cells are used to make a passage leaving them out. This means these cells will then become walls in a passage leading to some varying sizes in passages.
- Partial braid- this maze will have loops and dead ends. This again will lead to an increase of difficulty.

With Routing some generation algorithms will unintentionally lead to bias in the maze. For my mazes, some algorithms will naturally have different routings and are more likely to make certain types than other algorithms.

Texture

Texture refers to the style of the passage. The passages tend to follow certain themes. The types of textures are:

- Bias- Bias means that the passages tend to go more horizontally or vertically. This means that if there are long horizontal passage there are short vertical passages and vice versa.
- Run- run means that passages in a maze will go in a continuous direction for more than a minimum of 4 cells if they have any less then it is a short run.
- Elite- mazes can be described as elitist. The level of elitism is the measure of how short the direct solution of the maze is to the size of the rest of the maze. A maze can be non-elitist if the solution takes up most of the maze. Elitism can affect the difficulty of the maze.
- Symmetric- a symmetric maze has symmetric passages. Mazes don't have to be fully symmetrical to be classed as symmetric they can be partially symmetric to fit.

- River- when a maze has a higher river characteristic it means that when creating the maze, the algorithm will clear out the cells next to the current one being cleared this will happen until all the cells have been cleared. If a maze has less “river” it will lead to more dead ends that are short whereas more “river” leads to less dead ends, but they are longer.

Just like routing some generations algorithms will lead to bias for example sidewinder and binary tree which completely relies on a pre-determined bias. Getting a mix of river and elite mazes and bias mazes would be a nice thing to have as it could lead to more interesting results, but I will not forcibly implement this into the artefact I will only take it into consideration in my evaluation where I will briefly talk about it.

Focus

Focus refers to two types of maze creation: wall adders and passage carvers. This is only a difference when generating this cannot really be visually seen. Wall adders focus on the walls of the maze. This will happen when an empty area with boundaries is set out and then the algorithm will add walls. Whereas passage carver will get an area full of walls and then get rid of walls to make passage. Some algorithms only can do one or the other where others can do both depending on how they are implemented I don’t have any aim to implement one or the other but if there is an option to use either of them passage carver will most likely be picked.

Maze generation and solving

Generation techniques

For my artefact I will be implementing a 3D orthogonal maze type this means that the maze will be made up of squares and in these squares, there will be entrances to the different levels of the maze. This is because this type of maze is what is relevant to my dissertation unlike other types of mazes that I have talked about above such as braid and weave, but some algorithms may naturally lead to these types of mazes.

There are two types of maze generation types this being graph based and cellular automata. Graph based maze generation creates the mazes by building a spanning tree this is a type of graph that I will talk about further in the chapter. The tree can be made different ways depending on the algorithms that are used to generate the maze for example Prim's algorithm will take the root of the tree and continuously grow from that root or smaller trees will be joined together to make one bigger tree this can be found in Kruskal's algorithm. As already explained mazes are created using a wall adder and or a passage carver. Different algorithms will use different types but the are some algorithms that will be able to use one or the other Prim's algorithm is an example of this as well. Some algorithms encompassed in graph-based algorithms can be classed as simpler than others this can be in terms of how they work but it can be also in terms of how much each algorithm takes from memory to store the maze. In my artefact I will implement a mix of "simple" and "complicated" algorithms.

Cellular automata can be defined as "a collection of cells arranged in a grid of specified shape, such that each cell changes state as a function of time, according to a defined set of rules driven by the states of neighbouring cells" (Awati, 2021 #7). Cellular automata have two well-known rule string that will generate maze these being B3/S12345 and B3/S1234 these rules help to determine how longs cells "survive" for and also whether it can be "born" this will determine empty cells and walls in the maze this can be well demonstrated in Conway's game of life. Cellular automaton rules are deterministic so each maze start from a randomly generated pattern which makes it unique but because of these starting patterns they can be easily predictable before they are fully formed this can be seen as a big drawback as they follow such obvious patterns. For my artefact I will be using a graph-based generation method this is because of the wider variety of algorithms I can use to generate and solve the mazes. This will also help me get much more varied and interesting results.

When something is generated procedurally what this usually means is that data has been algorithmically created, this is through a set of pre-defined rules and some computer-generated randomness which creates something different each time. The algorithms I intend on using already have this as a part of how they work so I will not have to intentionally implement it.

Basic concepts

The next part of this chapter will present basic concepts on relevant parts of graph theory that need to be explained. This will not be very in depth it will cover topics such as trees, spanning trees and weighted graphs.

Graph Theory

Graph theory is said to first appear in Bridges of Königsberg by Leonhard Euler {Euler, 1956 #8}. In this paper they tried to solve the question that they were wondering if it was possible to start in any of the four land masses of the city and then cross all the bridges only once and then end up where you started. Instead of doing this in person multiple times with different variations Euler decided to simplify this into vertices and edges. (He did end up discovering this would be impossible). This was later expanded upon by the Vandermonde in the 'knight problem' paper and 'analysis situs' by Leibniz.

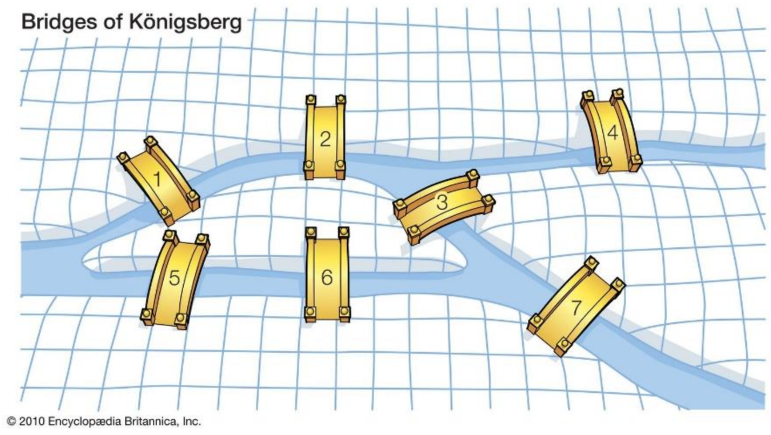


Figure 2 Seven Bridges of Königsberg problem.

Graphs

A graph is the most basic part of graph theory. In its most basic form, a graph is composed of two parts. A set of vertexes (V) and a set of edges (E) this can be simplified to G = (V, E) this is known as an undirected simple graph. The formula for edges is written as E ⊆ $\{\{x, y\} \mid x, y \in V \text{ and } x \neq y\}$ what this means is that the vertices {x, y} are the endpoints and the edges are incident this and the graph can have a vertex with no edges. Two or more edges cannot be connected to the same vertices. To counteract this an incidence function will fix this this will be written as $\phi: E \longrightarrow \{\{x, y\} \mid x, y \in V \text{ and } x \neq y\}$. This will map every edge to an unorder pair of vertices. This known as an undirected multigraph.

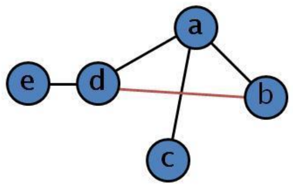
Graphs can also be directed meaning that the edges have orientations and the notation e = (x, y) describes and ordered pair and its orientation from x to y. Undirected graphs on the other has no order as the incident vertices are organised into sets. This is sufficient and will work in terms of maze generation because if one vertex is connected to more than two vertices this could be a problem in maze generation leading to paths being cut off. Graphs permit loops as well this means that a vertex can be connected to itself, but this would be pointless and unnecessary in maze generation.

Trees

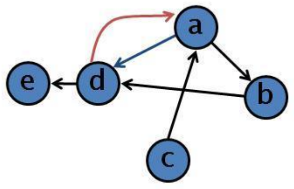
Trees are a type of graph. They are undirected in which any tow vertices are connected by only one path. A tree has n vertices and n-1 edges. Removing any one edge would break the tree whereas adding edges would create a cycle but if that was the case it would no longer be classed as a tree.

Spanning Tree

A spanning tree is a subset of graph G such that all the vertices are connected using minimum possible number of edges and it is also a subgraph of G this is where every edge in the tree belong to G. this can also be defined as the "maximal set of edges if G that contains no cycle, or as a minimal set of edges that connect all vertices". The number of spanning trees in graphs can be calculated using Kirchhoff's matrix-tree theorem (vaibhav, 2023 #9), (S Chaiken, 1977 #10). But this won't be necessary for this dissertation.



$$\begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 1 & 0 \\ b & 1 & 0 & 0 & 1 & 0 \\ c & 1 & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 0 & 0 & 1 \\ e & 0 & 0 & 0 & 1 & 0 \end{array}$$



$$\begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & 0 & 1 & 0 & 1 & 0 \\ b & 0 & 0 & 0 & 1 & 0 \\ c & 1 & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 0 & 0 & 1 \\ e & 0 & 0 & 0 & 0 & 0 \end{array}$$

Figure 3 Kirchhoff's matrix-tree.

Weighted Graphs

A graph can also be weighted. This means that each edge has a so-called weight this being a number. This allows for things like minimum spanning trees which is a spanning tree with its weight being less than or equal to all the other possible spanning tress of the graph. Although this is important for graphs in general it will not be important when creating mazes. But some algorithms use minimum spanning trees to generate their mazes. Although most of this aspect of graph theory have been presented separately, they all play an integral part in graph theory but as I have a stated only a certain amount of it is important in maze creation for example in the maze solving algorithm Dijkstra or generation algorithms like Kruskal's or Prim's.

Graphs in terms of Mazes

In this dissertation all my generation algorithms will be graph based but all of the algorithms will use graph theory in different ways in such as traversing and searching but all the basic logic of all the algorithms will be untouched. Mazes are made up of cells as stated already, in the tessellation section, they can be in many different shapes, but they are all made up of vertices and edges. This is how walls are made in mazes. Some parts of mazes though cannot be found in graphs this being entrance and exit cells, isolated cells, portals, crossings etc. If a maze was to be turned back into a graph these would be turned into vertices. Paths in a maze are made up of continuous edges and vertices. As I have stated previously, I will be following orthogonal tessellation for my maze this means it will be made up of hundreds of square shaped cells. A spanning tree of connected graph G is a tree composed of all vertices of G thus fulfilling the parameter. Any cell in the entire maze can be designated as the root. This will allow some algorithms like depth first search (DPS) can easily find its way through the spanning tree.

Solving Techniques

When solving a maze there are two categories, they can fall into those being known environment and unknown. The difference in these is quite self-explanatory in that the known environment algorithms is where the solver has prior knowledge of the maze this could be size, number of edges, the amount of junction and corners or distance between cells. Unknown environment algorithms have no knowledge of the maze they are about to solve (Shatha Alamri, 2021 #12). In this dissertation I will be using unknown environment algorithms, but I do think it is necessary for me to cover the known environment algorithms. Some of these algorithms are dead-end filling, maze routing, flood-fill, and lee's algorithm. Although these algorithms can work when they do not know the maze already, they work at their most optimal when they have knowledge for example when using flood fill, the solver will try and follow the best path possible but that path will constantly change as it will go down different paths following the most optimal route until it actually finds that route which then means it has to go again to make it more efficient as it now knows the best path. Many consider flood fill to be the best algorithm out of the known environment algorithms, in micro mouse competitions this is the algorithm used by most participants (Swati Mishra, 2008 #11). Because these algorithms have to go through the maze more than once to get the best answer and the unknown environment algorithms do not, that is why I have chosen not to use them. This is because if my artefact maze has large dimensions, it will take a very long time for them to be solved. Although some will not take a long-time, I know other algorithms will so there is no point in testing and evaluating the results.

Some Unknown environment algorithms are wall follower, Trémaux's, random mouse, pledge and Kruskal's. These algorithms vary widely in the way they work but they all do the same thing. But this means the results I will get from my artefact will be obtainable but also quite interesting to evaluate. I will talk more about more about these types of algorithms later in the chapter.

Graph based maze generation algorithms.

Binary tree algorithm

From Pullen's point of view, he believes that binary tree mazes are the "simplest and fastest algorithm possible however mazes produced by it have a very biased texture" (Pullen, 2022 #3). The way this works is:

1. Decide whether it will carve passages or add walls.
2. If it carves passages, go left or up in the grid.
3. If it adds walls, for each vertex add a wall leading down or right.

- You can't do both directions at once and if you reach the end of a row and can't go in a direction you go in the other direction.

Because each cell is independent from one another you don't have to hold any memory about the other cells when creating the other cells. This mean that this algorithm is memory less this also then means that you can create a maze with no worry about size limitations. This is called a binary tree maze because of how the passages turn out this can be best seen in figure 4 and 5.

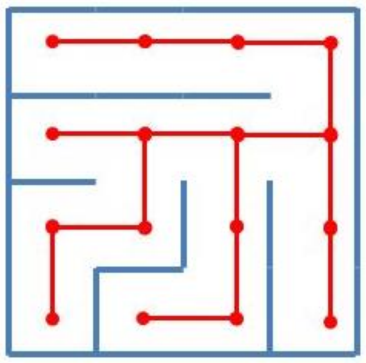


Figure 4 paths in a binary tree maze.

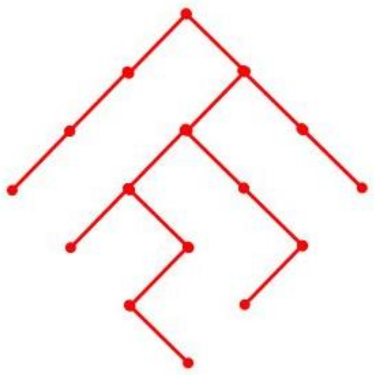


Figure 5 the passages after walls removed and rotated 45 degrees.

One major problem of this as described by Pullen is that there is a large amount of bias and this cannot be solved this is because the algorithm naturally causes it this isn't necessarily a problem though, because my aim is to evaluate the algorithms all as equally as possible but the difference in the algorithms is what will make the results more varied. This can be mitigated so that it is not so severe. This can be done by something called random walking which randomly decides which direction the algorithm goes if it is able to (Buck, 2011 #13).

Eller's algorithm

Pullen says that Eller's algorithm "is not only faster than all the others that don't have obvious biases or blemishes, but its creation is also the most memory efficient. The way this algorithm works is:

1. Initialize the cells of the first row to each exist in their own set.
2. Then randomly join adjacent cells but only if they are not in the same set. When joining adjacent cells merge the cell of both sets into a single set
3. For each set randomly create vertical connections downward to the next row. Each remaining set must have at least one vertical connection. The cells in the next row thus connected must share the set of the cell above them.
4. Flesh out the next row by putting any remaining cells into their own sets.
5. Then repeat until the last row is reached.
6. Once the last row is reached, join all adjacent cells that do not share a set and omit the vertical connections.

During the process once the algorithm is done with one set it doesn't have to store it in memory in more so it will just remove and then add then next set to its memory. Jamis buck says "The set that a cell belongs to tells the algorithm who its siblings were, are, and will be. It's the crystal ball that lets the algorithm gaze into the future (and the past!) and avoid adding cycles and isolates to the maze" (Buck, 2010 #14). Because we need every cell to be connected by the end so every set needs to be made into one big set that is why every set in a row has at least one vertical passage if this didn't happen there would be isolated cells. An example of the algorithm working can be seen in figure.

```
 ___________________
|           |       |
| 1   1   1 | 9   9 |
|    ___    |___    |
|       |       |   |
| 1   1 | 1   1 | 9 |
|___    |_______|   |
|   |   |           |
| 8 | 1 | 9   9   9 |
|   |   |___     ___|
         ___     ___
|   |   |   |   |   |
| 8 | 1 | 3 | 9 | 5 |
|___|___|___|___|___|
```

Figure 6 Eller's in progress.

```
 ___________________
|           |       |
|           |       |
|    ___    |___    |
|       |       |   |
|       |       |   |
|___    |_______|   |
|   |   |           |
|   |   |           |
|   |   |___     ___|
|                   |
|                   |
|___________________|
```

Figure 7 Eller's algorithm maze.

Wilson's algorithm

This is a more improved version of the Aldous Broder algorithm. This algorithm uses random walk which when completed will correspond to a spanning tree to be more exact a uniform spanning tree (UST). "A UST is any one of the possible spanning trees of a graph, selected randomly with equal possibility" says Jamis Buck (Buck, 2011 #15). This algorithm works like this:

1. Choose any vertex at random and add it to the UST.
2. Select any vertex that is not already in the UST and perform a random walk until it encounters a vertex that is in the UST.
3. Add the vertices and edges touched in the random walk to the UST.
4. Repeat steps 2 and 3 until all the vertices have been added to the UST.

One problem that all sources seem to talk about is that initially Wilson's algorithm is quite fast as it covers a lot of cells quickly, but it struggles to find the last couple of cells because of the random number generator in the algorithm which will mean that it takes a long time for it to fully finish the maze. Sometimes this algorithm will create loops this one way this can altered be done is using Darryl Nester's solution (Nester, 2018 #16). This involves finding the path from start to finish first and then removing the all the necessary walls and then if there are any remaining cells find the rest of them and fill them in this gets rid of loops as there will be one path that is correct in the maze so will most likely try and implement this later. You can see the algorithm working in figure 8.

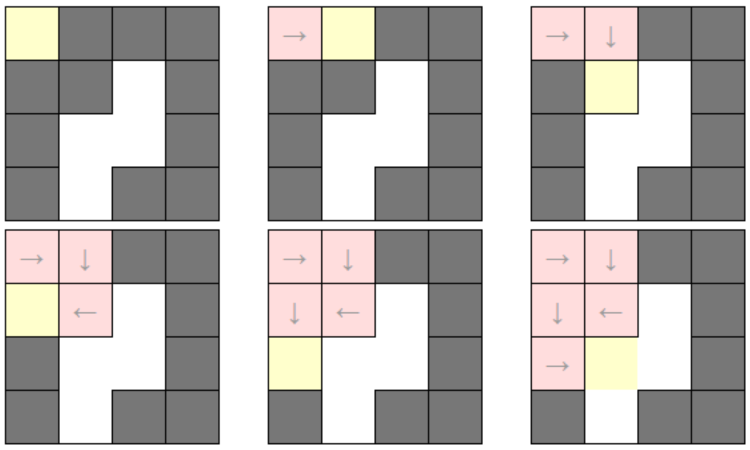


Figure 8 Wilson's algorithm in progress.

Backtracking Generator

Backtracking generator is based on depth first search (DFS). The DFS algorithm that the next visited vertex is a child of the previously visited one. If there are no children to move onto then the algorithm will backtrack and then check if that vertex is a new child. This finishes when all the vertices have been visited as the algorithm moves along it stores all visited edges and vertices which will lead to a spanning tree. This algorithm is different from the other algorithms in that only passage carving really works with this one as all the other algorithms I have mentioned already can use both wall adders and passage carvers. This is because if you did it in wall adder all the walls will be on the outside of the maze. This algorithm work like this:

1. Choose a starting point in the grid randomly.
2. Randomly choose a wall at that point and carve a passage through to the adjacent cell, but only if the adjacent cell has not been visited yet. This becomes the new current cell.

3. If all adjacent cells have been visited, back up to the last cell that has uncarved wall and repeat.
4. The algorithm ends when the process has backed all the way up to the starting point.

Walter D Pullen says that when storing the movement, the algorithm can use a stack and then every time the algorithm moves it can store it on the stack and then when you can't move anymore pop the stack and backtrack once all the stack has been popped the maze is completed. The maze creation can be seen in figure 9.

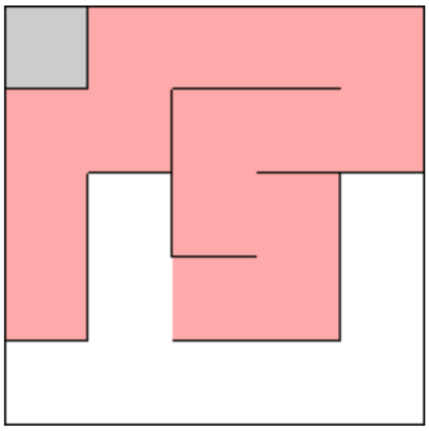


Figure 9 backtracking generator in progress.

Why these algorithms

The reason I have chosen to use these algorithms is that they all walk in very different ways from one another. Because of this I believe that this might make a large difference in the solving algorithms completion time and memory usage. As I have said, and others have said some of the algorithms I have chosen are more superior versions of existing algorithms or have at least tried to for example Wilson's algorithm is a superior version of Aldous-Broder algorithm as discovered from this paper (Gabrovšek, 2019 #18). In this paper the author explains various maze generation algorithms and then compares properties of these algorithms some these being time to complete, number of intersections, number of dead ends, size, etc. He then uses these findings to determine the most difficult algorithm. They find that recursive backtracking is the easiest and Aldous-Broder is the hardest with Wilson being second. I decide that it would be best to algorithms from the opposite side of these findings. But he goes on to explain that they used the same algorithm to solve all these mazes and didn't say which algorithm they used that is why I want to use multiple solvers as well. Binary tree is also seen as the most basic as well which means that it will be a good base for other algorithms.

Other generation algorithms

Kruskal's algorithm

This algorithm will produce a minimal spanning tree (MST) from a weighted graph meaning it is greedy. This algorithm will also take storage proportional to the size of the maze. There are a few variations of this algorithm, but I will show the most used version for maze generation this being randomized which works like this:

1. Label each cell with a unique ID then loop over all the edges in a random order.
2. For each edge after picking a random cell either.
    i. if the cells on either side of it have different ID's, then get rid of that wall, and then change that new cells ID to the current ID.
    ii. if the cells on either side have the same ID keep the wall so a loop isn't created.
3. Repeat 1 and 2 until all cells have the same ID.

When making it the IDs have to be stored in two separate data sets meaning that the thing that will make this algorithm work slowly is merging the two data sets together over the process of the algorithm this is because Kruskal's algorithm is implemented using a loop. Pullen says that one way to solve this is "by using the union-find algorithm: Place each cell in a tree structure, with the id at the root, in which merging is done quickly by splicing two trees together. Done right, this algorithm runs reasonably fast, but is slower than most because of the edge list and set management" (Pullen, 2022 #3).

Hunt and kill algorithm.

This algorithm is commonly compared to backtracking generator, but it really can't be compared because backtracking generator is depth first search (DPS) but with DPS it will backtrack but for this it will randomly find a cell that fits the parameters of the rules. This is how it works:

1. Choose a starting location at random and mark it as visited.
2. Perform a random walk, carving passages to unvisited neighbours until the current cell has no unvisited neighbours.
3. Enter "hunting" mode where the grid is scanned for an unvisited cell that is adjacent to a visited cell. If found carve a passage between the two and let the formerly unvisited cell be the new starting location
4. Repeat steps 2 and 3 until all cells have been visited and "hunting" mode finds no unvisited cells.

The slowest part of this algorithm will be the "hunting" part as it will continuously search for an unvisited cell if the maze is big this could take some time for it to be found and especially since the search is done row by row this is also a problem. This can be seen in figure 10 below.

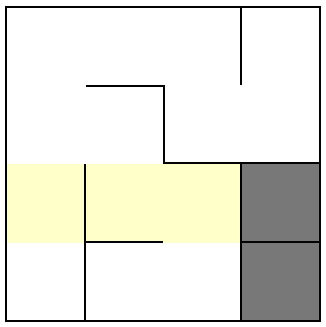


Figure 10 example of the "hunting".

Prim's algorithm

Prim's algorithm is very similar to Kruskal's algorithm, but this algorithm approaches the problem from a different angle. Instead of randomly selecting edges from the graph and adding them to the maze if the connected disjoint trees this algorithm will work from one point and grows outward. There are many variations of this algorithm I will cover one version:

1. Give all edges the same weight.
2. Start with a random vertex.
3. Randomly select a passage edge connecting the maze to a point not already in it and attach it to the maze.
4. Repeat step 3 until there are no more edges to be added to the maze.

This will create a minimum spanning tree (MST). Since the edges are unweighted and unordered these can be stored in a list and as selecting elements from a list is very fast (constant time). This makes it faster than other variation like true prim's which give all the edges a random weight.

Why not these algorithms

There are many more algorithms that I can talk about, but this would mean I am talking about multiple algorithms that I will not be using for my artefact, so I believe there is no need to do so. The main reason I have not chosen these algorithms is because of my original choice of focusing on the four main algorithm types (can be seen in the "**why these algorithms**" section above). These algorithms do somewhat follow these, but some just don't, So I didn't see the need of picking them. Other factors include the fact that some of these algorithms are already very similar to some I have picked already but are more complicated. Also, from the paper (Gabrovšek, 2019 #18) their findings say that both Kruskal's and Prim's algorithm are the middle of the pack in terms of difficulty and very similar to one another I would prefer to see the supposed worst and best compared with different solving algorithms.

Solving algorithms

Random mouse algorithm

In the most basic terms, it moves randomly. It is seen as a very inefficient algorithm. The way this works is that it keeps moving in one direction until it reaches a junction and then randomly pick one direction out of the possible three and then continue like that. As this does not store any memory there is no way for it to remember where it has been meaning that it can go to the same place it has already been multiple times. This algorithm can sometimes not find the end especially if the maze is big but sometimes it can get lucky and find it quickly but most of the time this is highly unlikely.

Shortest path Finder algorithm

Pullen says that shortest path finder is "basically the A* path finding algorithm without a heuristic, so all movement is given equal weight" (Pullen, 2022 #3). A* algorithm is seen as an extension of Dijkstra algorithm this is because of its use of goal-directed-heuristics which many people believe makes it superior. Unlike other algorithms this will work on all maze types and will need the and will need extra memory proportional to the size of the maze. The way this works is by using "water" as described earlier and from the start the maze is flooded such that all distances are filled in at the same time in other words this is breadth first search (BFS) and once any of the paths reach the end, the solution is found then track back to the beginning cell this is the solution. This algorithm's main goal is to find the best possible path, so it doesn't really care about speed or memory as depending on how many passages there are the longer and more memory is taken up it only cares about finding the best solution. The first path to reach the end is assumed to be the best as BFS is used meaning all paths are being solved at the same time so the first should be the best. This algorithm can be seen in figure 12 where the shortest path is red, but all the other paths can be seen as green.

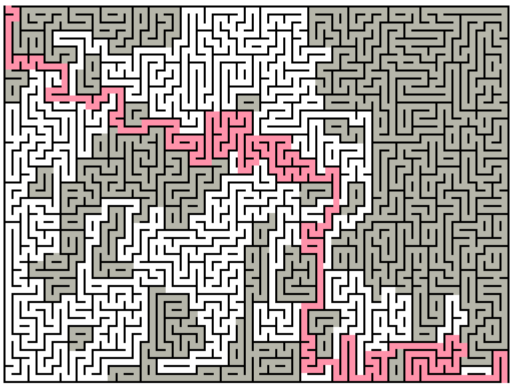


Figure 11 shortest path finder completed.

Trémaux's algorithm

This algorithm was invented by Charles Pierre Trémaux and is designed to be used by people inside of a maze. This algorithm will work for all mazes. The way it works is like this:

1. Walk down a passage and as it does draw a line behind to mark a path.
    a. If you hit a dead end turn around and go back
    b. If you encounter a junction you haven't visited before pick any passage at random
    c. If you are going down a new passage and reach a junction you have visited treat it like dead end and go back (this helps to prevent it going in circles)
    d. If there is a junction and there is aa new passage available always take it otherwise go down a visited passage at random
2. Once the end is reaching all passages marked once will lead back to the start of the maze.

If all passages have been marked twice and you have returned to the start, then there is no solution. As you might be able to tell this algorithm doesn't necessarily always find the shortest path for some mazes.

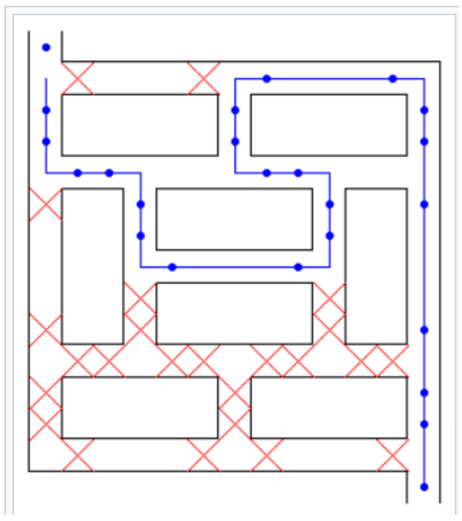


Figure 12 Trémaux algorithm blue dots single pass red cross double pass.

Backtracking solver

This will not find the fastest solution that is not what it is for normally. It uses stacks for its storage this stack will become the size of the maze at least. The way this works is that if you are at a wall or in an area you have already visited return a failure but if you reach the finish return a success otherwise try and recursively move in all four directions. When it moves in a new direction plot a line behind it but if there's a failure it erases the line and backtracks to the last success because of this once it reaches the end there will be a line from the beginning to the end. Although optional it is best to mark the area where there was a failure this is so that the algorithm doesn't go to that area again through a different path. This is pretty much a depth first search. Stacks are best for this because when you go down a new path you would push it onto the stack but when backtracking just pop the cell off the stack. But this might not be the case when implementing.

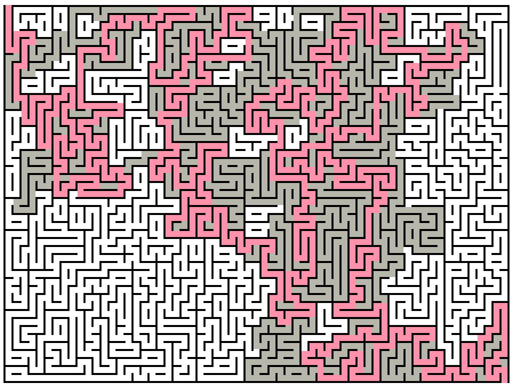


Figure 13 backtracking solver algorithm solved.

Why these algorithms

In path finding there are four algorithms that are considered the main algorithms these being BFS, DFS, Dijkstra and A*. Because of this I chose these algorithms or other algorithms that are based on these algorithms, but I have not chosen Dijkstra as it is similar to A* and just like in the generation I want to use a variety of algorithms and I want superior versions of algorithms if possible. This decision was helped by this paper (Permana, 2018 #17). In this paper the authors took BFS, Dijkstra and A* and evaluate them against each other in a game called maze runner. In this paper they discovered that in most circumstances A* is the superior algorithm in terms of computation time but sometimes Dijkstra does work better, but more memory will always be used. I think this paper is very well done but I do have a problem with it this being the maze game they use to test the algorithms. The results they found were all against the game so that means that A* is only the best in this game not in other types of mazes that is what I will try and determine. The reason I have chosen random mouse is that it is a base line as you really can't get any more inefficient than random mouse.

Other algorithms

Wall follower algorithm
The wall follower algorithm is very simple. This can be applied to real life in that if you stick your left or right hand on a maze wall and then keep following that wall you will eventually find the finish. For the algorithm to work is for it to turn right or left every single time it reaches a junction. Although unnecessary you can mark down what cells have been visited as well but this won't necessarily find the shortest solution it will just find a solution. This algorithm will not work if the end is at the centre of the maze in a closed circuit as the wall you originally start following will never reach it meaning you will go all the way back to the beginning. This can work in 3D mazes, but it is not straightforward as you must make passages in a deterministic manner and the algorithm has to know the orientation of the maze.

Dead end filler algorithm
This algorithm begins by scanning the maze and then will fill in each dead end that it finds until it reaches a junction. But then new dead ends will be created because of this filling in which is when those dead ends are filled in as well. This will find multiple solutions if available as well as single solutions.

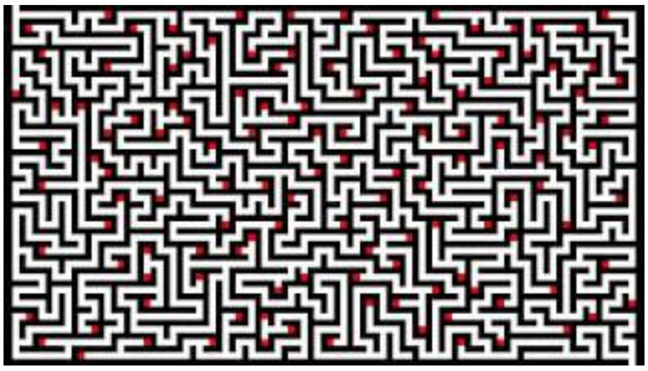


Figure 14 dead end filling after scan.

Pledge algorithm.
This is a modified version of the wall follower algorithm that able to overcome the problems of that algorithm. This will only work when escaping from mazes not going into the maze. To do this pick a direction that you want to travel and then follow the wall in that direction and always try to move in that direction then when you turn increase a counter for the way you turned by 1 for right and -1 for left this will keep increasing or decreasing of you are stuck in a spiral and you can only let go of the wall once the counter has reached 0. Because of this it does lead to a lot of backtracking meaning you can naturally end up at the start point again during the process eventually the exit will be reached. As paths aren't marked the only way to know if a maze is unsolvable is by seeing whether the turn count just keep increasing or decreasing meaning it is just going in a loop but for mazes that are spirals this naturally happens, so it is not always easy to tell.

Dijkstra algorithm

Dijkstra algorithm was originally used for finding the shortest path between nodes in a weighted graph this could be a short distance or long distances once the algorithm finishes it produces something called a shortest path tree. This obviously must be changed somewhat to work inside of mazes. What happens instead is that it works a lot like a flood fill algorithm, but it assigns numbers to each cell it visits these numbers are the distance to the starting node. Usually, this algorithm is implemented using a min-heap priority queue this means that the smallest value is usually at the front of the queue, but his can also be done with a stack.

The way this algorithm works is as follows:

1. Pick the starting point of the maze (usually the northwest corner but can be any corner)
2. Record the cost of reaching that cell:0.
3. Then find that cells neighbours.
4. For each neighbour, record the travel cost for getting to the new neighbour.: cost of previous neighbour +1.
5. Repeat steps 3 and 4 until all cells are visited and no cells are revisited.

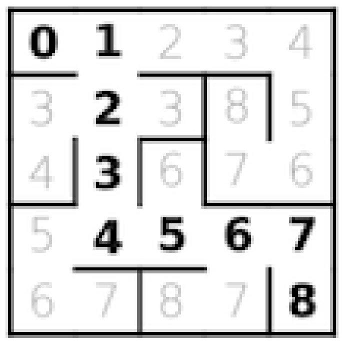


Figure 15 end result of Dijkstra algorithm.

Why not these algorithms

For the wall follower algorithm this could be used but my aim is to generate mazes that can have the end in the middle this will mean that the wall follower is actually going to be useless this is the same with the pledge algorithm if it starts on the outside edge of the maze I don't think it would be good to use algorithms that might not ever actually solve the maze or can't. The reason I haven't picked algorithms like dead-end filler and other like it is because of its need to scan the maze beforehand this makes it different from the other algorithms as they will go into the maze blind, and I would like to use those types of algorithms as I believe that it will make it fairer when I evaluate the algorithms.

More Literature Review

Although this paper is focusing on AI in mazes, I thought it would be interesting and a good idea to talk about it just to expand further on the entire topic of mazes and cover more advanced topics like AI. In this paper (Peachey, 2022 #51) they focus on using neural networks and parameterized maze generation algorithms and other algorithms to determine and define different levels of difficulty in mazes as they find, and I also believe that mazes have different definitions of difficulty based on whether a human or computer is solving it. What they found was that mazes that have more short passages and less directional bias are much more difficult than ones with longer passages and that they believe that using something like a neural network they can fully convert difficulty into a set of inputs. Although they did seem to use multiple different solving algorithms there was no focus on those algorithms just how they did against certain types of mazes, but this does have its problems as with the solution length this helps to determine the difficulty but mazes that have short solution are being determined as equal to long solutions which just isn't correct. What I do like about this is that they seem to keep the testing quite fair in that each algorithm starts at the same place when starting the solve and in the same maze and although this doesn't apply to this dissertation as it is procedural it is interesting to see different ways to test maze algorithms.

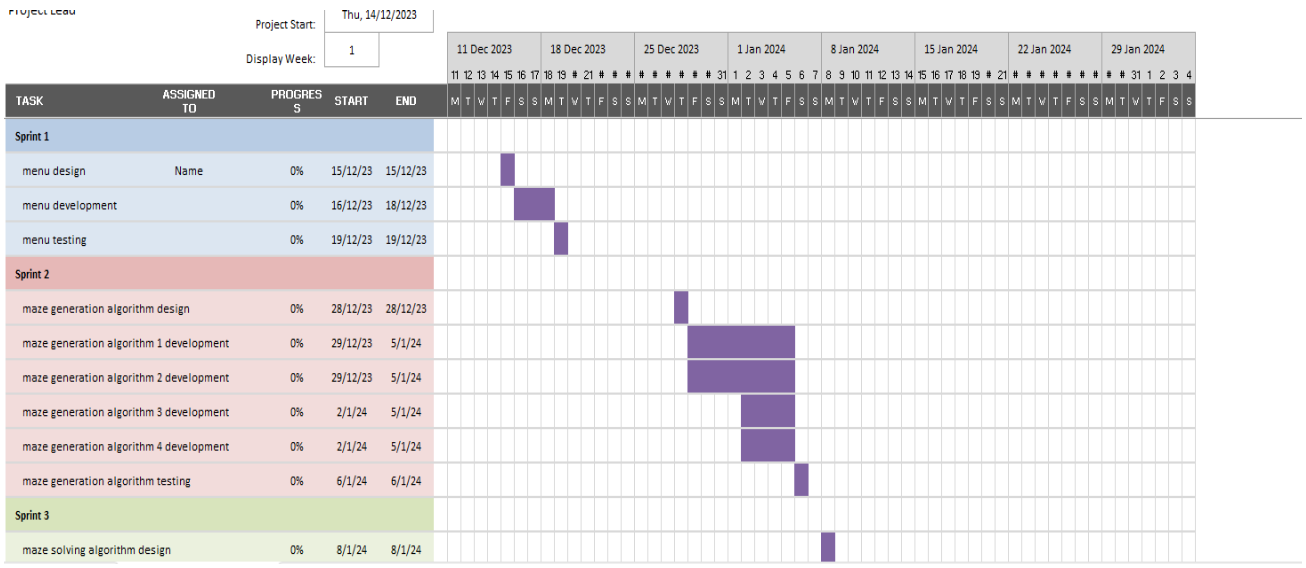
Requirement analysis and methodologies

Programming languages

For this dissertation I will be using python to create my artefact. This is mainly because this is the language, I am most comfortable with, but it also has the capability to complete this task. The main requirement to complete this task is that it has object orientated capabilities other languages have this capability like C++, C#, Ruby, Java, etc. But I have already created some games in python more specifically pygame. Pygame is an open-source library that can be used for multimedia applications like games this could be used to make the artefact there is also other things like matplotlib which can be used to create the maze as well there is also libraries like numpy as well that allow for easy manipulations of data structures specifically arrays. A great example of what I want to implement can be found in Walter D Pullen's Daedalus (Walter.D. Pullen, 2023 #25). This is a windows application that will allow anyone to create, solve, analyse, view, and walk through mazes. This application is made out of a combination of C++, C and HTML. This is a fantastic tool, but is would be impossible for me to do something like it in the time I have, and it is completely unnecessary for this dissertation. Another example can be found in Jamis Buck's book "Mazes for Programmers" in this he explains different aspects of mazes and also teaches the reader to program algorithms. This is done in Ruby, but it's done in a way that can easily converted into any language as he clearly explains all of the algorithms (Buck, 2015 #26). For this artefact to be made all I need is a computer or laptop that is able to run python and the libraries in python and this is not a problem as most computers today can easily run these languages. If I was to make the maze enormous there might be some problems with memory, but this would have to be a big maze which I do plan on doing so there might be problems when it comes to running these larger mazes. But this does mean that if I was to use a more powerful computer this might make processing faster and less memory intensive. But for most uses there should not be a problem with the hardware I have access to and plan on using I will just have to be more careful at higher dimensions of things like memory leaks.

Methodologies

When I create the code for my artefact, I will be following an agile methodology this is because this allows me to keep a steady pace on the development of my project, but it also allows me to test my algorithms but also change them later on once I have tested them it will also allow me to continuously improve my artefact until I am satisfied with it.



Project Lead

Project Start: Thu, 14/12/2023

Display Week: 1

11 Dec 2023 | 18 Dec 2023 | 25 Dec 2023 | 1 Jan 2024 | 8 Jan 2024 | 15 Jan 2024 | 22 Jan 2024 | 29 Jan 2024

TASK	ASSIGNED TO	PROGRESS	START	END
Sprint 1				
menu design	Name	0%	15/12/23	15/12/23
menu development		0%	16/12/23	18/12/23
menu testing		0%	19/12/23	19/12/23
Sprint 2				
maze generation algorithm design		0%	28/12/23	28/12/23
maze generation algorithm 1 development		0%	29/12/23	5/1/24
maze generation algorithm 2 development		0%	29/12/23	5/1/24
maze generation algorithm 3 development		0%	2/1/24	5/1/24
maze generation algorithm 4 development		0%	2/1/24	5/1/24
maze generation algorithm testing		0%	6/1/24	6/1/24
Sprint 3				
maze solving algorithm design		0%	8/1/24	8/1/24

Figure 16 Gantt chart of artefact development.

As you can see in the Gantt chart in figure 16, my plan is to have weeklong sprints of doing different parts of the project most of my time will be taken up by implementing the different algorithms I will be using to create and solve the maze. But I will also implement a menu system that will allow myself and users to run the different algorithms together. Also, I will need to implement some measures into my algorithms such as time and space. I believe that I can complete this in around 1 month as long as the agile methodology is followed. As you can see in figure 16 I will go through cycles of design, implementation and testing although I do intend on following this cycle of short design and testing times I believe that some parts of the artefact might require more emphasis on design and testing for example the algorithms will have to be rigorously tested but the menu system will be tested but not to the extent of the algorithms this is because I have already created a menu system before and it is not as complicated as the algorithms will be so it should work as expected. For the design I intend on showing some pseudocode and also some use case and UML diagrams. To make these I will be using draw.io for the flowchart, use case diagrams and the UML diagrams as well. This is because it is what I am used to using the most, but I know that if I need to tool like Microsoft Visio can also be used to create these diagrams as well. For the pseudocode I will be writing it inside of visual studio code as that has pseudocode language recognition which just helps when writing the pseudocode.

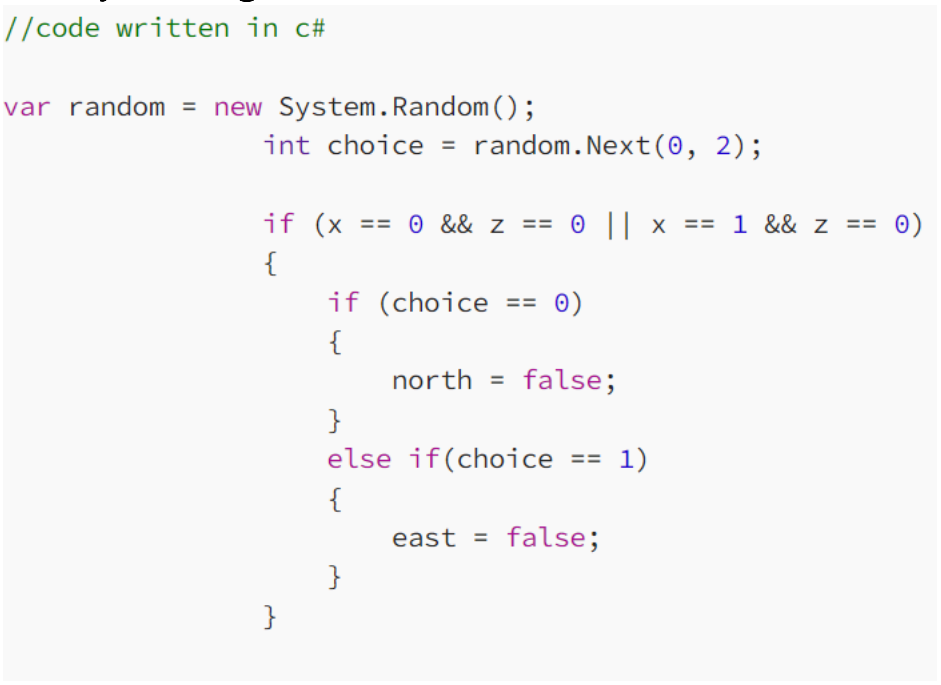
Artefact Design

Existing examples code and pseudocode

When creating something it is best to look back, if possible, to other examples of what you are trying to make I have done this in the previous chapter but not in much detail. I will do this now as it will help me to determine what I want to have in my artefact. As I have said previously many algorithms for this topic have already been created but they have been implemented in many different ways across many different languages because of this it would be very time consuming to show you all of them that is why I will be showing existing code and pseudocode for the algorithms I have covered already. I will show code from different resources and my own pseudocode when I believe it is necessary.

Generation algorithms

Binary tree algorithm



```
//code written in c#

var random = new System.Random();
                int choice = random.Next(0, 2);

                if (x == 0 && z == 0 || x == 1 && z == 0)
                {
                    if (choice == 0)
                    {
                        north = false;
                    }
                    else if(choice == 1)
                    {
                        east = false;
                    }
                }
```

Figure 17 binary tree algorithm code from Medium.

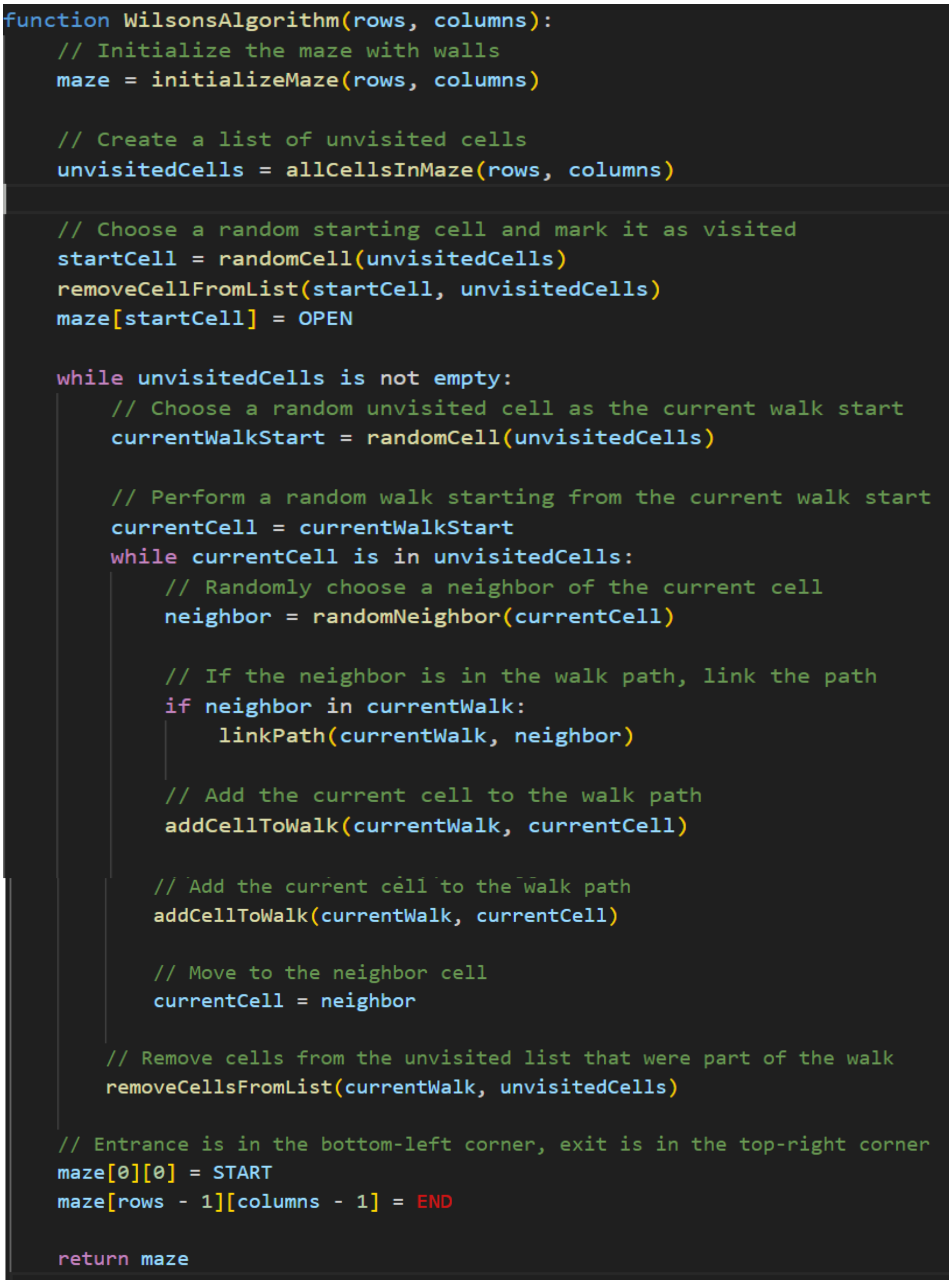
So, the main part to take away from this is the decision of which direction the algorithm will move this can be north/east, north/west, south/east, south/west. As you can see from the code they are moving north/east. This really doesn't matter a huge amount it is just something to consider when implementing it just means that whatever directions I chose there will be a diagonal bias in that direction. I could implement that so that it allows me to pick which direction it goes but this is unnecessary at the moment but could be something I end up doing in the end but for the testing this will be set in a constant direction as to not have a huge amount of data on the binary tree algorithm compared to the other algorithms.

Eller's algorithm

```
def step(state, finish=false)
  connected_sets = []
  connected_set = [0]

  # ---
  # create the set of horizontally connected corridors in this row
  # ---

  (state.width-1).times do |c|
   if state.same?(c, c+1) || (!finish && rand(2) > 0)
     # cells are not joined by a passage, so we start a new connec
     connected_sets << connected_set
     connected_set = [c+1]
   else
     state.merge(c, c+1)
     connected_set << c+1
   end
  end

  connected_sets << connected_set

 verticals = []
 next_state = state.next

 unless finish
   state.each_set do |id, set|
     cells_to_connect = set.sort_by { rand }[0, 1 + rand(set.length-1)]
     verticals.concat(cells_to_connect)
     cells_to_connect.each { |cell| next_state.add(cell, id) }
   end
 end
```

Figure 18 Eller's algorithm code from "Mazes for programmers".

As you can see in figure 18 there is a big emphasis on the vertical and horizontal passages of the maze this is mainly because of the way the algorithm works but it does lead to biases. The main problem will be the sets, but I need to make that each row has its own set but then by the last row all set have been merged into one set this will be the main problem for this algorithm. This process may seem small but will probably have to be spread out across multiple different functions. The sets will be most likely be put into arrays.



```
function WilsonsAlgorithm(rows, columns):
    // Initialize the maze with walls
    maze = initializeMaze(rows, columns)

    // Create a list of unvisited cells
    unvisitedCells = allCellsInMaze(rows, columns)

    // Choose a random starting cell and mark it as visited
    startCell = randomCell(unvisitedCells)
    removeCellFromList(startCell, unvisitedCells)
    maze[startCell] = OPEN

    while unvisitedCells is not empty:
        // Choose a random unvisited cell as the current walk start
        currentWalkStart = randomCell(unvisitedCells)

        // Perform a random walk starting from the current walk start
        currentCell = currentWalkStart
        while currentCell is in unvisitedCells:
            // Randomly choose a neighbor of the current cell
            neighbor = randomNeighbor(currentCell)

            // If the neighbor is in the walk path, link the path
            if neighbor in currentWalk:
                linkPath(currentWalk, neighbor)

            // Add the current cell to the walk path
            addCellToWalk(currentWalk, currentCell)

            // Add the current cell to the walk path
            addCellToWalk(currentWalk, currentCell)

            // Move to the neighbor cell
            currentCell = neighbor

        // Remove cells from the unvisited list that were part of the walk
        removeCellsFromList(currentWalk, unvisitedCells)

    // Entrance is in the bottom-left corner, exit is in the top-right corner
    maze[0][0] = START
    maze[rows - 1][columns - 1] = END

    return maze
```

Figure 19 Wilson's algorithm pseudocode.

Wilson’s algorithm

The main part of this algorithm is the random walk that takes place this means that it is sometimes down to luck on the speed but during the algorithm a lot of things will take up the memory this being things like current position and the paths that have been walked through etc. After the walk is done though this frees up the memory but then the process starts again until the maze is finished. Just like in other algorithms some parts of it seem smaller than they will actually be as the random walk will be made up of many different functions all working together as there will be walk and then randomness that comes from that but then that walk again needs to be solved in itself. So, a couple of data structures might have to be used implement this.

Backtracking generation algorithm

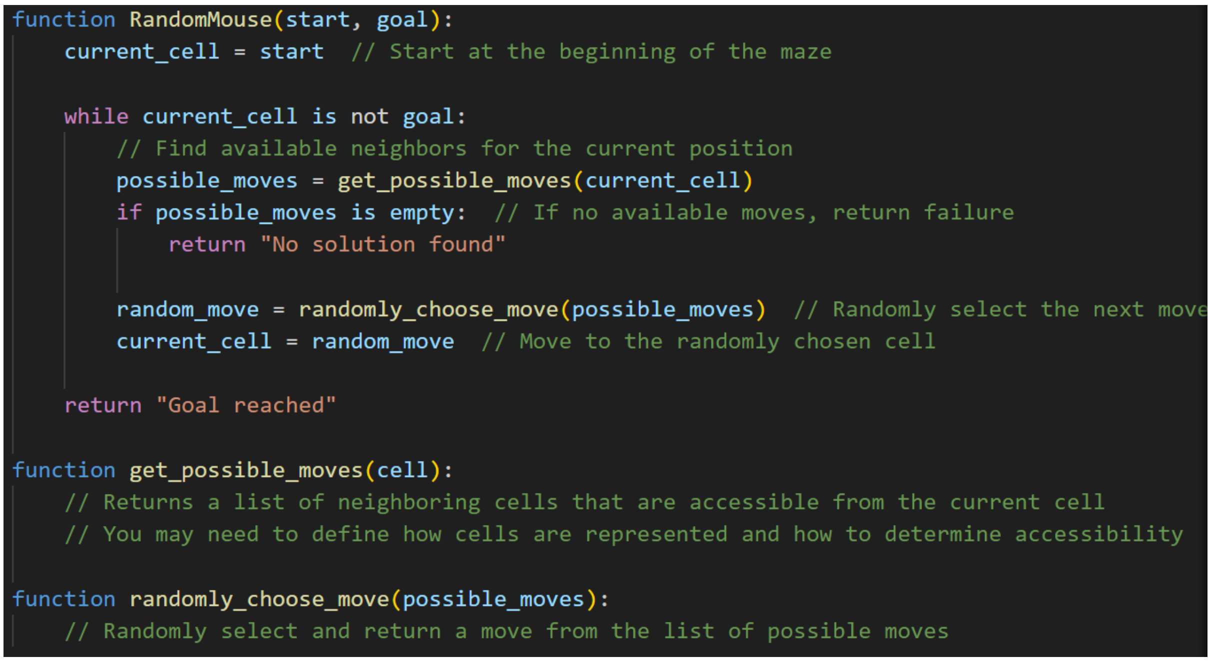
```
def carve_passages_from(cx, cy, grid)
  directions = [N, S, E, W].sort_by{rand}

  directions.each do |direction|
    nx, ny = cx + DX[direction], cy + DY[direction]

    if ny.between?(0, grid.length-1) && nx.between?(0, grid[ny]
      grid[cy][cx] |= direction
      grid[ny][nx] |= OPPOSITE[direction]
      carve_passages_from(nx, ny, grid)
    end
  end
end

carve_passages_from(0, 0, grid)
```

Figure 20 backtracking pseudocode.

The main thing about this algorithm is the fact that the entire maze does need to be stored in a stack compared to the other algorithms this is something to think about, but the only other part is the random direction part that can be seen in figure 20 and the ability to recognize visited and unvisited cells as this plays a part in other algorithms as well once it is implemented in either it will be less of a challenge.

Solving algorithms

Random Mouse

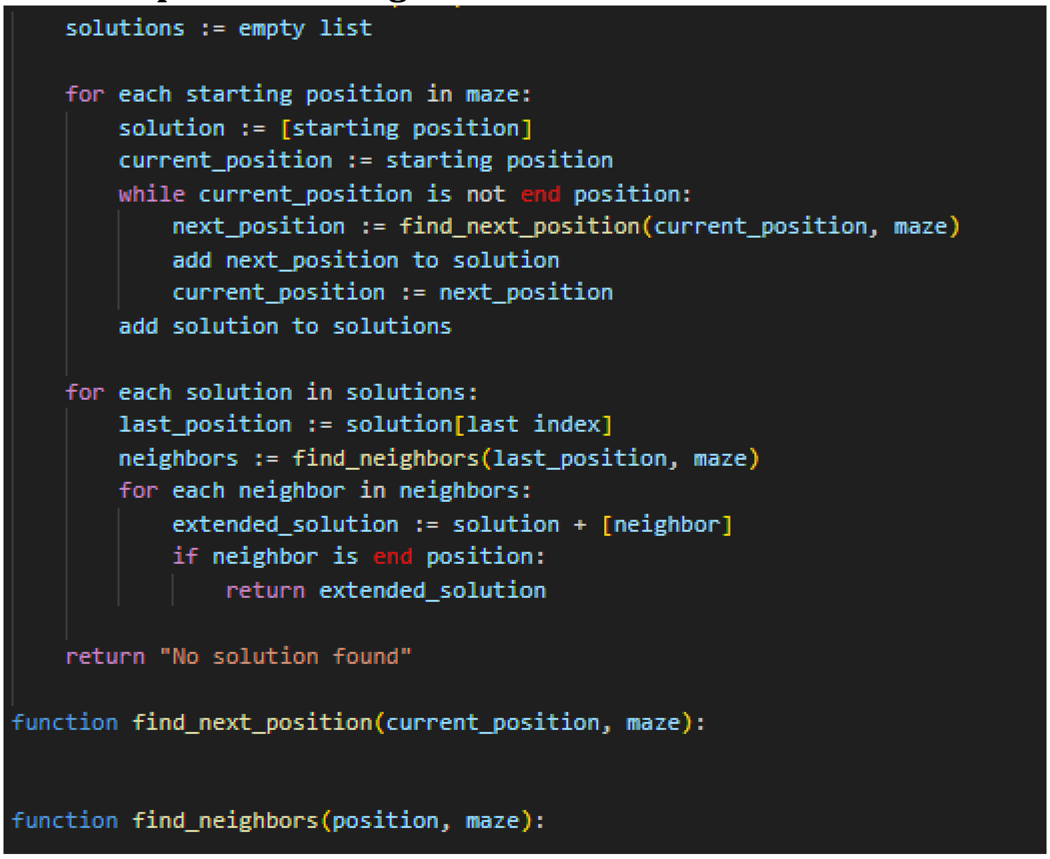


```
function RandomMouse(start, goal):
    current_cell = start  // Start at the beginning of the maze

    while current_cell is not goal:
        // Find available neighbors for the current position
        possible_moves = get_possible_moves(current_cell)
        if possible_moves is empty:  // If no available moves, return failure
            return "No solution found"

        random_move = randomly_choose_move(possible_moves)  // Randomly select the next move
        current_cell = random_move  // Move to the randomly chosen cell

    return "Goal reached"

function get_possible_moves(cell):
    // Returns a list of neighboring cells that are accessible from the current cell
    // You may need to define how cells are represented and how to determine accessibility

function randomly_choose_move(possible_moves):
    // Randomly select and return a move from the list of possible moves
```

Figure 21 Random mouse pseudocode.

This algorithm is quite simple in that the only thing that needs to be stored is the final solution this algorithm stores only that as it doesn't need to remember where it has been. Every move will be put onto a list and that will be the final solution. I will need to bring in a random choice library as well so that the algorithm can be random.

Shortest path finder algorithm

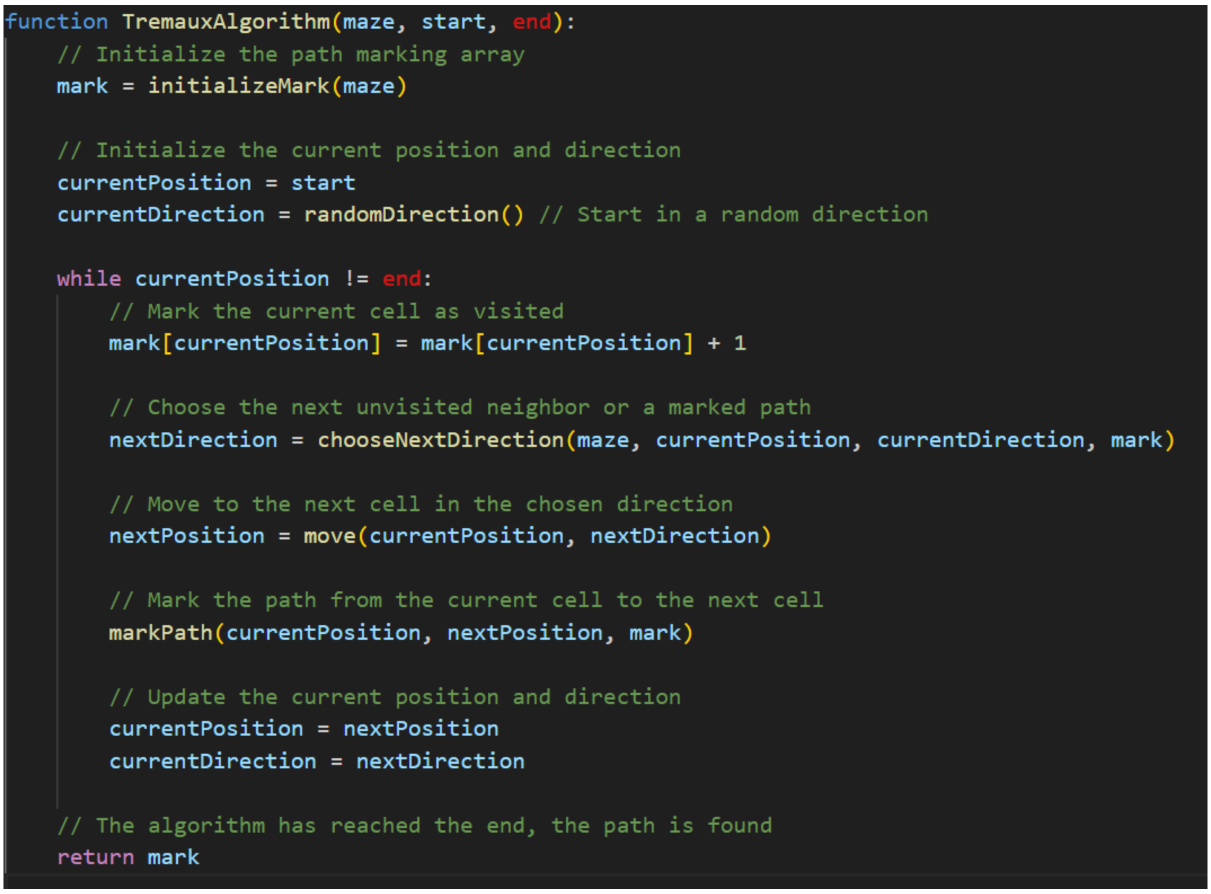


```
    solutions := empty list

    for each starting position in maze:
        solution := [starting position]
        current_position := starting position
        while current_position is not end position:
            next_position := find_next_position(current_position, maze)
            add next_position to solution
            current_position := next_position
        add solution to solutions

    for each solution in solutions:
        last_position := solution[last index]
        neighbors := find_neighbors(last_position, maze)
        for each neighbor in neighbors:
            extended_solution := solution + [neighbor]
            if neighbor is end position:
                return extended_solution

    return "No solution found"

function find_next_position(current_position, maze):


function find_neighbors(position, maze):
```

Figure 22 shortest path pseudocode.

As this algorithm is a combination of A* and BFS with all cells having equal weight it is just the two algorithms combined together. When implementing I will need have more memory available for this algorithm because until the very end all travelled paths are stored in memory and then the paths that lead to the finish are kept in memory but the path that makes it there first is the one that gets displayed. This will probably mean that for each path I will have to have separate lists or arrays to hold the path travelled.

Trémaux's algorithm

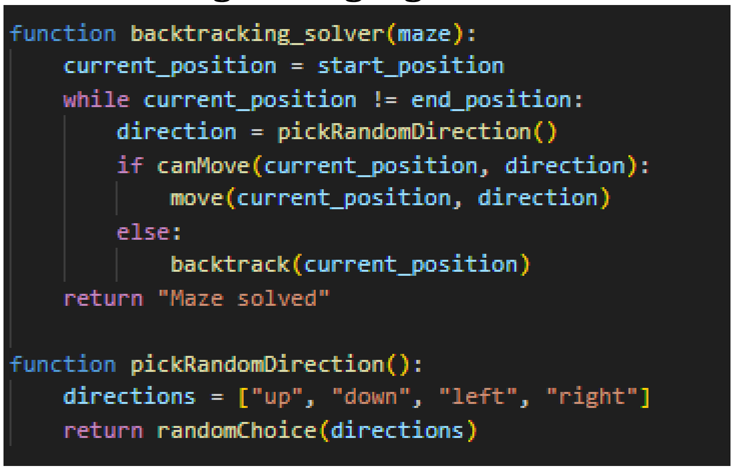


```
function TremauxAlgorithm(maze, start, end):
    // Initialize the path marking array
    mark = initializeMark(maze)

    // Initialize the current position and direction
    currentPosition = start
    currentDirection = randomDirection() // Start in a random direction

    while currentPosition != end:
        // Mark the current cell as visited
        mark[currentPosition] = mark[currentPosition] + 1

        // Choose the next unvisited neighbor or a marked path
        nextDirection = chooseNextDirection(maze, currentPosition, currentDirection, mark)

        // Move to the next cell in the chosen direction
        nextPosition = move(currentPosition, nextDirection)

        // Mark the path from the current cell to the next cell
        markPath(currentPosition, nextPosition, mark)

        // Update the current position and direction
        currentPosition = nextPosition
        currentDirection = nextDirection

    // The algorithm has reached the end, the path is found
    return mark
```

Figure 23 Tremaux algorithm pseudocode.

With this algorithm the main feature of it is that the path that is travelled is marked out this is another thing that takes up memory as well as the maze itself as these markings do not need to be removed, we do not need a data structure that removes its contents as the markings will stay whilst the path is being solved. But as this works similarly to algorithms like backtracking solver and DFS they will be similarly implemented.

Backtracking solving algorithm.



```
function backtracking_solver(maze):
    current_position = start_position
    while current_position != end_position:
        direction = pickRandomDirection()
        if canMove(current_position, direction):
            move(current_position, direction)
        else:
            backtrack(current_position)
    return "Maze solved"

function pickRandomDirection():
    directions = ["up", "down", "left", "right"]
    return randomChoice(directions)
```

Figure 24 backtracking solver pseudocode.

This works a lot like random mouse and DFS but the difference with that is that random mouse doesn't need extra memory unlike this algorithm which stores where it has visited this is so it doesn't go back on itself so it will find it eventually otherwise it is completely random again this is only stored in memory until completion but it will take up more memory as all visited cells are stored which means that it can sometimes take up the full size of the maze. This can probably be stored in a stack or list.

Data structures

Some algorithms will need extra memory this is because extra data needs to be stored to carry out the algorithm this can be seen in all of the solving algorithms I am implementing. These pieces of data differ depending on the algorithm and when and how they should be stored meaning data structures play a big part when implementing some algorithms. Most if not, all can actually be simply implemented using a stack. But when actually implementing into code this can lead to inefficiencies and sometimes algorithms will work best with different data structures for example Eller's algorithm uses sets but could work with linked lists. So, when I implement the algorithms, this will be something I have to decide on. I have already stated or shown the data sets I will use for each algorithm so far, but this could change during implementation these being either lists or dictionaries or stacks but in practice I will probably have to implement using arrays a lot as well this will be talked about in the implementation chapter.

UML Diagrams

Before starting to code it is best to organize and visualize what you have in mind and what you plan on implementing. One of the best ways to do this is through UML diagrams, UML diagrams allow for accurate modelling and visualization and makes complex systems easier to understand. There are a couple of types of UML diagrams but the ones that are relevant to me are class diagrams and use case diagrams. The class diagrams will allow me to layout and better determine how my program will function in terms of the classes in it. This diagram will help me to get a better grasp on the back end of my artefact. Along with this I will need to show the front end of the artefact to do this I will show a use case diagram which will help me to better understand how users will interact with the artefact.

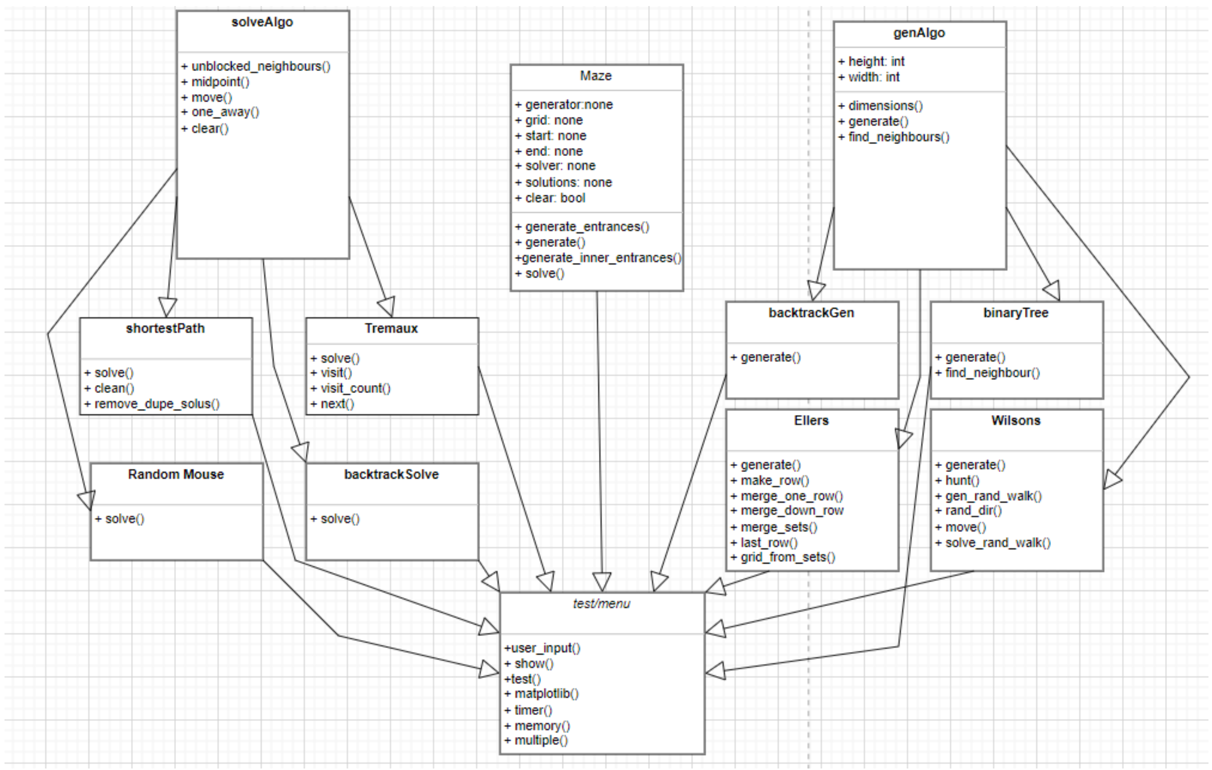


Figure 25 UML Class diagram.

Now that the UML class diagram is done, I will explain some of the classes and functions that can be found in figure 25 this is what I believe will be needed in the artefact, but this might change as I go into the development process.

- Solve () – this can be found in all of the solving algorithms this function contains the actual algorithm for the solving process. This will be different for each algorithm but when it is run in the test/menu class then it will use the inherited solve from the maze class which will just be a culmination of solution from the algorithms.
- Solve algo- solve () – this is a helper method for the solve () functions in the algorithms. That will start the solving process but will not have any algorithm behind it. It will have things to help find paths and neighbours and movement of the algorithms.
- Visit (), visit count () & next ()- these are the functions that allow for the solve () to work in Trémaux algorithm they keep track of what cells have been visited and then helps decide for the algorithm what direction is best to go through.
- Clean () – will get rid of the other partially completed solutions that will generate from the shortest path algorithm.
- Unblocked neighbours () – this will help find the cells next to the current cell that the algorithm is in that is actually accessible to be able to be moved to as each cell can have up to 4 other cells next to it, but some are blocked by walls whilst some aren't this will determine that.
- Midpoint () – finds the halfway point of two cells, as the maze will most likely be stored in an array this means that every other point in the grid will be a possible wall (as long as it hasn't already been removed) this finds those point when solving.
- Move () – changes the current position of the algorithm.
- One away () – this checks whether the end is within one move from the current position.
- Clear () – this will clear up all the paths that the solving algorithms have taken this will be able to be turned on and off when the user decides the algorithms.

- Generate () – this is found in all the generation algorithms and contains the algorithm for the generation process. This will be different for each algorithm but just like the solver they will be a basic constructor that will be used to put the algorithm into the menu/test class so different algorithms can be called.
- Find neighbour () – this will find the next cell that is a wall, this will find out the location of all the walls closest to the current cell these being north, south, east, west this will be a helper method for all generation algorithms.
- Ellers functions – all of the functions found in Eller's algorithm are mainly for the sets and combining them together as this is how the algorithm works and is its main feature.
- Wilson's functions – all of the functions involve generating and then solving the random walk that is used for the algorithm. There are also functions for picking directions randomly. There will also be functions for solving the random walk.
- genAlgo- this is a constructor for the generation algorithms as a whole and will set the minimum dimensions for the mazes. It also has methods to help the algorithms.
- Maze ()- in maze generate will create the mazes with the algorithms but also things like the entrances and also exit. It will do this by importing the different algorithms and then using them to actually generate and solve the maze. Also, my entrance and exit will only be located inside the maze this is just for simplicity but obviously there are many examples of mazes with external entrances and exits this just means I will be using the 'ladder' method for the 3D aspect.
- test/menu class – this is how the user would interact with the algorithms allowing them to generate and solve. I plan on displaying the maze using matplotlib as well so a function will be need for that. There will also need to be a function that allows me to test the algorithms this will be accompanied with a function for getting the time and space of the algorithms. This has to be different from the matplotlib function as this will not give the true time as the time is dependent on how quickly you get matplotlib to plot the maze and memory usage can naturally be increased when using matplotlib. This matplotlib section I see it as not a necessary as the testing section as it will be more visually focused by just allowing me and users to have a better view of the mazes, so I see this as more of an advanced feature of the artefact.

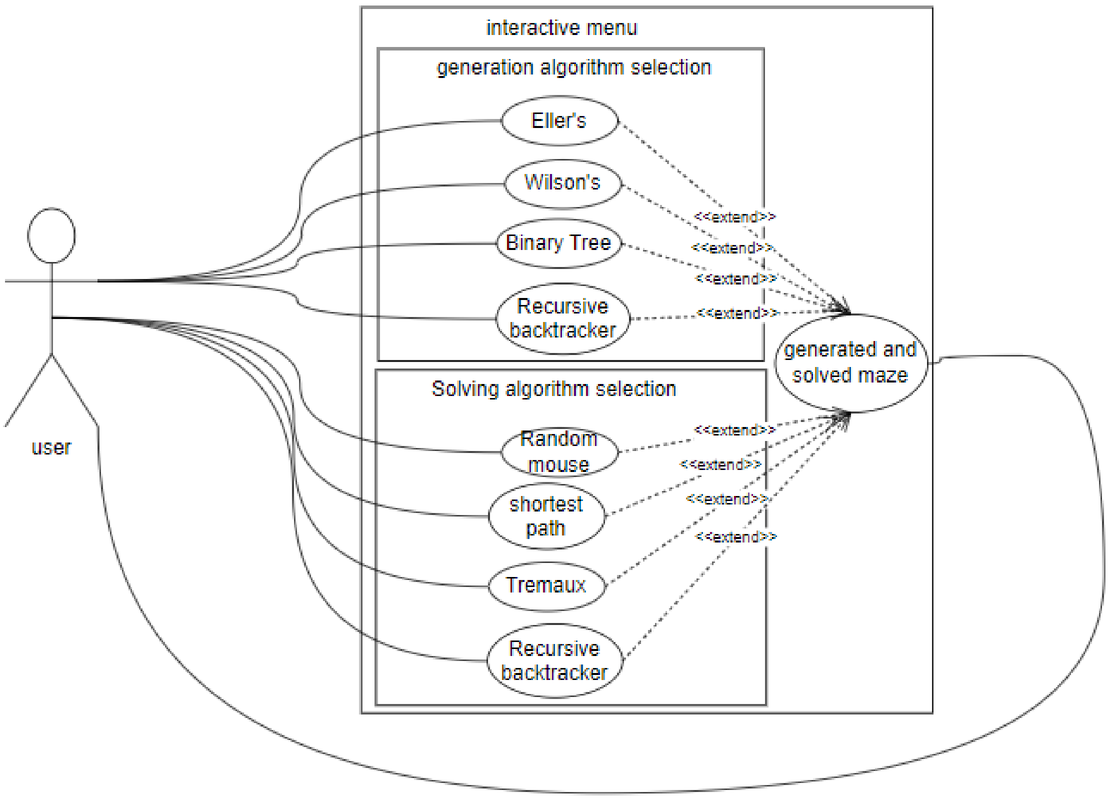


Figure 26 UML Use Case Diagram

My plan for the interactivity of the artefact can be seen in figure 24 what this shows is that the user is able to select a generation algorithm and a solving algorithm which in turn will then produce a maze that is made and solved.

The overall plan for the UI is that it is a simple text popup on the console that will give the user the option to input what type of generator they want and then the same will happen with the solver after that. This will then result in a display of the algorithms working together in this I plan of having a timer somewhere on the screen or as a popup at the end saying how long each algorithm took to finish this will be the same for overall memory usage as well. As this is an interactive menu, I plan on allowing the entire thing to loop on itself in that once the entire process of generation and then solving is completed the user will just have to press a button and then the user will be able to repeat the process over again I will also give the option to control how many mazes the user wants to make so that they have control of the dimensions and I will also give the option for the size of the maze as well. The UI I have thought of is simple because that is all it has to be as this will not be widely accessible and I do not plan on getting people to use it for me to get my results so having the UI as simple as possible makes sense as it will only really be ease of use for me when I come to evaluate and test the artefact. I plan on using matplotlib to display the mazes along with a simpler string method this will allow me to plot the maze generation and the maze solving and allow me to see how the algorithms I implemented will work but I also plan on making another function so that I can just print out the maze and the solving path at the same time this is where I will use my memory function and time function as this is a better representation of the time and memory used.

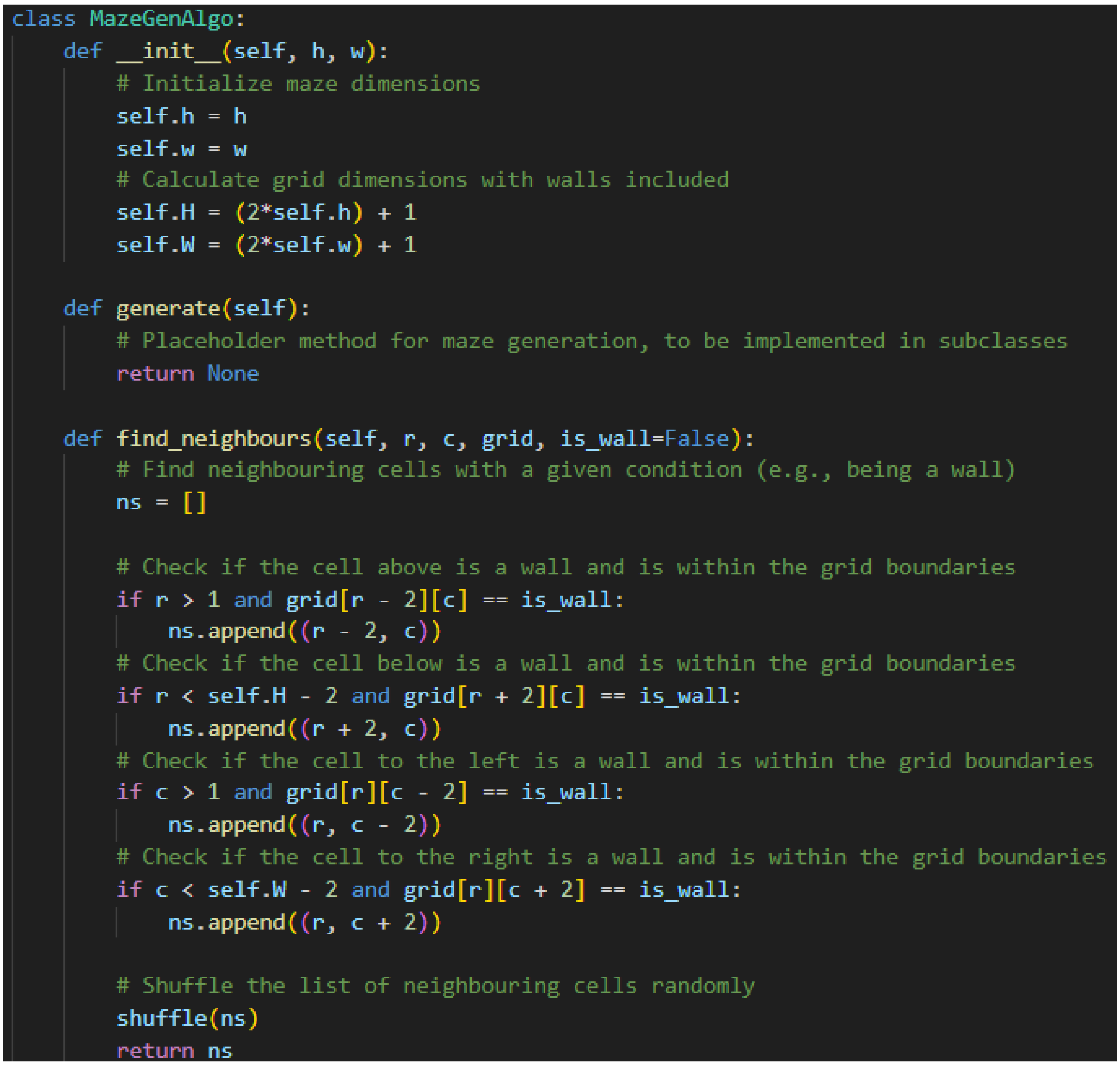
Functional requirements and non-functional requirements

The maze will have a certain look and set out to it. There are mostly functional requirements except for some that I will mention. As of now I plan to have 2 types of mazes one is in matplotlib the reason for this is because this will allow me to create and accurate but more visually appealing maze for users and for myself. My plan for this maze is once the criteria for the maze has been entered (e.g., the algorithms and the height, width, and depth) the maze will be generated first and then once it has been generated you will then see the maze be solved. In this I have decided that the entrances will be on the inside of the maze as this gives it more of a cohesive structure look as if the start and finish were on the edges of the maze then it will feel more like they are not going up a level just along. For one of the non-functional the solution path will just have dots that go along to show the path that is taken these will be displayed one by one. Once the first maze is done then the next maze will generate and so on depending on the amount has been entered. The other maze is more for testing in that I will show this using string in that the maze will be printed to the screen using things like '#' and '+' to show the maze and the solve path, respectively. This is so that time is not wasted waiting for matplotlib to show what it is plotting. Another non-functional requirement when inputting the things like algorithms chosen and the dimensions of the maze this will be simplified to pressing '1' for binary tree and then '1' for shortest path for example this is because it is simpler for myself and potential users but isn't necessary for it to work. This will have to be tested and will be shown later.

Artefact Implementation

In Design I said that I would use the data structures that were displayed with the pseudocode or talked about. This did change as once I started making the artefact, I realised that it would be much easier for development if they all use the same data structures this being an 2D array, lists and dictionaries. some other data structures have been used but I will talk about that in the sections they link to. One of the main reasons for using an array was so that I could use matplotlib to actually plot and animate the maze process as matplotlib works best with array specifically NumPy arrays which is what I have used. This is one of the biggest changes from the design. Also, it made it easier for me in development if I used less types of data structures.

genAlgo

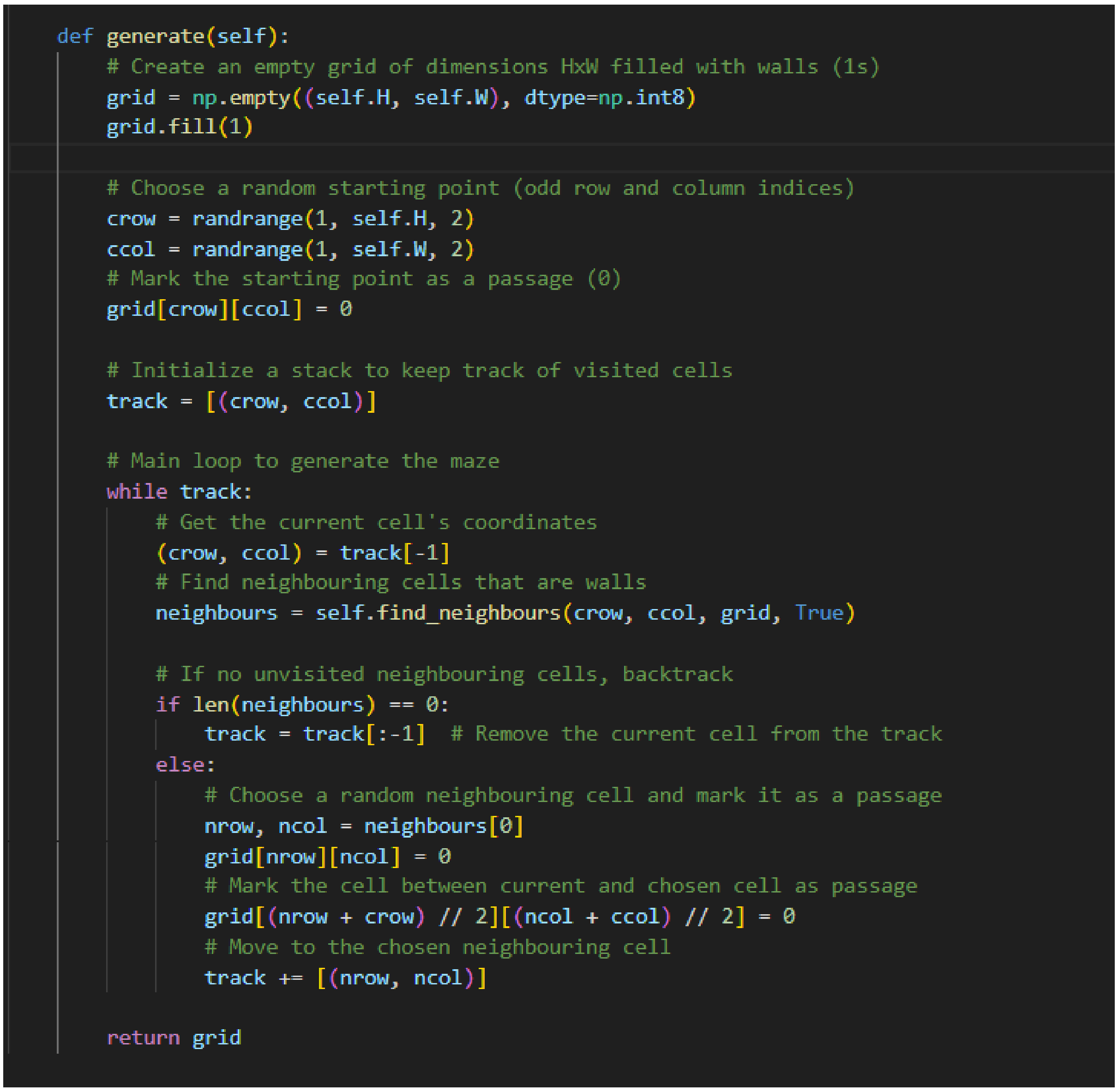


```python
class MazeGenAlgo:
    def __init__(self, h, w):
        # Initialize maze dimensions
        self.h = h
        self.w = w
        # Calculate grid dimensions with walls included
        self.H = (2*self.h) + 1
        self.W = (2*self.w) + 1

    def generate(self):
        # Placeholder method for maze generation, to be implemented in subclasses
        return None

    def find_neighbours(self, r, c, grid, is_wall=False):
        # Find neighbouring cells with a given condition (e.g., being a wall)
        ns = []

        # Check if the cell above is a wall and is within the grid boundaries
        if r > 1 and grid[r - 2][c] == is_wall:
            ns.append((r - 2, c))
        # Check if the cell below is a wall and is within the grid boundaries
        if r < self.H - 2 and grid[r + 2][c] == is_wall:
            ns.append((r + 2, c))
        # Check if the cell to the left is a wall and is within the grid boundaries
        if c > 1 and grid[r][c - 2] == is_wall:
            ns.append((r, c - 2))
        # Check if the cell to the right is a wall and is within the grid boundaries
        if c < self.W - 2 and grid[r][c + 2] == is_wall:
            ns.append((r, c + 2))

        # Shuffle the list of neighbouring cells randomly
        shuffle(ns)
        return ns
```

Figure 27 genAlgo code snippet.

So in this file this where the helpers and constructors for all the other generation algorithms can be found in this you can see that the maze height and width are initialized this done in a way so that for the array there will be a wall then an empty space and so on but because of this we need to times the initial number by 2 but this would leave the last part of the array empty so we add 1 that the all sides end in a wall. Generate has been left empty as the generates are done through the actual algorithm classes not this one. The find neighbours function will find neighbours in all four directions from the current position and will check whether there is wall or not this is then appended to the list, which is shuffled to introduce randomness into the process, but it also will make sure that the algorithms don't just carry on past the edge of the maze.

Backtracking generator

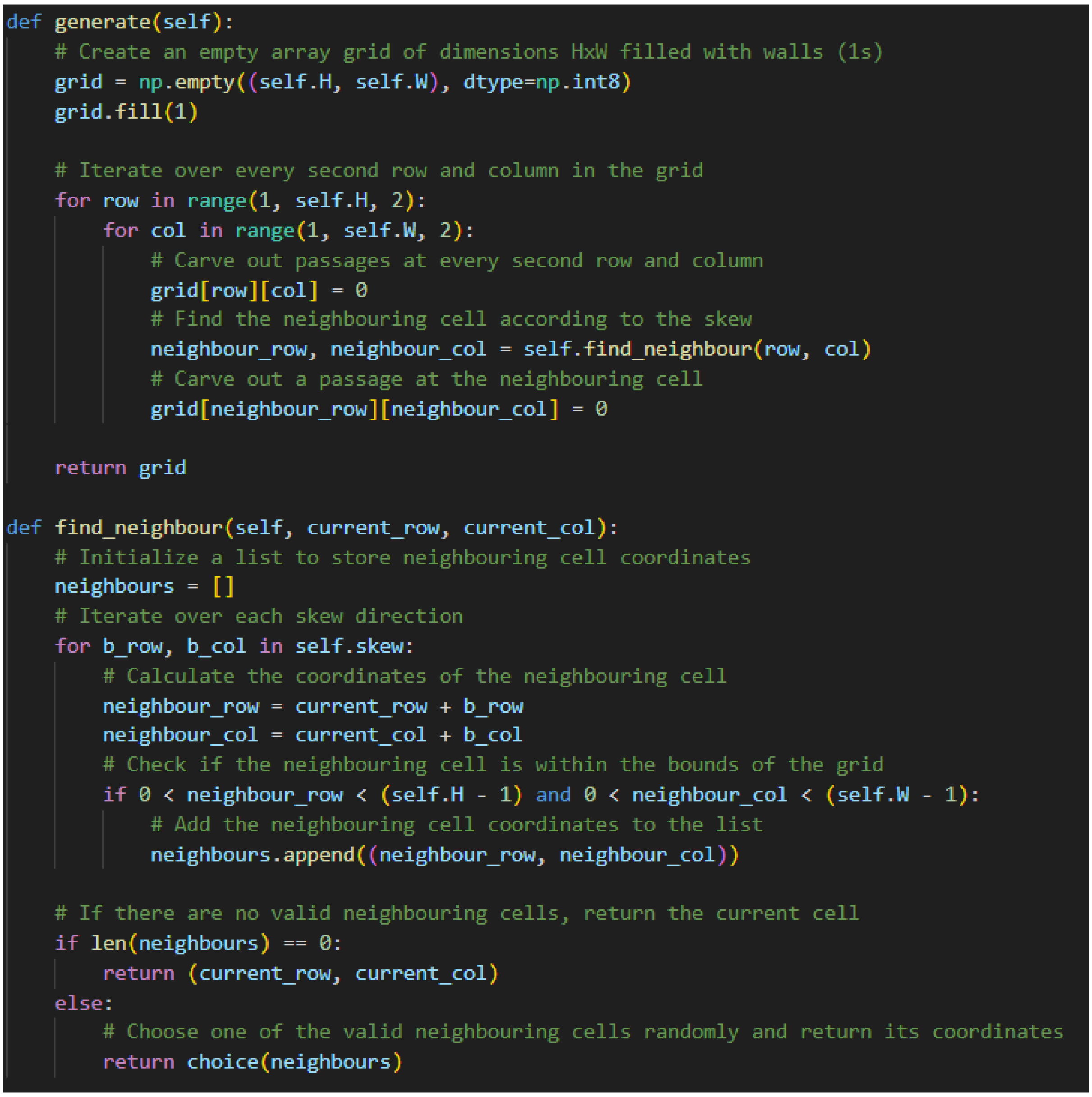


```python
def generate(self):
    # Create an empty grid of dimensions HxW filled with walls (1s)
    grid = np.empty((self.H, self.W), dtype=np.int8)
    grid.fill(1)

    # Choose a random starting point (odd row and column indices)
    crow = randrange(1, self.H, 2)
    ccol = randrange(1, self.W, 2)
    # Mark the starting point as a passage (0)
    grid[crow][ccol] = 0

    # Initialize a stack to keep track of visited cells
    track = [(crow, ccol)]

    # Main loop to generate the maze
    while track:
        # Get the current cell's coordinates
        (crow, ccol) = track[-1]
        # Find neighbouring cells that are walls
        neighbours = self.find_neighbours(crow, ccol, grid, True)

        # If no unvisited neighbouring cells, backtrack
        if len(neighbours) == 0:
            track = track[:-1]  # Remove the current cell from the track
        else:
            # Choose a random neighbouring cell and mark it as a passage
            nrow, ncol = neighbours[0]
            grid[nrow][ncol] = 0
            # Mark the cell between current and chosen cell as passage
            grid[(nrow + crow) // 2][(ncol + ccol) // 2] = 0
            # Move to the chosen neighbouring cell
            track += [(nrow, ncol)]

    return grid
```

Figure 28 backtracking generator code snippet.

The way this works is that it takes the dimensions of the maze from genAlgo this will then give shape to the empty array which we then fill with 1s then we pick a random starting point and mark that as 0 and then we will use a stack to track were we are currently then using a loop I used the find neighbours function to find the neighbours of the current cell and then from there I just followed how the algorithm works if there are available neighbours it randomly chooses one and goes that way each time changing the 1s to 0s to denote a passage if it can't find an unvisited neighbour it backtracks and starts again. Each movement is put to the top of the stack. This has had no real changes to the original design as it didn't have to.

Binary tree

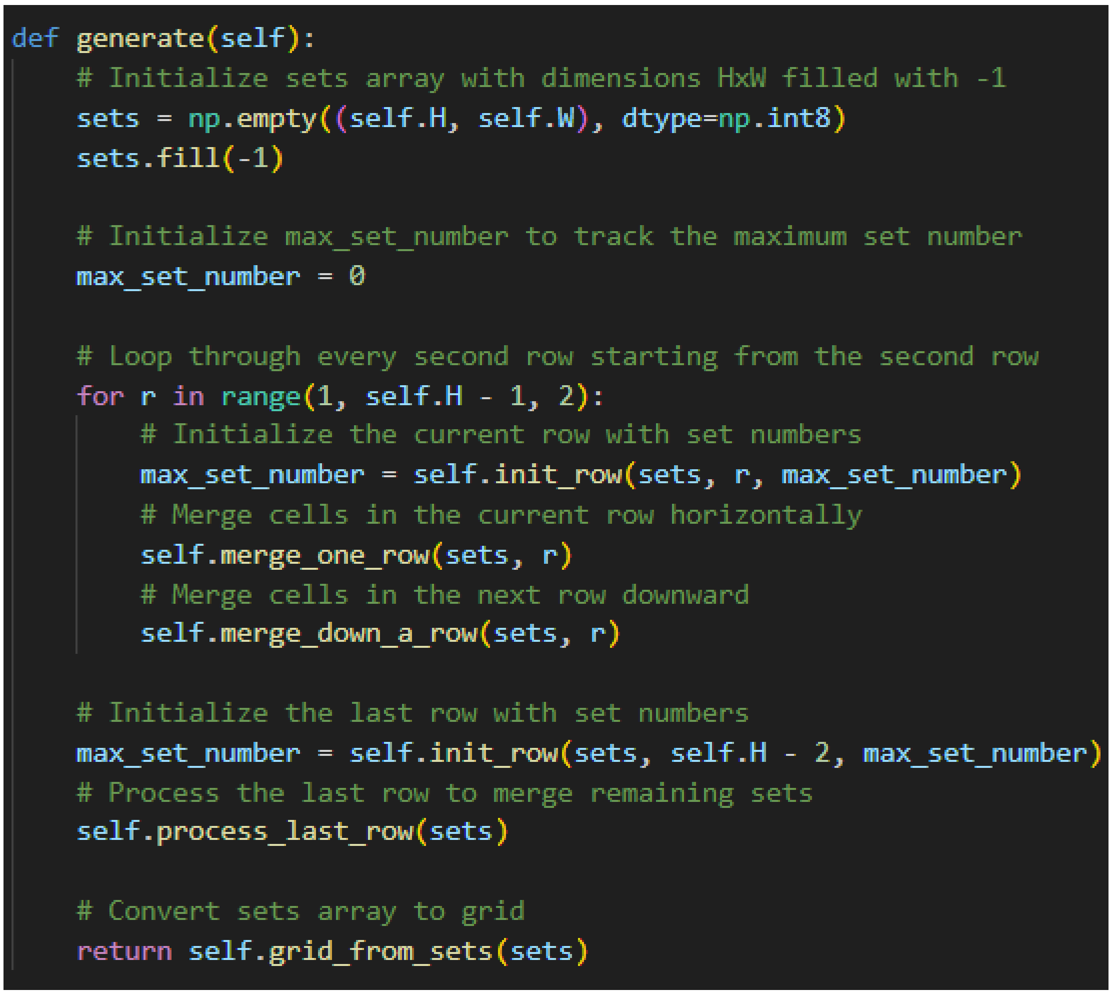


```python
def generate(self):
    # Create an empty array grid of dimensions HxW filled with walls (1s)
    grid = np.empty((self.H, self.W), dtype=np.int8)
    grid.fill(1)

    # Iterate over every second row and column in the grid
    for row in range(1, self.H, 2):
        for col in range(1, self.W, 2):
            # Carve out passages at every second row and column
            grid[row][col] = 0
            # Find the neighbouring cell according to the skew
            neighbour_row, neighbour_col = self.find_neighbour(row, col)
            # Carve out a passage at the neighbouring cell
            grid[neighbour_row][neighbour_col] = 0

    return grid

def find_neighbour(self, current_row, current_col):
    # Initialize a list to store neighbouring cell coordinates
    neighbours = []
    # Iterate over each skew direction
    for b_row, b_col in self.skew:
        # Calculate the coordinates of the neighbouring cell
        neighbour_row = current_row + b_row
        neighbour_col = current_col + b_col
        # Check if the neighbouring cell is within the bounds of the grid
        if 0 < neighbour_row < (self.H - 1) and 0 < neighbour_col < (self.W - 1):
            # Add the neighbouring cell coordinates to the list
            neighbours.append((neighbour_row, neighbour_col))

    # If there are no valid neighbouring cells, return the current cell
    if len(neighbours) == 0:
        return (current_row, current_col)
    else:
        # Choose one of the valid neighbouring cells randomly and return its coordinates
        return choice(neighbours)
```

Figure 29 Binary Tree code snippet.

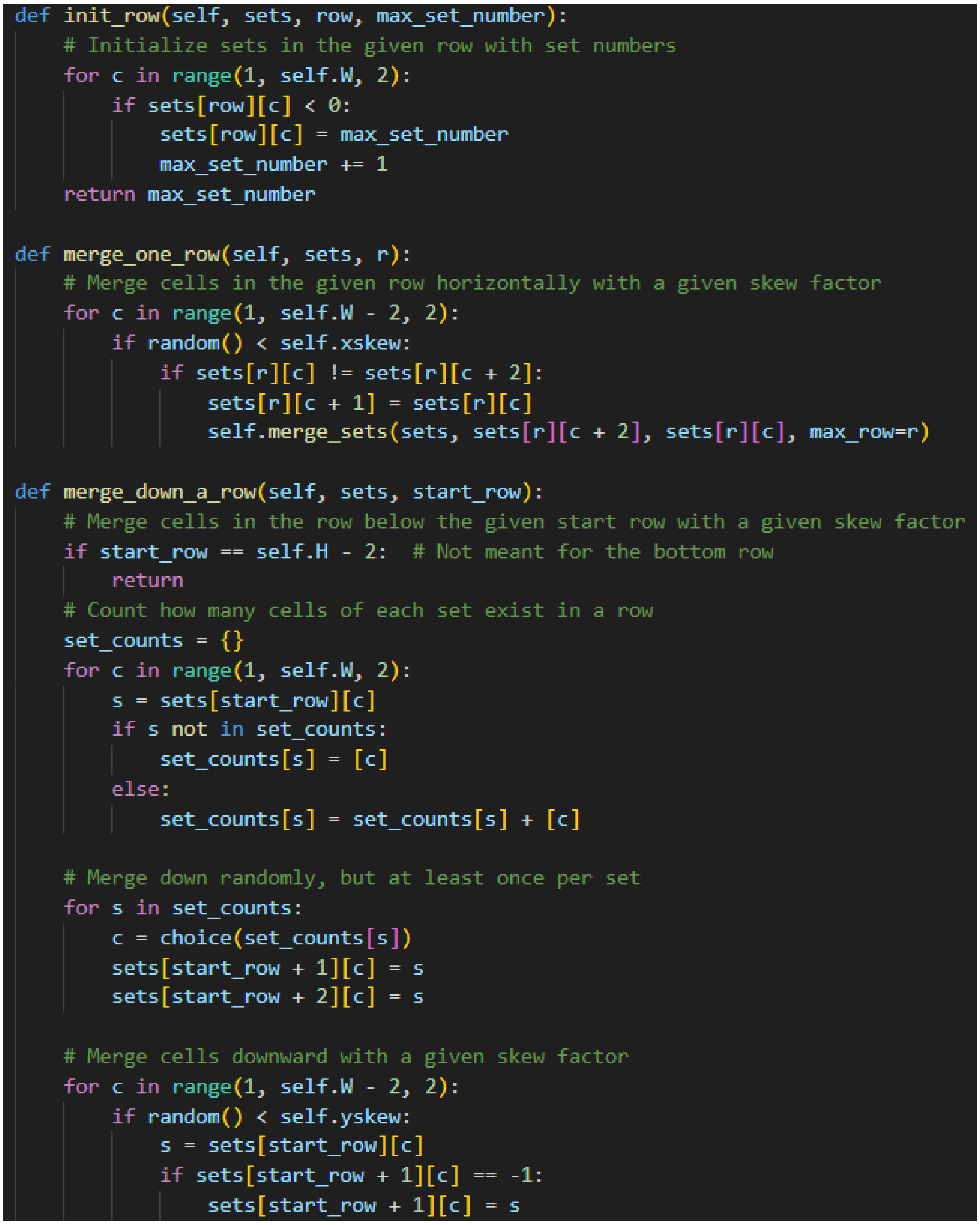
Just like the last algorithm an array is filled with 1s so that wall can be taken away during the process. In the find neighbours function this will use a skew function that has also been written for this class this can be seen in appendix B. In this function an empty list will be used to hold neighbours' coordinates. Using the skew, the coordinate of the neighbouring cell then checks whether they are in the bounds of the maze if they are they get added to the list and then if there are no valid neighbours the coordinates for the current cell are returned otherwise the neighbours do. This was done as the find neighbours of the genAlgo wouldn't really work with the binary tree algorithm as skew has to be used. For the generation, every second row and column are iterated over each iteration causes a passage of 0s to be made then the neighbouring cells is found and then 0s are added in the direction of the skew. This again hasn't changed much from initial design the only part that has really changed is that instead of keeping it to only one skew direction I've made it so that it can go in each direction randomly from the start I did this because it just made more sense as otherwise, I would be restricting the functionality of the generation algorithm.

Eller’s algorithm



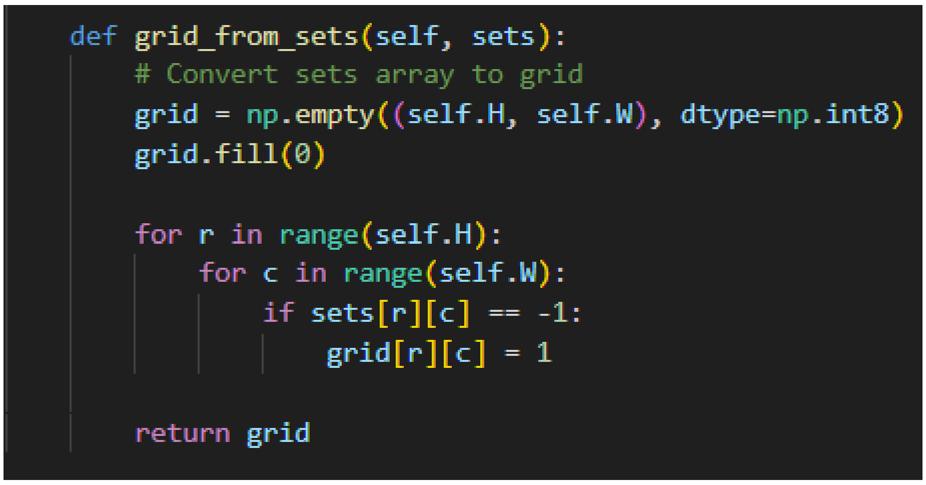
```python
def generate(self):
    # Initialize sets array with dimensions HxW filled with -1
    sets = np.empty((self.H, self.W), dtype=np.int8)
    sets.fill(-1)

    # Initialize max_set_number to track the maximum set number
    max_set_number = 0

    # Loop through every second row starting from the second row
    for r in range(1, self.H - 1, 2):
        # Initialize the current row with set numbers
        max_set_number = self.init_row(sets, r, max_set_number)
        # Merge cells in the current row horizontally
        self.merge_one_row(sets, r)
        # Merge cells in the next row downward
        self.merge_down_a_row(sets, r)

    # Initialize the last row with set numbers
    max_set_number = self.init_row(sets, self.H - 2, max_set_number)
    # Process the last row to merge remaining sets
    self.process_last_row(sets)

    # Convert sets array to grid
    return self.grid_from_sets(sets)
```

Figure 30 Ellers algorithm code snippet 1.



```python
def init_row(self, sets, row, max_set_number):
    # Initialize sets in the given row with set numbers
    for c in range(1, self.W, 2):
        if sets[row][c] < 0:
            sets[row][c] = max_set_number
            max_set_number += 1
    return max_set_number

def merge_one_row(self, sets, r):
    # Merge cells in the given row horizontally with a given skew factor
    for c in range(1, self.W - 2, 2):
        if random() < self.xskew:
            if sets[r][c] != sets[r][c + 2]:
                sets[r][c + 1] = sets[r][c]
                self.merge_sets(sets, sets[r][c + 2], sets[r][c], max_row=r)

def merge_down_a_row(self, sets, start_row):
    # Merge cells in the row below the given start row with a given skew factor
    if start_row == self.H - 2:  # Not meant for the bottom row
        return
    # Count how many cells of each set exist in a row
    set_counts = {}
    for c in range(1, self.W, 2):
        s = sets[start_row][c]
        if s not in set_counts:
            set_counts[s] = [c]
        else:
            set_counts[s] = set_counts[s] + [c]

    # Merge down randomly, but at least once per set
    for s in set_counts:
        c = choice(set_counts[s])
        sets[start_row + 1][c] = s
        sets[start_row + 2][c] = s

    # Merge cells downward with a given skew factor
    for c in range(1, self.W - 2, 2):
        if random() < self.yskew:
            s = sets[start_row][c]
            if sets[start_row + 1][c] == -1:
                sets[start_row + 1][c] = s
```

Figure 31 Ellers algorithms code snippet 2.

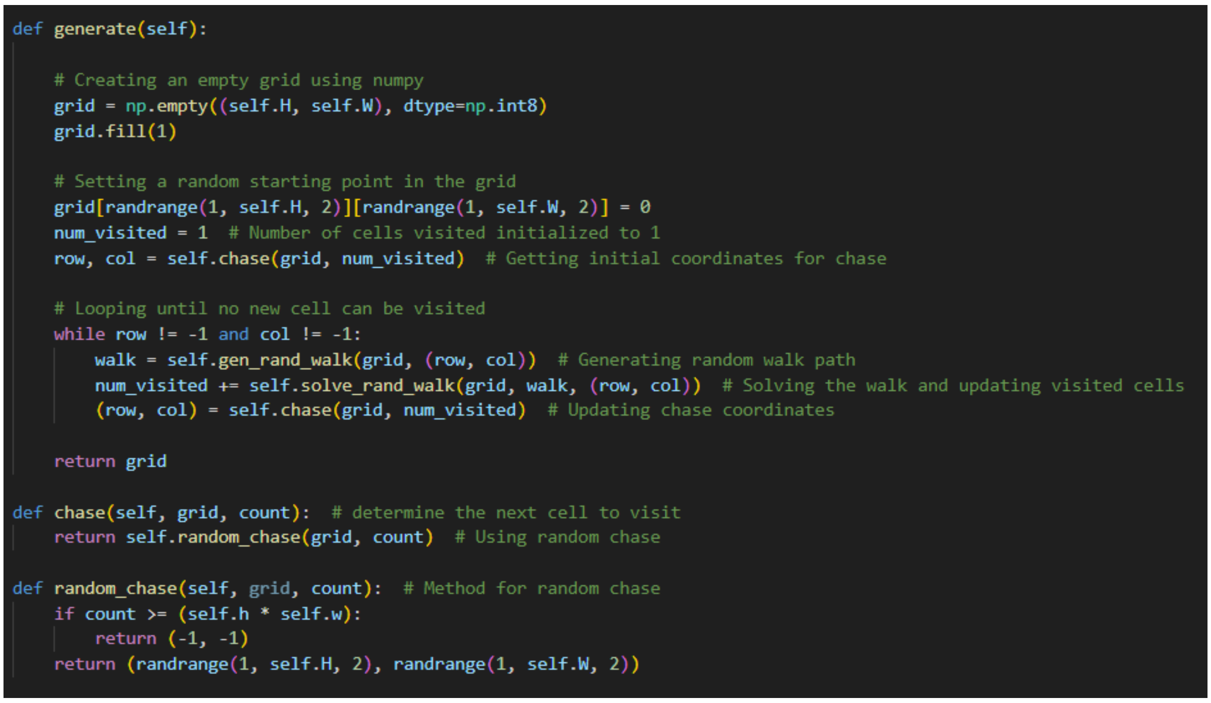


```
def grid_from_sets(self, sets):
    # Convert sets array to grid
    grid = np.empty((self.H, self.W), dtype=np.int8)
    grid.fill(0)

    for r in range(self.H):
        for c in range(self.W):
            if sets[r][c] == -1:
                grid[r][c] = 1

    return grid
```

Figure 32 Ellers algorithm code snippet part 3.

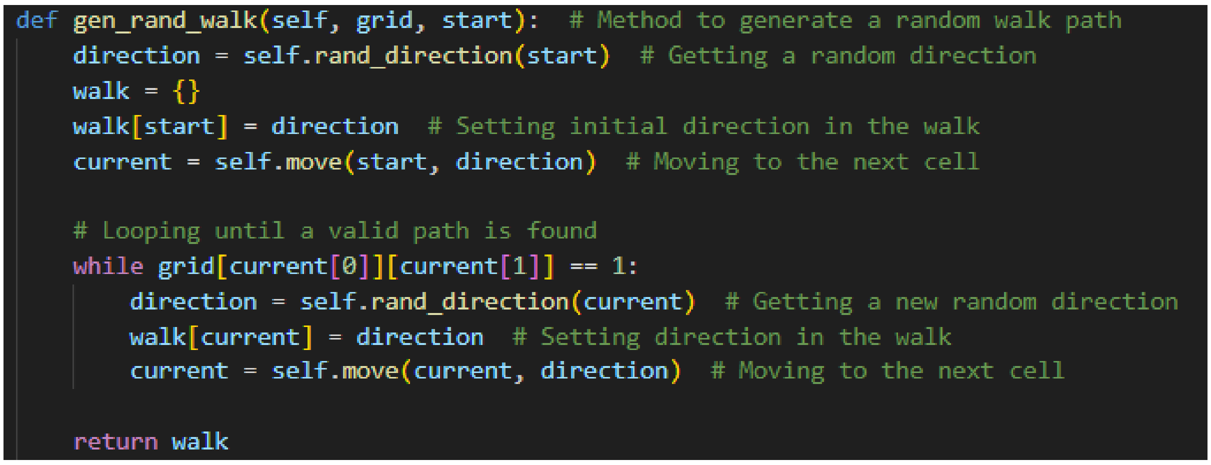
Skew again has been used to influence the horizontal and vertical merging of the sets this can be seen in appendix B along with other missing functions that can't be seen here. Following the generate function, the array is filled to the height and width, but this is filled with -1s to indicate uninitialized sets. The algorithm will iterate through every second row starting the first to the last row. Each row is then given a set number so that we can tell which cell belongs to which set. Then it will merge the given rows together depending on the skew factor meaning some will be connected were some wont this will lead to the walls of the maze being made. This again is implemented with parameters so that outer edges of the maze are not included this is done in merge_sets which can be seen in appendix B. Then after that cells need to be merged to the row below this again is influenced by the vertical skew factor and creates vertical connections between cells in consecutive rows. But this will also make sure that every row has at least one vertical connection otherwise parts of the maze would be cut off this can be seen in the merge_down_a_row function. Once the for loop reaches the last row then it ends, and process last row is called this can be seen in appendix B this just merges the cells in the last row together. Once everything is done the array is still full of -1s so the array is then filled with 0s and then the walls are added using 1s and the grid is returned. There are no real changes from the design for this algorithm.

Wilson’s algorithm



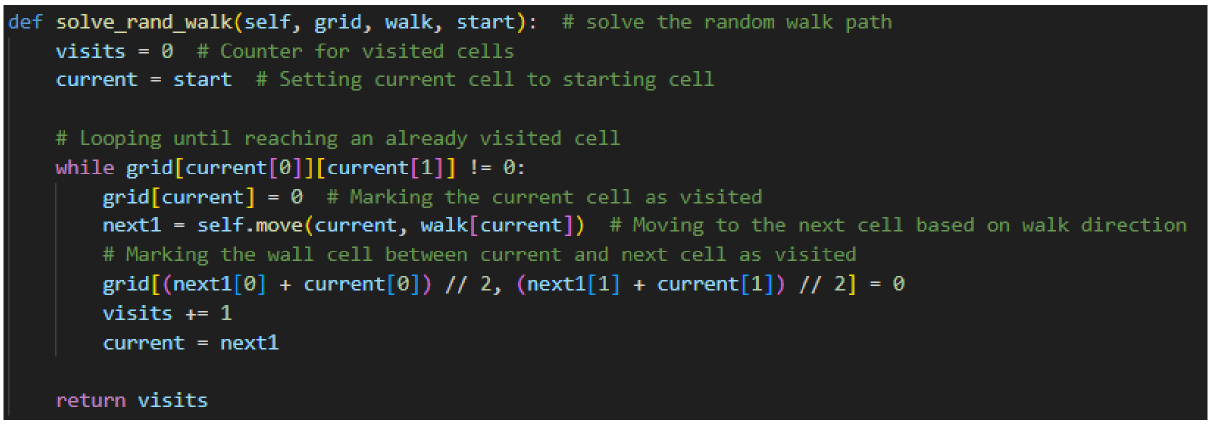
```python
def generate(self):

    # Creating an empty grid using numpy
    grid = np.empty((self.H, self.W), dtype=np.int8)
    grid.fill(1)

    # Setting a random starting point in the grid
    grid[randrange(1, self.H, 2)][randrange(1, self.W, 2)] = 0
    num_visited = 1  # Number of cells visited initialized to 1
    row, col = self.chase(grid, num_visited)  # Getting initial coordinates for chase

    # Looping until no new cell can be visited
    while row != -1 and col != -1:
        walk = self.gen_rand_walk(grid, (row, col))  # Generating random walk path
        num_visited += self.solve_rand_walk(grid, walk, (row, col))  # Solving the walk and updating visited cells
        (row, col) = self.chase(grid, num_visited)  # Updating chase coordinates

    return grid

def chase(self, grid, count):  # determine the next cell to visit
    return self.random_chase(grid, count)  # Using random chase

def random_chase(self, grid, count):  # Method for random chase
    if count >= (self.h * self.w):
        return (-1, -1)
    return (randrange(1, self.H, 2), randrange(1, self.W, 2))
```

Figure 33 Wilsons algorithm code snippet 1.



```python
def gen_rand_walk(self, grid, start):  # Method to generate a random walk path
    direction = self.rand_direction(start)  # Getting a random direction
    walk = {}
    walk[start] = direction  # Setting initial direction in the walk
    current = self.move(start, direction)  # Moving to the next cell

    # Looping until a valid path is found
    while grid[current[0]][current[1]] == 1:
        direction = self.rand_direction(current)  # Getting a new random direction
        walk[current] = direction  # Setting direction in the walk
        current = self.move(current, direction)  # Moving to the next cell

    return walk
```

Figure 34 Wilsons algorithm code snippet 2.

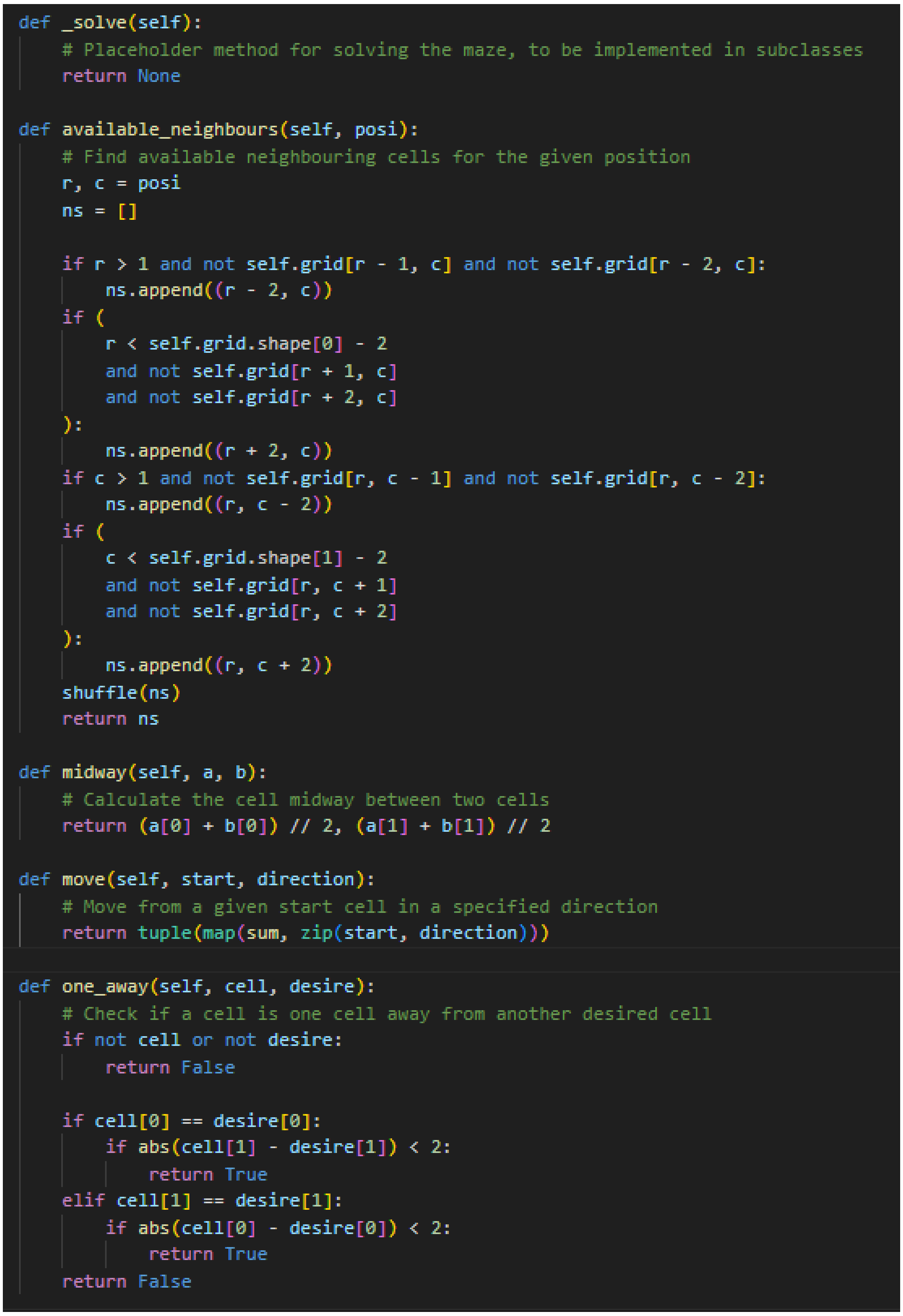


```
def solve_rand_walk(self, grid, walk, start):  # solve the random walk path
    visits = 0  # Counter for visited cells
    current = start  # Setting current cell to starting cell

    # Looping until reaching an already visited cell
    while grid[current[0]][current[1]] != 0:
        grid[current] = 0  # Marking the current cell as visited
        next1 = self.move(current, walk[current])  # Moving to the next cell based on walk direction
        # Marking the wall cell between current and next cell as visited
        grid[(next1[0] + current[0]) // 2, (next1[1] + current[1]) // 2] = 0
        visits += 1
        current = next1

    return visits
```

Figure 35 Wilsons algorithm code snippet 3.

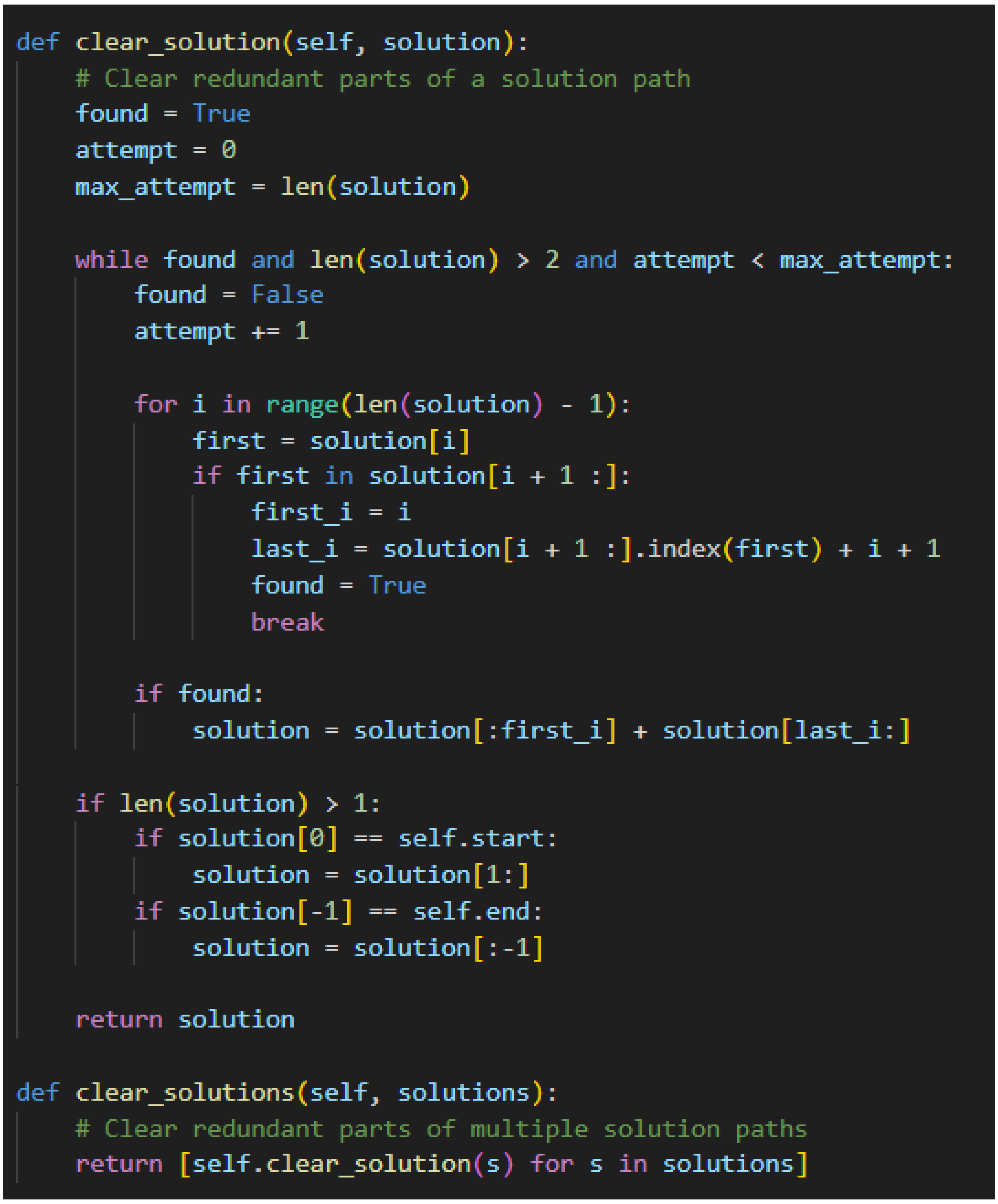
Again, the grid is setup as an array full of 1s. Then a random starting point is selected in the grid and then a counter for how many cells have been visited is set to 1. Then the cell that the random walk will start from will be chosen this as well will be done randomly again. Then everything else will loop until all cells have been visited. After this the random walk will be generated a dictionary will be returned that is mapping the cell to a direction and if the walk goes over the same cell twice the direction is overwritten changing it. In the generate_rand_walk function the move function is used this is what is used to change the current position in the random walk this can again be seen in appendix B. Once the first cell and the rand walk meet then it needs to be solved the way this is done is by following the rand walk and when a 1 is come by in the array this will be changed to a 0 this will create the passages of the maze. Then a new starting location for the random walk is given these loops until all cells are visited cells visited is always incremented at the end of the solving for the random walk. One thing that has changed from the design is that in the design only lists were used but in the end tuple and dictionaries were used this is because they were necessary for the implementation especially because of the directions that are used. Another part is that for the selection cell for the chase although I did plan on using random selection other styles can be used such as just cycling through each cell in order to find an unvisited cell, but I wanted to stick with more randomness as the other algorithms have random elements in them.

SolveAlgo



```python
def _solve(self):
    # Placeholder method for solving the maze, to be implemented in subclasses
    return None

def available_neighbours(self, posi):
    # Find available neighbouring cells for the given position
    r, c = posi
    ns = []

    if r > 1 and not self.grid[r - 1, c] and not self.grid[r - 2, c]:
        ns.append((r - 2, c))
    if (
        r < self.grid.shape[0] - 2
        and not self.grid[r + 1, c]
        and not self.grid[r + 2, c]
    ):
        ns.append((r + 2, c))
    if c > 1 and not self.grid[r, c - 1] and not self.grid[r, c - 2]:
        ns.append((r, c - 2))
    if (
        c < self.grid.shape[1] - 2
        and not self.grid[r, c + 1]
        and not self.grid[r, c + 2]
    ):
        ns.append((r, c + 2))
    shuffle(ns)
    return ns

def midway(self, a, b):
    # Calculate the cell midway between two cells
    return (a[0] + b[0]) // 2, (a[1] + b[1]) // 2

def move(self, start, direction):
    # Move from a given start cell in a specified direction
    return tuple(map(sum, zip(start, direction)))

def one_away(self, cell, desire):
    # Check if a cell is one cell away from another desired cell
    if not cell or not desire:
        return False

    if cell[0] == desire[0]:
        if abs(cell[1] - desire[1]) < 2:
            return True
    elif cell[1] == desire[1]:
        if abs(cell[0] - desire[0]) < 2:
            return True
    return False
```

Figure 36 solveAlgo code snippet 1.

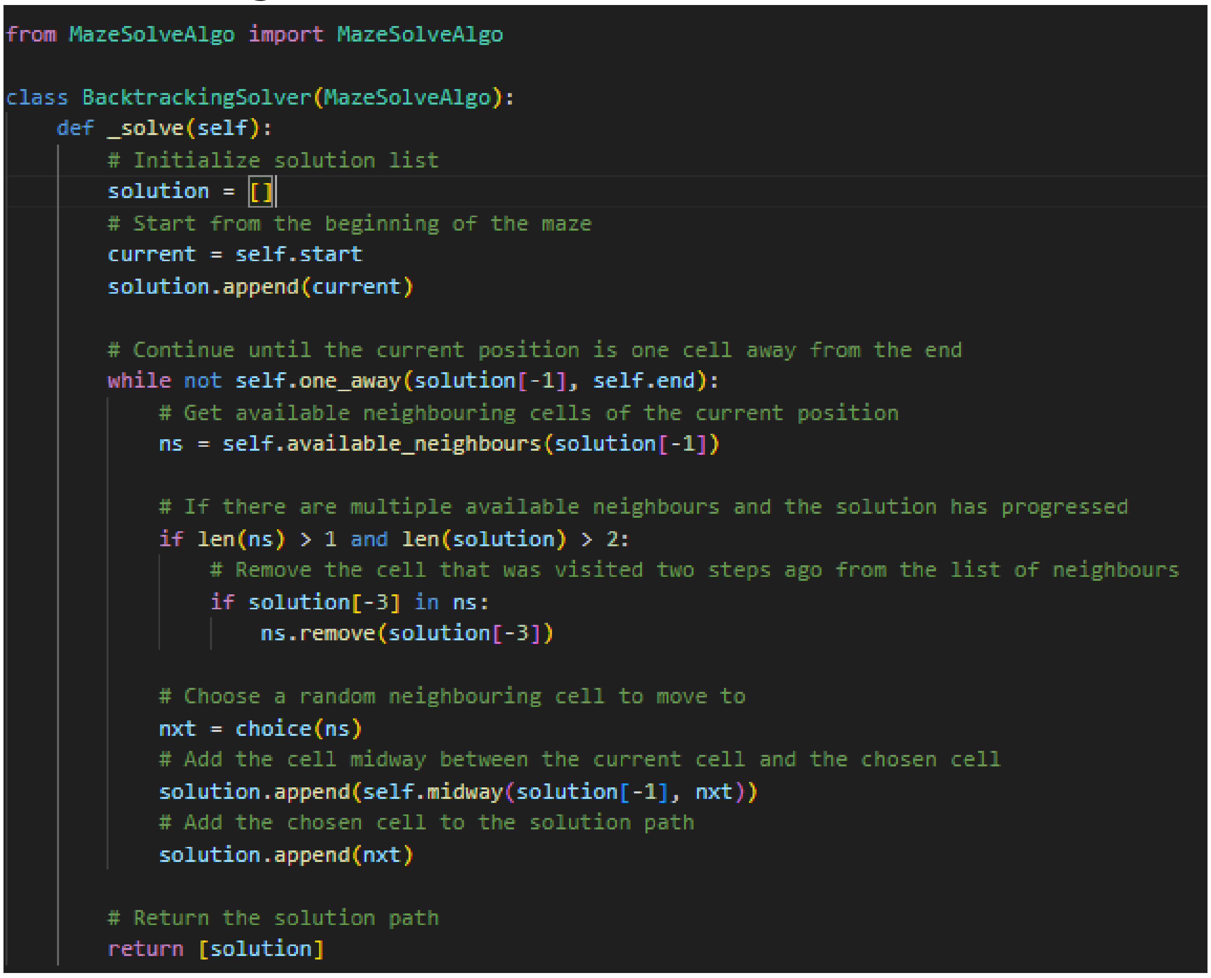


```python
def clear_solution(self, solution):
    # Clear redundant parts of a solution path
    found = True
    attempt = 0
    max_attempt = len(solution)

    while found and len(solution) > 2 and attempt < max_attempt:
        found = False
        attempt += 1

        for i in range(len(solution) - 1):
            first = solution[i]
            if first in solution[i + 1 :]:
                first_i = i
                last_i = solution[i + 1 :].index(first) + i + 1
                found = True
                break

        if found:
            solution = solution[:first_i] + solution[last_i:]

    if len(solution) > 1:
        if solution[0] == self.start:
            solution = solution[1:]
        if solution[-1] == self.end:
            solution = solution[:-1]

    return solution

def clear_solutions(self, solutions):
    # Clear redundant parts of multiple solution paths
    return [self.clear_solution(s) for s in solutions]
```

Figure 37 solveAlgo code snippet 2.

This is similar to genAlgo in that it is made up of a lot of helper methods that can be used in all of the solving algorithms. Just like in genAlgo a placeholder is made for the solving algorithms that can be used on the mazes. Available neighbours will take the current position through a tuple and then it will check all adjacent cells and return only the ones that are not walls. Midway will be used to find the position of the wall cell between two passage cells this is done by calculating the average of the row and column indices of the two cells. Move is used to move between by using the current position and the direction a new position is created. On away is used to check if the current position is one cell away from the end this can be a full move or could be on a midpoint cell this will return a Boolean value. Clear solution is used to remove any unnecessary backtracking or loops in the final outcome it does this by iterating through each solution path this function also makes sure that the start and end points are not duplicated in the final solution path. This is the same for clear solutions this is just used when their multiple solutions.

Backtracking solver

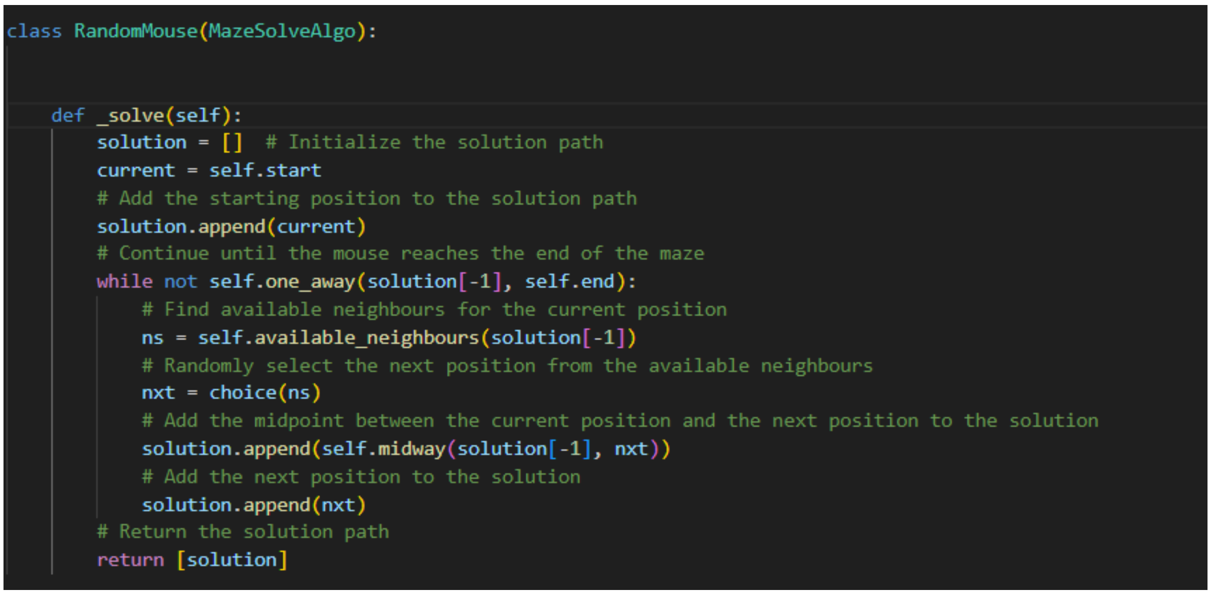


```python
from MazeSolveAlgo import MazeSolveAlgo

class BacktrackingSolver(MazeSolveAlgo):
    def _solve(self):
        # Initialize solution list
        solution = []
        # Start from the beginning of the maze
        current = self.start
        solution.append(current)

        # Continue until the current position is one cell away from the end
        while not self.one_away(solution[-1], self.end):
            # Get available neighbouring cells of the current position
            ns = self.available_neighbours(solution[-1])

            # If there are multiple available neighbours and the solution has progressed
            if len(ns) > 1 and len(solution) > 2:
                # Remove the cell that was visited two steps ago from the list of neighbours
                if solution[-3] in ns:
                    ns.remove(solution[-3])

            # Choose a random neighbouring cell to move to
            nxt = choice(ns)
            # Add the cell midway between the current cell and the chosen cell
            solution.append(self.midway(solution[-1], nxt))
            # Add the chosen cell to the solution path
            solution.append(nxt)

        # Return the solution path
        return [solution]
```

Figure 38 Backtracking solver algorithm code snippet.

Now because all of the helper algorithms are setup, they can be used to make these algorithms. So, solve will now override the _solve method inherited from solvealgo. The solution path will be stored using a list. It starts with the first co-ordinate being saved to the list. This will iterate until the current position is one away from the end this done by using one away from solvealgo. It gets the available neighbouring cells for the current position. If there are multiple direction for the solver to go and then they go down that direction it will remove the cell that had been visited two steps ago this is, so backtracking is avoided. Using choice () it will randomly pick an available neighbouring cell and move to it and then that movement as well as the midway cell will also be added to the solution. This has had no real changes from the design this because there hasn't needed to be as the algorithm is the same through many examples.

Random Mouse

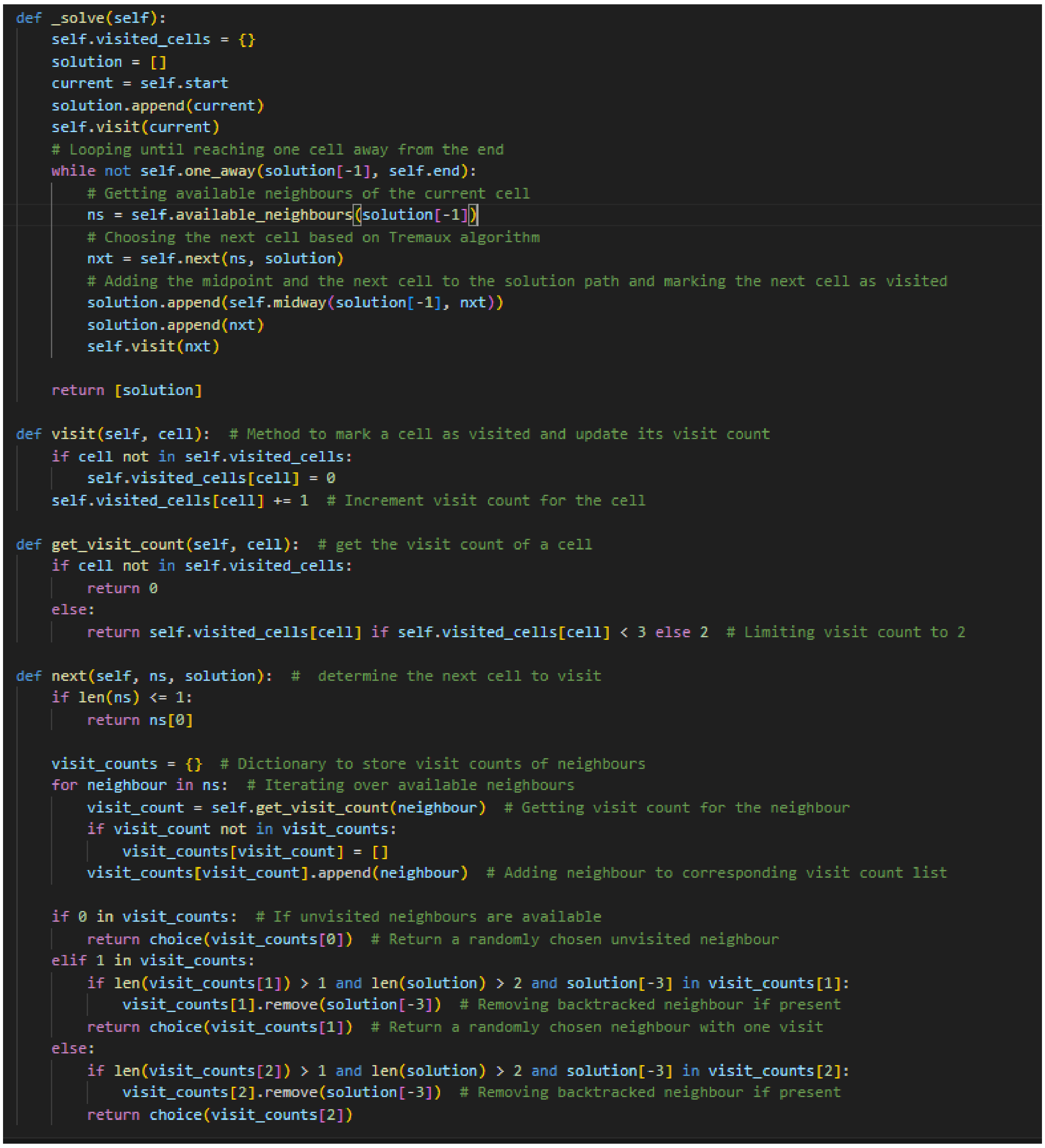


```
class RandomMouse(MazeSolveAlgo):


    def _solve(self):
        solution = []  # Initialize the solution path
        current = self.start
        # Add the starting position to the solution path
        solution.append(current)
        # Continue until the mouse reaches the end of the maze
        while not self.one_away(solution[-1], self.end):
            # Find available neighbours for the current position
            ns = self.available_neighbours(solution[-1])
            # Randomly select the next position from the available neighbours
            nxt = choice(ns)
            # Add the midpoint between the current position and the next position to the solution
            solution.append(self.midway(solution[-1], nxt))
            # Add the next position to the solution
            solution.append(nxt)
        # Return the solution path
        return [solution]
```

Figure 39 Random mouse code snippet.

The way this works is that again the solution is stored in list and the first item is that starting position is then it loops until it is one away from the end and then find available neighbours and then picks one using choice and moves in that direction and then the current position is added to the list along with the midway cell that was moved over to get to it. Then the solution is returned. This is very similar to backtracking solver it just doesn’t remove past cells travelled and again nothing has really changed from the design again as this was a simple algorithm.

Trémaux algorithm

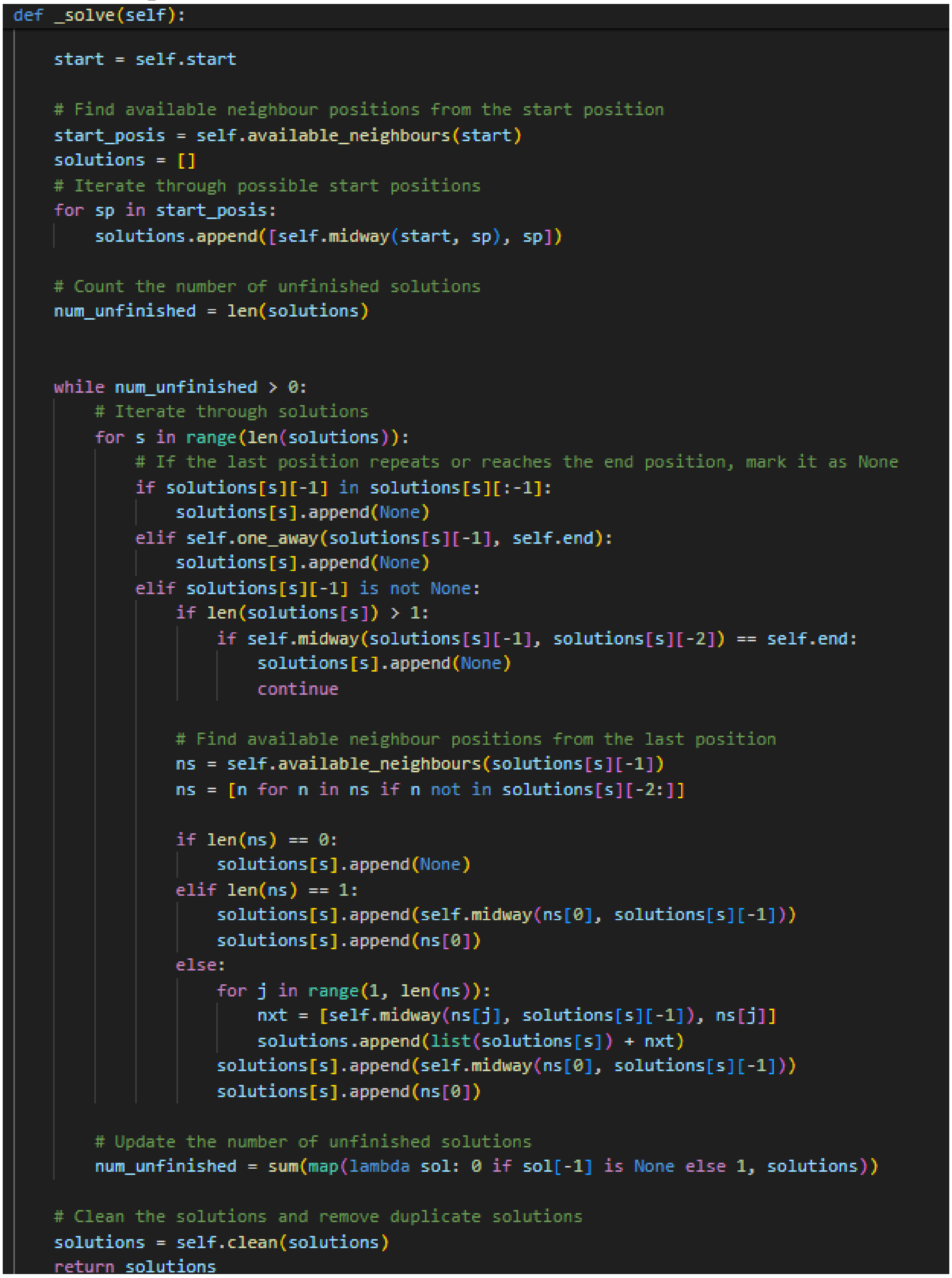


```python
def _solve(self):
    self.visited_cells = {}
    solution = []
    current = self.start
    solution.append(current)
    self.visit(current)
    # Looping until reaching one cell away from the end
    while not self.one_away(solution[-1], self.end):
        # Getting available neighbours of the current cell
        ns = self.available_neighbours(solution[-1])
        # Choosing the next cell based on Tremaux algorithm
        nxt = self.next(ns, solution)
        # Adding the midpoint and the next cell to the solution path and marking the next cell as visited
        solution.append(self.midway(solution[-1], nxt))
        solution.append(nxt)
        self.visit(nxt)

    return [solution]

def visit(self, cell):  # Method to mark a cell as visited and update its visit count
    if cell not in self.visited_cells:
        self.visited_cells[cell] = 0
    self.visited_cells[cell] += 1  # Increment visit count for the cell

def get_visit_count(self, cell):  # get the visit count of a cell
    if cell not in self.visited_cells:
        return 0
    else:
        return self.visited_cells[cell] if self.visited_cells[cell] < 3 else 2  # Limiting visit count to 2

def next(self, ns, solution):  #  determine the next cell to visit
    if len(ns) <= 1:
        return ns[0]

    visit_counts = {}  # Dictionary to store visit counts of neighbours
    for neighbour in ns:  # Iterating over available neighbours
        visit_count = self.get_visit_count(neighbour)  # Getting visit count for the neighbour
        if visit_count not in visit_counts:
            visit_counts[visit_count] = []
        visit_counts[visit_count].append(neighbour)  # Adding neighbour to corresponding visit count list

    if 0 in visit_counts:  # If unvisited neighbours are available
        return choice(visit_counts[0])  # Return a randomly chosen unvisited neighbour
    elif 1 in visit_counts:
        if len(visit_counts[1]) > 1 and len(solution) > 2 and solution[-3] in visit_counts[1]:
            visit_counts[1].remove(solution[-3])  # Removing backtracked neighbour if present
        return choice(visit_counts[1])  # Return a randomly chosen neighbour with one visit
    else:
        if len(visit_counts[2]) > 1 and len(solution) > 2 and solution[-3] in visit_counts[2]:
            visit_counts[2].remove(solution[-3])  # Removing backtracked neighbour if present
        return choice(visit_counts[2])
```

Figure 40 Trémaux algorithm code snippet.

Trémaux algorithm starts like the other algorithms with the first position being saved to the solution but this time we need to keep track of the visited cells so that the algorithm can make informed decisions. Trémaux works similar to the other algorithms I have shown in terms of the order everything is done just that the way the steps are done are different this is because of functions like visit and get visit count when a cell is visited along with the coordinates of that cell in the dictionary there is an increase of one and with get_visit_count which will limit that to 2 as if it goes over that it is always brought back down to 2 this is because this is how the algorithm is meant to work. The next function is how the algorithm determines how to actually move and in which direction. It does this by getting each neighbouring cells visit count and then the visit count of the neighbours is used as a key in the dictionary and then the cell itself is added to the corresponding list of cells with that visit count. Then it needs to decide where to go next to do this it checks with if statements whether the neighbouring cells have been visited none, once or twice and then this is hierarchical in that if there is one with none and the rest have been visited it chooses the none and it makes sure that it can't backtrack. For the solve again the only difference is that for the move it uses the Trémaux informed move so that it gets the next cell that fits the criteria but as always that is added to the solution and so is the midway cell it just then marks that cell as visited. Nothing has really changed from the design.

Shortest path

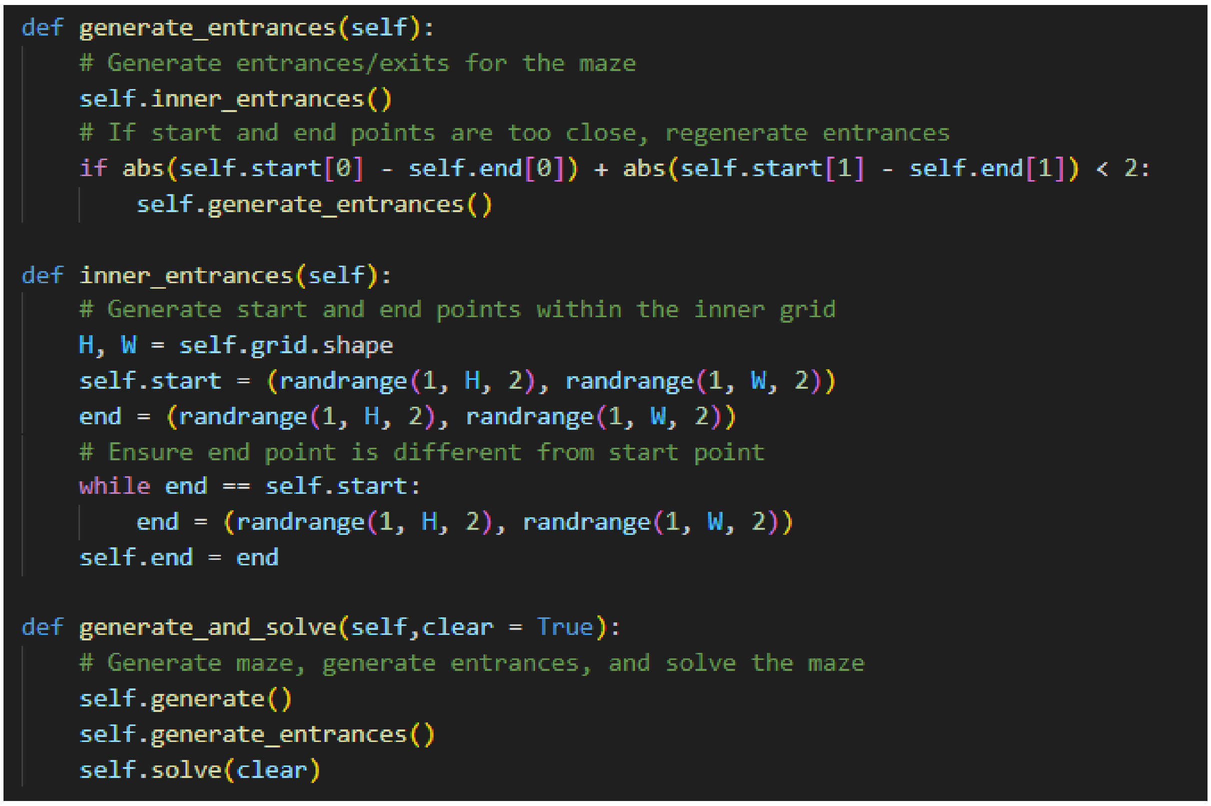


```python
def _solve(self):

    start = self.start

    # Find available neighbour positions from the start position
    start_posis = self.available_neighbours(start)
    solutions = []
    # Iterate through possible start positions
    for sp in start_posis:
        solutions.append([self.midway(start, sp), sp])

    # Count the number of unfinished solutions
    num_unfinished = len(solutions)


    while num_unfinished > 0:
        # Iterate through solutions
        for s in range(len(solutions)):
            # If the last position repeats or reaches the end position, mark it as None
            if solutions[s][-1] in solutions[s][:-1]:
                solutions[s].append(None)
            elif self.one_away(solutions[s][-1], self.end):
                solutions[s].append(None)
            elif solutions[s][-1] is not None:
                if len(solutions[s]) > 1:
                    if self.midway(solutions[s][-1], solutions[s][-2]) == self.end:
                        solutions[s].append(None)
                        continue

                # Find available neighbour positions from the last position
                ns = self.available_neighbours(solutions[s][-1])
                ns = [n for n in ns if n not in solutions[s][-2:]]

                if len(ns) == 0:
                    solutions[s].append(None)
                elif len(ns) == 1:
                    solutions[s].append(self.midway(ns[0], solutions[s][-1]))
                    solutions[s].append(ns[0])
                else:
                    for j in range(1, len(ns)):
                        nxt = [self.midway(ns[j], solutions[s][-1]), ns[j]]
                        solutions.append(list(solutions[s]) + nxt)
                    solutions[s].append(self.midway(ns[0], solutions[s][-1]))
                    solutions[s].append(ns[0])

        # Update the number of unfinished solutions
        num_unfinished = sum(map(lambda sol: 0 if sol[-1] is None else 1, solutions))

    # Clean the solutions and remove duplicate solutions
    solutions = self.clean(solutions)
    return solutions
```

Figure 41 Shortest path code snippet.

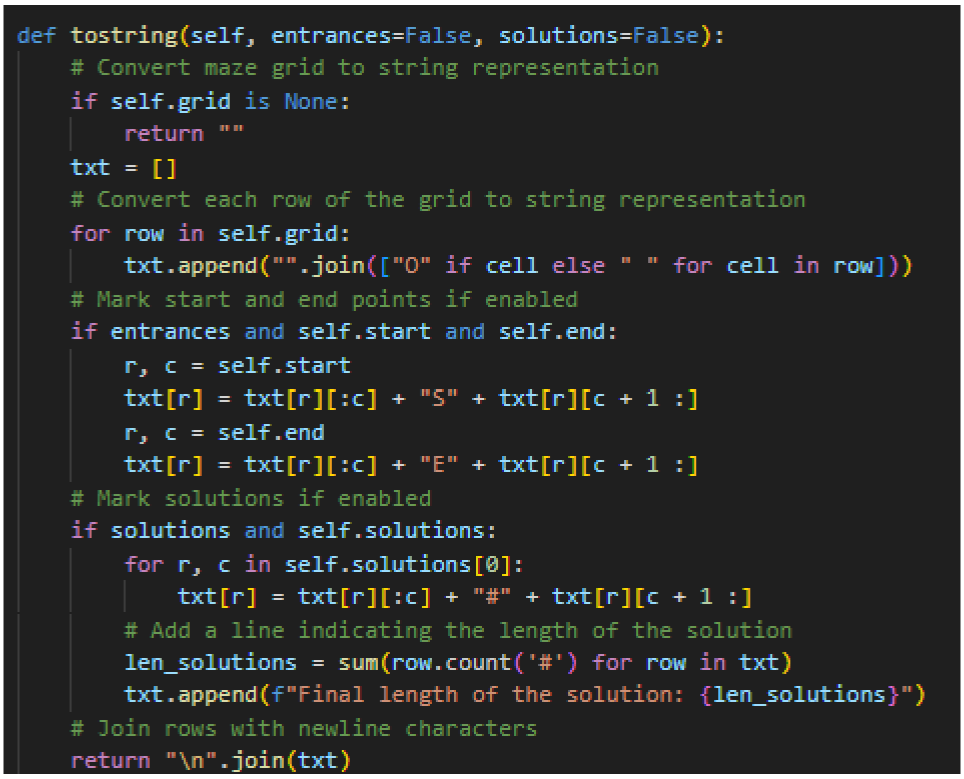
The way shortest path works is different from the other algorithms naturally as this is a BFS based algorithm. It starts off by finding available neighbours from the starting position and also makes a list for the solutions the algorithm will find. It then iterates through each possible start position and then adds the initial path from the start to the other starting position and the midway cell between those cells. Then a loop begins that doesn't end until there are no unfinished solutions. If the last position repeats of reaches the end position it is, then marked as none but if it not none then it checks the midway cell that is between the last position and the position before that to see if it is the end and if it is it is marked as none. Then it will find the available neighbour positions from the last position and then it will filter out the already visited neighbours. If there is only one path that it can go down after checking for the visit, then it picks that and the midway and the cell are appended otherwise if there are multiple then it makes a new solution for each possible way and appends them to solutions. After the solution has been found then the other paths need to be cleaned this can be seen in appendix B. This clean will remove any paths where the second to last position is one cell away from the end as the paths could have been more efficient meaning that the first path found should always be the fastest and then it removes duplicate solutions it does this by converting the list to a tuple and then a tuple to a set to remove duplicates and then back to a list, so the solution is unique. This hasn't really changed from my original design.

Maze



```
def generate_entrances(self):
    # Generate entrances/exits for the maze
    self.inner_entrances()
    # If start and end points are too close, regenerate entrances
    if abs(self.start[0] - self.end[0]) + abs(self.start[1] - self.end[1]) < 2:
        self.generate_entrances()

def inner_entrances(self):
    # Generate start and end points within the inner grid
    H, W = self.grid.shape
    self.start = (randrange(1, H, 2), randrange(1, W, 2))
    end = (randrange(1, H, 2), randrange(1, W, 2))
    # Ensure end point is different from start point
    while end == self.start:
        end = (randrange(1, H, 2), randrange(1, W, 2))
    self.end = end

def generate_and_solve(self,clear = True):
    # Generate maze, generate entrances, and solve the maze
    self.generate()
    self.generate_entrances()
    self.solve(clear)
```

Figure 42 maze code snippet 1.

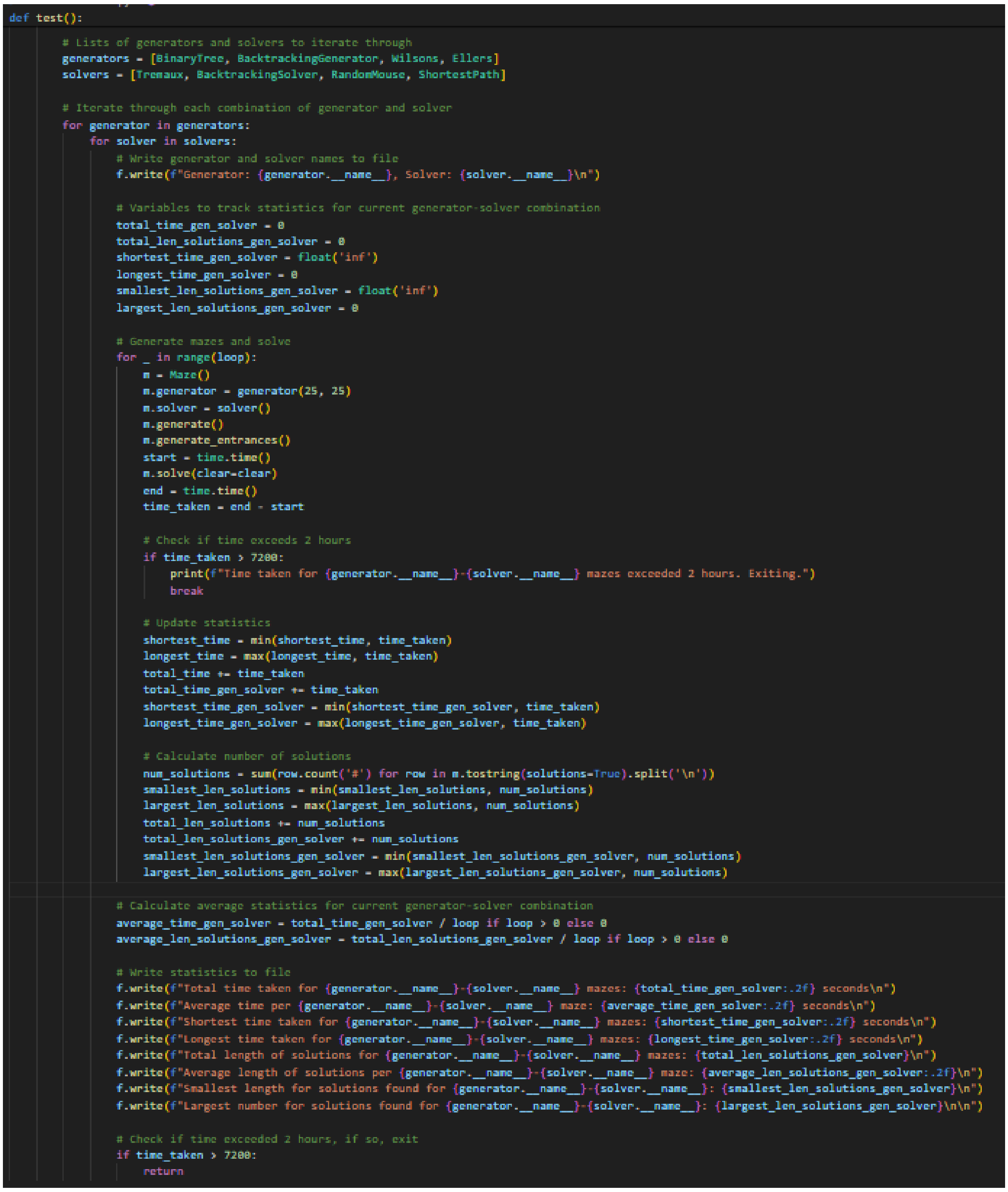


```python
def tostring(self, entrances=False, solutions=False):
    # Convert maze grid to string representation
    if self.grid is None:
        return ""
    txt = []
    # Convert each row of the grid to string representation
    for row in self.grid:
        txt.append("".join(["0" if cell else " " for cell in row]))
    # Mark start and end points if enabled
    if entrances and self.start and self.end:
        r, c = self.start
        txt[r] = txt[r][:c] + "S" + txt[r][c + 1 :]
        r, c = self.end
        txt[r] = txt[r][:c] + "E" + txt[r][c + 1 :]
    # Mark solutions if enabled
    if solutions and self.solutions:
        for r, c in self.solutions[0]:
            txt[r] = txt[r][:c] + "#" + txt[r][c + 1 :]
        # Add a line indicating the length of the solution
        len_solutions = sum(row.count('#') for row in txt)
        txt.append(f"Final length of the solution: {len_solutions}")
    # Join rows with newline characters
    return "\n".join(txt)
```

Figure 43 maze code snippet 2.

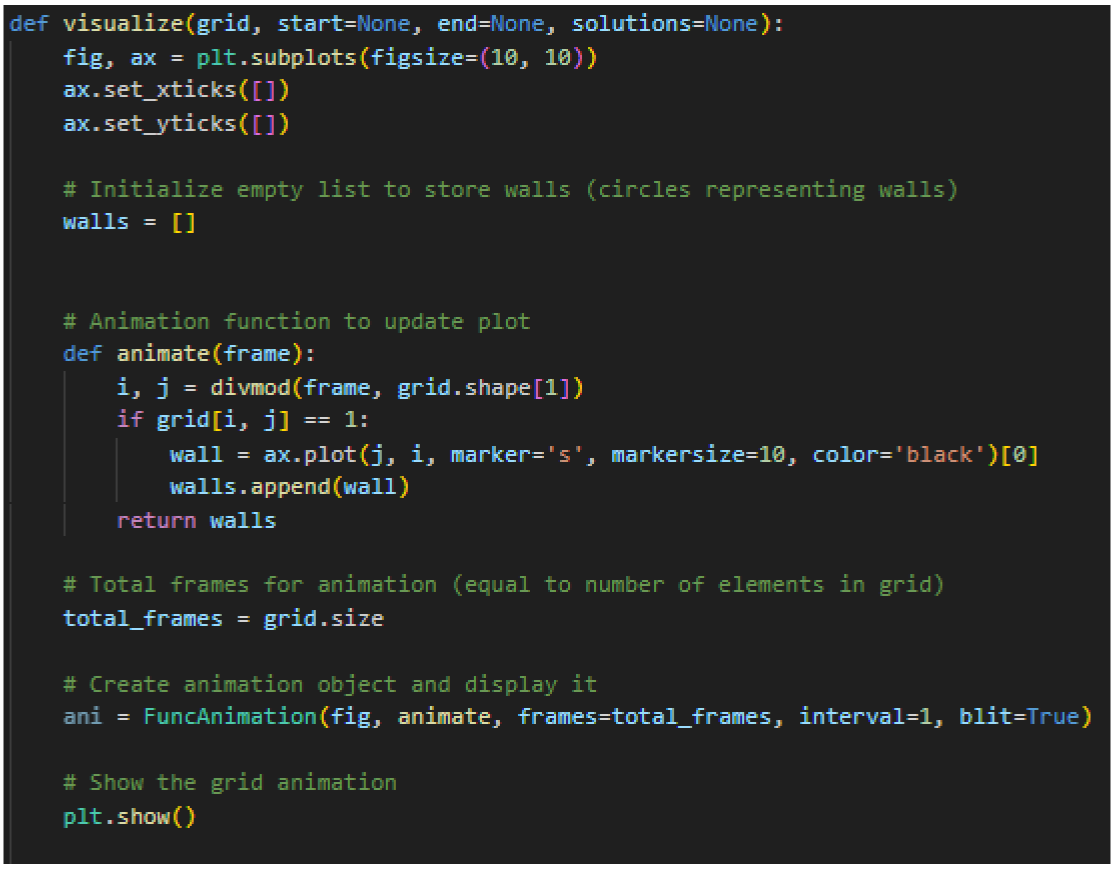
This file has instance variables at the top that initializes parts of the maze such as the start and end along with the generator and the solver along with the solutions. There is also a call for the generate method of the generator object this creates the maze. This can be seen in appendix B. The main part of this file is to generate the entrances of the maze just like what I said in the design these entrances can only be made on the inside of the maze. The way this works is that the inner entrances method will randomly select the co-ordinates for the start and end points and makes sure that they are inside the maze and also are placed on odd columns and rows this is so that they are not placed on any walls. It also checks that the start and end points are not placed on top of each other. Then in generate entrances it checks whether the start and end are too close to one another it does this by calculating the absolute value between them and if it is less than 2, they are too close together meaning the path would be pointless. If this is the case, then it just calls itself again and repeats the process. In the generate and solve function you can see in the solve function we pass clear I will expand on this in the test file section. The other main part of this file is the string representation of the maze. What this does is it converts everything like the grid and the start, end, and solution into string by using chars first of all the wall of the maze are converted into '0' and the empty spaces are left blank. The start is marked with and 'S' and the end with an 'E' and the solution is converted into'#.' This is all appended to the list txt. With that as well there is also a count to get the length of the solution this just count how many instances there are of '#' in the final list which will get the length of the solution this will be expanded upon in the test file part as well. This has not really changed from my original design other than what I originally planned to put the chars as I put them to what I originally wanted in the design, but it didn't look that clear and it was hard to determine certain parts of the maze.

Test


```python
def test():

        # Lists of generators and solvers to iterate through
        generators = [BinaryTree, BacktrackingGenerator, Wilsons, Ellers]
        solvers = [Tremaux, BacktrackingSolver, RandomMouse, ShortestPath]

        # Iterate through each combination of generator and solver
        for generator in generators:
            for solver in solvers:
                # Write generator and solver names to file
                f.write(f"Generator: {generator.__name__}, Solver: {solver.__name__}\n")

                # Variables to track statistics for current generator-solver combination
                total_time_gen_solver = 0
                total_len_solutions_gen_solver = 0
                shortest_time_gen_solver = float('inf')
                longest_time_gen_solver = 0
                smallest_len_solutions_gen_solver = float('inf')
                largest_len_solutions_gen_solver = 0

                # Generate mazes and solve
                for _ in range(loop):
                    m = Maze()
                    m.generator = generator(25, 25)
                    m.solver = solver()
                    m.generate()
                    m.generate_entrances()
                    start = time.time()
                    m.solve(clear=clear)
                    end = time.time()
                    time_taken = end - start

                    # Check if time exceeds 2 hours
                    if time_taken > 7200:
                        print(f"Time taken for {generator.__name__}-{solver.__name__} mazes exceeded 2 hours. Exiting.")
                        break

                    # Update statistics
                    shortest_time = min(shortest_time, time_taken)
                    longest_time = max(longest_time, time_taken)
                    total_time += time_taken
                    total_time_gen_solver += time_taken
                    shortest_time_gen_solver = min(shortest_time_gen_solver, time_taken)
                    longest_time_gen_solver = max(longest_time_gen_solver, time_taken)

                    # Calculate number of solutions
                    num_solutions = sum(row.count('#') for row in m.tostring(solutions=True).split('\n'))
                    smallest_len_solutions = min(smallest_len_solutions, num_solutions)
                    largest_len_solutions = max(largest_len_solutions, num_solutions)
                    total_len_solutions += num_solutions
                    total_len_solutions_gen_solver += num_solutions
                    smallest_len_solutions_gen_solver = min(smallest_len_solutions_gen_solver, num_solutions)
                    largest_len_solutions_gen_solver = max(largest_len_solutions_gen_solver, num_solutions)

                # Calculate average statistics for current generator-solver combination
                average_time_gen_solver = total_time_gen_solver / loop if loop > 0 else 0
                average_len_solutions_gen_solver = total_len_solutions_gen_solver / loop if loop > 0 else 0

                # Write statistics to file
                f.write(f"Total time taken for {generator.__name__}-{solver.__name__} mazes: {total_time_gen_solver:.2f} seconds\n")
                f.write(f"Average time per {generator.__name__}-{solver.__name__} maze: {average_time_gen_solver:.2f} seconds\n")
                f.write(f"Shortest time taken for {generator.__name__}-{solver.__name__} mazes: {shortest_time_gen_solver:.2f} seconds\n")
                f.write(f"Longest time taken for {generator.__name__}-{solver.__name__} mazes: {longest_time_gen_solver:.2f} seconds\n")
                f.write(f"Total length of solutions for {generator.__name__}-{solver.__name__} mazes: {total_len_solutions_gen_solver}\n")
                f.write(f"Average length of solutions per {generator.__name__}-{solver.__name__} maze: {average_len_solutions_gen_solver:.2f}\n")
                f.write(f"Smallest length for solutions found for {generator.__name__}-{solver.__name__}: {smallest_len_solutions_gen_solver}\n")
                f.write(f"Largest number for solutions found for {generator.__name__}-{solver.__name__}: {largest_len_solutions_gen_solver}\n\n")

                # Check if time exceeded 2 hours, if so, exit
                if time_taken > 7200:
                    return
```
Figure 44 test code snippet.

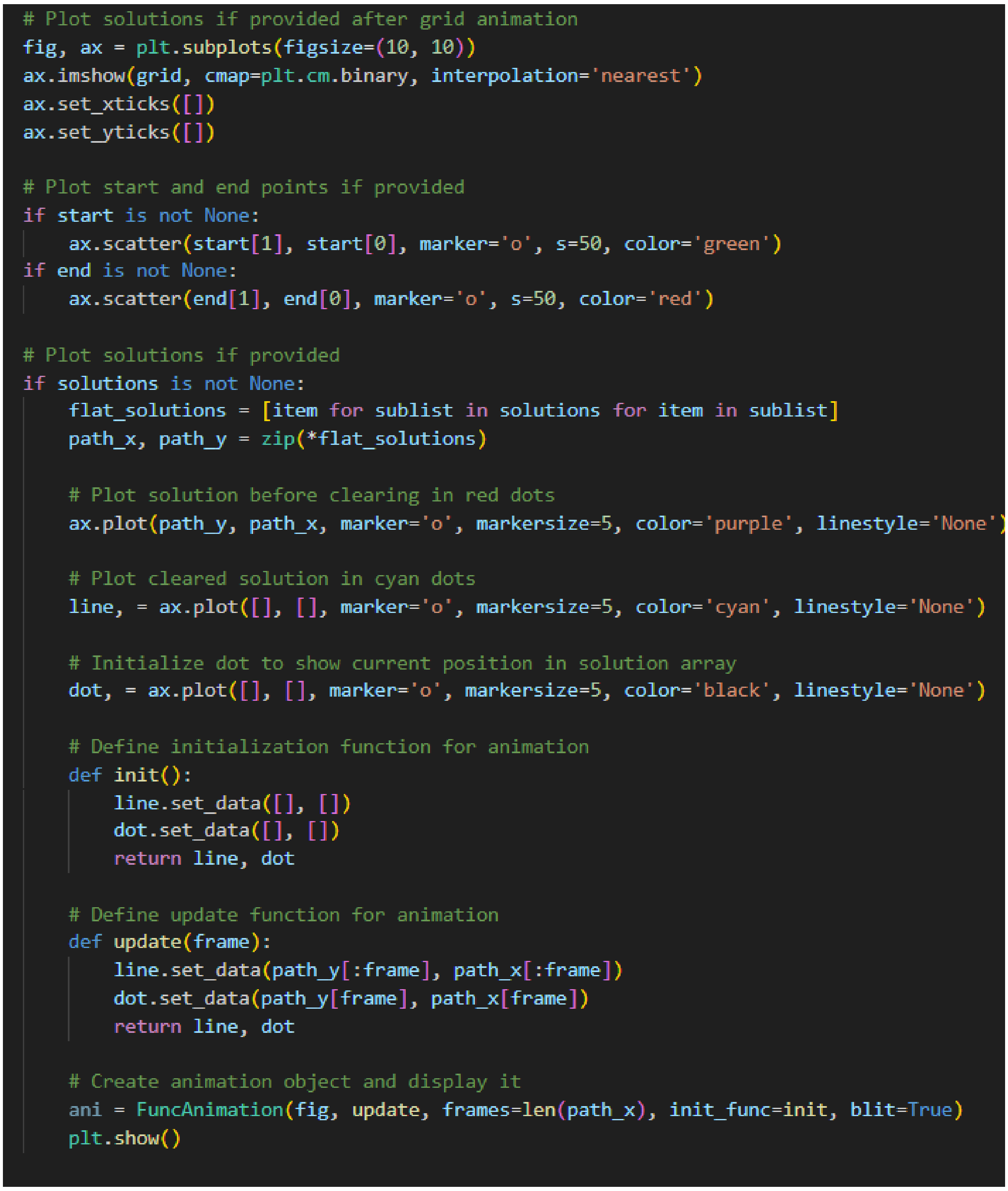
Although this is not shown this begins by asking for the how many mazes are wanted this is for the 3D aspect of the maze. It also asks whether the user wants to see the full solve path or just the solution this is where the clear that I passed through solve comes from this just allows for more options but for me in the testing I will really only be using the full solution. This code can be seen in the appendix. I have also made so that there is a timer for the solving algorithm this just takes the start time and takes it away from the end time to get the final time this again is printed along with the string mazes. To make the mazes we inherit the maze class and then use the functions in maze to generate and solve the maze to do this we use the generator attribute and solver attribute and then give them an instance of a generation algorithm and solving algorithm, respectively. The dimensions have to be set manually by me, but I have made it so that once I have picked the dimensions it will run these dimensions against every combination of algorithms possible. As you can see, I have made it so that other metrics are for all the generators and the solver for things like average time and average maze length for example these are then all saved to an external file so that later for the evaluation they are there saved for me to get easy access to I have also made it like this so that I can perform tests that could take a long time without needing to watch the tests happen. This has changed from the original design as initially I just planned on taking initial measurements and then not saving them to a file at all but now, I am taking more measurements although they are just time and space like originally planned it now allows me to better evaluate the algorithms. Also, I originally planned to manually change the algorithms I use and then take the measurements now I don't have to do that. I am also predicting that in the testing that some algorithms might just take too long to test especially random mouse fully this will be a problem when we get to bigger mazes that is why I have put in a timer that will check if it goes over 2 hours and if it does it will close the program.

Menu



```python
def visualize(grid, start=None, end=None, solutions=None):
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.set_xticks([])
    ax.set_yticks([])

    # Initialize empty list to store walls (circles representing walls)
    walls = []


    # Animation function to update plot
    def animate(frame):
        i, j = divmod(frame, grid.shape[1])
        if grid[i, j] == 1:
            wall = ax.plot(j, i, marker='s', markersize=10, color='black')[0]
            walls.append(wall)
        return walls

    # Total frames for animation (equal to number of elements in grid)
    total_frames = grid.size

    # Create animation object and display it
    ani = FuncAnimation(fig, animate, frames=total_frames, interval=1, blit=True)

    # Show the grid animation
    plt.show()
```

Figure 45 menu code snippet 1.



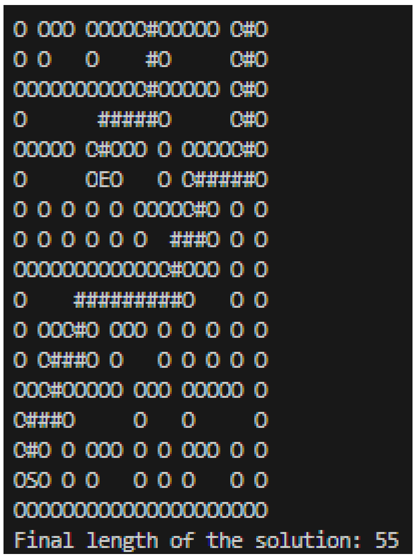
```python
# Plot solutions if provided after grid animation
fig, ax = plt.subplots(figsize=(10, 10))
ax.imshow(grid, cmap=plt.cm.binary, interpolation='nearest')
ax.set_xticks([])
ax.set_yticks([])

# Plot start and end points if provided
if start is not None:
    ax.scatter(start[1], start[0], marker='o', s=50, color='green')
if end is not None:
    ax.scatter(end[1], end[0], marker='o', s=50, color='red')

# Plot solutions if provided
if solutions is not None:
    flat_solutions = [item for sublist in solutions for item in sublist]
    path_x, path_y = zip(*flat_solutions)

    # Plot solution before clearing in red dots
    ax.plot(path_y, path_x, marker='o', markersize=5, color='purple', linestyle='None')

    # Plot cleared solution in cyan dots
    line, = ax.plot([], [], marker='o', markersize=5, color='cyan', linestyle='None')

    # Initialize dot to show current position in solution array
    dot, = ax.plot([], [], marker='o', markersize=5, color='black', linestyle='None')

    # Define initialization function for animation
    def init():
        line.set_data([], [])
        dot.set_data([], [])
        return line, dot

    # Define update function for animation
    def update(frame):
        line.set_data(path_y[:frame], path_x[:frame])
        dot.set_data(path_y[frame], path_x[frame])
        return line, dot

    # Create animation object and display it
    ani = FuncAnimation(fig, update, frames=len(path_x), init_func=init, blit=True)
    plt.show()
```

Figure 46 menu code snippet 2.

In menu you will find a more user focused approach in that there is a lot more to be selected with input unlike in test were I have to manually change everything in this the input are for dimensions and it also asks to pick a generator and solver through inputting a number between 1-4 for each and it will also ask ,like in test, for the depth of the maze as well as whether the full solve path should be shown or not. This can be seen in the appendix. The main part of this is the matplotlib section in the visualize function. The first part is what displays the maze generation the way this works is by using matplotlib's animate function it will plot a black square every frame for where the walls correspond to in the generated maze this means that it will work row by row and the empty spaces are just left blank. Once a maze has been generated then it moves onto the solution being shown. It starts showing the maze as a still image but then it will use the scatter function to plot the start and finish as green and red dots, respectively. Then it will plot the solution paths as purple dots the amount will depend on whether the user has chosen to see all of the path taken by the algorithm or not. Then once that has all been plotted you will see the solution be plotted out this is done by a black dot that moves through the solution path turning the purple dots to cyan to show the places it has already gone, and it also allows me to see were to algorithm is at in the solution in any moment. This has changed a bit from my original design as what I wanted it to do was to go from the generation straight to the solving, but this isn't how I have done it in that the maze will be generated but it is not until you close the window for the generation does the solving actually show and begin. The reason I made this change is because if the solving process showed up instantly it would mean that there was not time to see what the generation algorithms had done.

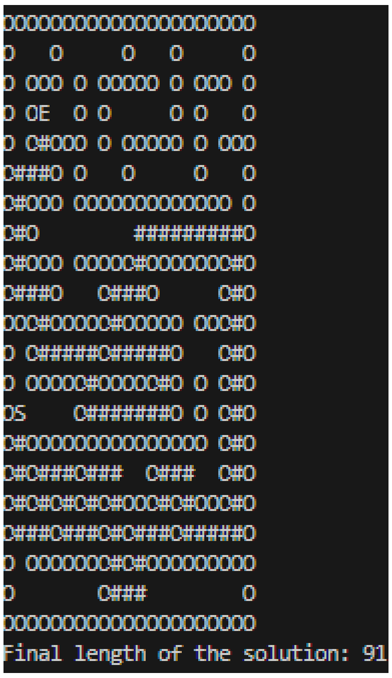
Outputs – test

All solved with Trémaux.



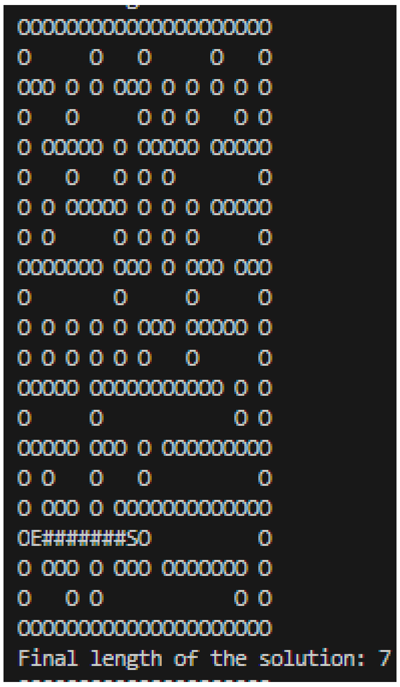
```
O OOO OOOOC#OOOOO C#O
O O   O    #O     C#O
OOOOOOOOOOC#OOOOO C#O
O      #####O     C#O
OOOOO C#OOO O OOOOC#O
O     OEO   O C#####O
O O O O O OOOOC#O O O
O O O O O O  ###O O O
OOOOOOOOOOOOC#OOO O O
O    #########O   O O
O OOC#O OOO O O O O O
O C###O O   O O O O O
OOC#OOOOO OOO OOOOO O
C###O     O   O     O
C#O O OOO O O OOO O O
OSO O O   O O O   O O
OOOOOOOOOOOOOOOOOOOOO
Final length of the solution: 55
```

Figure 47 String maze output example 10x10 binary tree.



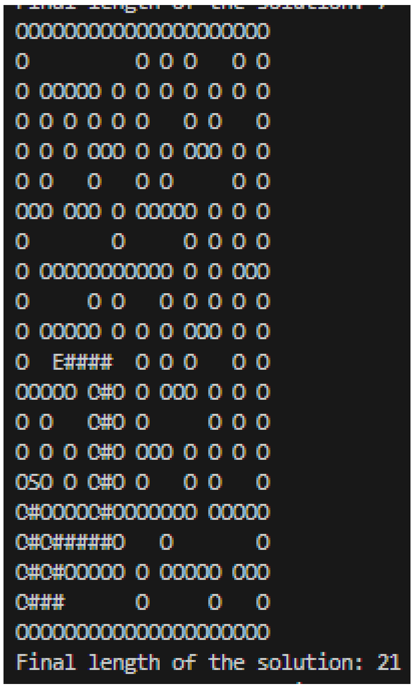
```
OOOOOOOOOOOOOOOOOOOOO
O   O     O   O     O
O OOO O OOOOO O OOO O
O OE  O O     O O   O
O C#OOO O OOOOO O OOO
C###O O   O     O   O
C#OOO OOOOOOOOOOOOO O
C#O        #########O
C#OOO OOOOC#OOOOOOC#O
C###O   C###O     C#O
OOC#OOOOC#OOOOO OOC#O
O C#####C#####O   C#O
O OOOOC#OOOOC#O O C#O
OS    C#######O O C#O
C#OOOOOOOOOOOOOOO C#O
C#C###C###  C###  C#O
C#C#C#C#OOOC#C#OOOC#O
C###C###C#C###C#####O
O OOOOOOC#C#OOOOOOOOO
O       C###        O
OOOOOOOOOOOOOOOOOOOOO
Final length of the solution: 91
```

Figure 48 String maze output example 10x10 backtracking generator.



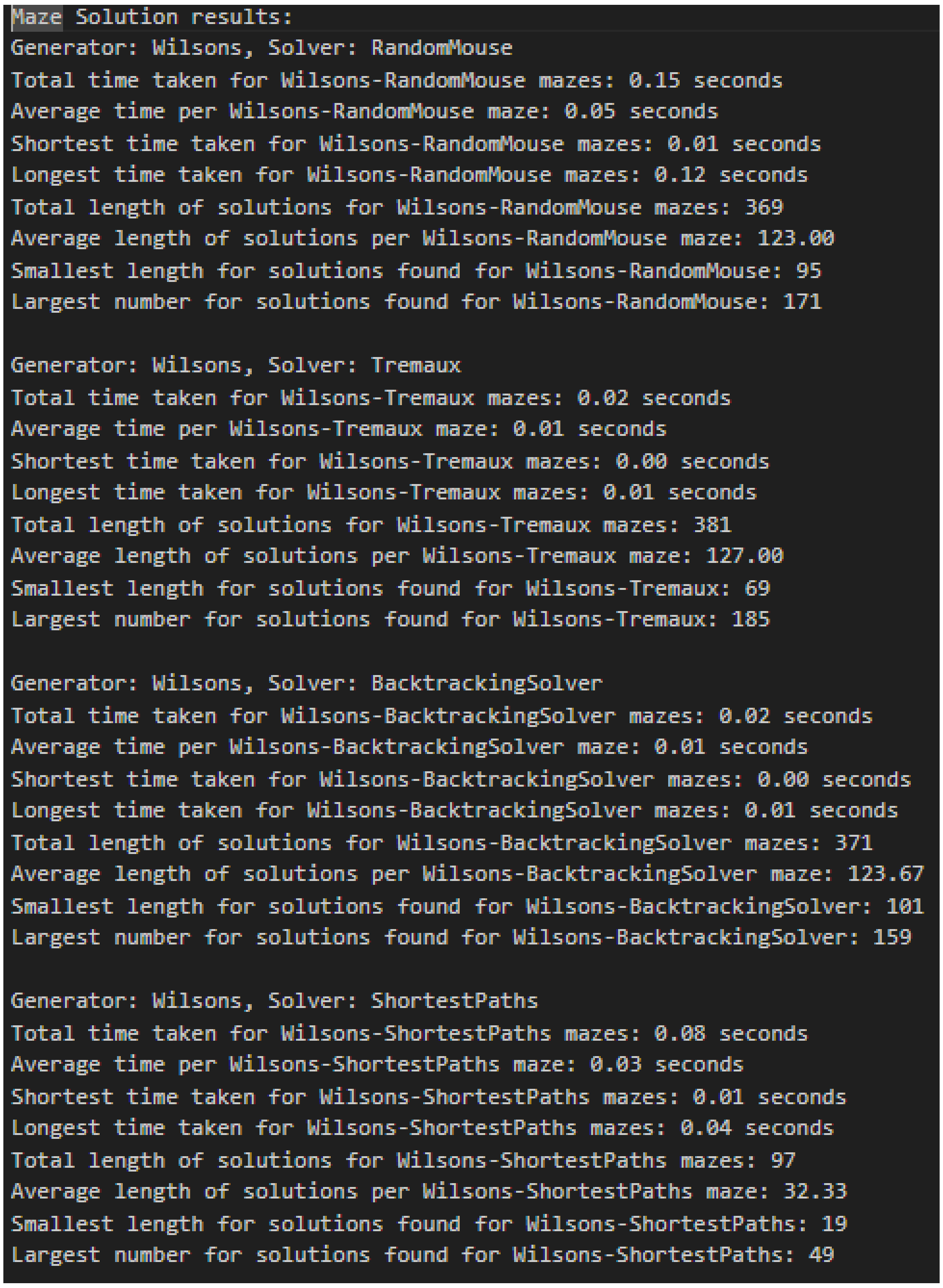
```
OOOOOOOOOOOOOOOOOOOOO
O     O   O     O   O
OOO O O OOO O O O O O
O   O     O O O   O O
O OOOOO O OOOOO OOOOO
O   O   O O O       O
O O OOOOO O O O OOOOO
O O     O O O O     O
OOOOOOO OOO O OOO OOO
O       O     O     O
O O O O O OOO OOOOO O
O O O O O O   O     O
OOOOO OOOOOOOOOOO O O
O     O           O O
OOOOO OOO O OOOOOOOOO
O O   O   O         O
O OOO O OOOOOOOOOOOOO
OE#######SO         O
O OOO O OOO OOOOOOO O
O   O O           O O
OOOOOOOOOOOOOOOOOOOOO
Final length of the solution: 7
```

Figure 49 String maze output example 10x10 Wilsons.



```
OOOOOOOOOOOOOOOOOOOOO
O         O O O   O O
O OOOOO O O O O O O O
O O O O O O   O O   O
O O O OOO O O OOO O O
O O   O   O O     O O
OOO OOO O OOOOO O O O
O       O     O O O O
O OOOOOOOOOOO O O OOO
O     O O   O O O O O
O OOOOO O O O OOO O O
O  E####  O O O   O O
OOOOO C#O O OOO O O O
O O   C#O O     O O O
O O O C#O OOO O O O O
OSO O C#O O   O O   O
C#OOOOC#OOOOOOO OOOOO
C#C#####O   O       O
C#C#OOOOO O OOOOO OOO
C###      O     O   O
OOOOOOOOOOOOOOOOOOOOO
Final length of the solution: 21
```

Figure 50 String maze output example 10x10 Ellers.



```
Maze Solution results:
Generator: Wilsons, Solver: RandomMouse
Total time taken for Wilsons-RandomMouse mazes: 0.15 seconds
Average time per Wilsons-RandomMouse maze: 0.05 seconds
Shortest time taken for Wilsons-RandomMouse mazes: 0.01 seconds
Longest time taken for Wilsons-RandomMouse mazes: 0.12 seconds
Total length of solutions for Wilsons-RandomMouse mazes: 369
Average length of solutions per Wilsons-RandomMouse maze: 123.00
Smallest length for solutions found for Wilsons-RandomMouse: 95
Largest number for solutions found for Wilsons-RandomMouse: 171

Generator: Wilsons, Solver: Tremaux
Total time taken for Wilsons-Tremaux mazes: 0.02 seconds
Average time per Wilsons-Tremaux maze: 0.01 seconds
Shortest time taken for Wilsons-Tremaux mazes: 0.00 seconds
Longest time taken for Wilsons-Tremaux mazes: 0.01 seconds
Total length of solutions for Wilsons-Tremaux mazes: 381
Average length of solutions per Wilsons-Tremaux maze: 127.00
Smallest length for solutions found for Wilsons-Tremaux: 69
Largest number for solutions found for Wilsons-Tremaux: 185

Generator: Wilsons, Solver: BacktrackingSolver
Total time taken for Wilsons-BacktrackingSolver mazes: 0.02 seconds
Average time per Wilsons-BacktrackingSolver maze: 0.01 seconds
Shortest time taken for Wilsons-BacktrackingSolver mazes: 0.00 seconds
Longest time taken for Wilsons-BacktrackingSolver mazes: 0.01 seconds
Total length of solutions for Wilsons-BacktrackingSolver mazes: 371
Average length of solutions per Wilsons-BacktrackingSolver maze: 123.67
Smallest length for solutions found for Wilsons-BacktrackingSolver: 101
Largest number for solutions found for Wilsons-BacktrackingSolver: 159

Generator: Wilsons, Solver: ShortestPaths
Total time taken for Wilsons-ShortestPaths mazes: 0.08 seconds
Average time per Wilsons-ShortestPaths maze: 0.03 seconds
Shortest time taken for Wilsons-ShortestPaths mazes: 0.01 seconds
Longest time taken for Wilsons-ShortestPaths mazes: 0.04 seconds
Total length of solutions for Wilsons-ShortestPaths mazes: 97
Average length of solutions per Wilsons-ShortestPaths maze: 32.33
Smallest length for solutions found for Wilsons-ShortestPaths: 19
Largest number for solutions found for Wilsons-ShortestPaths: 49
```

Figure 51 test output into example file.

In this you can see that once I have chosen the dimensions, and I have given the depth of the mazes then it will result in output like you can see above. As you can see it follows the chars I set out in maze, and it shows the solution path and the start and end. You can see the contents of the file that I have written to after doing a test run with three mazes as you can see it gives accurate and easy to read statistics on that test as it tells you which algorithms have been used together this is after a 25x25x50 test dimensions.

Outputs - menu

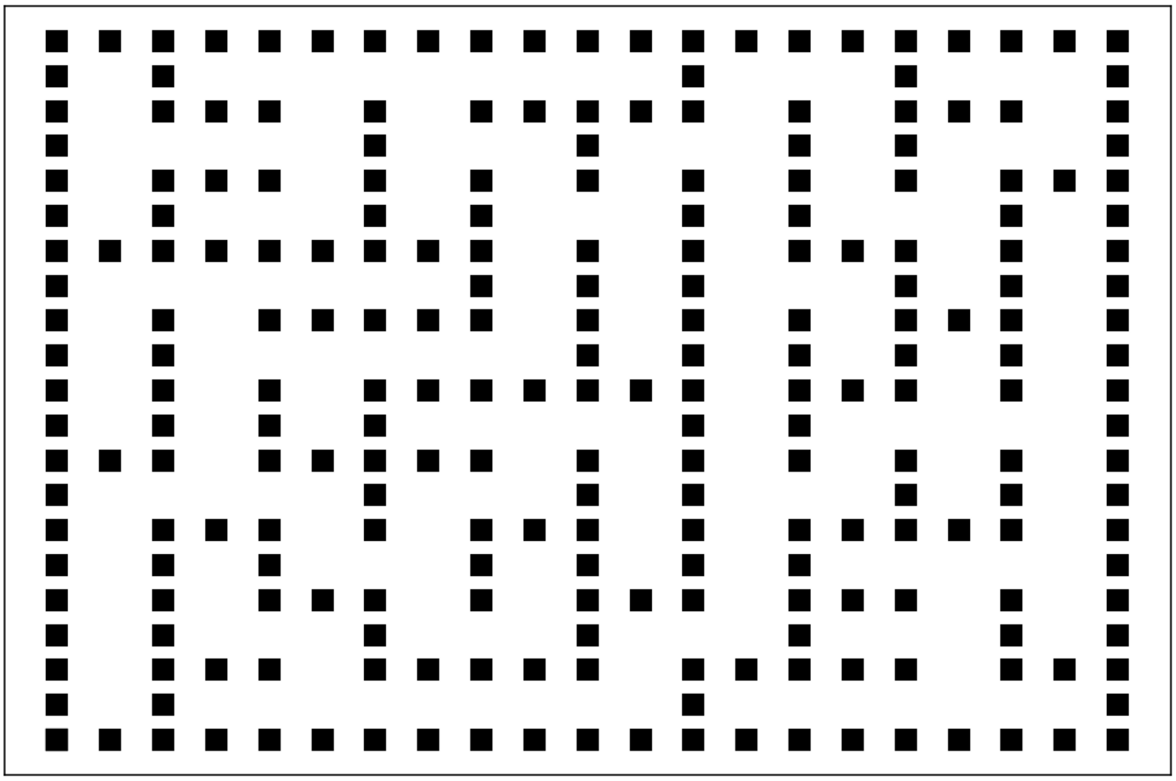


Figure 52 menu maze generation output.

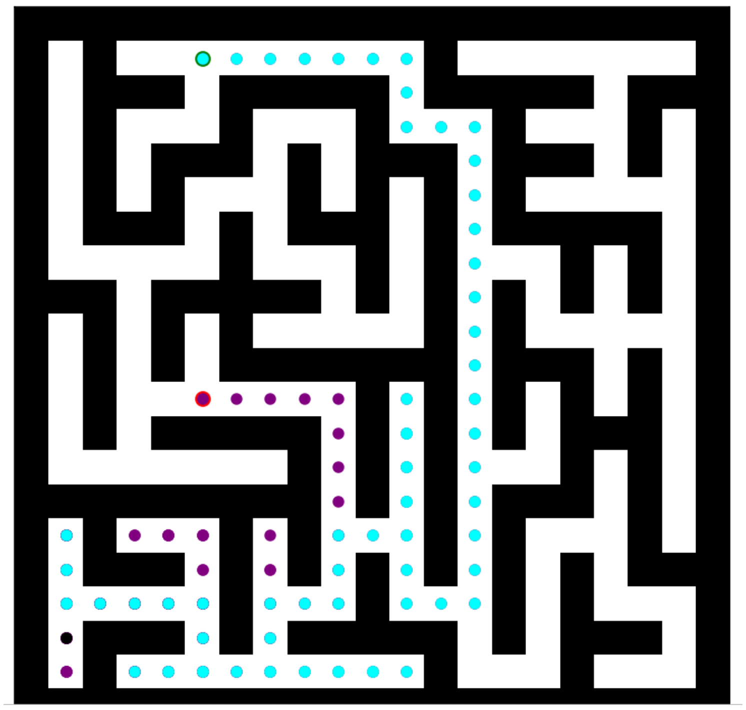


Figure 53 maze solving mid process in matplotlib.

This is the outputs from the menu the images above are mid animation the first one shows the output of the maze generation where it is like the test in that it is just segmented squares that make up the wall of the maze these squares are plotted one by one. Below that is a maze being solved as you can see just like I described in the menu the path travelled is marked by cyan dots and the current location is a black dot and the path that is yet to be travelled is in purple and the start and end are the dots that have green and red on them.

Testing and Evaluation of Artefact

Testing methodology

Now that the implementation is finished, I can now generate results through testing my artefact. For the testing I will be using the test file this was specifically created for this chapter. With this file I can select all the algorithms and test them together and save the results to a file. For the testing I need to get a decent number of results so that I can say my testing has been extensive and so that the accuracy of my results for things like averages is increased. The way I am going to test the artefact is as follows. I am going to do tests for every generation algorithm with every solving algorithm and vice versa. For these algorithms I will do three tests for each. In these tests I will have the depth set at a consistent fifty throughout the only thing that will change is the height and width of the mazes for this I plan on doing 5 stages the stages will have a consistent increase in height and width. The first stage will be 25x25 the next stage will be 50x50 then 100x100 then 250x250 and finally 500x500. From these tests my main goal is to get quantitative data this will be for space in how much spaces it takes for the solving algorithms to complete the maze and for time which is how long it takes for them to solve the mazes this will all be displayed in separate tables with the combinations going down the side with the dimension and test number going along the top. Whilst the quantitative data is my main goal for this, I will also be able to acquire some qualitative data this includes things like how the maze look like if there are obvious biases some more obvious passages. I can also take a look at how well the algorithm did like if it took the obvious path or if it went down the wrong path which led to worse metrics. I can also see how solvable some mazes are for humans if there are again obvious paths for humans that the algorithms took a longer time to get. Also, as mazes are now being used increasingly in the real world, I can look at the generated mazes and determine whether they would be useful to be used as examples of real-world subjects like rooms or streets. But my focus is on the quantitative data that will be produced as qualitative data can be very subjective in some circumstances.

Results

Total time taken results.

Total time taken in seconds	25x25 test 1	25x25 test 2	25x25 test 3	50x50 test 1	50x50 test 2	50x50 test 3	100x100 test 1	100x100 test 2	100x100 test 3	250x250 test 1	250x250 test 2	250x250 test 3	500x500 test 1	500x500 test 2	500x500 test 3
Binary Tree - Random mouse	33.12	34.45	33.42	329.16	315.67	330.14	2345.93	2290.78	2459.49	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00
Binary Tree - Trémaux	8.61	6.01	5.97	17.08	15.55	15.33	158.73	139.42	152.20	472.62	476.10	481.36	2550.55	2549.22	2571.94
Binary Tree – Backtracking solver	6.03	7.99	7.83	60.18	57.34	65.11	571.43	572.75	568.25	1182.18	1223.15	1185.39	7200.00	7200.00	7200.00
Binary Tree - Shortest path	18.29	18.88	17.67	148.03	140.53	146.98	1381.65	1361.21	1345.84	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00
Backtracking generator – Random mouse	137.21	132.67	144.23	1507.01	1480.56	1519.21	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00
Backtracking generator – Trémaux	1.39	1.42	1.48	6.20	6.50	6.35	25.24	28.41	28.96	128.40	115.52	125.99	635.01	621.44	631.55
Backtracking generator – Backtracking solver	5.10	4.87	4.99	58.39	63.10	61.21	552.47	576.19	589.29	4987.41	4942.11	5078.39	7200.00	7200.00	7200.00
Backtracking generator – Shortest path	54.85	52.84	60.01	1197.29	1201.75	1190.59	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00
Ellers – Random mouse	19.88	21.02	20.87	49.86	50.85	49.99	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00
Ellers – Trémaux	4.18	3.96	3.66	17.16	17.02	18.88	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00
Ellers – Backtracking solver	5.48	6.20	5.84	22.46	21.99	21.53	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00

Ellers – Shortest path	9.14	9.76	9.81	19.76	20.11	19.45	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00
Wilsons – Random mouse	45.92	46.78	52.11	485.07	479.81	486.15	3634.20	3629.40	3601.23	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00
Wilsons – Trémaux	8.72	8.48	8.13	30.56	32.05	31.61	128.14	125.17	128.34	505.41	508.00	510.19	2022.11	2013.45	2030.55
Wilsons – Backtracking solver	8.59	8.44	8.98	91.01	89.48	89.77	738.82	732.66	730.46	6,642.50	6629.52	6651.99	7200.00	7200.00	7200.00
Wilsons – Shortest path	24.89	25.33	24.55	333.05	328.76	325.51	3643.66	3640.99	3650.51	7200.00	7200.00	7200.00	7200.00	7200.00	7200.00

Figure 54 Total time taken results table.

Average time taken results.

Average time taken in seconds	25x25 test 1	25x25 test 2	25x25 test 3	50x50 test 1	50x50 test 2	50x50 test 3	100x100 test 1	100x100 test 2	100x100 test 3	250x250 test 1	250x250 test 2	250x250 test 2	500x500 test 1	500x500 test 2	500x500 test 3
Binary Tree - Random mouse	0.66	0.68	0.66	6.58	6.31	6.60	46.91	45.81	49.18	1440.00	1440.00	1800.00	0	0	0
Binary Tree - Trémaux	0.17	0.12	0.11	0.34	0.31	0.30	3.17	2.78	3.04	9.45	9.52	9.62	51.01	50.98	51.43
Binary Tree – Backtracking solver	0.12	0.16	0.16	1.20	1.15	1.30	11.43	11.46	11.37	23.64	24.46	23.71	180.00	171.42	184.61
Binary Tree - Shortest path	0.37	0.38	0.35	2.96	2.81	2.94	27.63	27.22	26.92	720.00	800.00	720.00	1440.00	2400.00	1200.00
Backtracking generator – Random mouse	2.74	2.65	2.88	30.14	29.61	30.38	7200.00	0	2400.00	0	0	0	0	0	0

Backtracking generator – Trémaux	0.03	0.03	0.03	0.12	0.13	0.13	0.50	0.57	0.58	2.57	2.31	2.52	12.70	12.43	12.63
Backtracking generator – Backtracking solver	0.10	0.10	0.10	1.17	1.26	1.22	11.05	11.52	11.79	99.75	98.84	101.57	194.59	175.60	180.00
Backtracking generator – Shortest path	1.10	1.06	1.20	23.95	24.03	23.81	1440.00	2400.00	3600.00	0	0	0	0	0	0
Ellers – Random mouse	0.40	0.42	0.42	1.00	1.02	1.00	0	0	0	0	0	0	0	0	0
Ellers – Trémaux	0.08	0.08	0.07	0.34	0.34	0.38	0	0	0	0	0	0	0	0	0
Ellers – Backtracking solver	0.11	0.12	0.12	0.45	0.44	0.43	0	0	0	0	0	0	0	0	0
Ellers – Shortest path	0.18	0.20	0.20	0.40	0.40	0.39	0	0	0	0	0	0	0	0	0
Wilsons – Random mouse	0.92	0.94	1.04	9.70	9.60	9.72	72.68	72.59	72.02	1800.00	7200.00	0	0	7200.00	0
Wilsons – Trémaux	0.17	0.17	0.16	0.61	0.64	0.63	2.56	2.50	2.57	10.11	10.16	10.20	40.44	40.27	40.61
Wilsons – Backtracking solver	0.17	0.17	0.18	1.82	1.79	1.80	14.78	14.65	14.61	132.85	132.59	133.04	288.00	240.00	276.92
Wilsons – Shortest path	0.50	0.51	0.49	6.66	6.58	6.51	72.87	72.82	73.01	900.00	900.00	1028.57	1800.00	1800.00	2400.00

Figure 55 Average time taken results table.

Shortest time taken results.

Shortest time taken in seconds	25x25 test 1	25x25 test 2	25x25 test 3	50x50 test 1	50x50 test 2	50x50 test 3	100x100 test 1	100x100 test 2	100x100 test 3	250x250 test 1	250x250 test 2	250x250 test 2	500x500 test 1	500x500 test 2	500x500 test 3

Binary Tree - Random mouse	0.00	0.48	1.22	0.00	12.44	2.01	0.27	5.10	2.12	10.55	50.39	4.50	0	0	0
Binary Tree - Trémaux	0.00	0.01	0.00	0.00	0.04	0.01	0.02	0.02	0.05	0.20	3.45	1.32	34.77	2.52	20.31
Binary Tree – Backtracking solver	0.00	0.04	0.78	0.23	0.10	0.22	2.19	1.99	0.49	12.01	3.51	0.41	20.50	28.32	5.69
Binary Tree - Shortest path	0.02	0.04	0.09	0.15	0.63	0.22	2.11	3.95	1.00	29.04	50.28	32.09	104.78	90.55	207.33
Backtracking generator – Random mouse	0.01	0.00	0.23	0.11	0.03	0.00	7200.00	0	1051.78	0	0	0	0	0	0
Backtracking generator – Trémaux	0.00	0.00	0.03	0.00	0.01	0.02	0.05	0.02	0.09	0.45	0.39	0.61	0.24	0.69	0.55
Backtracking generator – Backtracking solver	0.00	0.01	0.00	0.00	0.10	0.00	0.02	0.09	0.01	10.53	15.76	12.22	40.98	19.45	31.93
Backtracking generator – Shortest path	0.01	0.03	0.02	0.01	0.05	0.01	500.50	181.51	1381.20	0	0	0	0	0	0
Ellers – Random mouse	0.00	0.02	0.08	0.07	0.02	0.10	0	0	0	0	0	0	0	0	0
Ellers – Trémaux	0.00	0.00	0.00	0.02	0.04	0.01	0	0	0	0	0	0	0	0	0
Ellers – Backtracking solver	0.00	0.01	0.01	0.01	0.00	0.05	0	0	0	0	0	0	0	0	0
Ellers – Shortest path	0.00	0.02	0.00	0.03	0.00	0.02	0	0	0	0	0	0	0	0	0
Wilsons – Random mouse	0.00	0.02	0.00	0.01	0.05	0.04	0.02	0.03	0.04	371.40	7200.00	0	0	7200.00	0
Wilsons – Trémaux	0.00	0.00	0.01	0.03	0.00	0.02	0.05	0.04	0.05	5.21	4.89	3.77	9.81	11.67	10.11
Wilsons – Backtracking solver	0.00	0.00	0.00	0.01	0.02	0.01	0.04	0.07	0.01	8.99	6.83	7.11	69.45	93.07	125.01

Wilsons – Shortest path	0.03	0.02	0.00	0.81	0.34	0.32	26.74	27.89	40.99	400.78	387.66	869.12	900.56	1025.61	1101.55

Figure 56 Shortest time taken results table.

Longest time taken results.

Longest time taken in seconds	25x25 test 1	25x25 test 2	25x25 test 3	50x50 test 1	50x50 test 2	50x50 test 3	100x100 test 1	100x100 test 2	100x100 test 3	250x250 test 1	250x250 test 2	250x250 test 2	500x500 test 1	500x500 test 2	500x500 test 3
Binary Tree - Random mouse	6.31	5.99	6.82	27.48	22.12	20.99	183.59	167.33	201.57	2342.05	1999.10	2198.42	0	0	0
Binary Tree - Trémaux	1.21	1.89	1.23	1.79	1.60	1.88	16.94	12.38	17.80	29.75	32.41	35.65	37.89	50.23	52.56
Binary Tree – Backtracking solver	0.50	0.90	1.45	6.61	5.42	4.99	78.14	67.99	73.78	110.51	107.32	115.22	181.73	177.61	190.01
Binary Tree - Shortest path	0.70	1.02	0.69	5.57	6.12	5.58	55.93	61.78	62.49	7200.00	969.82	7200.00	1889.43	3602.90	1999.31
Backtracking generator – Random mouse	13.43	9.88	15.22	199.24	201.56	225.81	7200.00	0	4201.45	0	0	0	0	0	0
Backtracking generator – Trémaux	0.07	0.10	0.05	0.47	0.52	0.44	1.58	2.01	1.59	4.32	5.77	5.39	8.91	9.99	11.59
Backtracking generator – Backtracking solver	0.54	0.66	0.27	3.56	6.12	4.88	71.33	48.69	51.22	163.95	152.37	159.11	269.24	244.81	219.10
Backtracking generator – Shortest path	3.40	5.60	3.99	52.62	59.01	67.82	2899.05	4700.39	5818.8	0	0	0	0	0	0
Ellers – Random mouse	2.83	4.50	4.42	5.55	3.88	4.78	0	0	0	0	0	0	0	0	0
Ellers – Trémaux	0.15	0.60	0.22	2.79	3.33	2.98	0	0	0	0	0	0	0	0	0

Ellers – Backtracking solver	0.35	0.20	0.33	1.15	2.22	2.98	0	0	0	0	0	0	0	0	0
Ellers – Shortest path	3.12	3.57	2.80	5.21	5.91	5.23	0	0	0	0	0	0	0	0	0
Wilsons – Random mouse	9.96	8.47	7.98	53.75	50.12	48.90	377.28	355.89	376.92	3025.23	7200.00	0	0	7200.00	0
Wilsons – Trémaux	1.16	2.01	1.78	4.66	5.34	5.71	11.97	13.27	15.99	28.97	25.61	22.39	69.61	66.77	72.21
Wilsons – Backtracking solver	0.61	1.50	1.02	7.68	5.98	8.20	68.27	60.11	59.11	210.29	2031.22	198.48	391.07	379.28	376.99
Wilsons – Shortest path	1.39	1.76	1.22	16.50	17.88	16.76	187.50	182.30	169.98	1503.12	1689.48	1891.89	2951.88	2897.05	3607.81

Figure 57 longest time taken results table.

Total length of solutions results.

Total length of solutions	25x25 test 1	25x25 test 2	25x25 test 3	50x50 test 1	50x50 test 2	50x50 test 3	100x100 test 1	100x100 test 2	100x100 test 3	250x250 test 1	250x250 test 2	250x250 test 3	500x500 test 1	500x500 test 2	500x500 test 3
Binary Tree - Random mouse	28778	30456	30619	132460	135781	136801	490438	533379	521915	380386	400,525	386861	0	0	0
Binary Tree - Trémaux	31002	28111	27891	125546	136801	122755	477982	475129	480012	2549450	2751093	2456910	19849850	19857500	18482141
Binary Tree – Backtracking solver	30192	29740	30445	128008	129856	131009	548616	529333	533696	3784450	3610998	3719301	13300100	12097361	11604219
Binary Tree - Shortest path	29990	29517	29126	122148	198890	128978	468912	472029	469845	509235	506591	501224	1807559	1782890	1879926
Backtracking generator – Random mouse	35436	32497	33346	132300	133982	132003	10987	0	21283	0	0	0	0	0	0

Backtracking generator – Trémaux	32438	33349	30119	124204	129349	128992	514618	509135	510991	4147040	4095162	4115928	16588166	17012361	16857330
Backtracking generator – Backtracking solver	31182	32781	31827	136992	133826	134119	530038	529774	530619	3995504	4059980	4026491	11826691	13316734	12884771
Backtracking generator – Shortest path	38986	37445	36009	187709	179999	185521	24501	18665	12983	0	0	0	0	0	0
Ellers – Random mouse	36104	36005	38102	181494	176440	187991	0	0	0	0	0	0	0	0	0
Ellers – Trémaux	30949	29992	30874	145545	149120	144660	0	0	0	0	0	0	0	0	0
Ellers – Backtracking solver	34490	35555	35099	171167	169037	168111	0	0	0	0	0	0	0	0	0
Ellers – Shortest path	32917	31004	30118	155098	153331	153973	0	0	0	0	0	0	0	0	0
Wilsons – Random mouse	31028	32091	31199	127114	127556	129001	496404	488110	495657	310252	62050	0	0	248202	0
Wilsons – Trémaux	29604	30222	30628	133052	132909	129998	491012	496918	492550	5588950	5697851	5890819	20355800	21051801	20071917
Wilsons – Backtracking solver	32862	33092	32119	134298	133872	133092	526602	522009	523167	5991262	6014592	5959921	11982524	14435020	12396633
Wilsons – Shortest path	30829	31092	31174	129838	130982	130223	493827	493289	498221	523826	529773	499129	1087654	1075323	1037407

Figure 58 Total length of solutions results table.

Average length of solutions results.

Average length of solutions	25x25 test 1	25x25 test 2	25x25 test 3	50x50 test 1	50x50 test 2	50x50 test 3	100x100 test 1	100x100 test 2	100x100 test 3	250x250 test 1	250x250 test 2	250x250 test 2	500x500 test 1	500x500 test 2	500x500 test 3

Binary Tree - Random mouse	575.56	609.12	612.38	2649.20	2715.62	2736.02	9808.76	10667.58	10438.30	76077.20	80105.00	96715.25	0	0	0
Binary Tree - Trémaux	620.04	562.22	557.82	2510.92	2736.02	2455.10	9559.64	9502.58	9600.24	50989.00	55021.86	49138.20	396997.00	397150.00	369642.82
Binary Tree – Backtracking solver	603.84	594.80	608.90	2560.16	2597.12	2620.18	10972.32	10586.66	10673.92	75689.00	72219.96	74386.02	332,502.50	288,032.40	297,544.07
Binary Tree - Shortest path	599.80	590.34	582.52	2442.96	3,977.80	2579.56	9378.24	9440.58	9396.90	10184.70	10131.82	10024.48	36151.18	35657.80	37598.52
Backtracking generator – Random mouse	708.72	649.94	666.92	2646.00	2679.64	2640.06	10987.00	0	7094.33	0	0	0	0	0	0
Backtracking generator – Trémaux	648.76	666.98	602.38	2484.08	2586.98	2579.84	10292.36	10182.70	10219.82	82940.80	81903.24	82318.56	331763.32	340247.22	337146.60
Backtracking generator – Backtracking solver	623.64	655.62	636.54	2739.84	2676.52	2682.38	10600.76	10595.48	10612.38	79910.08	81199.60	80529.82	319,640.29	324,798.39	322,119.27
Backtracking generator – Shortest path	779.72	748.90	720.18	3754.18	3600.18	3710.42	4,900.20	6,221.66	6,491.50	0	0	0	0	0	0
Ellers – Random mouse	722.08	720.10	762.04	3629.88	3528.80	3759.82	0	0	0	0	0	0	0	0	0
Ellers – Trémaux	618.98	599.84	617.48	2910.90	2982.40	2893.20	0	0	0	0	0	0	0	0	0
Ellers – Backtracking solver	689.80	711.10	701.98	3423.34	3380.74	3362.22	0	0	0	0	0	0	0	0	0
Ellers – Shortest path	658.34	620.08	602.36	3101.96	3066.62	3079.46	0	0	0	0	0	0	0	0	0
Wilsons – Random mouse	620.56	641.82	623.98	2542.28	2551.12	2580.02	9928.08	9762.20	9913.14	77563.00	62050.00	0	0	248202.00	0

Wilsons – Trémaux	592.08	604.44	612.56	2661.04	2658.18	2599.96	9820.24	9938.36	9851.00	111779.00	113957.02	117816.38	407116.00	421036.02	401438.34
Wilsons – Backtracking solver	657.24	661.84	642.38	2685.96	2677.44	2661.84	10532.04	10440.18	10463.34	119825.24	120291.84	119198.42	479,300.96	481,167.33	476,793.57
Wilsons – Shortest path	616.58	621.84	623.48	2596.76	2619.64	2604.46	9876.54	9865.78	9964.42	65478.25	66221.63	71289.85	271913.50	268830.75	345802.33

Figure 59 Average length of solution results table.

Smallest length of solutions results.

Smallest length of solutions	25x25 test 1	25x25 test 2	25x25 test 3	50x50 test 1	50x50 test 2	50x50 test 3	100x100 test 1	100x100 test 2	100x100 test 3	250x250 test 1	250x250 test 2	250x250 test 2	500x500 test 1	500x500 test 2	500x500 test 3
Binary Tree - Random mouse	33	55	12	5	88	24	673	445	399	1394	2777	3223	0	0	0
Binary Tree - Trémaux	37	9	11	47	44	19	549	298	511	1355	5922	498	12953	22103	8990
Binary Tree – Backtracking solver	51	38	66	5	99	24	198	226	498	3091	927	13995	19022	25999	555
Binary Tree - Shortest path	73	12	25	44	101	73	442	192	333	1401	1145	835	9026	15677	12234
Backtracking generator –	3	20	88	41	83	22	10987	0	766	0	0	0	0	0	0

Random mouse															
Backtracking generator – Trémaux	13	8	49	59	56	91	99	398	514	5011	2983	1253	19002	12443	3339
Backtracking generator – Backtracking solver	9	19	26	37	81	40	834	335	55	8192	7204	2330	30911	26119	19922
Backtracking generator – Shortest path	10	25	26	47	64	22	725	100	165	0	0	0	0	0	0
Ellers – Random mouse	17	49	98	3	60	82	0	0	0	0	0	0	0	0	0
Ellers – Trémaux	23	12	53	72	57	40	0	0	0	0	0	0	0	0	0
Ellers – Backtracking solver	34	61	11	61	12	101	0	0	0	0	0	0	0	0	0
Ellers – Shortest path	22	36	44	71	49	82	0	0	0	0	0	0	0	0	0
Wilsons – Random mouse	13	22	78	34	91	60	402	48	783	10259	62050.00	0	0	248202.00	0
Wilsons – Trémaux	29	31	45	125	88	103	366	837	611	2911	828	1888	9923	14567	12829
Wilsons – Backtracking solver	29	55	86	101	126	111	220	240	699	8734	19022	5502	26782	19835	19202
Wilsons – Shortest path	9	25	66	89	72	102	188	207	221	5938	3920	6793	28943	23244	32838

Figure 60 Smallest length of solutions results table.

Longest length of solutions results.

Longest length of solutions	25x25 test 1	25x25 test 2	25x25 test 3	50x50 test 1	50x50 test 2	50x50 test 3	100x100 test 1	100x100 test 2	100x100 test 3	250x250 test 1	250x250 test 2	250x250 test 2	500x500 test 1	500x500 test 2	500x500 test 3

Binary Tree - Random mouse	1245	1382	1209	4985	5012	4898	18616	20268	19832	144147	152099	183758	0	0	0
Binary Tree - Trémaux	1178	1068	1059	4770	5198	4664	18163	18095	18240	96878	104541	93362	754294	754585	701321
Binary Tree – Backtracking solver	1146	1130	1157	4864	4934	4978	20847	20114	20319	143709	137218	141434	631754	547260	565333
Binary Tree - Shortest path	1138	1120	1107	4641	7,557	4900	17819	17936	17854	19350	19250	19045	68687	67749	71537
Backtracking generator – Random mouse	1239	1203	1213	4683	4769	4778	10987	0	12487	0	0	0	0	0	0
Backtracking generator – Trémaux	1214	1235	1138	4419	4603	4675	18018	18653	18183	154925	144912	149028	593769	638851	599363
Backtracking generator – Backtracking solver	1166	1174	1159	4869	5029	4741	20044	18967	19631	142804	151994	143356	600773	574953	576249
Backtracking generator – Shortest path	1419	1415	1289	6682	6804	6976	8575	11059	12018	0	0	0	0	0	0
Ellers – Random mouse	1335	1289	1472	6419	6777	6848	0	0	0	0	0	0	0	0	0
Ellers – Trémaux	1170	1176	1160	5328	5069	5642	0	0	0	0	0	0	0	0	0
Ellers – Backtracking solver	1242	1310	1346	6478	6629	6653	0	0	0	0	0	0	0	0	0
Ellers – Shortest path	1118	1203	1071	5827	5866	6009	0	0	0	0	0	0	0	0	0
Wilsons – Random mouse	1072	1187	1171	4625	4533	4668	17945	18229	19504	135869	62050	0	0	248202	0
Wilsons – Trémaux	1024	1082	1189	5109	5026	5049	19311	18008	19295	219964	218845	222784	769563	799969	787052

Wilsons – Backtracking solver	1150	1249	1228	5323	5065	4732	19815	18581	18613	213390	229732	225370	943299	855771	863917
Wilsons – Shortest path	1084	1207	1171	4882	4663	4841	18550	18941	18134	119182	127128	136985	525256	519746	622444

Figure 61 Longest length of solutions results table.

Number of mazes completed.

Number of mazes completed	25x25 test 1	25x25 test 2	25x25 test 3	50x50 test 1	50x50 test 2	50x50 test 3	100x100 test 1	100x100 test 2	100x100 test 3	250x250 test 1	250x250 test 2	250x250 test 2	500x500 test 1	500x500 test 2	500x500 test 3
Binary Tree - Random mouse	50	50	50	50	50	50	50	50	50	5	5	4	0	0	0
Binary Tree - Trémaux	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
Binary Tree – Backtracking solver	50	50	50	50	50	50	50	50	50	50	50	50	40	42	39
Binary Tree - Shortest path	50	50	50	50	50	50	50	50	50	10	9	10	5	3	6
Backtracking generator – Random mouse	50	50	50	50	50	50	1	0	3	0	0	0	0	0	0
Backtracking generator – Trémaux	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
Backtracking generator – Backtracking solver	50	50	50	50	50	50	50	50	50	50	50	50	37	41	40
Backtracking generator – Shortest path	50	50	50	50	50	50	5	3	2	0	0	0	0	0	0

Ellers – Random mouse	50	50	50	50	50	50	0	0	0	0	0	0	0	0	0
Ellers – Trémaux	50	50	50	50	50	50	0	0	0	0	0	0	0	0	0
Ellers – Backtracking solver	50	50	50	50	50	50	0	0	0	0	0	0	0	0	0
Ellers – Shortest path	50	50	50	50	50	50	0	0	0	0	0	0	0	0	0
Wilsons – Random mouse	50	50	50	50	50	50	50	50	50	4	1	0	0	1	0
Wilsons – Trémaux	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
Wilsons – Backtracking solver	50	50	50	50	50	50	50	50	50	50	50	50	25	30	26
Wilsons – Shortest path	50	50	50	50	50	50	50	50	50	8	8	7	4	4	3

Figure 62 number of mazes completed results table.

Results Findings – Quantitative

From the results that I have gathered there are a couple of things that we can take away from them. First, as expected both the time and space that each test took increased in size based on the dimensions of the maze this was not particularly unexpected, but some do have more of a natural increase than other algorithm combinations. Some important things to mention are that because of the time limit I had set being 2 hours some of them did reach that mark resulting in not all the mazes being complete this was to be expected this mostly takes place in random mouse which is not so surprising as this is one of the few algorithms that has no guarantee to ever finish but shortest path also had a problem with time. It would always complete more than random mouse but still sometimes wouldn't finish ,this was in backtracking generator, this is mostly in the bigger dimensions that is why there are actually lower averages for the higher dimensions than some of the lower dimensions as for things like random mouse and shortest path in the 2 hours sometimes none were solved others it would be a really low amount of mazes solved this was actually the case for backtracking solver as well at the 500x500 dimension but this would at least finish around 40 mazes in the 2 hours. One thing that did surprise me was Ellers algorithms in that the thing that took most of the time was not the solving algorithms but the generation of the mazes once it got to 100 x 100 it would not even be able to generate a maze in the time frame that is why there are a lot of 0s in the results in that area. The reason I believe Ellers took so long to create mazes was most likely due to my implementation, but I could not think of many other ways to do it and although bigger mazes struggle to be created the smaller sizes take a more reasonable time. Another reason is most likely my laptop that I ran the tests on as it is not optimised or very powerful for running code.

From the results that are available you can see they are similar in most cases especially when the dimensions are lower but as the dimensions increase the differences do appear, but they are generally not that vast in their differences. From the results I can say on average that the best to worst solving algorithms are Trémaux and then backtracking solver then shortest path and then random mouse. This only on average there are some cases where this isn't the case and sometimes backtracking solver does sometimes get better results this is at lower dimensions though but in most circumstances, this isn't the case and the ranking I gave before is the most accurate ranking this is showable by looking at the total and average for both time and length of solutions. You can see this in the total timetable as with every algorithm except for Trémaux they didn't complete all the mazes at 500x500 whereas Trémaux got full times around the 2000 second mark other than in backtracking generator where that completed in around 600 seconds. This is also the same for space across the different mazes where Trémaux had solved all the mazes so that is the reason for higher values in the tables but from the parts where all algorithms completed all 50 mazes on most occasion Trémaux had less space taken to solve the mazes. This can be seen in the average length of solution table where Trémaux across all algorithms except backtracking generator had lower averages than every other algorithm on most occasions except for the few outliers but those are to be expected. It is only initially beaten at lower dimensions in backtracking generator by backtracking solver but as the dimensions size increases so too does the gap between Trémaux and backtracking solver, so Trémaux is better in almost all circumstances. This is mostly based off total for both time and length this is because the averages for the mazes are going to be a bit skewed as some algorithms didn't finish making all of the mazes, so the results are harder to compare. Another interesting thing is how most of the solving algorithms all have an exponential increase in average time except for Trémaux as with Trémaux with Binary Tree the times go from 0.11 to 0.30 to 3.04 to 9.62 then 51.43 which is a lot more linear compared to something like backtracking solver with Binary Tree which goes 0.16 to 1.20 to 11.43 to 23.64 to 180.00 which is a lot more drastic than Trémaux.

What these results don't show is how space is acquired this shows how many empty spaces have been filled in with solution symbols'#' so this is why they are generally similar in how much space is being taken up. When I have ran these tests, I made it so that the redundant paths that don't necessarily lead to a solution are accounted for from using the opposite of this where it just shows the solution and the matplotlib feature I discovered a lot. This will be discussed in the qualitative section. From the results as well, I can say that in most circumstances that backtracking generator mazes are the most compatible for most of the solving algorithms then it would be binary tree and then Wilsons is the hardest. I unfortunately can't confidently say where Eller's would place due to the lack of results but from the results, I do have I can guess that it would be in between binary tree and Wilson's in terms of compatibility with the solvers.

What I mean by compatibility is how many good results an algorithm gets from the tests compared to the other algorithms. This is also in terms of size as I have said previously the size naturally leads to bigger results, but the backtracking generator is faster and takes less space when being solved at bigger dimensions. Wilsons takes a lot more space to solve as well. This is not that surprising as in this paper (Permana, 2018 #17) Wilsons and backtracking are on the other sides of difficulty and dead ends met and intersections travelled but just as in my tests they are comparatively equal in how many steps it takes to solve them thus reinforcing what I have found. From what I can see with number of mazes solved Trémaux again is the best option for the solvers with backtracking solver being very close behind that is with the number of mazes solved taken into consideration meaning that for both time and space the order from best to worst is the same.

Results Findings - Qualitative

So, for the qualitative data the topics I will be talking about are at the start of this chapter. To get this data I used a mixture of the test file and the menu file as this allowed me to see end results and how the solving went through each maze. For how the mazes are solved I noticed that for shortest path when I would pass clear as true, I would find that the amount of space taken would be the shortest possible path, I know this as through testing in the range of 10x10 and 25x25 and 50x50 as going any bigger would be too time consuming for me to solve. I went through the mazes finding the answers and with all the other algorithms they would find answers and, but they would not necessarily always be the best solution unlike shortest path. In all of the solvers as well I noticed that during the process a lot of them would backtrack and go over already visited squares on the grid and this was very prevalent in random mouse where there would be a ten second periods were it would go backwards and forwards in the same direction. Also, there was not a single algorithm that sometimes would go through big chunks of the maze when the finish was right next to the start all because of the random direction choices but there were many instances when they would pick the direction and get to the finish quickly.

I also did do some solving of mazes without the use of solving algorithms myself to see how well a human could solve them personally at the lower dimensions I wouldn't say there was any particular differences in terms of difficulty but as the mazes did get bigger there were some more noticeable things about the maze these being that for binary maze there were some very distinct passage made in the direction of the bias that did make it easier to solve than the other maze and again in Eller's there were sometimes when the horizontal and vertical biases were more noticeable sometimes making it easier to solve and because of its ability to produce islands in the maze it made it easier as it sometimes allowed for much shorter and easier to see solutions. Along with this "river" was very noticeable in some of the mazes like Wilsons with a very big mix of "river" this is because at the start there are long passages with dead end but by the end of the process there are many short passages with dead ends whereas with something like binary tree there is more "river" than other mazes as there are many long passages. Wilsons and backtracking generator were more difficult than the others as because of how they are made there were a lot more passages that would lead to dead ends this made bigger mazes more difficult as there were longer passages that you would go down and then must go all the way back. One of the other pieces of qualitative data I wanted to get was to see if any of the mazes could be compared to real world things like maps and roads or building layouts. Personally, I believe Binary and backtracking generator in most circumstances don't really have many uses but maybe you could use binary for road layouts but that's about it. But Wilsons and Ellers I could see being able to be used in the real world especially Ellers as it can make islands which you could consider building in a road layout or destinations on a map and because Wilsons gets to the point where it makes compact branches it could be seen as something like a cul-de-sac. These can be seen in figures 63 and 64.

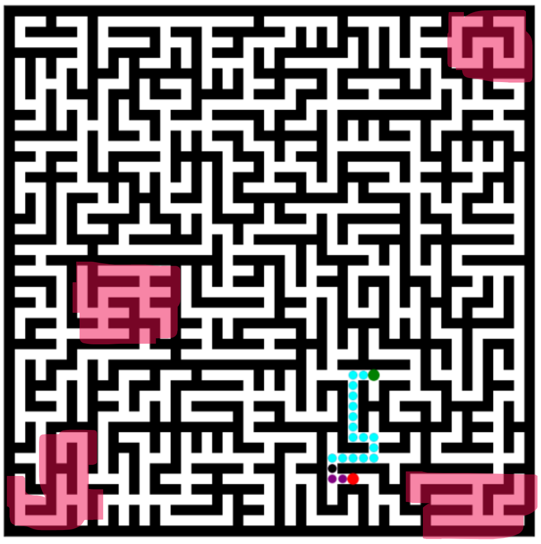


Figure 63 Wilsons output showing cul-de-sac.

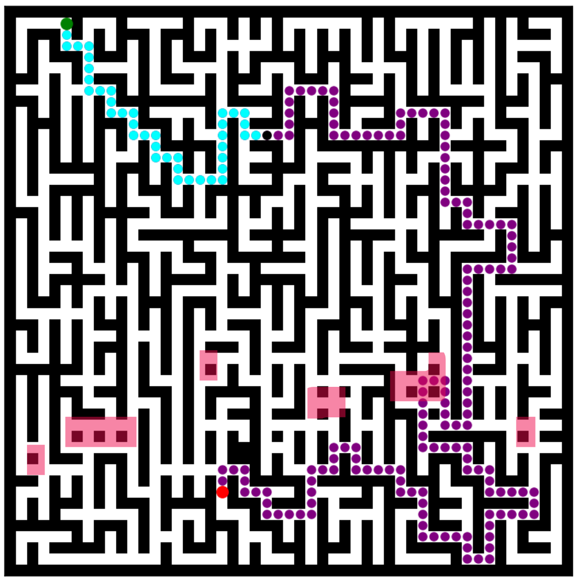
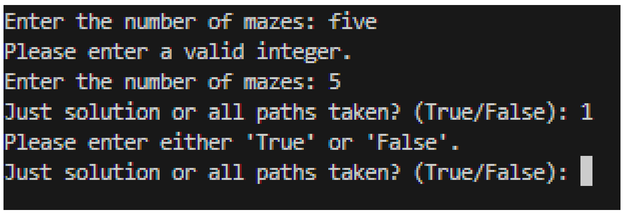


Figure 64 Ellers output showing islands.

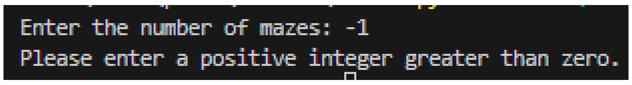
Input Testing

In the functional and non-functional requirements section in the design I talked mostly about how I and potential other people would interact with the system in this and throughout the dissertation I have said that my focus is not on the user but the algorithms themselves but as I have some inputs I thought it best to implement some input validation into the code this section will show the tests and outcomes of testing the input validation. As I only have two main instances of input validation I will only be talking about them as for the testing file that is specifically meant for me and although there is validation for some parts of it there are some parts that don't mainly picking the x and y dimensions which is meant to be hard coded before running the application so I never intended for this to need inputs unlike in the menu file where it was intended. This was tested however and as suspected this just leads to an error making the code un runnable.



```
Enter the number of mazes: five
Please enter a valid integer.
Enter the number of mazes: 5
Just solution or all paths taken? (True/False): 1
Please enter either 'True' or 'False'.
Just solution or all paths taken? (True/False):
```

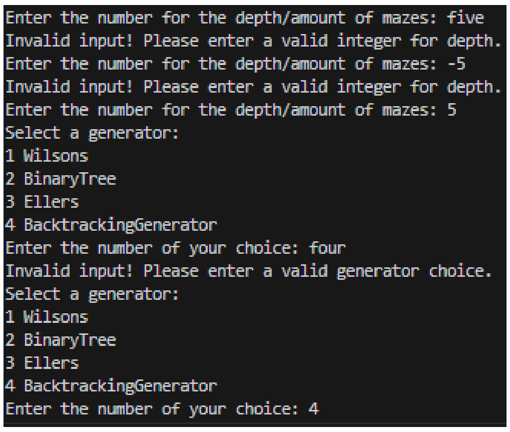
Figure 65 input testing on test file.



```
Enter the number of mazes: -1
Please enter a positive integer greater than zero.
```

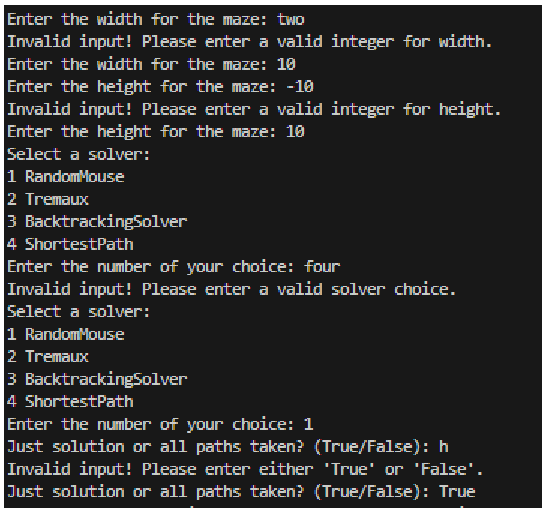
Figure 66 input test 2 on test file.

As you can see in the figures above input validation does work as I had intended it to. Where there is expected to be an integer and anything other than an integer is inputted then it asks for a valid integer and if a negative is given it asks for a number above 0 just as intended. Then you can see in the figures below that also for the menu file the input validation does what is expected when the input is wrong it says it is and asks you to input the correct value just as I had intended this input is more important as this would be the file used by other people if that was to ever happen. The code for all of this can be seen in appendix B.



```
Enter the number for the depth/amount of mazes: five
Invalid input! Please enter a valid integer for depth.
Enter the number for the depth/amount of mazes: -5
Invalid input! Please enter a valid integer for depth.
Enter the number for the depth/amount of mazes: 5
Select a generator:
1 Wilsons
2 BinaryTree
3 Ellers
4 BacktrackingGenerator
Enter the number of your choice: four
Invalid input! Please enter a valid generator choice.
Select a generator:
1 Wilsons
2 BinaryTree
3 Ellers
4 BacktrackingGenerator
Enter the number of your choice: 4
```

Figure 67 input test on menu file.



```
Enter the width for the maze: two
Invalid input! Please enter a valid integer for width.
Enter the width for the maze: 10
Enter the height for the maze: -10
Invalid input! Please enter a valid integer for height.
Enter the height for the maze: 10
Select a solver:
1 RandomMouse
2 Tremaux
3 BacktrackingSolver
4 ShortestPath
Enter the number of your choice: four
Invalid input! Please enter a valid solver choice.
Select a solver:
1 RandomMouse
2 Tremaux
3 BacktrackingSolver
4 ShortestPath
Enter the number of your choice: 1
Just solution or all paths taken? (True/False): h
Invalid input! Please enter either 'True' or 'False'.
Just solution or all paths taken? (True/False): True
```

Figure 68 input test on menu file part 2.

Evaluation of Artefact

Compared to my original design and ideas I had for my artefact I would say that there have not been many changes, but changes have mostly been in the interactive menus. I believe that I have my met my original requirements these being able to compare the effects of multiple algorithms against one another as in my artefact you can select any maze generation and maze solving algorithm and get tangible results from running the program whilst also being able to change the dimensions of the mazes. You can also see this visually as well which I had also planned to do this as well can be seen in two ways one being through strings and the other through the use of animation in matplotlib which originally was not meant to be as complicated as it was as this was just meant to be a visual aid to help me see how the algorithms work. There is a lot more user focused approaches that I took when implementing the artefact this was solely for me as so that during creation and testing that I could increase the ease of use of the artefact which unintentionally led to more of a focus on how someone else might use it.

Artefact Conclusion

By testing and the artefact that I have made I have revealed in general which algorithms work best together but in real circumstances this may not be the case as it really depends on the needs of the person that is using it if they want a small maze that is easily solvable by all I would say that in most circumstances Ellers or backtracking generator might be the best pick but at bigger dimensions Ellers becomes too time consuming in the generation process which if time is not an issue it would be fine but this rarely isn't something someone cares about that is why I found it so important to make sure to get results from mazes of that higher dimension. This can be said of the solvers as well if you want time and space Trémaux will most likely be the best option you have but it won't necessarily produce the best solution to the finish shortest path would be. As I have said it solely depends on what the us needed from the algorithms.

Conclusion and Future Work

This dissertation has been my first experience with this kind of work. At first, I didn't know or realise the actual importance of mazes as a topic of study and their actual real world uses which in turn made it a much more interesting and complicated topic than I could have realised. Which makes me strongly believe that this subject deserves a more in depth studying than it currently has. Currently there is no concentrated source of information on mazes it is very much a scattered topic in terms of information many of the important subjects that play big roles in maze generation and solving are not easy to come by and explain that is why throughout my first chapters I tried my best to help explain some concepts to the average reader as I believe it is necessary to introduce more people to the interesting subject.

My original goal for this dissertation was to be able to compare different maze generation algorithms with different maze solving algorithms in 3D and see what interesting results can come from that I believe that I have achieved this goal. But this did not happen without some problems my main problem would be with time management in that there were times during the dissertation that I fell behind my planned work schedule although I did catch up to my planned work schedule this was definitely something I struggled with at the time but because of this it allowed me to finish with enough time spare to allow for further development of previous chapters and taught me a valuable lesson. Also, I would say another problem was my underestimation of some chapters more specifically my literature review and background research in that before starting I previously believed that the section would be quite easy and not that hard but as I have said information is scattered and hard to organise so this took up a lot more time than I previously though it would. Along with these problems other difficulties occurred during the implementation process due to unexpected complexities of some of the algorithms and problems with my inexperience with matplotlib produced some setbacks. In the middle of the dissertation process the presentation happened. I believe that this went well although it was challenging due to the time it was set meaning I could only show so much about what I had written and the natural dislike I have of presentations I still think it went well and overall, I believe my dissertation has gone well. Some things I believe did go well as well such as the implementation during which yes, I had the usual problems, but I actually enjoyed the creation of the artefact and solving those problems. Along with Being able to finish earlier than expected allowed me time to add more and refine the project.

As I have said mazes in general is a fascinating topic that can me delved into quite deeply this is the same for my dissertation and artefact as I believe there is a lot more work that can be done in the future to deepen the understanding of this topic. Some of these include a more extensive testing what I mean by this is that my testing only covers the basic measurements of mazes this being space and time and I have focused a bit more on the solving algorithms in my testing compared to the generation algorithms also other measurements could be how many dead ends there are in a maze how many unbroken passages there are etc. Other works can be to introduce more algorithms into the system because as of now I only have four for both generation and solving but there are many more that can be introduced. Although I did not intend for my artefact to be used by other people the artefact could be expanded upon more to make it more presentable to other potential users and it could be made to allow human users to solve the mazes instead or make the mazes.

The topic of mazes in general is considered shallow but underneath it is quite important and interesting but has really only found important uses now due technology expanding. Uses in things like self-driving cars and GPS is what mazes are now mainly used for, but it is still widely considered as a something simple like a game or brain teaser. I hope that through my dissertation a higher interest in the topic of mazes can be made and although I am happy with my artefact and dissertation, I believe that future work on the topic can produce a lot more interesting results and further expand the topic.

References

Matthews, W.H., 1970. *Mazes and labyrinths: their history and development*. Courier Corporation.

Pullen, W. D. 2022. *Maze Classification* [Online]. Walter D Pullen. Available: https://www.astrolog.org/labyrnth/algrithm.htm [Accessed 30/10/ 2023].

2023. *dimension* [Online]. Oxford English Dictionary. Available: https://www.oed.com/dictionary/dimension_n?tab=factsheet#6810473 [Accessed 30/10/ 2023].

Britannica, E.O.E. 2009. *Euclidean space* [Online]. Britannica Available: https://www.britannica.com/science/Euclidean-space [Accessed 30/10/ 2023].

AWATI, R. 2021. *cellular automaton* [Online]. TechTarget. Available#: ~ttps://www.techtarget.com/searchenterprisedesktop/definition/cellular-automaton#:~:text=A%20cellular%20automaton%20(CA)%20is,the%20states%20of%20neighboring%20cells. [Accessed 06/11/ 2023].

EULER, L. 1956. The seven bridges of Königsberg. *The world of mathematics,* 1**,** 573-580.

VAIBHAV. 2023. *Spanning Tree* [Online]. GeeksforGeeks. Available: https://www.geeksforgeeks.org/spanning-tree/ [Accessed 07/11/ 2023].

s chaiken, d. j. k., department of mathematics, massachusetts institute of technology, cambridge, massachusetts 1977. matrix tree theorems. *combinatorial theory,* 24**,** 377-381.

shatha alamri, s. a., wejdan alshehri, hadeel alamri, ahad alaklabi, tareq alhmiedat 2021. autonomous maze solving robotics: algorithms and systems.

swati mishra, p. b. 2008. maze solving algorithms for micro mouse. *2008 IEEE international conference.* IEEE.

BUCK, J. 2011. *Maze Generation: Binary Tree Algorithm* [Online]. Available: https://weblog.jamisbuck.org/2011/2/1/maze-generation-binary-tree-algorithm [Accessed 12/11/ 2023].

BUCK, J. 2010. *Maze Generation: Eller's Algorithm* [Online]. Available: https://weblog.jamisbuck.org/2010/12/29/maze-generation-eller-s-algorithm [Accessed 12/11/ 2023].

BUCK, J. 2011. *Maze Generation: Wilson's algorithm* [Online]. The Buckblog. Available: https://weblog.jamisbuck.org/2011/1/20/maze-generation-wilson-s-algorithm [Accessed 13/11/ 2023].

NESTER, D. 2018. *Maze Profiler* [Online]. Bluffton University website. Available: https://homepages.bluffton.edu/~nesterd/apps/mazeprofiler.html [Accessed 13/11/ 2023].

PERMANA, S. H., BINTORO, K. Y., ARIFITAMA, B. & SYAHPUTRA, A. 2018. Comparative analysis of pathfinding algorithms A*, Dijkstra, and BFS on maze runner game. *IJISTECH (International J. Inf. Syst. Technol., vol. 1, no. 2, p. 1*.

GABROVŠEK, P. 2019. Analysis of maze generating algorithms. *IPSI Transactions on Internet Research,* 15**,** 23-30.

WALTER.D. PULLEN. 2023. *Daedalus 3.4* [Online]. Available: https://www.astrolog.org/labyrnth/daedalus.htm [Accessed 25/11/ 2023].

BUCK, J. 2015. *Mazes for programmers*.

Parameterized maze generation algorithm for specific difficulty maze generation L. Peachey. Association for Computing Machinery 2022 Vol. 1 Issue 1

Appendix A – Project management

Project Specification

(Dang et al., 2010)School of Computer Science and Mathematics

6200COMP Project
Project Specification Form

Complete this page and write your specification starting on the next page using the headings provided. This template is to be used to provide the initial outline for your project. It does not count toward the final assessment, rather it is to help you get started with your project. You can change the title or ideas at any time, but you should negotiate this with your supervisor.

The **First Draft** should be submitted to your supervisor by **Friday 13th October 2023**.

The final/approved version of the **Project Specification** should be uploaded to Canvas by **Friday 27th October 2023**.

1. Project details

Name	Luke Curran
Student No	958598
Programme	Computer Science
Supervisor	Reino Niskanen
Initial Project Title	Solving Procedurally Generated 3D Mazes
Brief description (Up to 100 words)	Procedurally generate 3D mazes using different algorithms and then creating agents that will solve these mazes whilst each agent is using a different algorithm to solve them. As they are procedurally generated it means that each time a new maze is created it is different than the last, but it could be made using the same algorithm or a completely different one.

2. Checklist

Meeting	I have met with my supervisor to discuss my ideas	[✓ YES]/NO
Approval	I have received my supervisor's approval for the specification	[✓ YES]/NO
Management	I have arranged regular meetings with my supervisor	[✓ YES]/NO

Ethics	I have discussed ethical approval for the project	[✓ YES]/NO
Domain	I have agreed which subject areas or problem domain I will be undertaking the project in	[✓ YES]/NO

1. **Background**

Mazes and maze solving at first might seem they are only used for things like games but that couldn't be further from reality as maze solving can provide us with useful information that can be used in everyone's daily lives. Maze solving and their algorithms directly relate to things such as autonomous vehicles, GPS, network routing, architecture and urban planning, emergency response routes and medical operations.

Mazes have two parts, the actual maze and the things solving it. In real life this can be simple as someone grabbing a pen and paper drawing a simple 2D mazes and then getting someone to solve it. But this is completely different for computers as they need to be told the rules in how to make and solve the maze. This can be done in many ways as there are many different algorithms to make and solve mazes.

Some of the algorithms that are used for maze generation include depth first search, randomized prims algorithm, binary tree algorithm and sidewinder algorithm and many more. All of these algorithms can be used in many different situations outside of maze generation, but all of them can be used to generate mazes, but they all do it in a different way which can result in very different looking mazes from one another ,this is called bias it means that the way each maze is made by each algorithm they can be told apart in a glance just by what the end result looks like. For example, binary tree algorithms will methodically go through each cell and delete 1 of 2 walls this is normally either the north, east, south, or west wall (depending on the current cells location). Whereas the sidewinder although very similar to the binary tree algorithm does things row by row instead and will do cell runs.

Once the mazes are made that is where the maze solving agents are introduced. These agents will solve the mazes. Just like the maze generation there are many algorithms for this too such as A* algorithm, Dijkstra algorithm, shortest path algorithm and random mouse algorithm. All of these algorithms work with each maze, but some algorithms don't work well together for example the shortest path algorithm solver might not work well with a maze made by the binary tree algorithm. This is because each algorithm solves the maze in a different way for example the random mouse algorithm is very simple in that it will move along the path until reaching a junction and then will make a random decision in which way to go this will eventually lead to the maze being solved but if the maze was just made up of loads of junctions and dead ends it might get lucky and pick the right way but there is a high probability that won't happen which means it will take a while to solve the maze. This is why this project requires evaluation and analysis of the results as they will be able to be compared with one another and this will allow me to come to conclusions about the algorithms.

2. Problem

In this dissertation, there is two main problems these being the maze generation and the maze solving agent. As already talked about there is many algorithms that can be implemented for each of these problems so the real overall problem for these is implementation.

Maze generation will have to be done in a language like python for example as pygame will then be an option for the display of the actual maze. When someone thinks about a maze, they will normally think about a 2D maze but in this dissertation 3D mazes will be generated instead. 2D mazes already have their own problems such as bias as explained in the section before but as a new dimension is going to be added this again will make more problem such as how the agent will progress to the next level this will probably be solved by either using a "ladder" on the first and second level of the maze or a row of cells as "stairs" that will lead to the next level. The size of the maze could also cause problems although I do plan on limiting the maze the minimum dimension will be 5 x 5 x 3 but if it goes any bigger some algorithms may take longer than others to create the maze.

Again, the maze solvers will be done in python as well and just like the maze generator there are many algorithms for the solvers to follow and use to solve the mazes. One problem that will most likely arise is that although I don't actually need to see where the agent has been if I don't implement something like a trail that follows the agent, and some problem occurs with agent it will help me to solve the problem and it does also look visually better as will help display how the algorithms work.

Solving both of these problems can be done by humans with no problem except for the fact that if you were to do this for a maze that has the dimensions of 100 x 100 x 3 it would take a very long time to make the maze and to solve. That is why this problem is best solved using computers as it will take computers a fraction of the time it would take a human to make and solve the mazes and it does also get rid of human error as a human might double back on themselves when solving a computer wouldn't do that.

I have a good amount of experience with python, and I have done a couple of projects myself in pygame. I also have prior knowledge to some of the algorithms that will be implemented but that is only at a theoretical level, and I have never implemented them in any code before so I will have to learn how to do that.

3. Aims and Objectives

Aims:

- Create code that will procedurally generate a 3D maze using different algorithms.
- Create code that will make an agent that will solve these mazes by using algorithms.
- Evaluate and analyse the results of the code in terms of time it takes each algorithm to solve each maze and also the space each agent takes when solving the maze.

Objectives:

- Create a grid of n x n size (still TBD)
- Implement existing maze generation algorithms to create a 2D maze: binary tree, sidewinder, Wilson's algorithm, and backtracking.
- Figure out how to stack 2D mazes on top of each other to make a 3D maze.

- Display the maze.
- Implement existing maze solving algorithms: Dijkstra, shortest path, Trémaux's algorithm, wall follower algorithm.
- Add an agent tracer in that where the agent goes a line or colour follows.
- Add a clock so evaluation is easier at the end.
- Add a 'counter' to determine the space each agent takes.

4. Hardware and Software Requirements

There is no specific hardware or software requirements, all I will need is a computer/laptop that can run python and the libraries in python also Microsoft office.

5. Project Plan

dissertation gantt.xlsx

Gantt link provided above.

As you can see my Gantt is split into two sections report writing and then coding and implementation. Although I have set out this timeline it will most likely vary and change as I go through my project for example although I will get most of the report done before Christmas after I have finished my code or even during it, I might be able to go back and add more information to the report and I will have to do evaluation and testing which will all be written in the report.

In my first couple of weeks, I have said that I will do background research and I will do most of my research then I believe that as I progress through the project, I will most likely have to do more research. This Gantt has also taken into consideration that I am not exactly certain in when the presentation will be and that again could change the timeline for parts of the project.

6. References

Aqel, M.O., Issa, A., Khdair, M., ElHabbash, M., AbuBaker, M. and Massoud, M., 2017, October. Intelligent maze solving robot based on image processing and graph theory algorithms. In *2017 International Conference on Promising Electronic Technologies (ICPET)* (pp. 48-53). IEEE.

Kozlova, A., Brown, J.A. and Reading, E., 2015, August. Examination of representational expression in maze generation algorithms. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 532-533). IEEE.

Kaur, N.K.S., 2019. A review of various maze solving algorithms based on graph theory. *IJSRD*, *6*(12), pp.431-434.

Sagming, M.N., Heymann, R. and Hurwitz, E., 2019, September. Visualising and solving a maze using an artificial intelligence technique. In *2019 IEEE AFRICON* (pp. 1-7). IEEE.

Schrijver, A., 2012. On the history of the shortest path problem. *Documenta Mathematica*, *17*(1), pp.155-167.

Teigland, A.K., 2022. Evaluation of a Modified Trémaux's Maze Solving Algorithm.

Niemczyk, R. and Zawiślak, S., 2018, December. Review of Maze Solving Algorithms for 2D Maze and Their Visualisation. In *International Conference of Students, PhD Students and Young Scientists" Engineer of the XXI Century"* (pp. 239-252). Cham: Springer International Publishing.

Kim, P.H., 2019. *Intelligent maze generation*. The Ohio State University.

Signature (student)	Luke Curran	Date	09/10/23
Signature (supervisor)	Reino Niskanen	Date	20/10/23

Signed Monthly Reports

6200COMP Project
Monthly Supervision Meeting Record

Progress Report #1
Month: November 2023

This form should be completed in the first instance by the student based on the progress up to 10 November 2023. The first draft should be sent by email to the supervisor by that date. The student should use the next progress meeting to discuss the points raised in this form with their supervisor. A final signed form should be uploaded to Canvas by 17 November 2023.

Note: Timely adherence to monthly progress reporting schedule is part of the Project Management mark component. Failure to upload agreed and signed monthly report timely will affect your Project Management mark adversely.

Student's Name: Luke Curran

Supervisor's Name: Reino Niskanen

1. Main issues / Points of discussion / Progress made
I have completed my research stage and have created my first chapter which encompasses the basics of mazes overall tat is now completed and I am in the middle of writing my second chapter which will

talk about maze generation and solving algorithms and also the basics of graph theory and other subjects that are needed to better explain my artefact and also the algorithms. At first, I did struggle a bit with the referencing as I realised it slowed me down more than I thought it would but as I have gotten into writing the dissertation it isn't that big of a problem anymore. At the moment my main problem is the literature review as I have done what I think a literature review is, but I am not certain if it is up to the standard that is expected. If I can get my second chapter done by the time, I come back from reading week I will be on track. So that means I am currently on track.
2. List of actions for the next month
My plan for the next month is to start my artefact design section this will involve UML diagrams, pseudocode, and flow charts in this section I will also do my methodologies and also the hardware and software requirements as I believe these two things will fit better together in one chapter instead of two. I Plan to get this chapter done by the time we go off for Christmas. Some problems I might encounter is the literature review section of this chapter but I should be getting an example soon so that will probably not be a problem. I also do not really enjoy creating UML diagrams so that might be a problem, but they are necessary. I do know how to create UML diagrams, so it is not necessarily a problem of creation, but it might take me more time than the other parts of this chapter. Also, other coursework will take up my time as well so they will inevitably get in the way of the dissertation, but I plan on doing them as early as possible to get them out of the way.
3. List of deliverables for next time
Because I will be doing my design of the artefact next month I will most likely be producing pseudocode, UML diagrams and flowcharts. Whilst also carrying on with the report writing whilst I am writing about methodologies and hardware and software requirements. In this section as well, I will be doing more literature reviews throughout it instead one big section of just literature review so that will also be produced as well Because of the design including pseudocode this will also give me a basis for my actual code as well.
4. Other comments
Use this box to inform and record anything else that does not fall into any of the above category. For example, if you plan to go be away for a few days due to emergency which will be affecting your progress.

Signature (student)	Luke Curran	Date	10/11/23
Signature (supervisor)	[illegible]	Date	14/11/23

6200COMP Project
Monthly Supervision Meeting Record

Progress Report #2
Month: December 2023

This form should be completed in the first instance by the student based on the progress up to 8 December 2023. The first draft should be sent by email to the supervisor by that date. The student should use the next progress meeting to discuss the points raised in this form with their supervisor. A final signed form should be uploaded to Canvas by 15 December 2023.

Note: Timely adherence to monthly progress reporting schedule is part of the Project Management mark component. Failure to upload agreed and signed monthly report timely will affect your Project Management mark adversely.

Student's Name: Luke Curran

Supervisor's Name: Reino Niskanen

5. Main issues / Points of discussion / Progress made
This month I have done the problem analysis and the methodologies section of the report I have also went back to previously completed sections and added more content and polished some parts as well. I am currently behind schedule as I was hoping to have been part way through the design section of the report by now, but I have not started that section, yet this is mostly due to other coursework's.
6. List of actions for the next month
My plan for next month is to get the design finished by the time I return from winter break I will also have to create my presentation for the 12th so this will take up time as well, meaning my main problem will be lack of time this is because I will also have other deadlines over the winter break. Although my initial plan was to be starting the code around now, I still believe I am in a good position so there will be a bit of a change to the initial plan by pushing the coding section back to the start of the second semester.
7. List of deliverables for next time
I will be delivering some of the design components that I mentioned last time these being: • *UML diagrams* • *Pseudocode* • *Flowcharts* • *Literature reviews* • *Etc*

8. Other comments
Use this box to inform and record anything else that does not fall into any of the above category. For example, if you plan to go be away for a few days due to emergency which will be affecting your progress.

Signature (student)	Luke Curran	Date	08/12/2023
Signature (supervisor)	[illegible]	Date	11/12/23

6200COMP Project
Monthly Supervision Meeting Record

Progress Report #3
Month: January 2024

This form should be completed in the first instance by the student based on the progress up to 19 January 2024. The first draft should be sent by email to the supervisor by that date. The student should use the next progress meeting to discuss the points raised in this form with their supervisor. A final signed form should be uploaded to Canvas by 26 January 2024.

Note: Timely adherence to monthly progress reporting schedule is part of the Project Management mark component. Failure to upload agreed and signed monthly report timely will affect your Project Management mark adversely.

Student's Name: Luke Curran

Supervisor's Name: Reino Niskanen

9. Main issues / Points of discussion / Progress made
This month I have almost completed the design of the artefact it is very close to completion for my changed timeline I am on track as I had planned to have the design completed around this time. I have not run into any serious problems whilst completing this section.
10. List of actions for the next month

My plan for the next month is to start and complete my code I will undoubtedly run into problems to fix this I will get help from online resources and my supervisor. Problems I will run into could be the 3D implementation and the acquisition of data from the algorithms.
11. List of deliverables for next time
I plan on delivering most if not all the completed artefacts. This means the algorithms implemented and also the 3D mazes and the measurement code as well as the interactive menu.
12. Other comments

Signature (student)	Luke Curran	Date	24/01/24
Signature (supervisor)	[illegible]	Date	25/1/24

6200COMP Project
Monthly Supervision Meeting Record

Progress Report #1
Month: February 2024

This form should be completed in the first instance by the student based on the progress up to 16 February 2024. The first draft should be sent by email to the supervisor by that date. The student should use the next progress meeting to discuss the points raised in this form with their supervisor. A final signed form should be uploaded to Canvas by 23 February 2024.

Note: Timely adherence to monthly progress reporting schedule is part of the Project Management mark component. Failure to upload agreed and signed monthly report timely will affect your Project Management mark adversely.

Student's Name: Luke Curran

Supervisor's Name: Reino Niskanen

13. Main issues / Points of discussion / Progress made
This month I have nearly finished my code there are just some things I need to touch up on and fix but that should be fixed in the coming days. Some things I have had issues with is that I needed somethings clearing up on to do with certain aspects of the code for example how the 3D aspect should work and what should be animated and what shouldn't. In the code some algorithms have caused me more problems than other like shortest path solver and Eller's and Wilson's generators, but they are working now. I am currently on track with the schedule.
14. List of actions for the next month
My plan for the next month is to make the most of reading week and fully complete the implementation documentation during that time if that doesn't happen it will be done by the following week which will then leave me the rest of the month to do the evaluation and testing which will probably take me into the next month. If I encounter any problems, I will ask my supervisor for assistance and also online resources will be used.
15. List of deliverables for next time
Next month I plan on delivering documentation for my artefact in the report and maybe some of the evaluation and testing as well in my report.
16. Other comments

Signature (student)	Luke Curran	Date	21/02/2024
Signature (supervisor)	[illegible]	Date	21/2/24

6200COMP Project
Monthly Supervision Meeting Record

Progress Report #5
Month: March 2024

This form should be completed in the first instance by the student based on the progress up to 15 March 2024. The first draft should be sent by email to the supervisor by that date. The student should use the next progress meeting to discuss the points raised in this form with their supervisor. A final signed form should be uploaded to Canvas by 22 March 2024.

Note: Timely adherence to monthly progress reporting schedule is part of the Project Management mark component. Failure to upload agreed and signed monthly report timely will affect your Project Management mark adversely.

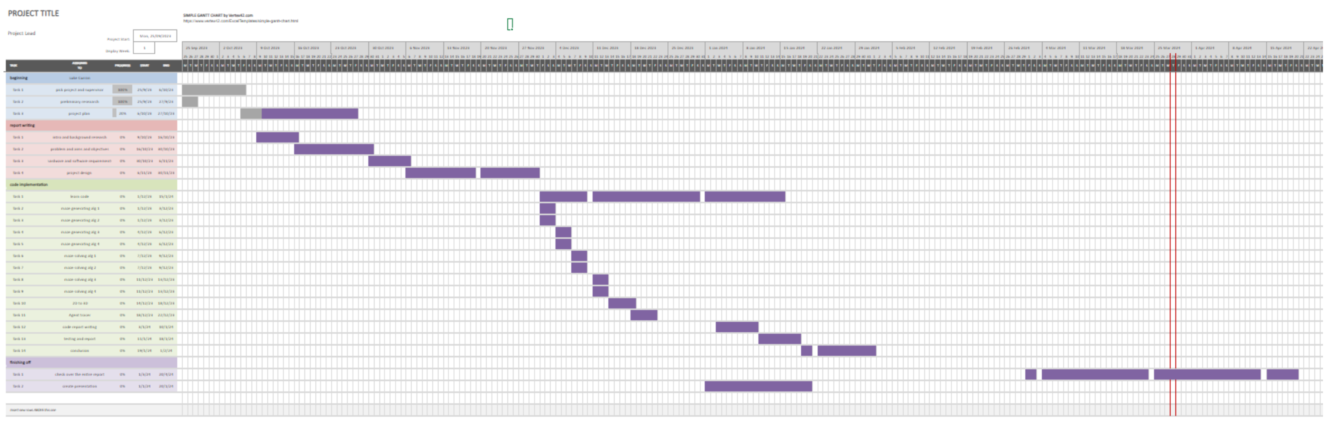
Student's Name: Luke Curran

Supervisor's Name: Reino Niskanen

17. Main issues / Points of discussion / Progress made
At this point I have finished the implementation section of my dissertation, and I am now almost finished with the testing section as well. One problem that I have encountered has been during my testing were my laptop has crashed a couple of times because of the testing but other than that there have not been any major problems. Currently I am still on track from what I previously had thought.
18. List of actions for the next month
My plan for the next month is to get the testing section finished as well as finish my dissertation with the conclusion and future works. This should leave me with sometime to go over what I have previously done and make changes and add content to my dissertation. If I encounter any problems, I will ask my advisor and use online resources.
19. List of deliverables for next time
Next month I plan to produce more of the report as it is really only conclusions and evaluations during my read over of the dissertation, I may discover the need for more work on the design section or more background research but that is if they are needed.
20. Other comments
Use this box to inform and record anything else that does not fall into any of the above category. For example, if you plan to go be away for a few days due to emergency which will be affecting your progress.

Signature (student)	Luke Curran	Date	15/03/24
Signature (supervisor)	[illegible]	Date	19/3/24

Initial project Gantt


PROJECT TITLE

Project Lead

Project Start: Mon, 25/09/2023

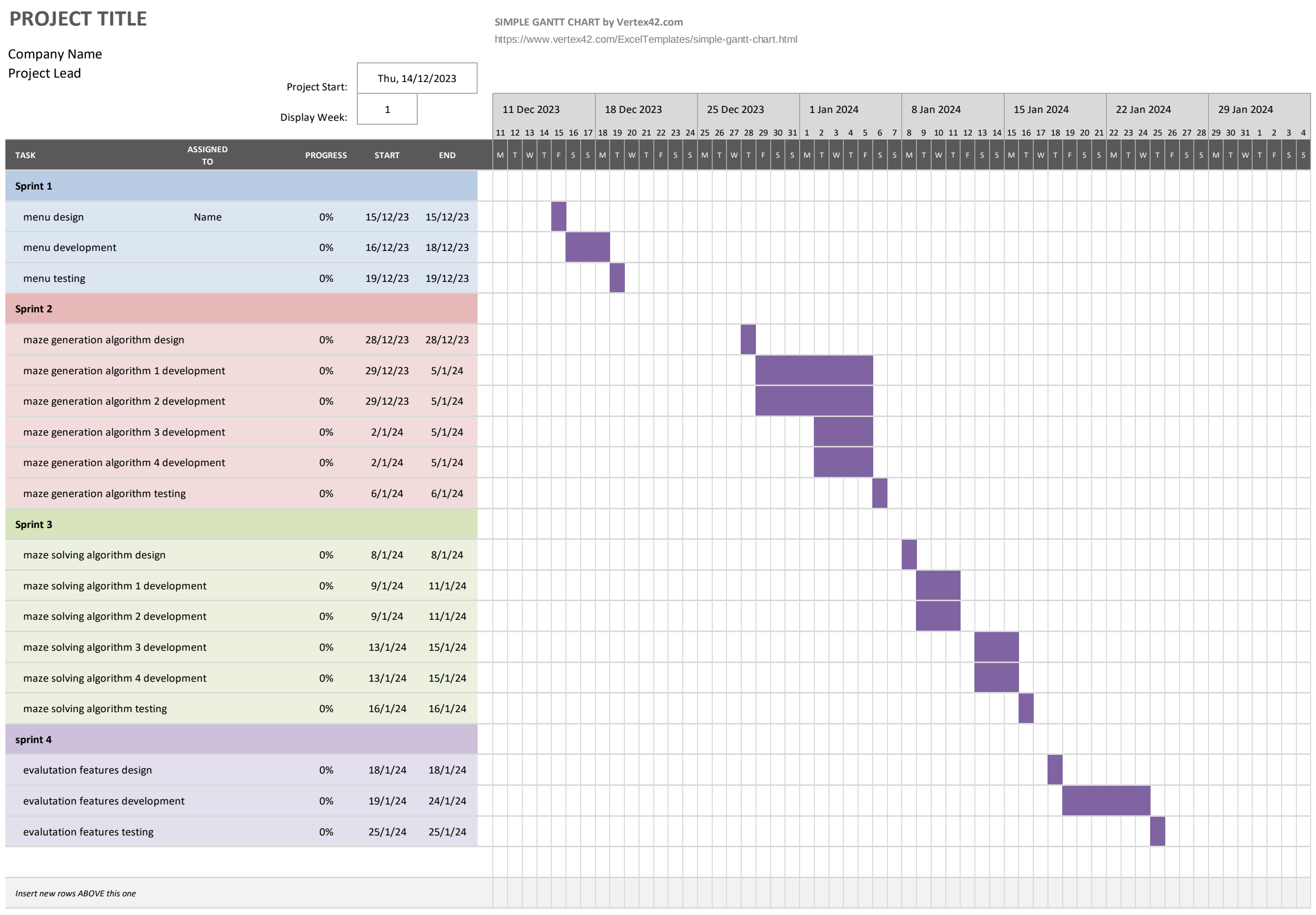
Display Week: 1

TASK	ASSIGNED TO	PROGRESS	START	END
beginning	Luke Carton			
Task 1	pick project and supervisor	100%	25/9/23	6/10/23
Task 2	preliminary research	100%	25/9/23	27/9/23
Task 3	project plan	20%	6/10/23	27/10/23
report writing				
Task 1	intro and background research	0%	9/10/23	16/10/23
Task 2	problem and aims and objectives	0%	16/10/23	30/10/23
Task 3	hardware and software requirement	0%	30/10/23	6/11/23
Task 4	project design	0%	6/11/23	30/11/23
code implementation				
Task 1	learn code	0%	1/12/23	15/1/24
Task 2	maze generating alg 1	0%	1/12/23	3/12/23
Task 3	maze generating alg 2	0%	1/12/23	3/12/23
Task 4	maze generating alg 3	0%	4/12/23	6/12/23
Task 5	maze generating alg 4	0%	4/12/23	6/12/23
Task 6	maze solving alg 1	0%	7/12/23	9/12/23
Task 7	maze solving alg 2	0%	7/12/23	9/12/23
Task 8	maze solving alg 3	0%	11/12/23	14/12/23
Task 9	maze solving alg 4	0%	11/12/23	14/12/23
Task 10	2D to 3D	0%	14/12/23	18/12/23
Task 11	Agent tracer	0%	18/12/23	22/12/23
Task 12	code report writing	0%	3/1/24	10/1/24
Task 13	testing and report	0%	11/1/24	18/1/24
Task 14	conclusion	0%	19/1/24	1/2/24
finishing off				
Task 1	check over the entire report	0%	1/3/24	30/4/24
Task 2	create presentation	0%	1/1/24	30/1/24

Insert new rows ABOVE this one

Figure 69 Dissertation Gantt chart planned out at the very start of the project.

Artefact Gantt chart


PROJECT TITLE

SIMPLE GANTT CHART by Vertex42.com
https://www.vertex42.com/ExcelTemplates/simple-gantt-chart.html

Company Name
Project Lead

Project Start: Thu, 14/12/2023

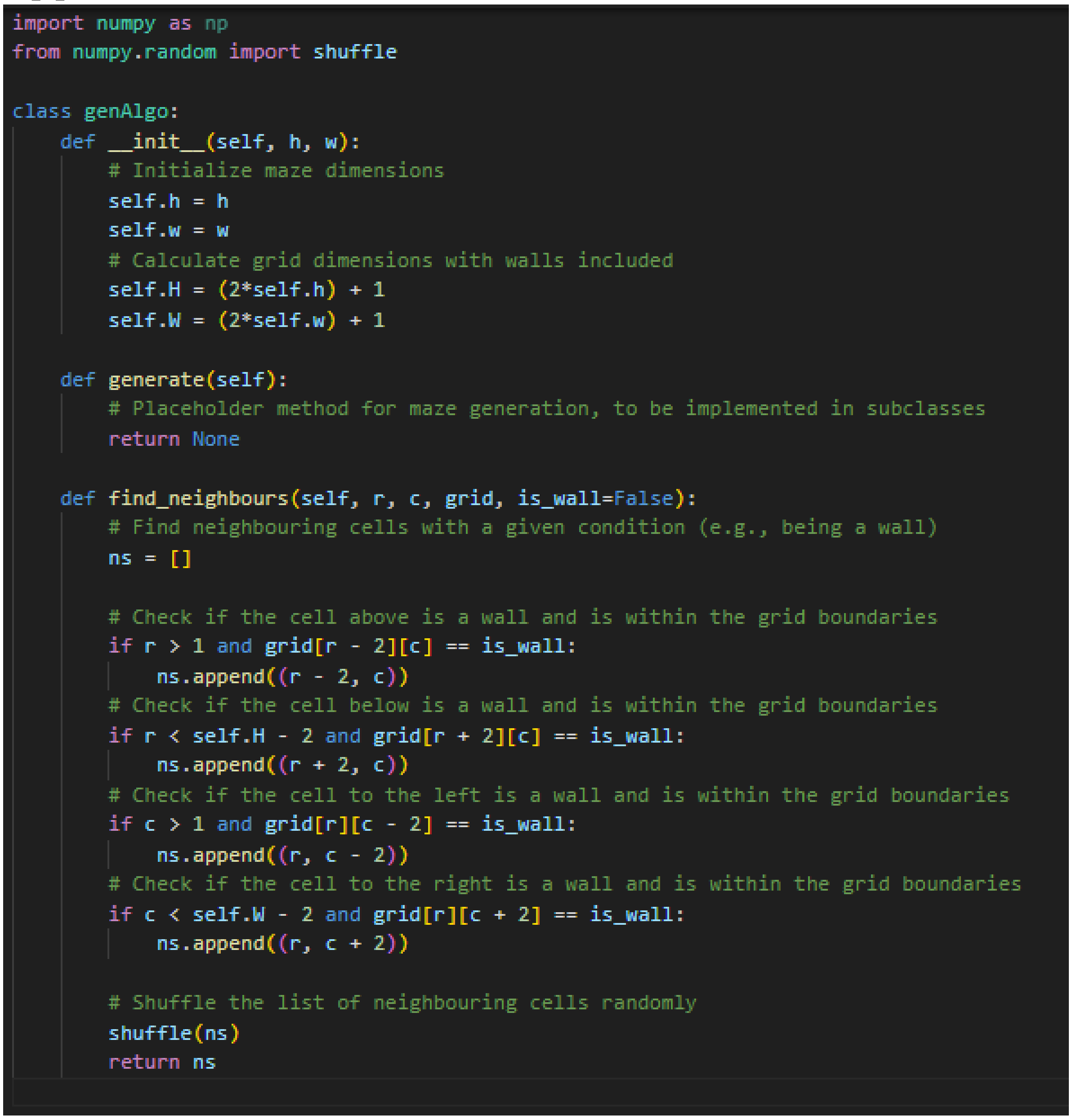
Display Week: 1

TASK	ASSIGNED TO	PROGRESS	START	END
Sprint 1				
menu design	Name	0%	15/12/23	15/12/23
menu development		0%	16/12/23	18/12/23
menu testing		0%	19/12/23	19/12/23
Sprint 2				
maze generation algorithm design		0%	28/12/23	28/12/23
maze generation algorithm 1 development		0%	29/12/23	5/1/24
maze generation algorithm 2 development		0%	29/12/23	5/1/24
maze generation algorithm 3 development		0%	2/1/24	5/1/24
maze generation algorithm 4 development		0%	2/1/24	5/1/24
maze generation algorithm testing		0%	6/1/24	6/1/24
Sprint 3				
maze solving algorithm design		0%	8/1/24	8/1/24
maze solving algorithm 1 development		0%	9/1/24	11/1/24
maze solving algorithm 2 development		0%	9/1/24	11/1/24
maze solving algorithm 3 development		0%	13/1/24	15/1/24
maze solving algorithm 4 development		0%	13/1/24	15/1/24
maze solving algorithm testing		0%	16/1/24	16/1/24
sprint 4				
evalutation features design		0%	18/1/24	18/1/24
evalutation features development		0%	19/1/24	24/1/24
evalutation features testing		0%	25/1/24	25/1/24

Insert new rows ABOVE this one

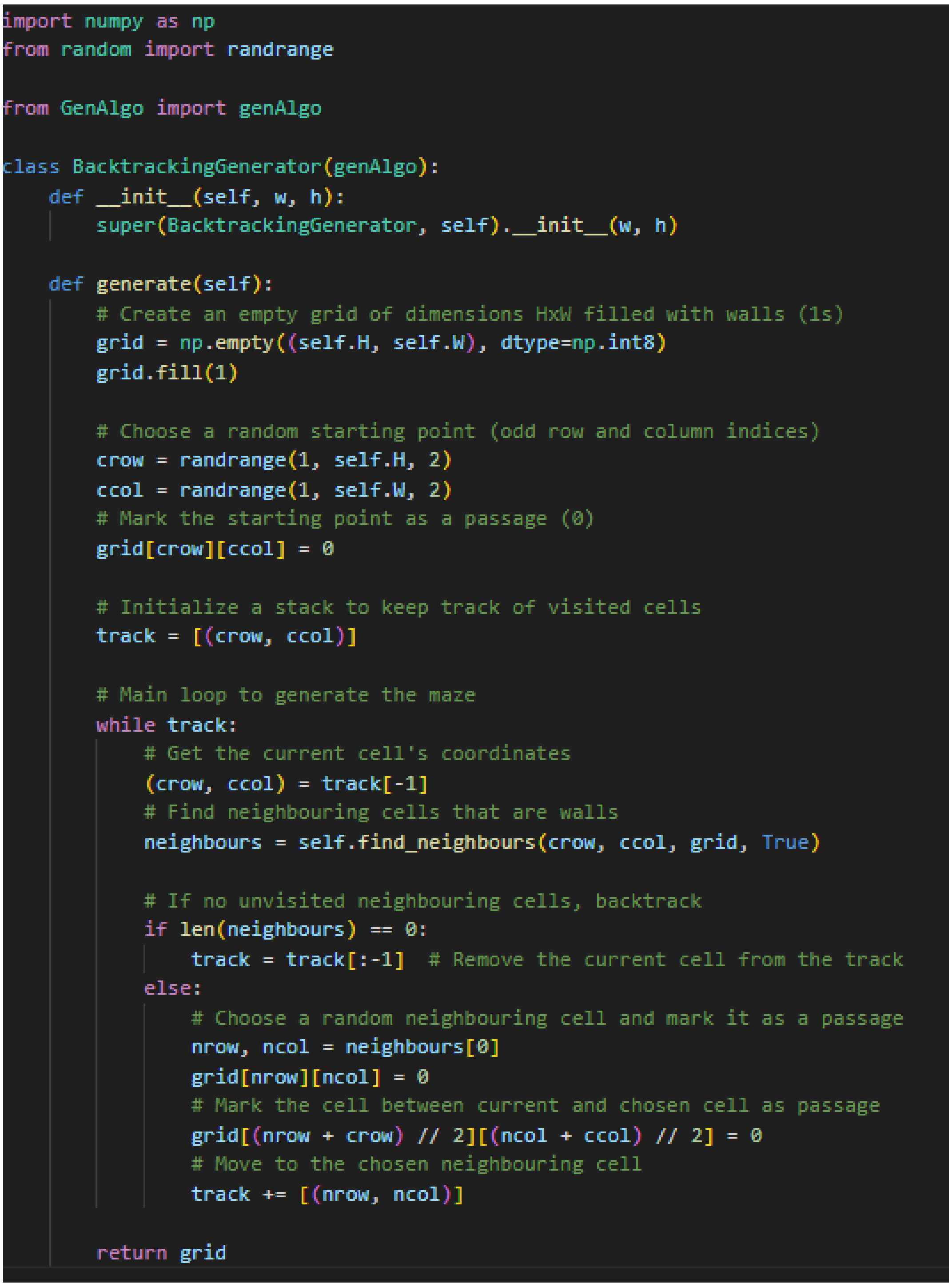
Figure 70 Gantt chart for artefact creation.

Appendix B – Full Code



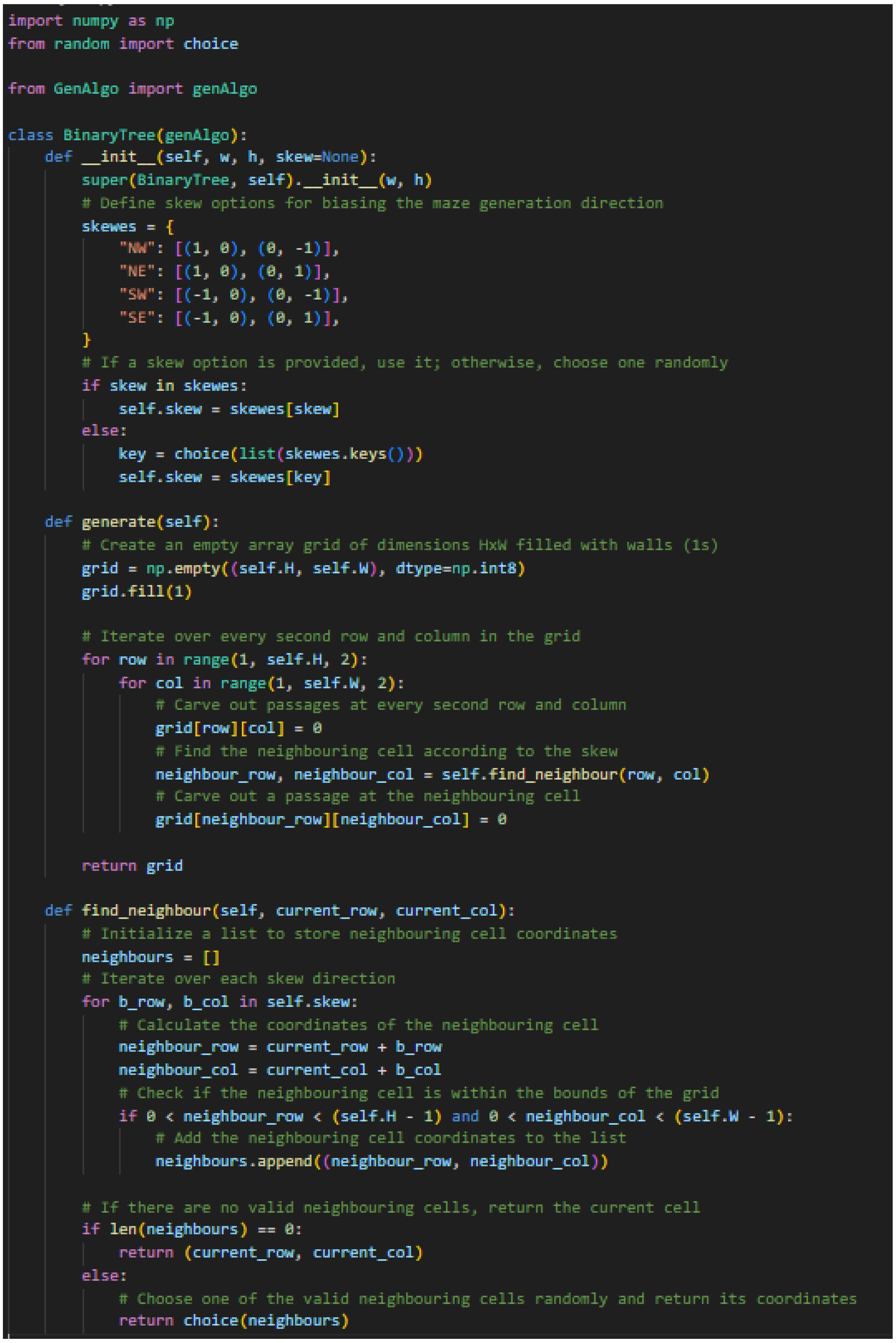
```
import numpy as np
from numpy.random import shuffle

class genAlgo:
    def __init__(self, h, w):
        # Initialize maze dimensions
        self.h = h
        self.w = w
        # Calculate grid dimensions with walls included
        self.H = (2*self.h) + 1
        self.W = (2*self.w) + 1

    def generate(self):
        # Placeholder method for maze generation, to be implemented in subclasses
        return None

    def find_neighbours(self, r, c, grid, is_wall=False):
        # Find neighbouring cells with a given condition (e.g., being a wall)
        ns = []

        # Check if the cell above is a wall and is within the grid boundaries
        if r > 1 and grid[r - 2][c] == is_wall:
            ns.append((r - 2, c))
        # Check if the cell below is a wall and is within the grid boundaries
        if r < self.H - 2 and grid[r + 2][c] == is_wall:
            ns.append((r + 2, c))
        # Check if the cell to the left is a wall and is within the grid boundaries
        if c > 1 and grid[r][c - 2] == is_wall:
            ns.append((r, c - 2))
        # Check if the cell to the right is a wall and is within the grid boundaries
        if c < self.W - 2 and grid[r][c + 2] == is_wall:
            ns.append((r, c + 2))

        # Shuffle the list of neighbouring cells randomly
        shuffle(ns)
        return ns
```

Figure 71 genAlgo full code.



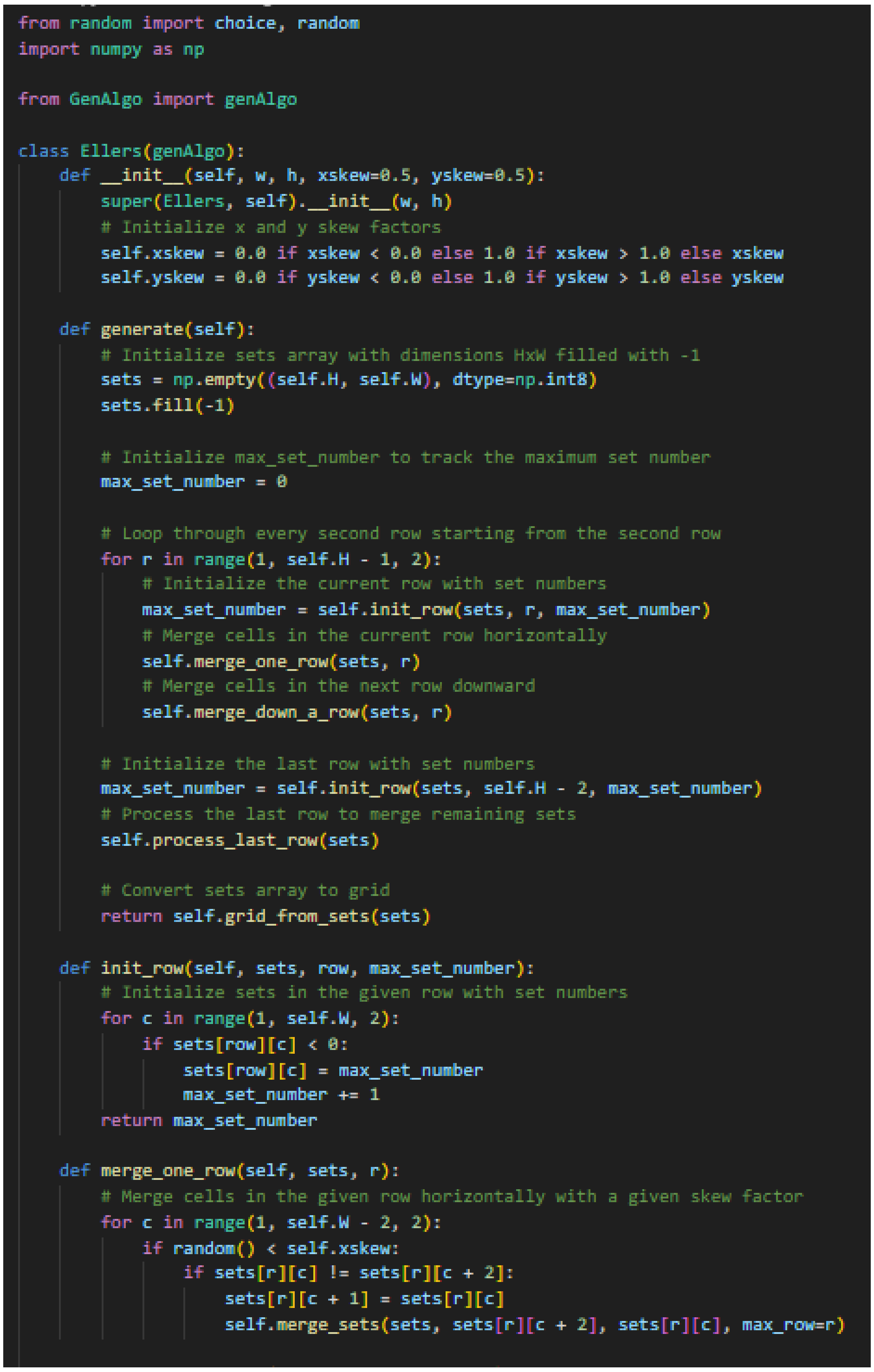
```python
import numpy as np
from random import randrange

from GenAlgo import genAlgo

class BacktrackingGenerator(genAlgo):
    def __init__(self, w, h):
        super(BacktrackingGenerator, self).__init__(w, h)

    def generate(self):
        # Create an empty grid of dimensions HxW filled with walls (1s)
        grid = np.empty((self.H, self.W), dtype=np.int8)
        grid.fill(1)

        # Choose a random starting point (odd row and column indices)
        crow = randrange(1, self.H, 2)
        ccol = randrange(1, self.W, 2)
        # Mark the starting point as a passage (0)
        grid[crow][ccol] = 0

        # Initialize a stack to keep track of visited cells
        track = [(crow, ccol)]

        # Main loop to generate the maze
        while track:
            # Get the current cell's coordinates
            (crow, ccol) = track[-1]
            # Find neighbouring cells that are walls
            neighbours = self.find_neighbours(crow, ccol, grid, True)

            # If no unvisited neighbouring cells, backtrack
            if len(neighbours) == 0:
                track = track[:-1]  # Remove the current cell from the track
            else:
                # Choose a random neighbouring cell and mark it as a passage
                nrow, ncol = neighbours[0]
                grid[nrow][ncol] = 0
                # Mark the cell between current and chosen cell as passage
                grid[(nrow + crow) // 2][(ncol + ccol) // 2] = 0
                # Move to the chosen neighbouring cell
                track += [(nrow, ncol)]

        return grid
```

Figure 72 Backtracking generator full code.



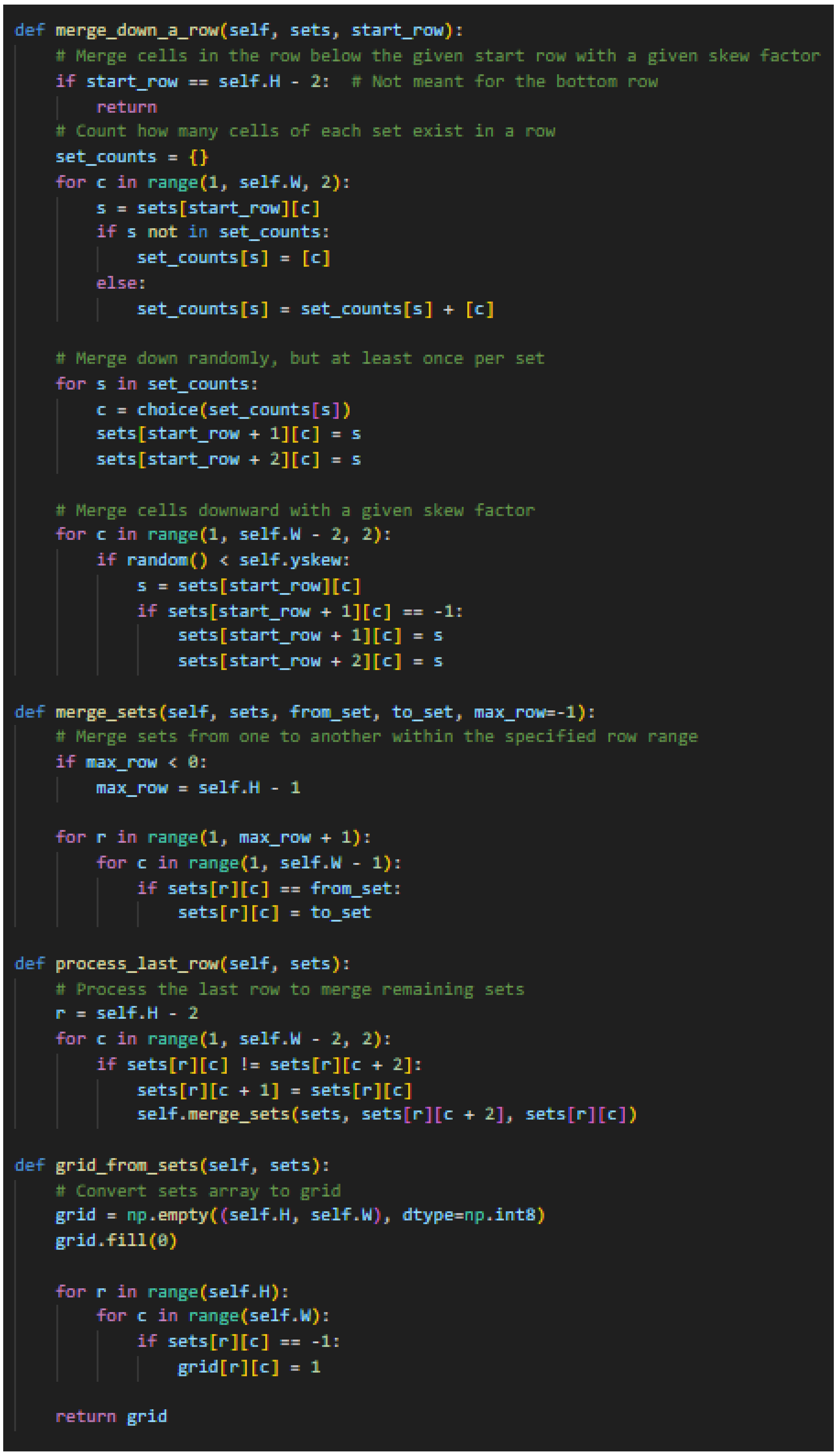
```python
import numpy as np
from random import choice

from GenAlgo import genAlgo

class BinaryTree(genAlgo):
    def __init__(self, w, h, skew=None):
        super(BinaryTree, self).__init__(w, h)
        # Define skew options for biasing the maze generation direction
        skewes = {
            "NW": [(1, 0), (0, -1)],
            "NE": [(1, 0), (0, 1)],
            "SW": [(-1, 0), (0, -1)],
            "SE": [(-1, 0), (0, 1)],
        }
        # If a skew option is provided, use it; otherwise, choose one randomly
        if skew in skewes:
            self.skew = skewes[skew]
        else:
            key = choice(list(skewes.keys()))
            self.skew = skewes[key]

    def generate(self):
        # Create an empty array grid of dimensions HxW filled with walls (1s)
        grid = np.empty((self.H, self.W), dtype=np.int8)
        grid.fill(1)

        # Iterate over every second row and column in the grid
        for row in range(1, self.H, 2):
            for col in range(1, self.W, 2):
                # Carve out passages at every second row and column
                grid[row][col] = 0
                # Find the neighbouring cell according to the skew
                neighbour_row, neighbour_col = self.find_neighbour(row, col)
                # Carve out a passage at the neighbouring cell
                grid[neighbour_row][neighbour_col] = 0

        return grid

    def find_neighbour(self, current_row, current_col):
        # Initialize a list to store neighbouring cell coordinates
        neighbours = []
        # Iterate over each skew direction
        for b_row, b_col in self.skew:
            # Calculate the coordinates of the neighbouring cell
            neighbour_row = current_row + b_row
            neighbour_col = current_col + b_col
            # Check if the neighbouring cell is within the bounds of the grid
            if 0 < neighbour_row < (self.H - 1) and 0 < neighbour_col < (self.W - 1):
                # Add the neighbouring cell coordinates to the list
                neighbours.append((neighbour_row, neighbour_col))

        # If there are no valid neighbouring cells, return the current cell
        if len(neighbours) == 0:
            return (current_row, current_col)
        else:
            # Choose one of the valid neighbouring cells randomly and return its coordinates
            return choice(neighbours)
```

Figure 73 Binary Tree algorithm full code.



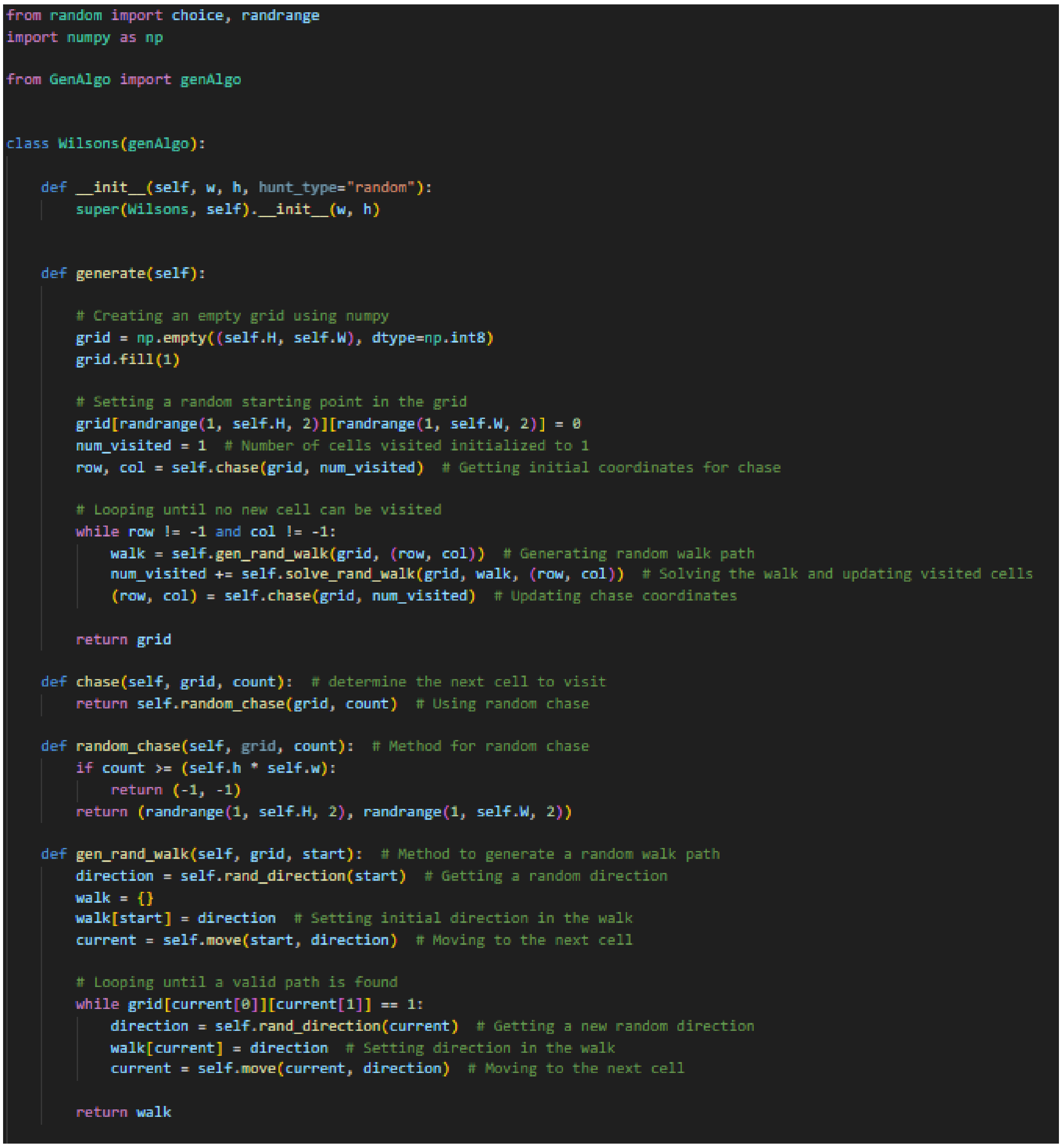
```python
from random import choice, random
import numpy as np

from GenAlgo import genAlgo

class Ellers(genAlgo):
    def __init__(self, w, h, xskew=0.5, yskew=0.5):
        super(Ellers, self).__init__(w, h)
        # Initialize x and y skew factors
        self.xskew = 0.0 if xskew < 0.0 else 1.0 if xskew > 1.0 else xskew
        self.yskew = 0.0 if yskew < 0.0 else 1.0 if yskew > 1.0 else yskew

    def generate(self):
        # Initialize sets array with dimensions HxW filled with -1
        sets = np.empty((self.H, self.W), dtype=np.int8)
        sets.fill(-1)

        # Initialize max_set_number to track the maximum set number
        max_set_number = 0

        # Loop through every second row starting from the second row
        for r in range(1, self.H - 1, 2):
            # Initialize the current row with set numbers
            max_set_number = self.init_row(sets, r, max_set_number)
            # Merge cells in the current row horizontally
            self.merge_one_row(sets, r)
            # Merge cells in the next row downward
            self.merge_down_a_row(sets, r)

        # Initialize the last row with set numbers
        max_set_number = self.init_row(sets, self.H - 2, max_set_number)
        # Process the last row to merge remaining sets
        self.process_last_row(sets)

        # Convert sets array to grid
        return self.grid_from_sets(sets)

    def init_row(self, sets, row, max_set_number):
        # Initialize sets in the given row with set numbers
        for c in range(1, self.W, 2):
            if sets[row][c] < 0:
                sets[row][c] = max_set_number
                max_set_number += 1
        return max_set_number

    def merge_one_row(self, sets, r):
        # Merge cells in the given row horizontally with a given skew factor
        for c in range(1, self.W - 2, 2):
            if random() < self.xskew:
                if sets[r][c] != sets[r][c + 2]:
                    sets[r][c + 1] = sets[r][c]
                    self.merge_sets(sets, sets[r][c + 2], sets[r][c], max_row=r)
```

Figure 74 Ellers algorithm full code 1.



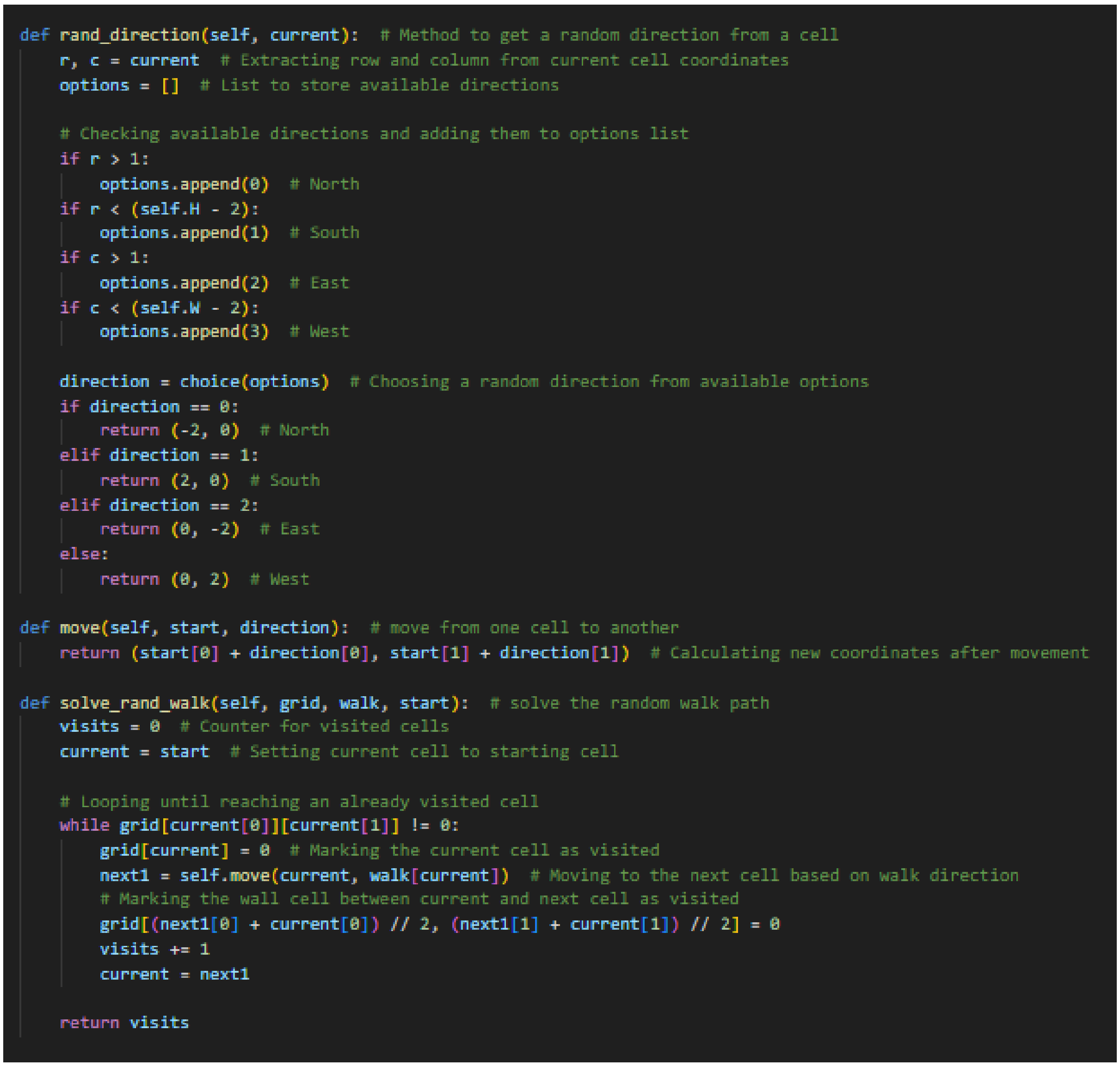
```python
def merge_down_a_row(self, sets, start_row):
    # Merge cells in the row below the given start row with a given skew factor
    if start_row == self.H - 2:  # Not meant for the bottom row
        return
    # Count how many cells of each set exist in a row
    set_counts = {}
    for c in range(1, self.W, 2):
        s = sets[start_row][c]
        if s not in set_counts:
            set_counts[s] = [c]
        else:
            set_counts[s] = set_counts[s] + [c]

    # Merge down randomly, but at least once per set
    for s in set_counts:
        c = choice(set_counts[s])
        sets[start_row + 1][c] = s
        sets[start_row + 2][c] = s

    # Merge cells downward with a given skew factor
    for c in range(1, self.W - 2, 2):
        if random() < self.yskew:
            s = sets[start_row][c]
            if sets[start_row + 1][c] == -1:
                sets[start_row + 1][c] = s
                sets[start_row + 2][c] = s

def merge_sets(self, sets, from_set, to_set, max_row=-1):
    # Merge sets from one to another within the specified row range
    if max_row < 0:
        max_row = self.H - 1

    for r in range(1, max_row + 1):
        for c in range(1, self.W - 1):
            if sets[r][c] == from_set:
                sets[r][c] = to_set

def process_last_row(self, sets):
    # Process the last row to merge remaining sets
    r = self.H - 2
    for c in range(1, self.W - 2, 2):
        if sets[r][c] != sets[r][c + 2]:
            sets[r][c + 1] = sets[r][c]
            self.merge_sets(sets, sets[r][c + 2], sets[r][c])

def grid_from_sets(self, sets):
    # Convert sets array to grid
    grid = np.empty((self.H, self.W), dtype=np.int8)
    grid.fill(0)

    for r in range(self.H):
        for c in range(self.W):
            if sets[r][c] == -1:
                grid[r][c] = 1

    return grid
```

Figure 75 Ellers algorithm full code 2.



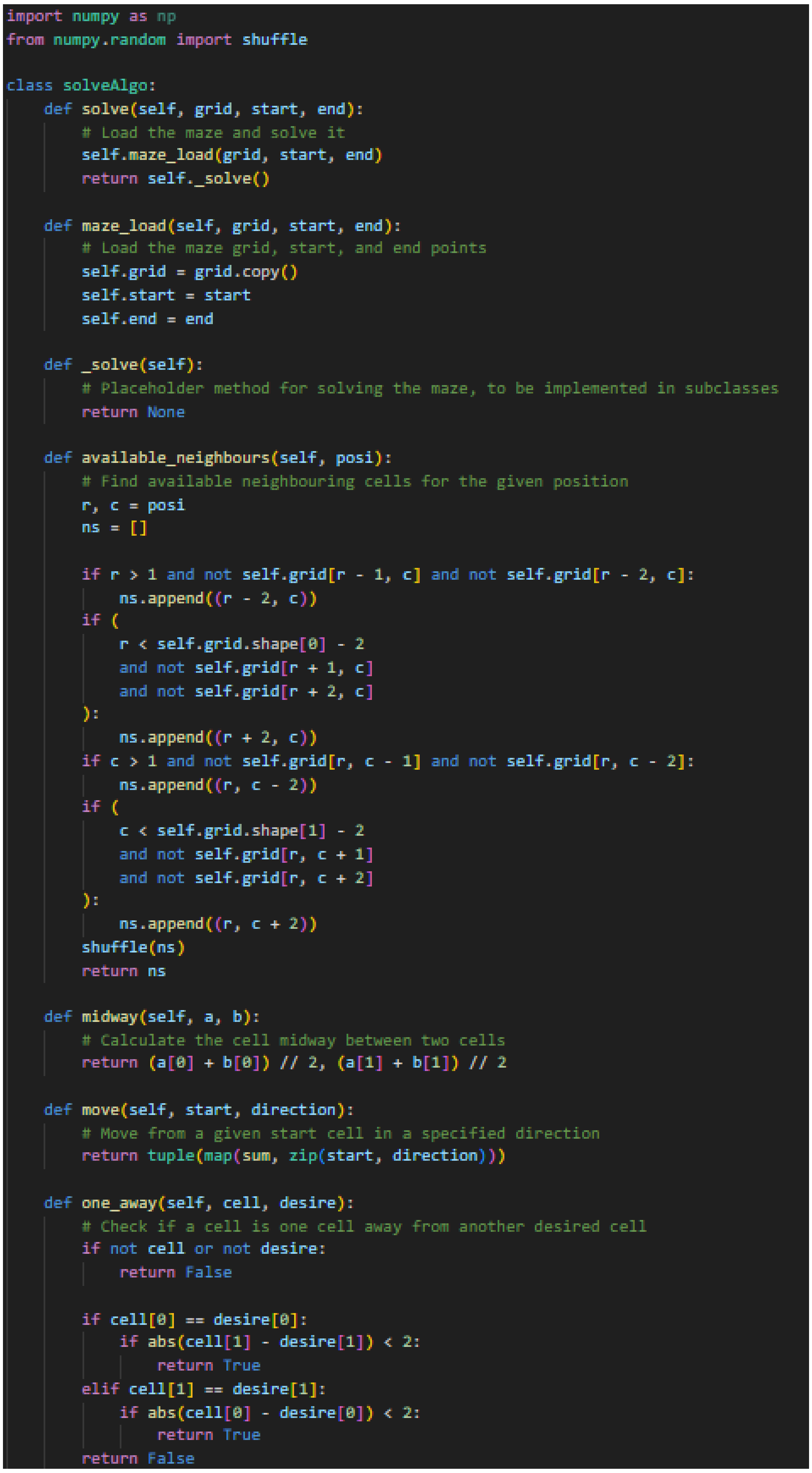
```python
from random import choice, randrange
import numpy as np

from GenAlgo import genAlgo


class Wilsons(genAlgo):

    def __init__(self, w, h, hunt_type="random"):
        super(Wilsons, self).__init__(w, h)


    def generate(self):

        # Creating an empty grid using numpy
        grid = np.empty((self.H, self.W), dtype=np.int8)
        grid.fill(1)

        # Setting a random starting point in the grid
        grid[randrange(1, self.H, 2)][randrange(1, self.W, 2)] = 0
        num_visited = 1  # Number of cells visited initialized to 1
        row, col = self.chase(grid, num_visited)  # Getting initial coordinates for chase

        # Looping until no new cell can be visited
        while row != -1 and col != -1:
            walk = self.gen_rand_walk(grid, (row, col))  # Generating random walk path
            num_visited += self.solve_rand_walk(grid, walk, (row, col))  # Solving the walk and updating visited cells
            (row, col) = self.chase(grid, num_visited)  # Updating chase coordinates

        return grid

    def chase(self, grid, count):  # determine the next cell to visit
        return self.random_chase(grid, count)  # Using random chase

    def random_chase(self, grid, count):  # Method for random chase
        if count >= (self.h * self.w):
            return (-1, -1)
        return (randrange(1, self.H, 2), randrange(1, self.W, 2))

    def gen_rand_walk(self, grid, start):  # Method to generate a random walk path
        direction = self.rand_direction(start)  # Getting a random direction
        walk = {}
        walk[start] = direction  # Setting initial direction in the walk
        current = self.move(start, direction)  # Moving to the next cell

        # Looping until a valid path is found
        while grid[current[0]][current[1]] == 1:
            direction = self.rand_direction(current)  # Getting a new random direction
            walk[current] = direction  # Setting direction in the walk
            current = self.move(current, direction)  # Moving to the next cell

        return walk
```

Figure 76 Wilsons algorithm full code 1.



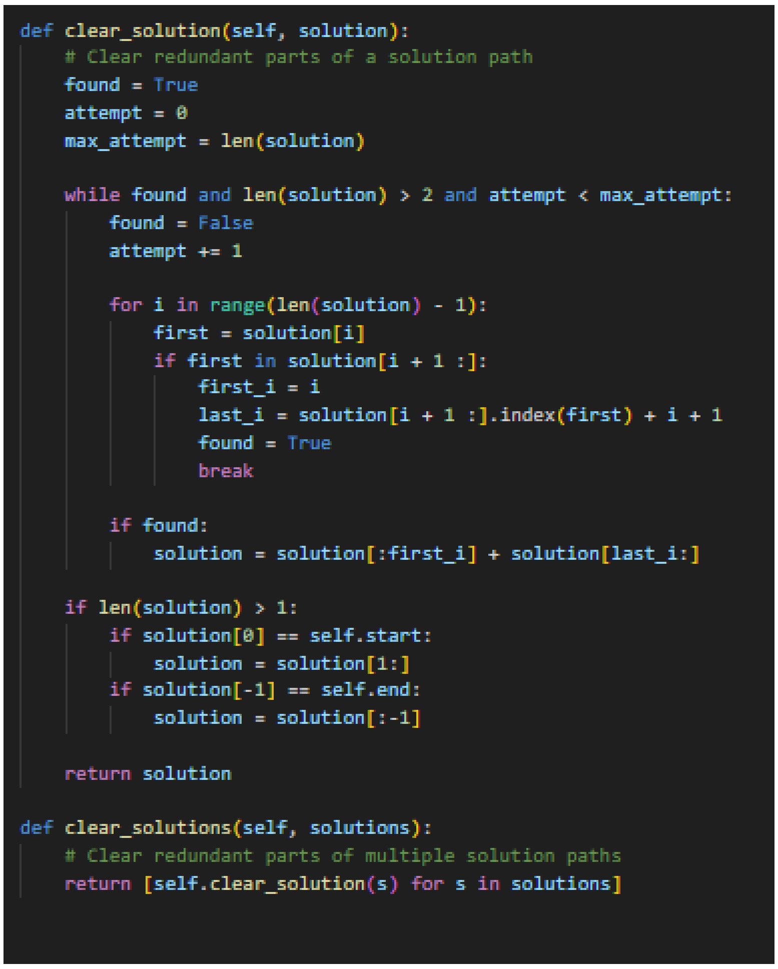
```python
def rand_direction(self, current):  # Method to get a random direction from a cell
    r, c = current  # Extracting row and column from current cell coordinates
    options = []  # List to store available directions

    # Checking available directions and adding them to options list
    if r > 1:
        options.append(0)  # North
    if r < (self.H - 2):
        options.append(1)  # South
    if c > 1:
        options.append(2)  # East
    if c < (self.W - 2):
        options.append(3)  # West

    direction = choice(options)  # Choosing a random direction from available options
    if direction == 0:
        return (-2, 0)  # North
    elif direction == 1:
        return (2, 0)  # South
    elif direction == 2:
        return (0, -2)  # East
    else:
        return (0, 2)  # West

def move(self, start, direction):  # move from one cell to another
    return (start[0] + direction[0], start[1] + direction[1])  # Calculating new coordinates after movement

def solve_rand_walk(self, grid, walk, start):  # solve the random walk path
    visits = 0  # Counter for visited cells
    current = start  # Setting current cell to starting cell

    # Looping until reaching an already visited cell
    while grid[current[0]][current[1]] != 0:
        grid[current] = 0  # Marking the current cell as visited
        next1 = self.move(current, walk[current])  # Moving to the next cell based on walk direction
        # Marking the wall cell between current and next cell as visited
        grid[(next1[0] + current[0]) // 2, (next1[1] + current[1]) // 2] = 0
        visits += 1
        current = next1

    return visits
```

Figure 77 Wilsons algorithm full code 2.



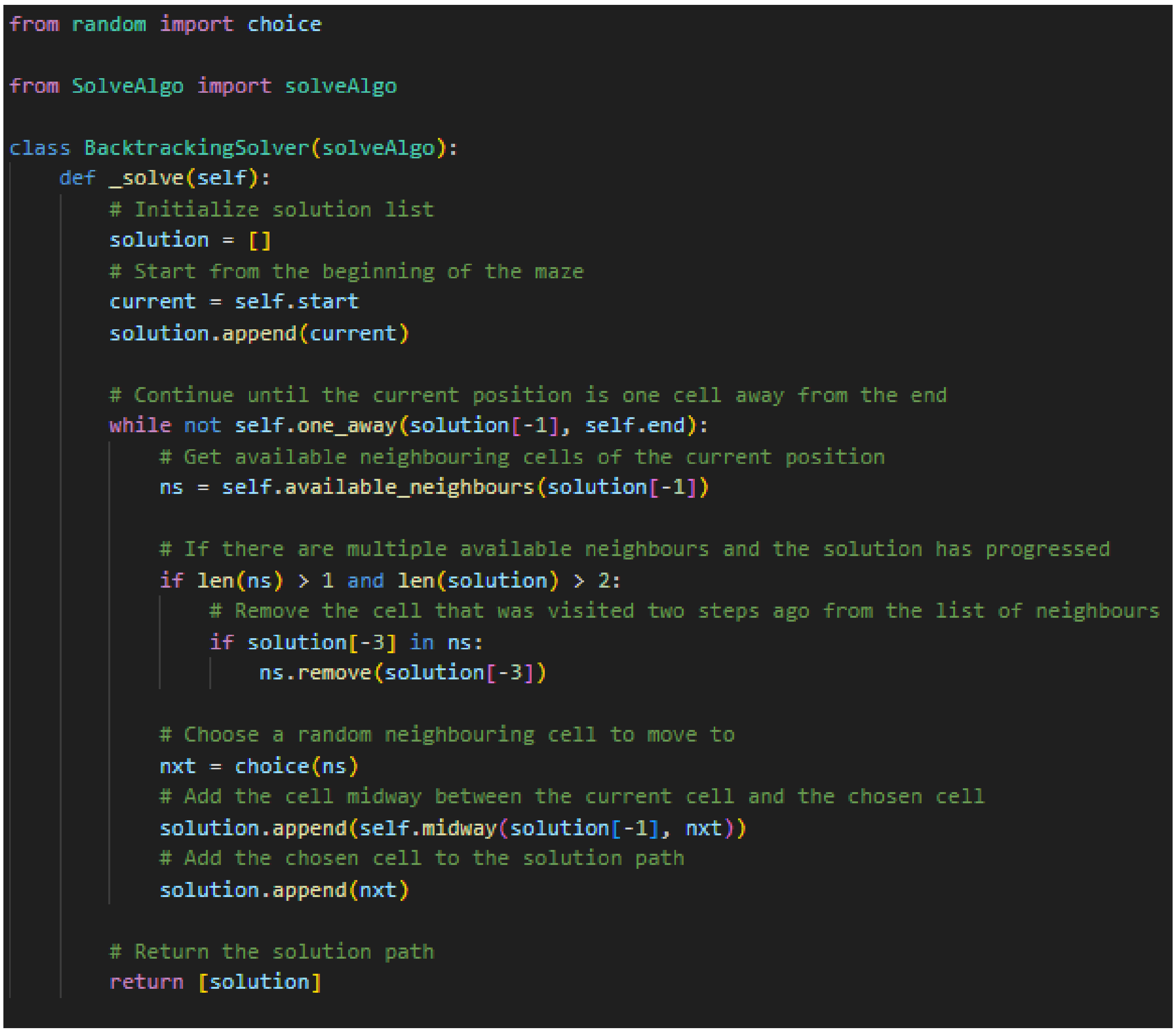
```python
import numpy as np
from numpy.random import shuffle

class solveAlgo:
    def solve(self, grid, start, end):
        # Load the maze and solve it
        self.maze_load(grid, start, end)
        return self._solve()

    def maze_load(self, grid, start, end):
        # Load the maze grid, start, and end points
        self.grid = grid.copy()
        self.start = start
        self.end = end

    def _solve(self):
        # Placeholder method for solving the maze, to be implemented in subclasses
        return None

    def available_neighbours(self, posi):
        # Find available neighbouring cells for the given position
        r, c = posi
        ns = []

        if r > 1 and not self.grid[r - 1, c] and not self.grid[r - 2, c]:
            ns.append((r - 2, c))
        if (
            r < self.grid.shape[0] - 2
            and not self.grid[r + 1, c]
            and not self.grid[r + 2, c]
        ):
            ns.append((r + 2, c))
        if c > 1 and not self.grid[r, c - 1] and not self.grid[r, c - 2]:
            ns.append((r, c - 2))
        if (
            c < self.grid.shape[1] - 2
            and not self.grid[r, c + 1]
            and not self.grid[r, c + 2]
        ):
            ns.append((r, c + 2))
        shuffle(ns)
        return ns

    def midway(self, a, b):
        # Calculate the cell midway between two cells
        return (a[0] + b[0]) // 2, (a[1] + b[1]) // 2

    def move(self, start, direction):
        # Move from a given start cell in a specified direction
        return tuple(map(sum, zip(start, direction)))

    def one_away(self, cell, desire):
        # Check if a cell is one cell away from another desired cell
        if not cell or not desire:
            return False

        if cell[0] == desire[0]:
            if abs(cell[1] - desire[1]) < 2:
                return True
        elif cell[1] == desire[1]:
            if abs(cell[0] - desire[0]) < 2:
                return True
        return False
```

Figure 78 Solvealgo full code 1.



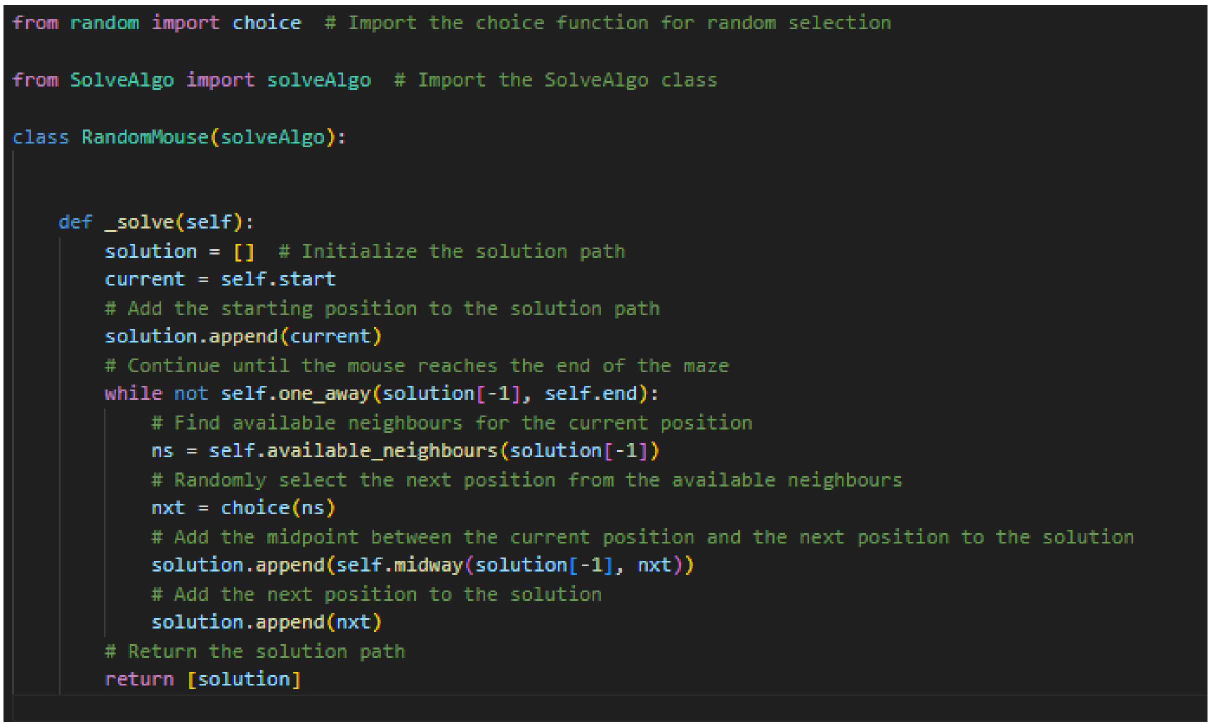
```python
def clear_solution(self, solution):
    # Clear redundant parts of a solution path
    found = True
    attempt = 0
    max_attempt = len(solution)

    while found and len(solution) > 2 and attempt < max_attempt:
        found = False
        attempt += 1

        for i in range(len(solution) - 1):
            first = solution[i]
            if first in solution[i + 1 :]:
                first_i = i
                last_i = solution[i + 1 :].index(first) + i + 1
                found = True
                break

        if found:
            solution = solution[:first_i] + solution[last_i:]

    if len(solution) > 1:
        if solution[0] == self.start:
            solution = solution[1:]
        if solution[-1] == self.end:
            solution = solution[:-1]

    return solution

def clear_solutions(self, solutions):
    # Clear redundant parts of multiple solution paths
    return [self.clear_solution(s) for s in solutions]
```

Figure 79 Solvealgo full code 2.



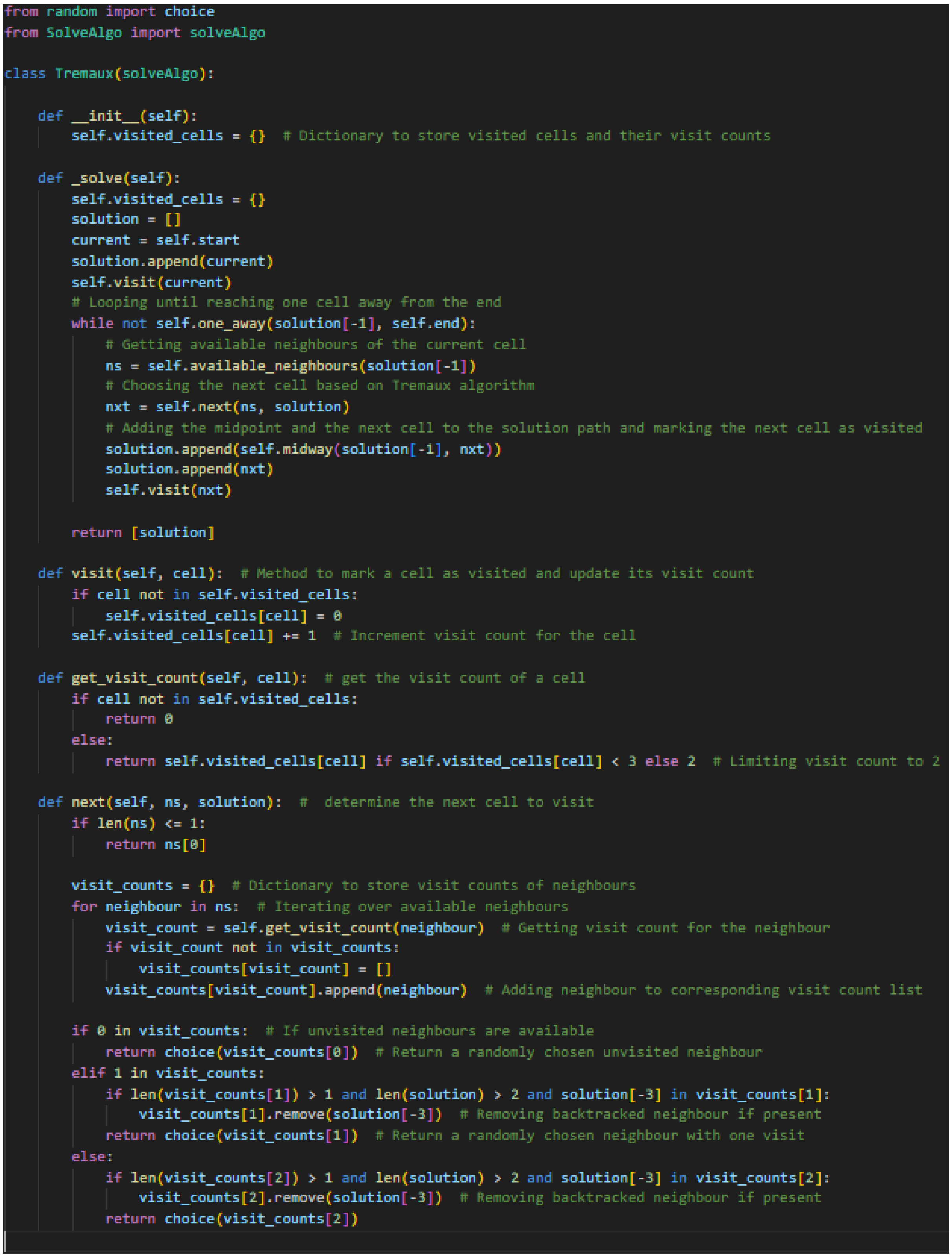
```python
from random import choice

from SolveAlgo import solveAlgo

class BacktrackingSolver(solveAlgo):
    def _solve(self):
        # Initialize solution list
        solution = []
        # Start from the beginning of the maze
        current = self.start
        solution.append(current)

        # Continue until the current position is one cell away from the end
        while not self.one_away(solution[-1], self.end):
            # Get available neighbouring cells of the current position
            ns = self.available_neighbours(solution[-1])

            # If there are multiple available neighbours and the solution has progressed
            if len(ns) > 1 and len(solution) > 2:
                # Remove the cell that was visited two steps ago from the list of neighbours
                if solution[-3] in ns:
                    ns.remove(solution[-3])

            # Choose a random neighbouring cell to move to
            nxt = choice(ns)
            # Add the cell midway between the current cell and the chosen cell
            solution.append(self.midway(solution[-1], nxt))
            # Add the chosen cell to the solution path
            solution.append(nxt)

        # Return the solution path
        return [solution]
```

Figure 80 Backtracking solver algorithm full code.



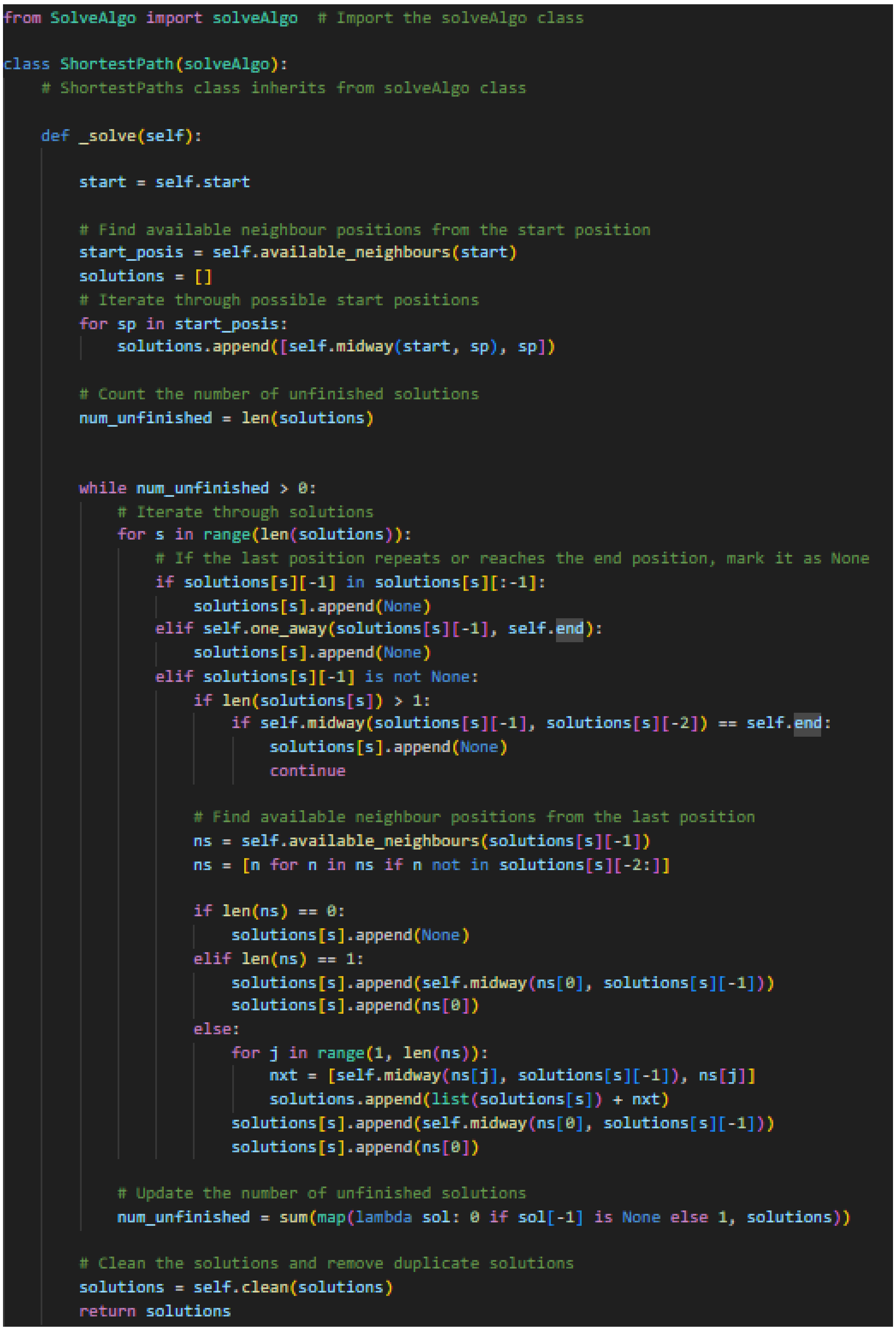
```
from random import choice  # Import the choice function for random selection

from SolveAlgo import solveAlgo  # Import the SolveAlgo class

class RandomMouse(solveAlgo):


    def _solve(self):
        solution = []  # Initialize the solution path
        current = self.start
        # Add the starting position to the solution path
        solution.append(current)
        # Continue until the mouse reaches the end of the maze
        while not self.one_away(solution[-1], self.end):
            # Find available neighbours for the current position
            ns = self.available_neighbours(solution[-1])
            # Randomly select the next position from the available neighbours
            nxt = choice(ns)
            # Add the midpoint between the current position and the next position to the solution
            solution.append(self.midway(solution[-1], nxt))
            # Add the next position to the solution
            solution.append(nxt)
        # Return the solution path
        return [solution]
```

Figure 81 Random mouse algorithm full code.



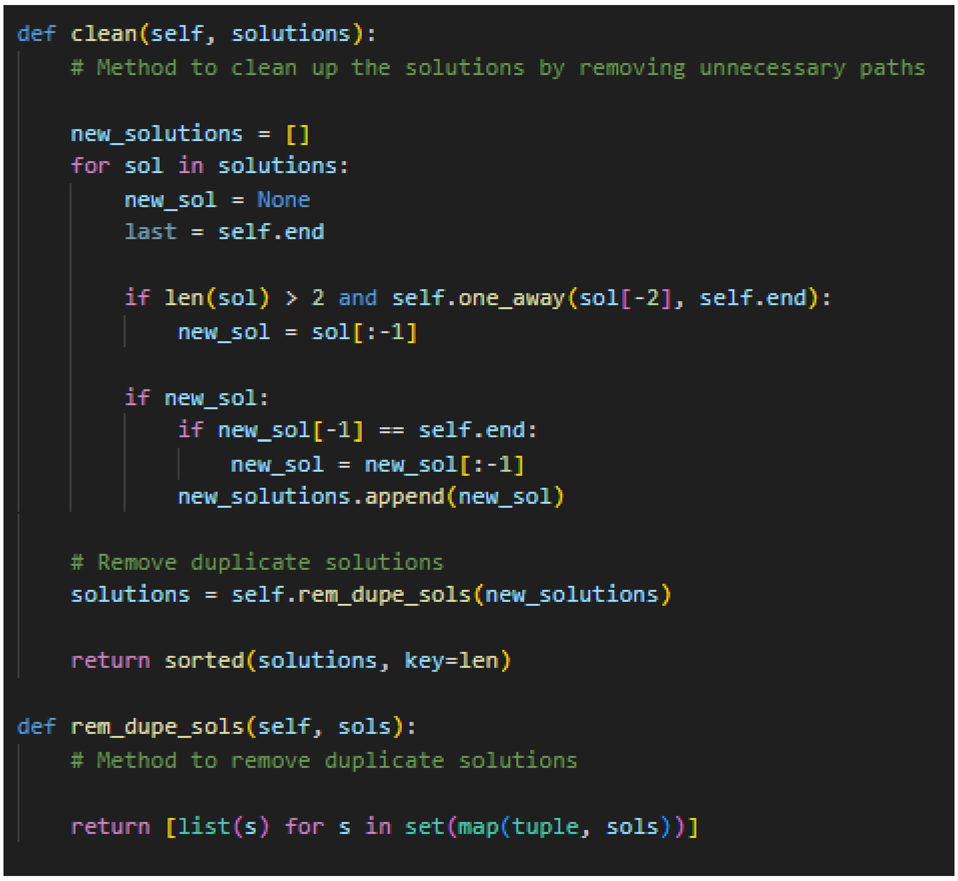
```python
from random import choice
from SolveAlgo import solveAlgo

class Tremaux(solveAlgo):

    def __init__(self):
        self.visited_cells = {}  # Dictionary to store visited cells and their visit counts

    def _solve(self):
        self.visited_cells = {}
        solution = []
        current = self.start
        solution.append(current)
        self.visit(current)
        # Looping until reaching one cell away from the end
        while not self.one_away(solution[-1], self.end):
            # Getting available neighbours of the current cell
            ns = self.available_neighbours(solution[-1])
            # Choosing the next cell based on Tremaux algorithm
            nxt = self.next(ns, solution)
            # Adding the midpoint and the next cell to the solution path and marking the next cell as visited
            solution.append(self.midway(solution[-1], nxt))
            solution.append(nxt)
            self.visit(nxt)

        return [solution]

    def visit(self, cell):  # Method to mark a cell as visited and update its visit count
        if cell not in self.visited_cells:
            self.visited_cells[cell] = 0
        self.visited_cells[cell] += 1  # Increment visit count for the cell

    def get_visit_count(self, cell):  # get the visit count of a cell
        if cell not in self.visited_cells:
            return 0
        else:
            return self.visited_cells[cell] if self.visited_cells[cell] < 3 else 2  # Limiting visit count to 2

    def next(self, ns, solution):  #  determine the next cell to visit
        if len(ns) <= 1:
            return ns[0]

        visit_counts = {}  # Dictionary to store visit counts of neighbours
        for neighbour in ns:  # Iterating over available neighbours
            visit_count = self.get_visit_count(neighbour)  # Getting visit count for the neighbour
            if visit_count not in visit_counts:
                visit_counts[visit_count] = []
            visit_counts[visit_count].append(neighbour)  # Adding neighbour to corresponding visit count list

        if 0 in visit_counts:  # If unvisited neighbours are available
            return choice(visit_counts[0])  # Return a randomly chosen unvisited neighbour
        elif 1 in visit_counts:
            if len(visit_counts[1]) > 1 and len(solution) > 2 and solution[-3] in visit_counts[1]:
                visit_counts[1].remove(solution[-3])  # Removing backtracked neighbour if present
            return choice(visit_counts[1])  # Return a randomly chosen neighbour with one visit
        else:
            if len(visit_counts[2]) > 1 and len(solution) > 2 and solution[-3] in visit_counts[2]:
                visit_counts[2].remove(solution[-3])  # Removing backtracked neighbour if present
            return choice(visit_counts[2])
```

Figure 82 Trémaux algorithm full code.



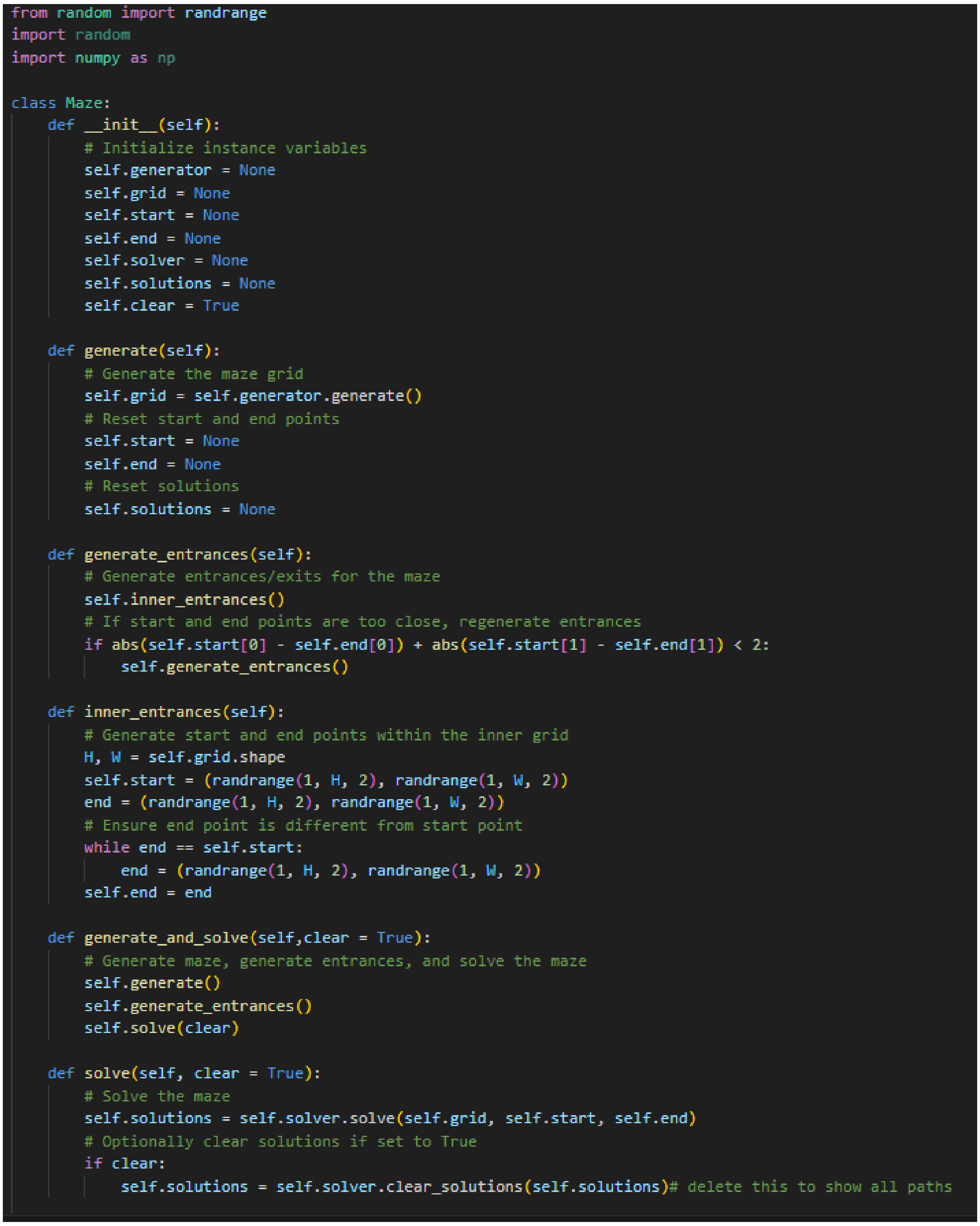
```python
from SolveAlgo import solveAlgo  # Import the solveAlgo class

class ShortestPath(solveAlgo):
    # ShortestPaths class inherits from solveAlgo class

    def _solve(self):

        start = self.start

        # Find available neighbour positions from the start position
        start_posis = self.available_neighbours(start)
        solutions = []
        # Iterate through possible start positions
        for sp in start_posis:
            solutions.append([self.midway(start, sp), sp])

        # Count the number of unfinished solutions
        num_unfinished = len(solutions)


        while num_unfinished > 0:
            # Iterate through solutions
            for s in range(len(solutions)):
                # If the last position repeats or reaches the end position, mark it as None
                if solutions[s][-1] in solutions[s][:-1]:
                    solutions[s].append(None)
                elif self.one_away(solutions[s][-1], self.end):
                    solutions[s].append(None)
                elif solutions[s][-1] is not None:
                    if len(solutions[s]) > 1:
                        if self.midway(solutions[s][-1], solutions[s][-2]) == self.end:
                            solutions[s].append(None)
                            continue

                    # Find available neighbour positions from the last position
                    ns = self.available_neighbours(solutions[s][-1])
                    ns = [n for n in ns if n not in solutions[s][-2:]]

                    if len(ns) == 0:
                        solutions[s].append(None)
                    elif len(ns) == 1:
                        solutions[s].append(self.midway(ns[0], solutions[s][-1]))
                        solutions[s].append(ns[0])
                    else:
                        for j in range(1, len(ns)):
                            nxt = [self.midway(ns[j], solutions[s][-1]), ns[j]]
                            solutions.append(list(solutions[s]) + nxt)
                        solutions[s].append(self.midway(ns[0], solutions[s][-1]))
                        solutions[s].append(ns[0])

            # Update the number of unfinished solutions
            num_unfinished = sum(map(lambda sol: 0 if sol[-1] is None else 1, solutions))

        # Clean the solutions and remove duplicate solutions
        solutions = self.clean(solutions)
        return solutions
```

Figure 83 Shortest path algorithm full code 1.



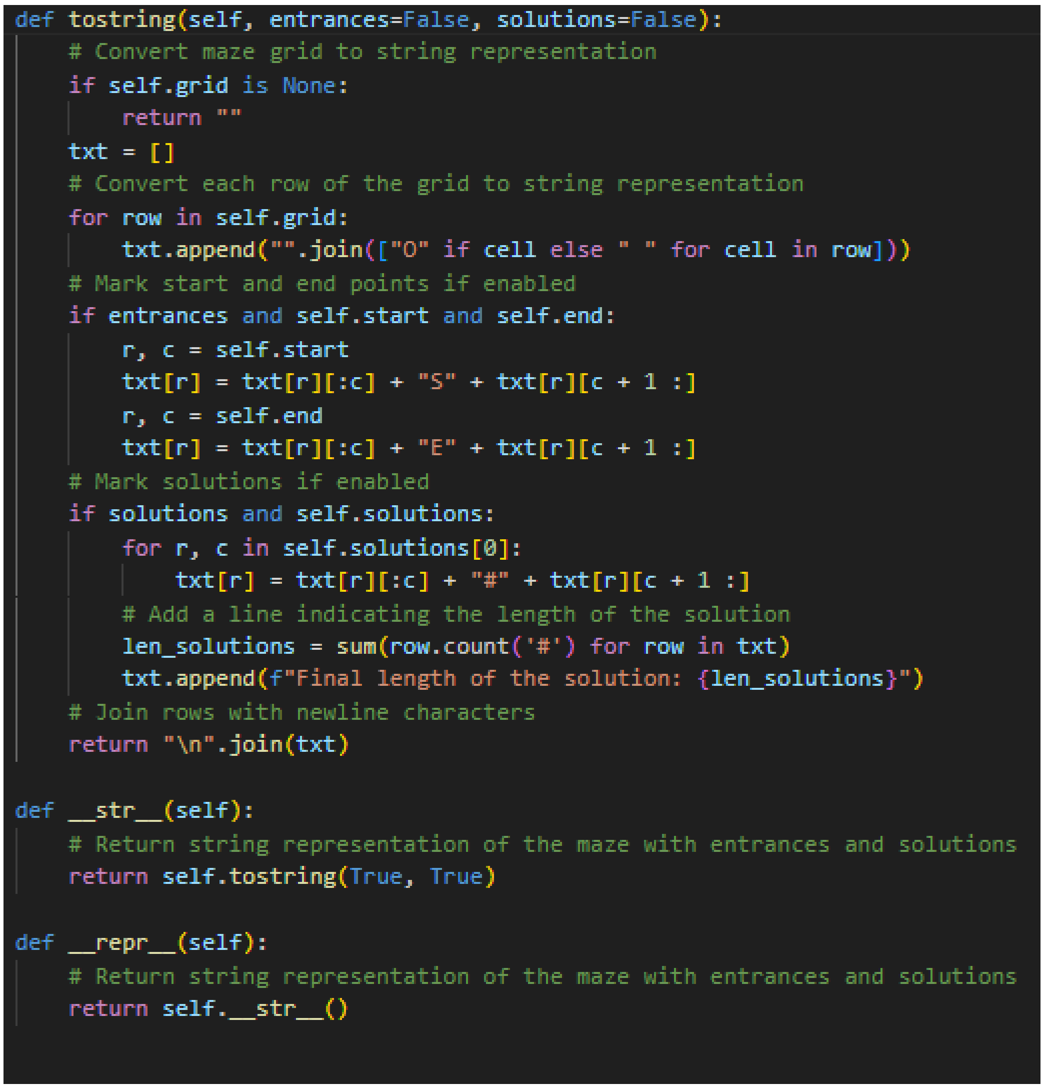
```python
def clean(self, solutions):
    # Method to clean up the solutions by removing unnecessary paths

    new_solutions = []
    for sol in solutions:
        new_sol = None
        last = self.end

        if len(sol) > 2 and self.one_away(sol[-2], self.end):
            new_sol = sol[:-1]

        if new_sol:
            if new_sol[-1] == self.end:
                new_sol = new_sol[:-1]
            new_solutions.append(new_sol)

    # Remove duplicate solutions
    solutions = self.rem_dupe_sols(new_solutions)

    return sorted(solutions, key=len)

def rem_dupe_sols(self, sols):
    # Method to remove duplicate solutions

    return [list(s) for s in set(map(tuple, sols))]
```

Figure 84 Shortest path algorithm full code 2.



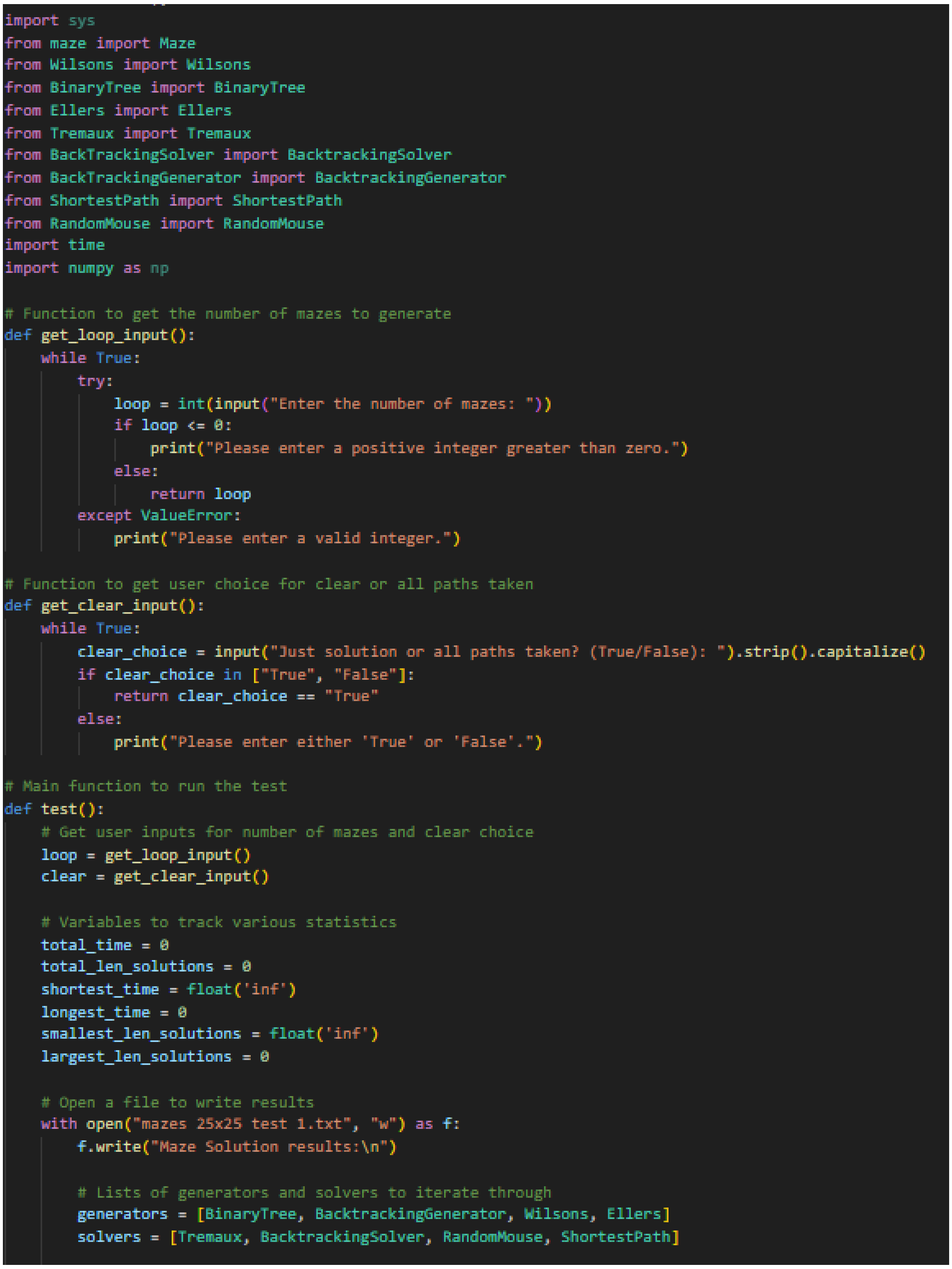
```python
from random import randrange
import random
import numpy as np

class Maze:
    def __init__(self):
        # Initialize instance variables
        self.generator = None
        self.grid = None
        self.start = None
        self.end = None
        self.solver = None
        self.solutions = None
        self.clear = True

    def generate(self):
        # Generate the maze grid
        self.grid = self.generator.generate()
        # Reset start and end points
        self.start = None
        self.end = None
        # Reset solutions
        self.solutions = None

    def generate_entrances(self):
        # Generate entrances/exits for the maze
        self.inner_entrances()
        # If start and end points are too close, regenerate entrances
        if abs(self.start[0] - self.end[0]) + abs(self.start[1] - self.end[1]) < 2:
            self.generate_entrances()

    def inner_entrances(self):
        # Generate start and end points within the inner grid
        H, W = self.grid.shape
        self.start = (randrange(1, H, 2), randrange(1, W, 2))
        end = (randrange(1, H, 2), randrange(1, W, 2))
        # Ensure end point is different from start point
        while end == self.start:
            end = (randrange(1, H, 2), randrange(1, W, 2))
        self.end = end

    def generate_and_solve(self,clear = True):
        # Generate maze, generate entrances, and solve the maze
        self.generate()
        self.generate_entrances()
        self.solve(clear)

    def solve(self, clear = True):
        # Solve the maze
        self.solutions = self.solver.solve(self.grid, self.start, self.end)
        # Optionally clear solutions if set to True
        if clear:
            self.solutions = self.solver.clear_solutions(self.solutions)# delete this to show all paths
```

Figure 85 maze full code 1.



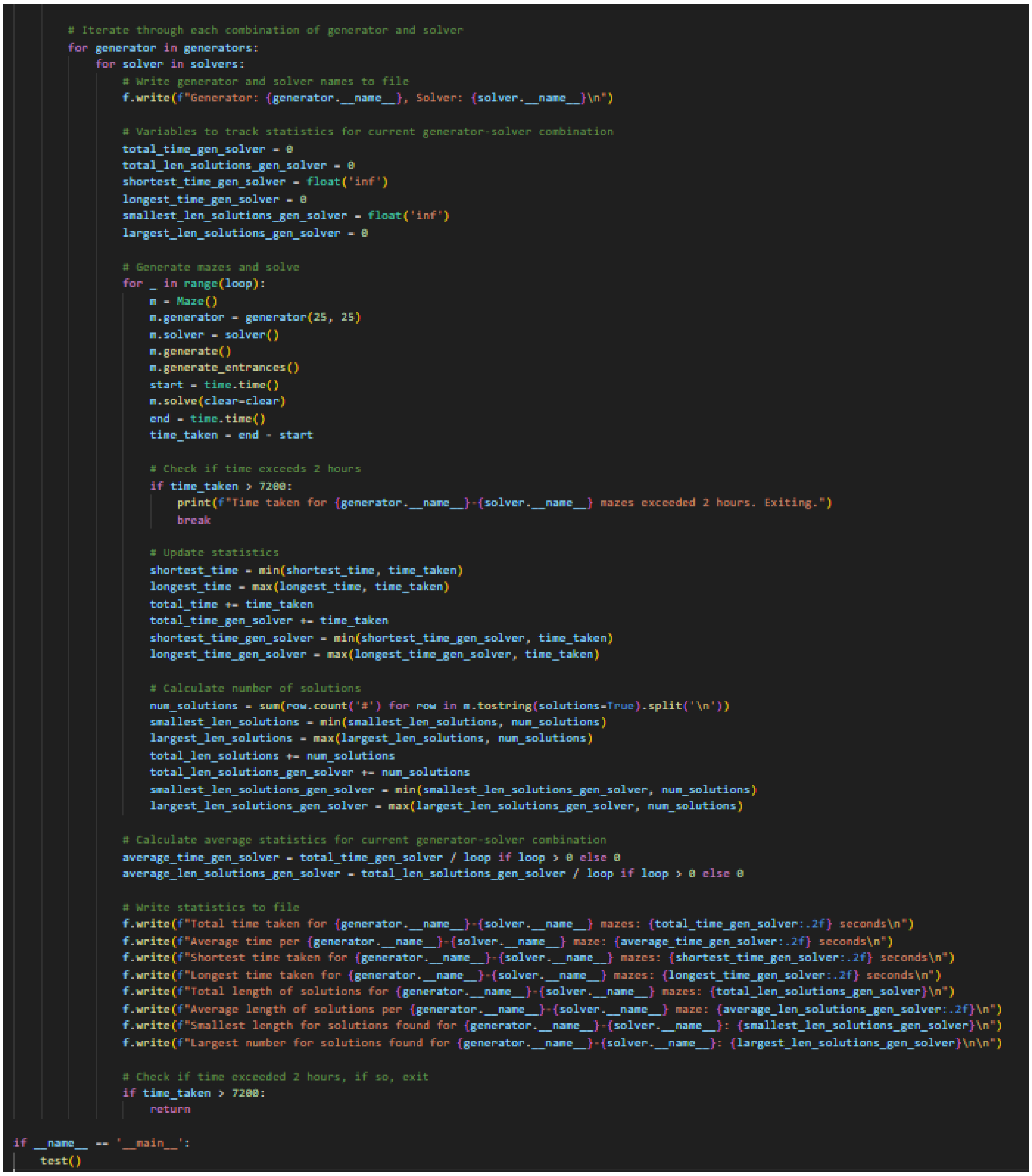
```python
def tostring(self, entrances=False, solutions=False):
    # Convert maze grid to string representation
    if self.grid is None:
        return ""
    txt = []
    # Convert each row of the grid to string representation
    for row in self.grid:
        txt.append("".join(["O" if cell else " " for cell in row]))
    # Mark start and end points if enabled
    if entrances and self.start and self.end:
        r, c = self.start
        txt[r] = txt[r][:c] + "S" + txt[r][c + 1 :]
        r, c = self.end
        txt[r] = txt[r][:c] + "E" + txt[r][c + 1 :]
    # Mark solutions if enabled
    if solutions and self.solutions:
        for r, c in self.solutions[0]:
            txt[r] = txt[r][:c] + "#" + txt[r][c + 1 :]
        # Add a line indicating the length of the solution
        len_solutions = sum(row.count('#') for row in txt)
        txt.append(f"Final length of the solution: {len_solutions}")
    # Join rows with newline characters
    return "\n".join(txt)

def __str__(self):
    # Return string representation of the maze with entrances and solutions
    return self.tostring(True, True)

def __repr__(self):
    # Return string representation of the maze with entrances and solutions
    return self.__str__()
```

Figure 86 Maze full code 2.



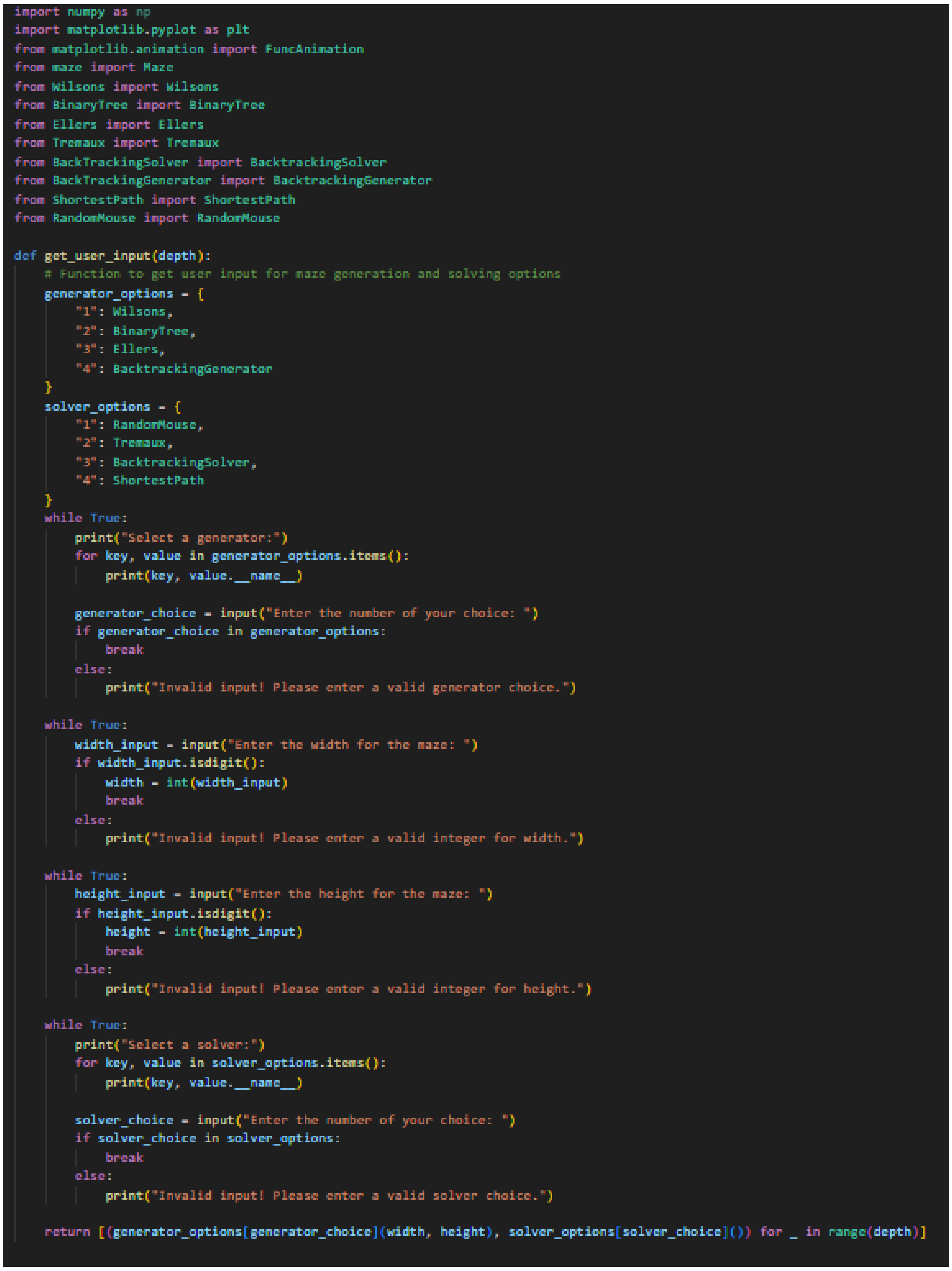
```python
import sys
from maze import Maze
from Wilsons import Wilsons
from BinaryTree import BinaryTree
from Ellers import Ellers
from Tremaux import Tremaux
from BackTrackingSolver import BacktrackingSolver
from BackTrackingGenerator import BacktrackingGenerator
from ShortestPath import ShortestPath
from RandomMouse import RandomMouse
import time
import numpy as np

# Function to get the number of mazes to generate
def get_loop_input():
    while True:
        try:
            loop = int(input("Enter the number of mazes: "))
            if loop <= 0:
                print("Please enter a positive integer greater than zero.")
            else:
                return loop
        except ValueError:
            print("Please enter a valid integer.")

# Function to get user choice for clear or all paths taken
def get_clear_input():
    while True:
        clear_choice = input("Just solution or all paths taken? (True/False): ").strip().capitalize()
        if clear_choice in ["True", "False"]:
            return clear_choice == "True"
        else:
            print("Please enter either 'True' or 'False'.")

# Main function to run the test
def test():
    # Get user inputs for number of mazes and clear choice
    loop = get_loop_input()
    clear = get_clear_input()

    # Variables to track various statistics
    total_time = 0
    total_len_solutions = 0
    shortest_time = float('inf')
    longest_time = 0
    smallest_len_solutions = float('inf')
    largest_len_solutions = 0

    # Open a file to write results
    with open("mazes 25x25 test 1.txt", "w") as f:
        f.write("Maze Solution results:\n")

        # Lists of generators and solvers to iterate through
        generators = [BinaryTree, BacktrackingGenerator, Wilsons, Ellers]
        solvers = [Tremaux, BacktrackingSolver, RandomMouse, ShortestPath]
```

Figure 87 Test full code 1.



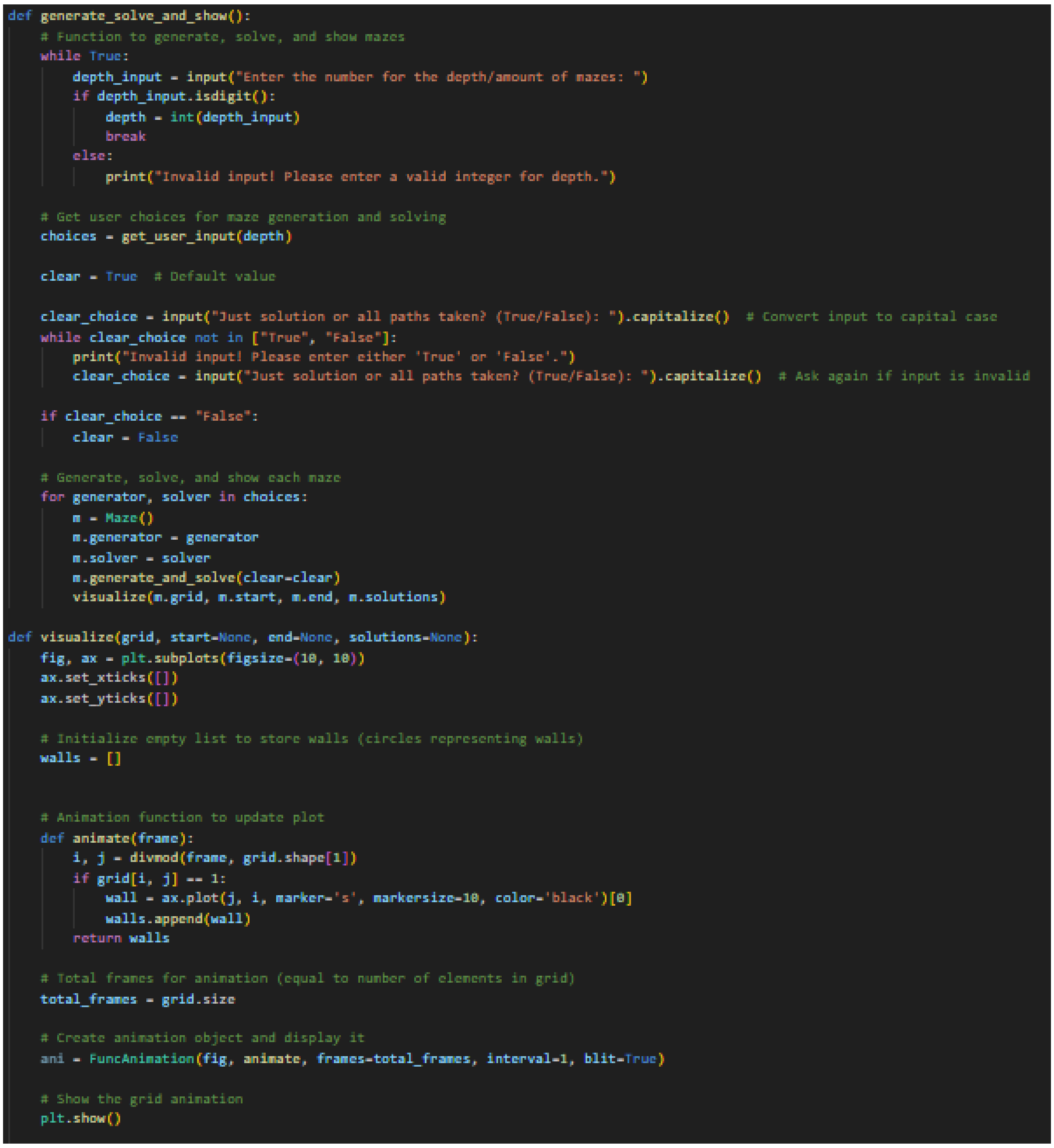
```python
        # Iterate through each combination of generator and solver
        for generator in generators:
            for solver in solvers:
                # Write generator and solver names to file
                f.write(f"Generator: {generator.__name__}, Solver: {solver.__name__}\n")

                # Variables to track statistics for current generator-solver combination
                total_time_gen_solver = 0
                total_len_solutions_gen_solver = 0
                shortest_time_gen_solver = float('inf')
                longest_time_gen_solver = 0
                smallest_len_solutions_gen_solver = float('inf')
                largest_len_solutions_gen_solver = 0

                # Generate mazes and solve
                for _ in range(loop):
                    m = Maze()
                    m.generator = generator(25, 25)
                    m.solver = solver()
                    m.generate()
                    m.generate_entrances()
                    start = time.time()
                    m.solve(clear=clear)
                    end = time.time()
                    time_taken = end - start

                    # Check if time exceeds 2 hours
                    if time_taken > 7200:
                        print(f"Time taken for {generator.__name__}-{solver.__name__} mazes exceeded 2 hours. Exiting.")
                        break

                    # Update statistics
                    shortest_time = min(shortest_time, time_taken)
                    longest_time = max(longest_time, time_taken)
                    total_time += time_taken
                    total_time_gen_solver += time_taken
                    shortest_time_gen_solver = min(shortest_time_gen_solver, time_taken)
                    longest_time_gen_solver = max(longest_time_gen_solver, time_taken)

                    # Calculate number of solutions
                    num_solutions = sum(row.count('#') for row in m.tostring(solutions=True).split('\n'))
                    smallest_len_solutions = min(smallest_len_solutions, num_solutions)
                    largest_len_solutions = max(largest_len_solutions, num_solutions)
                    total_len_solutions += num_solutions
                    total_len_solutions_gen_solver += num_solutions
                    smallest_len_solutions_gen_solver = min(smallest_len_solutions_gen_solver, num_solutions)
                    largest_len_solutions_gen_solver = max(largest_len_solutions_gen_solver, num_solutions)

                # Calculate average statistics for current generator-solver combination
                average_time_gen_solver = total_time_gen_solver / loop if loop > 0 else 0
                average_len_solutions_gen_solver = total_len_solutions_gen_solver / loop if loop > 0 else 0

                # Write statistics to file
                f.write(f"Total time taken for {generator.__name__}-{solver.__name__} mazes: {total_time_gen_solver:.2f} seconds\n")
                f.write(f"Average time per {generator.__name__}-{solver.__name__} maze: {average_time_gen_solver:.2f} seconds\n")
                f.write(f"Shortest time taken for {generator.__name__}-{solver.__name__} mazes: {shortest_time_gen_solver:.2f} seconds\n")
                f.write(f"Longest time taken for {generator.__name__}-{solver.__name__} mazes: {longest_time_gen_solver:.2f} seconds\n")
                f.write(f"Total length of solutions for {generator.__name__}-{solver.__name__} mazes: {total_len_solutions_gen_solver}\n")
                f.write(f"Average length of solutions per {generator.__name__}-{solver.__name__} maze: {average_len_solutions_gen_solver:.2f}\n")
                f.write(f"Smallest length for solutions found for {generator.__name__}-{solver.__name__}: {smallest_len_solutions_gen_solver}\n")
                f.write(f"Largest number for solutions found for {generator.__name__}-{solver.__name__}: {largest_len_solutions_gen_solver}\n\n")

                # Check if time exceeded 2 hours, if so, exit
                if time_taken > 7200:
                    return

if __name__ == '__main__':
    test()
```

Figure 88 Test full code 2.



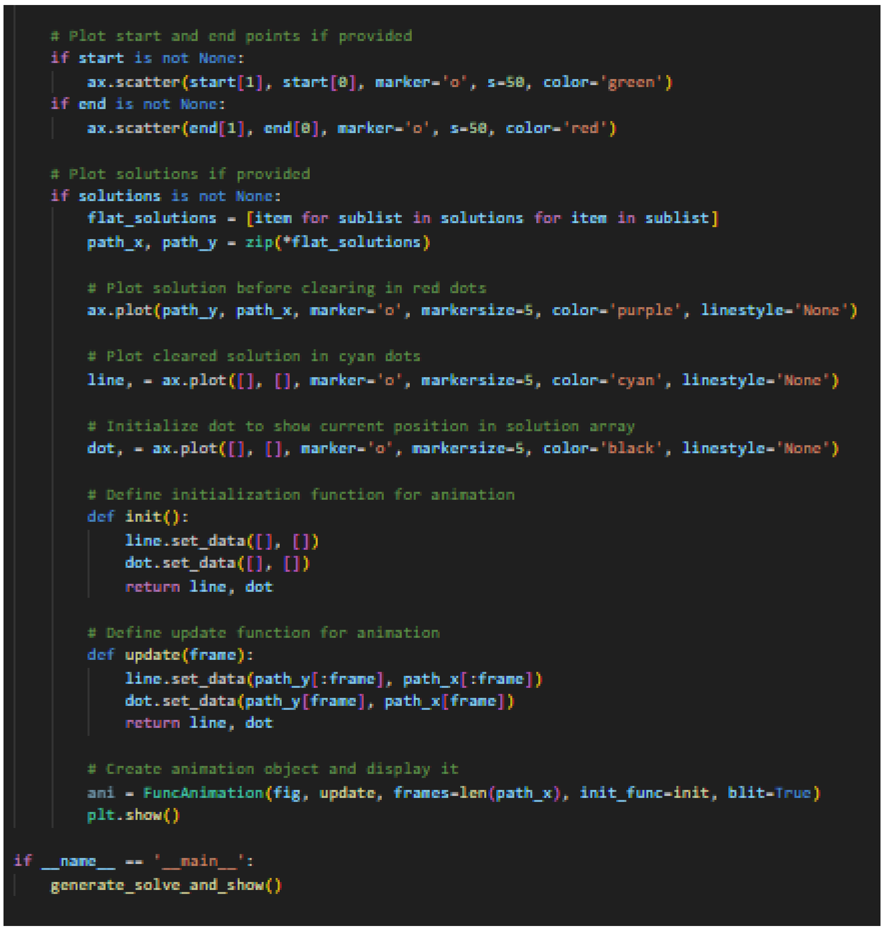
```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from maze import Maze
from Wilsons import Wilsons
from BinaryTree import BinaryTree
from Ellers import Ellers
from Tremaux import Tremaux
from BackTrackingSolver import BacktrackingSolver
from BackTrackingGenerator import BacktrackingGenerator
from ShortestPath import ShortestPath
from RandomMouse import RandomMouse

def get_user_input(depth):
    # Function to get user input for maze generation and solving options
    generator_options = {
        "1": Wilsons,
        "2": BinaryTree,
        "3": Ellers,
        "4": BacktrackingGenerator
    }
    solver_options = {
        "1": RandomMouse,
        "2": Tremaux,
        "3": BacktrackingSolver,
        "4": ShortestPath
    }
    while True:
        print("Select a generator:")
        for key, value in generator_options.items():
            print(key, value.__name__)

        generator_choice = input("Enter the number of your choice: ")
        if generator_choice in generator_options:
            break
        else:
            print("Invalid input! Please enter a valid generator choice.")

    while True:
        width_input = input("Enter the width for the maze: ")
        if width_input.isdigit():
            width = int(width_input)
            break
        else:
            print("Invalid input! Please enter a valid integer for width.")

    while True:
        height_input = input("Enter the height for the maze: ")
        if height_input.isdigit():
            height = int(height_input)
            break
        else:
            print("Invalid input! Please enter a valid integer for height.")

    while True:
        print("Select a solver:")
        for key, value in solver_options.items():
            print(key, value.__name__)

        solver_choice = input("Enter the number of your choice: ")
        if solver_choice in solver_options:
            break
        else:
            print("Invalid input! Please enter a valid solver choice.")

    return [(generator_options[generator_choice](width, height), solver_options[solver_choice]()) for _ in range(depth)]
```

Figure 89 Menu full code 1.



```python
def generate_solve_and_show():
    # Function to generate, solve, and show mazes
    while True:
        depth_input = input("Enter the number for the depth/amount of mazes: ")
        if depth_input.isdigit():
            depth = int(depth_input)
            break
        else:
            print("Invalid input! Please enter a valid integer for depth.")

    # Get user choices for maze generation and solving
    choices = get_user_input(depth)

    clear = True  # Default value

    clear_choice = input("Just solution or all paths taken? (True/False): ").capitalize()  # Convert input to capital case
    while clear_choice not in ["True", "False"]:
        print("Invalid input! Please enter either 'True' or 'False'.")
        clear_choice = input("Just solution or all paths taken? (True/False): ").capitalize()  # Ask again if input is invalid

    if clear_choice == "False":
        clear = False

    # Generate, solve, and show each maze
    for generator, solver in choices:
        m = Maze()
        m.generator = generator
        m.solver = solver
        m.generate_and_solve(clear=clear)
        visualize(m.grid, m.start, m.end, m.solutions)

def visualize(grid, start=None, end=None, solutions=None):
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.set_xticks([])
    ax.set_yticks([])

    # Initialize empty list to store walls (circles representing walls)
    walls = []


    # Animation function to update plot
    def animate(frame):
        i, j = divmod(frame, grid.shape[1])
        if grid[i, j] == 1:
            wall = ax.plot(j, i, marker='s', markersize=10, color='black')[0]
            walls.append(wall)
        return walls

    # Total frames for animation (equal to number of elements in grid)
    total_frames = grid.size

    # Create animation object and display it
    ani = FuncAnimation(fig, animate, frames=total_frames, interval=1, blit=True)

    # Show the grid animation
    plt.show()
```

Figure 90 Menu full code 2.



```
    # Plot start and end points if provided
    if start is not None:
        ax.scatter(start[1], start[0], marker='o', s=50, color='green')
    if end is not None:
        ax.scatter(end[1], end[0], marker='o', s=50, color='red')

    # Plot solutions if provided
    if solutions is not None:
        flat_solutions = [item for sublist in solutions for item in sublist]
        path_x, path_y = zip(*flat_solutions)

        # Plot solution before clearing in red dots
        ax.plot(path_y, path_x, marker='o', markersize=5, color='purple', linestyle='None')

        # Plot cleared solution in cyan dots
        line, = ax.plot([], [], marker='o', markersize=5, color='cyan', linestyle='None')

        # Initialize dot to show current position in solution array
        dot, = ax.plot([], [], marker='o', markersize=5, color='black', linestyle='None')

        # Define initialization function for animation
        def init():
            line.set_data([], [])
            dot.set_data([], [])
            return line, dot

        # Define update function for animation
        def update(frame):
            line.set_data(path_y[:frame], path_x[:frame])
            dot.set_data(path_y[frame], path_x[frame])
            return line, dot

        # Create animation object and display it
        ani = FuncAnimation(fig, update, frames=len(path_x), init_func=init, blit=True)
        plt.show()

if __name__ == '__main__':
    generate_solve_and_show()
```

Figure 91 Menu full code 3.