OCR GCE A COMPUTER SCIENCE PROJECT H446-03

Name: William King

Candidate Number: [redacted]

Title of Project: Pathfinding Algorithm Visualiser

Contents

Analysis

Project Outline

Description of the problem

A pathfinding algorithm attempts to find the shortest path between two points. There are many unique pathfinding algorithms, and they all utilise a variety of concepts and are built using different techniques – each having their own uses depending on their strengths and weaknesses. An example of how they vary is how they interpret information; some requiring more initial information than others or taking more steps to analyse the environment and information given.

However, there are too many plain resources about the theory of pathfinding algorithms either on the internet (such as articles, images and videos) or as physical copies (such as books), and not enough engaging and interactive material about how they work and compare to one another. This is a problem, because it can be difficult to understand and compare different algorithms using just theoretical information, especially since they are practical and fluid processes. There needs to be more programs that demonstrate pathfinding algorithms, where the user is at the centre and can observe every algorithmic process in an environment more relevant to their situation, having built it themselves or procedurally generated one. In this way, the user can get a firm grasp on the precise inner workings of a pathfinding algorithm, by observing how it works in different environments and how it directly compares to other algorithms.

This project aims to provide such a program, as described above, in order to reduce the problem of there being a lack of immersive resources about pathfinding algorithms.

Main features of the program

The pathfinding algorithm visualiser program will have many features that will allow the user to extensively explore and thoroughly observe how a chosen algorithm works in varying environments, in a hope that the user will be able to determine the right algorithms for their projects, and/or develop a greater understanding of the precise workings of the algorithm and how it compares to other algorithms.

The program will consist mainly of a grid of square nodes, where the pathfinding algorithm simulation will take place. Each node in the grid will have different properties and appearances, and will adapt during the visualisation of a pathfinding algorithm to represent its processes as they unfold. Some nodes will serve a purpose and have functionality, whereas other nodes will just be there to convey information graphically about their current state during a simulation; all nodes will 'work together' to effectively convey the processes of a pathfinding algorithm and visualise its mechanics.

The grid won't be the only component in the program – there will also be a toolbar, where the user can edit and manipulate the nodes in the grid, as well as control the simulation and pick the pathfinding algorithm to be visualised.

The user interface (UI) will remain the same throughout the entire program, all happening in the same window – there won't be a menu/start screen as everything can be controlled via the toolbar. The graphics will be quite simple, with a modern look. Sound effects will be limited – there may be a couple basic sounds to make it slightly more immersive.

Stakeholders

Audience and end users

This program is an educational resource intended for programmers and developers, as well as CS students (at college or university) and intellectuals that are exploring this area of Computer Science (pathfinding algorithms), looking to learn more. Therefore, the program will be used by a wide range of people and age groups, from beginner to expert programmers, as well as young students and even older people looking to learn something new and extend their knowledge. However, most of the audience will be relatively young, as it does require the user to have decent IT ability and knowledge.

Many programmers and developers implement pathfinding algorithms in their projects and programs, as they have many applications and uses ranging from NPC movement in games, to the more practical navigation software and satnavs. A problem they might face may be deciding which pathfinding algorithm to use for different parts of their project. A pathfinding algorithm visualiser would solve this problem by allowing them to test and observe different algorithms while they run and find the shortest route in different environments. They'd be able to set up ideal environments and conditions that represent scenarios similar to the ones in the programs they are developing, and hence find the best algorithms to implement into their project that best achieve their goals and meet their needs. A pathfinding algorithm visualiser would simplify the development of a project for a programmer or solutions architect, as they can more easily compare pathfinding algorithms in environments that they have created themselves, thus making it easier to research and choose the pathfinding algorithm that is best for their situation. Programmer's work is based entirely on the computer, and so they will definitely have access to the program and will have more than sufficient technology skills and knowledge to operate the program well and utilise all of its features. They will need this program to make the best, most informed decision about which pathfinding algorithm they should implement.

Another group of people with a problem might be learners trying to understand a particular pathfinding algorithm. They may find it difficult to visualise the processes involved, even if they know how the algorithm works in theory. A lot of learners (such as visual learners) would benefit from seeing that particular pathfinding algorithm work in different situations first hand to observe the step-by-step decisions the algorithm makes to find the shortest path to its goal. Many people would learn best by testing the algorithm in different scenarios that they have built themselves, and watching how the algorithm runs when a variety of obstacles and conditions are used. A pathfinding algorithm visualiser would solve all the problems a learner may face if they only had access to a textbook, websites or videos. An interactive visualiser would enable them to learn more effectively and gain a stronger understanding of individual pathfinding algorithms and how they work, as well as their uses and the contexts they're most effective in. Having access to this pathfinding algorithm visualiser will eliminate the problem of being restricted to basic, unstimulating learning resources, which will slow down the learning process for a lot of students; some of them will be too bored and unmotivated without an interactive visualiser, which would make their learning more interesting and simplify the learning process. It will be very accessible to students because most of them will have access to the internet and use devices for their college/university work, which they can use to run the program on.

Identification of the stakeholders

With these two groups of people in mind, I have chosen these people to be stakeholders in my project:

Name	Description	Reason
Lee	A network/cybersecurity engineer that manages and protects a large website for a booking company he works for.	He is an experienced (39 years old), self-taught programmer who has been obsessed with computers from a very young age. He will be useful to consult throughout the process as he will be familiar with a lot of the algorithms – he is my uncle and I see him fairly often. Also, if he finds the program quite useful for himself, it'll mean that the program will at least be effective for most beginner/intermediate programmers, if not people more experienced than him.
Jude	A year 8 who is considering taking Computer Science at GCSE; he is trying to learn to code.	I speak to my brother every day and give him coding lessons when I can. This regular contact means he will be an effective and efficient stakeholder. Being a very young student, his understanding of pathfinding algorithms is extremely limited, and so having his input throughout the process will help me make a very clear, visual and intuitive program so that beginners like Jude are comfortable and motivated to learn and use it for their studies or personal coding projects.
Kieran	A sixth form student currently studying Computer Science at A level.	Kieran is in my Computer Science classes at sixth form so we can easily communicate with each other. He will know more about pathfinding algorithms than Jude and enjoys programming, often coding in his free periods. This makes him a great candidate for representing the average end-user and his feedback will be valuable.

Due to Lee's expert perspectives and more informed/useful feedback, I have decided to make him the main interviewee. I have also chosen Jude because, although he isn't as mature as the others, his periodic questions and feedback will help prevent the program from becoming too complex or inaccessible for beginners. Kieran will also be able to personally provide valuable feedback when he tests my program in class, however his responses won't be recorded as frequently as the interviewees'.

Computational Methods

Justification for how the problem is amenable to a computational approach

The concept of a pathfinding algorithm itself plays a huge role in technology and computer theory, therefore it is clear that a computational approach to create a solution to the problem is the most logical approach.

In order visualise a pathfinding algorithm, and control the simulation as it runs to observe the steps it takes and how it responds to the environment requires a computational approach, since it's difficult to create a simulation without using technology – the demonstration wouldn't be dynamic or adaptable and would just add to the problem of repetitive, dull learning resources or research material. User input is very important in this solution, as the user needs to be able to easily manipulate the environment and control the simulation; they need to be able to interact effectively with the material, requiring computational methods (such as selection) – without which the

experience won't be immersive. Computational methods will be needed to create a resource that will respond to user inputs, process them and then output something with very quick response times. Also, a computer is designed to perform billions of tasks per second and, unlike a human or other contraption, can very rapidly update a display (I.e., monitor), and create a cooperative, responsive experience for the user. An Object-Oriented approach will suit the creation and control of the many nodes used in the visualisation, and elements of the program will contain procedural and random generation, which will also require computational methods (such as iteration).

Other approaches we could use to produce a solution involve physical resources, such as: books and videos. However, these have to be distributed and doesn't solve the problem, as there are too many of these static resources already.

Thinking abstractly

This project will require me to think abstractly, as the visualiser is, as the name suggests, just a visualisation of how the algorithm works – of course the algorithm's mathematical processes aren't shown, just some of the results of each decision towards finding the shortest path to the goal. It is important to remove unnecessary details that might distract the user from learning or comparing the pathfinding algorithms, which would prevent them from seeing and using the algorithm in its bare state.

Abstraction	Explanation
2-D grid of squares nodes	A lot of pathfinding algorithms are demonstrated (or implemented into programs) using a graph which contain nodes, where the distance between nodes has a weight. However, to make the process more visual and simpler for the user, the program will use a grid of equally spaced square 'nodes,' each with the same weight (with a few exceptions), to make the algorithm processes more visual and clearer to the user.
Different node types	In order to represent the most common environments in which pathfinding algorithms may be used (such as a satnav), without including any specific elements in that environment – which would make the visualiser too specialised – the user will be able to use types of nodes that will abstractly represent common variables within any environment. For example, a 'dense' node, which may represent traffic in a satnav or rough terrain in a game. These nodes will affect how the algorithm runs.

Thinking ahead

Here is some key data and inputs that will be required for my solution. The program will need to keep track of the data and update them constantly as the user interacts with the program. Some of the inputs will involve events, which are raised when something specific happens (such as a mouse click), calling a subroutine that handles the event.

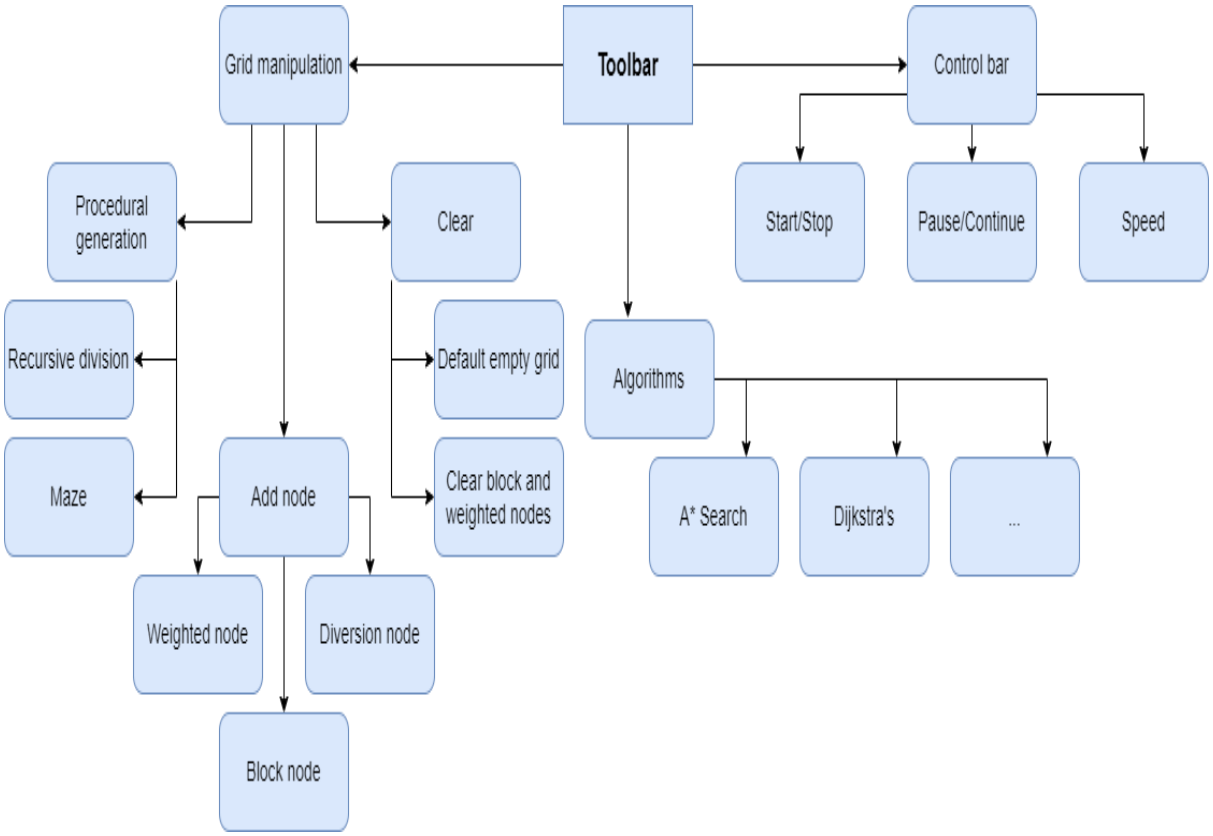
Description	Type	Explanation
Node type	**Data**	Some nodes will have different properties. The program needs to know the node type of each node so that it can interpret the environment, such as where the start and target nodes are, and which nodes are just standard, 'empty' nodes.

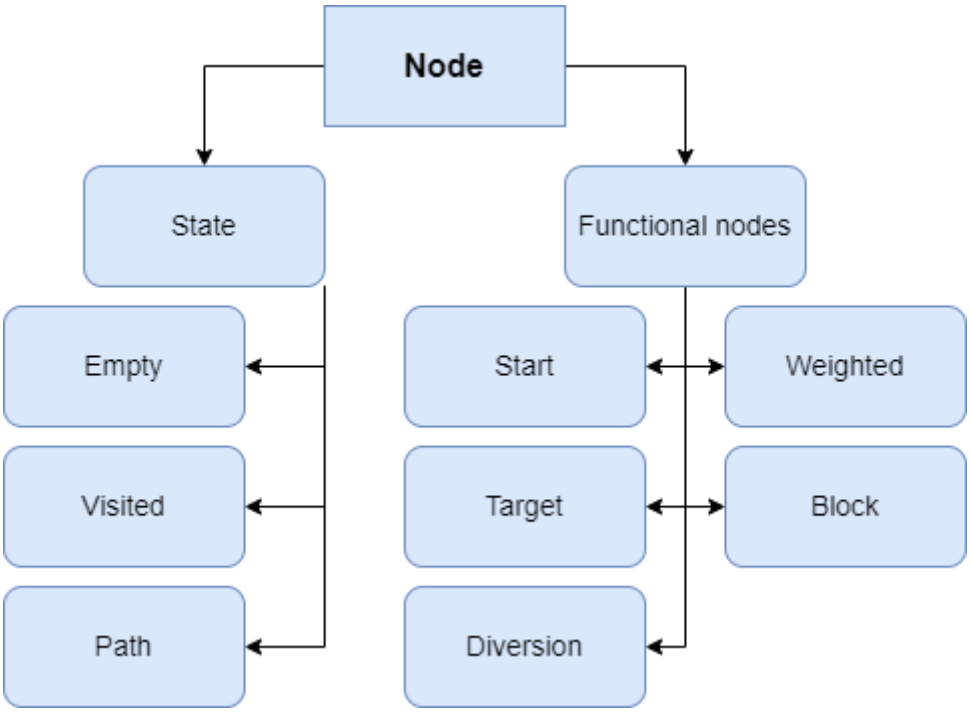
Node states	**Data**	While a pathfinding algorithm is running, data needs to be recorded about visited nodes and nodes in the shortest path, once calculated, so that the program can display the information appropriately to the user.
Node cost	**Data**	Whilst calculating the shortest path, the movement 'cost' of each node needs to be determined – the shortest path will be the path with the lowest cost.
Current algorithm	**Data**	The user will be able to select the pathfinding algorithm to be visualised. The program will need to know which algorithm to run during the simulation.
Simulation state	**Data**	The state of the simulation is determined by the user. The program needs to respond depending on whether the simulation is: running, paused or stopped.
Simulation speed	**Data**	The user will be able to adjust the simulation speed; the program will need to adjust accordingly.
Graphical information	**Data**	Data needs to be stored about how each node should appear (e.g., background colour) depending on their states and purposes.
Controls	**Data**	The program should respond appropriately to user input, so data must be stored about the inputs which will be controls.
Run/Stop simulation	**Input**	There will be a button to start the visualisation and stop/reset it – where all node states will be set back to unvisited.
Pause/continue simulation	**Input**	There will be a button, and keyboard shortcut, to pause the simulation and to resume the simulation.
Set speed	**Input**	There will be a drop-down menu, which the user can select the simulation speed from, as well as keyboard shortcuts.
Add node	**Input**	There will be a collection of node types that can be added to the grid.
Move node	**Input**	The user will be able to move some node types (start, target and diversion nodes) around using the mouse.
Add block	**Input**	The user will be able to create a block node using the mouse or keyboard.
Generate random grid	**Input**	There will be a button and a drop-down menu that the user can interact with if they wish to procedurally generate a grid, such as a random maze.
Clear grid	**Input**	There will be a button that will clear all the nodes on the grid – creating a completely empty grid. Alternatively, there could be an option to only clear certain parts of the grid.
Select algorithm	**Input**	There will be a drop-down menu where the user can select the pathfinding algorithm that they wish to see visualised.

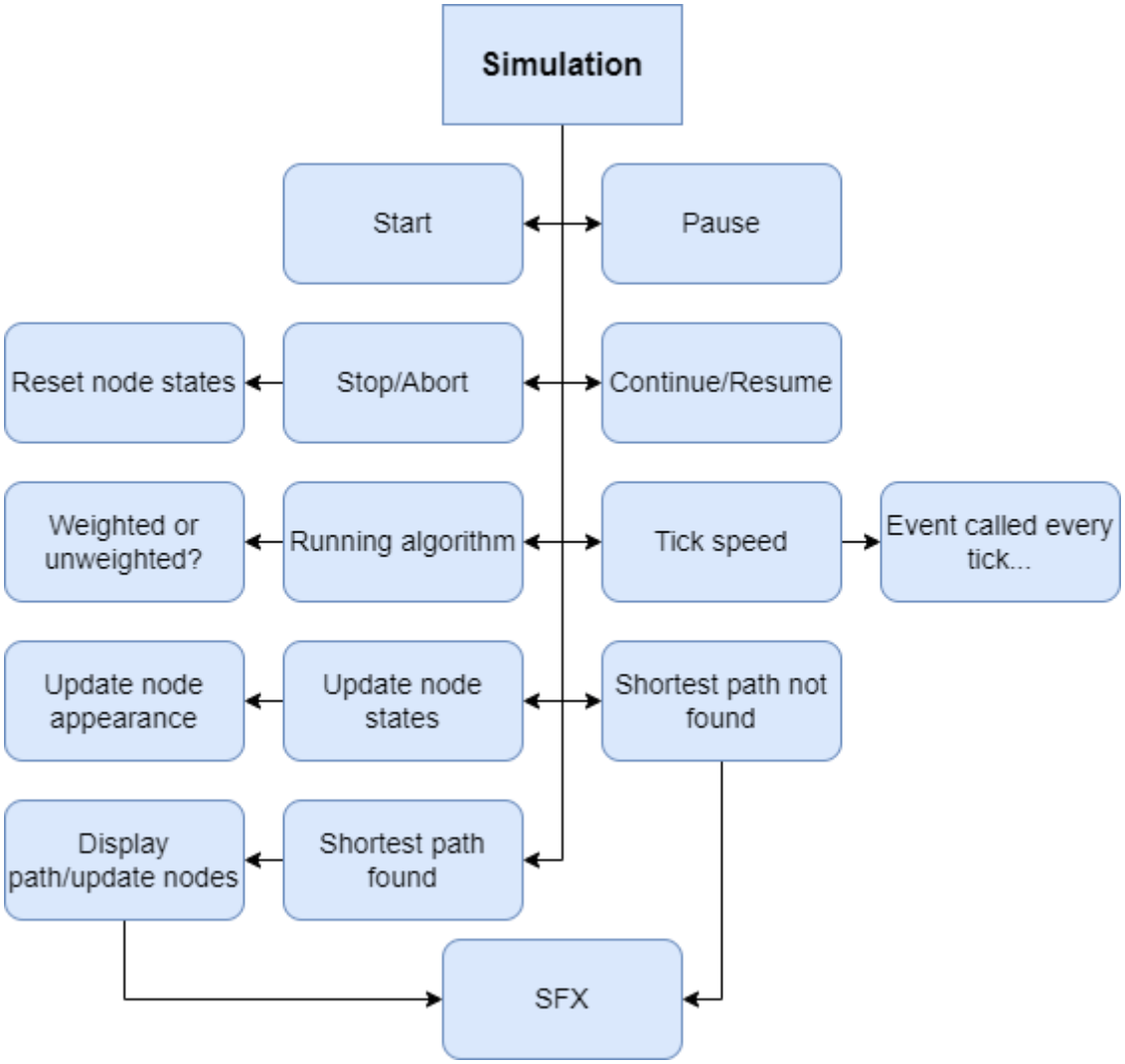
Thinking procedurally

Decomposition is a very important aspect of designing a solution. This program can be broken down into smaller chunks; each chunk can be tackled independently. I have broken down the solution into three main parts: the toolbar, the nodes (there will be a grid of these) and the simulation. Each algorithm will run differently under the simulation – each algorithm will, of course, have different behaviours and code, but the simulation is the framework which will run and control each algorithm.

This will be developed further in the design stage – where the 'chunks' could be classes and objects, and the individual parts could be procedures and functions:







Thinking logically

This involves identifying points where branching and repetition might occur, in which the outcome will differ depending on the conditions set and the current data values. This will be developed further during the design phase using flowcharts and pseudocode.

Situation	Explanation
Running an algorithm	The implemented algorithms use many branches, and loops, as they receive data about the changing nodes in the grid and seek the shortest path.
Procedural generation	The generation of a maze, or a randomised grid of node types using other methods, will require many loops and iterations.
Run until...	...The shortest path is found, or the simulation is manually stopped by the user, or if a path cannot be found.
Updating the grid	As an algorithm is visualised, the nodes on the grid will adapt and change in appearance. Nodes in the grid will have to be looped through and updated very frequently. The appearance of a node will depend in its states and different factors – branching will occur.

Key press	When the user presses a key on the keyboard, the key will be determined, and a large select-case/switch statement will be required to respond appropriately to that specific key stroke.

Thinking concurrently

It is important to perform tasks at the same time, where possible, to reduce waiting times and delays for the user.

Examples include:

- Updating the appearance of each node in the grid whilst the algorithm is running.
- Changing the speed of the simulation whilst the algorithm is running.
- The algorithms themselves may perform many tasks simultaneously. An extreme example is the bidirectional swarm algorithm, which attempts to find the shortest path (not guaranteed) by searching from both the start and target node at the same time.

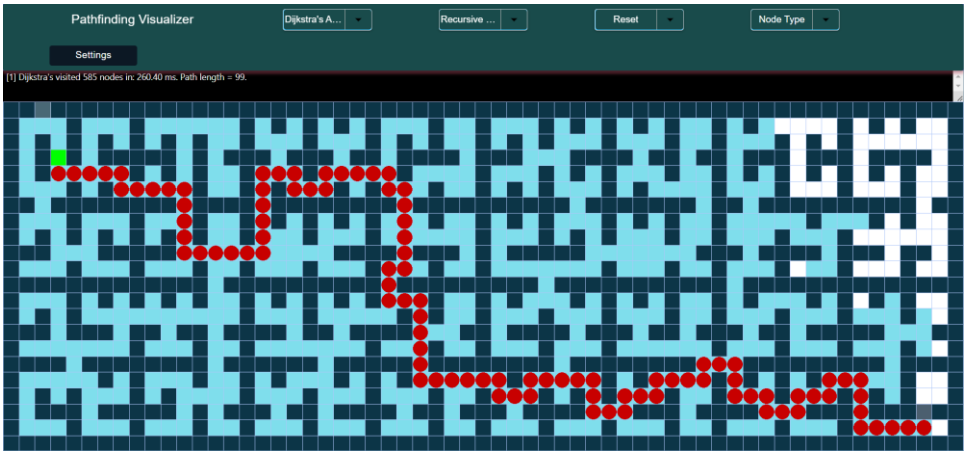
Research

Similar programs

I have researched and found a couple programs that visualise pathfinding algorithms on the internet. I have taken inspiration from these existing programs, as well as identified how my program will be different, by combining the most relevant features from both and not including unnecessary or specific ones.

Website 1- *pathfindout.com*:

This website visualises pathfinding algorithms using a grid of square cells. There is a start node (lime green) and end node (red), where the chosen pathfinding algorithm will try to find the shortest path between, which the user can drag around the grid. The user is able to choose the pathfinding algorithm from a drop-down menu at the top, and the visualisation begins as soon as they do so. Whilst the algorithm is running, the visited nodes are indicated by turning pale blue. Once the shortest path is found, each node in that path gains a dark red circle in an animation starting from the starting node, all the way to the end node. Also, once the visualisation is complete, the user is able to move the start node around, and the new shortest path is instantly calculated and displayed. By left clicking, the user is able to place 'walls', which the algorithm cannot pass through.



In the bar at the top of the screen there is a settings button, which allows the user to customise their experience and change the program's behaviour in incredible detail – such as changing the animation speed and algorithm settings, and setting the node weights for each node type. There is also a drop-down menu where the user can select a node type that they can then start assigning to nodes on the grid; there is a variety of node types, each with a different 'weight', which determines how 'difficult' the node is to pass through – the shortest path will be the path with the lowest sum of all the weights (the default node type is air). Each node type will have a different appearance to make them distinguishable. There is also a drop-down menu where the user can choose to generate terrain if want to. From this bar, they are also able to clear or reset the board. Here are all the node types and their weights, as well as an image which shows a Simplex Terrain generation:

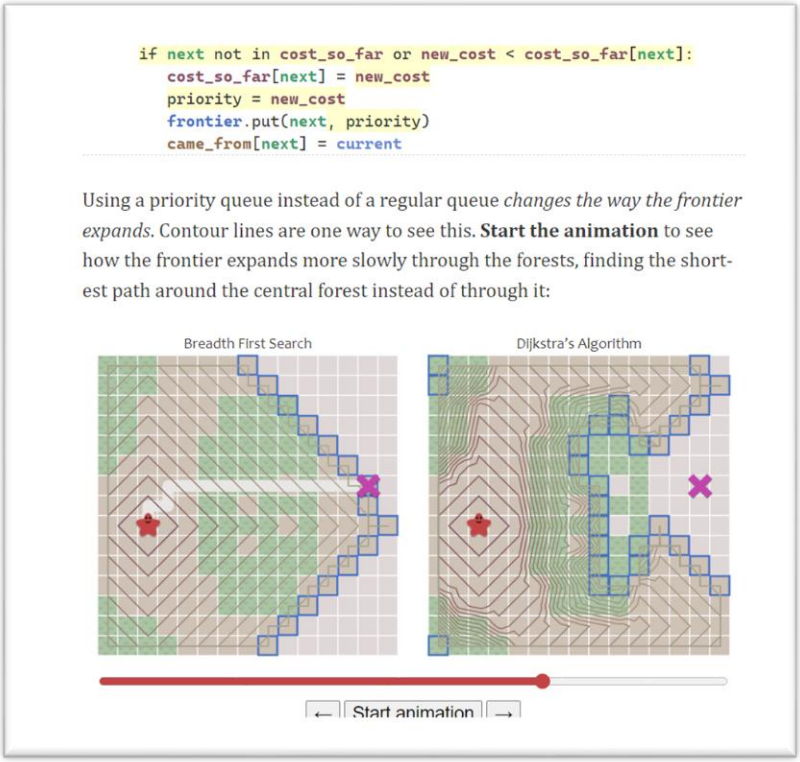
- Air [1]
- Grass [5]
- Sand [7]
- Stone [25]
- Granite [50]
- Water [50]
- Snow [75]
- Water Deep [100]
- Wall [infinity]

Website 2 - *redblobgames.com*:

This is an educational website that makes "interactive visual explanations of math and algorithms, using motivating examples from computer games." Although this website covers a wide range of topics, unlike my specialised pathfinding algorithm visualiser, this quote found from the top of the website sums up perfectly my aim with this project – to provide an educational, interactive resource that will visually explain pathfinding algorithms in a motivating way for students, developers and hobbyists alike; the website also mentions these groups of people when describing who the website is for.

As well as taking inspiration, this will be a very useful website for me as I learn pathfinding algorithms in more detail myself. *Amit Patel* (the creator of this website) has many interactive webpages, guides and tutorials on many topics – here are a few that are relevant to pathfinding algorithms:

- Introduction to A* and implementation guide (also: Dijkstra's Algorithm, Breadth First Search)
- Tower Defence pathfinding (also: Breadth First Search, Dijkstra Maps, Flow Field pathfinding)
- Intro to graph theory (also: how to represent grids as graphs)
- Pathfinding with A*
- Pathfinding with circular obstacles (also: bitangents)
- All pairs shortest paths for map analysis



Website	What did I identify that I can include in my project? Why?	What did I identify that I won't include in my project? Why?
Website 1 - ***pathfindout.com***	**Grid of square cells** – An effective and simple way of visualising 2-D space, a good abstraction from the commonly used graphs with weighted connection between separated nodes (as discussed in *thinking abstractly*). **Moveable start and end nodes** – The user needs to be able to easily manipulate the environment and determine where the algorithm will start, and where it should find the shortest path to. They should also be able to easily see where the start and end nodes are, hence, they should have contrasted colours. **Algorithm selection** – The user should be able to pick the pathfinding algorithm they want to see visualised from the list. **Node states update as the simulation runs** – It is vital that the user understands what the algorithm is doing whilst it is running. Therefore, users should see which nodes have been visited (by changing the colour, for example).	**Visualisation starting as soon as an algorithm is selected** – I found it a little bit frustrating as sometimes I selected the algorithm before building the environment fully, at which the simulation would immediately start. I want the user to be able to decide when the simulation should start; there should be a designated button for it. **Dark red circles on the nodes in the shortest path –** Although the animation is effective, I think the dark red circles are a bit too retro. I think a more modern, flat look would be more

	Shortest path animation – I like the simple animation, once the shortest path is found, as it improves the graphics, making it more appealing and fluid, all contributing to a more interactive experience. I will try to implement something similar in my project. **Rapid shortest path recalculation** – Once the shortest path has been visualised, and the board has not yet been reset, the user is able to move the start node around and the shortest path is quickly recalculated. This is helpful if the user doesn't want to visualise an algorithm in certain situations. **Impassable 'wall' nodes** – This is very important, as it restricts the algorithm and allows the user to see the processes unfold in more complex environments. **Weighted nodes and shortest path calculations (cost)** – This is an effective way of determining which is the shortest path. The weighted nodes will be another useful tool the user can use to explore how the pathfinding algorithms work. **Procedurally generating a board** – Apart from the *Simplex Terrain* generation, I will take inspiration from this. The recursive maze generation seems very attractive – it will be a quick way for the user to create an interesting grid with little effort. **Clearing the board –** The user will need to be able to clear the board (or parts of the board, such as resetting the path or clearing all the walls); it would be quite laborious to clear each node manually, it wouldn't be a fun experience for the user.	appealing – such as yellow matte nodes. **Settings –** Although this would improve the customisable aspect of my project, and allow the user to alter the environment on a lower level to suit their specific needs, it would be too complex to implement in the time I have, especially to the degree of detail in this website; it isn't essential. **A large range of node types (terrain)** – Although weighted nodes is a good concept, which I will be using, having a lot of different node types with different weights might be too much, and the user may become overwhelmed; it could be an overcomplication and may distract the user from the essence of the program – I don't want it to turn into a sandbox game about building terrain.
Website 2 – ***redblobgames.com***	**Pathfinding visualisation and demonstrations** – In many pages within this website, there are multiple sections which visualise and show the processes of many pathfinding algorithms. My project will take that to the extreme, being solely focussed on that goal.	**Wordy sections and explanations** – Although this website is very interactive and a great resource for exploring pathfinding algorithms, my project is trying to go even further to make an even less static and more engaging resource – the ultimate immersive material.

		A wide range of content and topics – My project will be solely focussed on visualising pathfinding algorithms, whereas this website also includes tutorials on other computing and maths concepts.

Interview transcript

I've asked the interviewees a series of questions relating to: the UI design and themes; the algorithms and visualisation process; and the customisability and desired features. Lee (red) and Jude (green) have produced responses to these questions, providing me with varying views and perspectives that will hopefully represent the project's audience.

Where should the toolbar be on the window?

"I would prefer it at the top of the screen, but that's just my personal opinion. I think it'd be okay on any of the sides; maybe a toolbar that unfolds when you hover over it would be nice."

"The top like it is on Microsoft Word and lots of other programs."

How big should each square node in the grid be?

"Slightly on the smaller side so that the user has a larger canvas and more nodes to utilise. Not too small though, otherwise it'll be really frustrating trying to select a specific node."

"Not very small otherwise the individual nodes will be difficult to see. Nice and clear but not too big as that would become boring since there would be fewer nodes to use."

What design theme should be chosen?

"Modern, simple and clean. A couple matte colours for the UI elements, such as a white and blue, and then contrasting colours for the main parts of the visualisation like green and red for the start and end nodes."

"Clean and not complex or confusing to look at. Geometric shapes and no shadows or 3-D shapes."

Should there be any sound?

"A few brief and simple sound effects might be okay, such as when the shortest path has been found, like a ding or something."

"Sounds might get annoying, especially if you run simulations in quick succession or have to keep editing the grid. If there is sound, there should be an option or setting to mute it."

How should different nodes states and functions be identified?

"Symbols or pictures would help identify important nodes. Nodes with different states could be colour coded."

"Clear enough so that I am able to see the difference and understand what each node is doing and what it means. I don't want too many colours and pictures as that would make it difficult to follow."

Are there any other node types that could be included?

"I think more nodes could be added at the end if you wish to expand the program, but I think you have enough for now."

"I think there's enough for me to learn already."

How should you navigate and manipulate the grid?

"The mouse should be the primary tool as it is quicker at targeting and selecting specific nodes than the keyboard. I want to be able to drag the mouse over multiple nodes to select or edit them. Binding a commonly used node, like the wall node, to the right mouse button will speed up the process of editing the environment."

"I think just using the mouse pointer is fine. But it would be cool to be able to use the arrow keys as well like a game."

Is animation when 'placing' block nodes and other nodes necessary?

"It's not needed, but subtle movement could improve user experience. Dynamic elements, such as gradually filling the nodes in the final shortest path, will make the program feel more responsive."

"It would be cool and would make the program more fun."

Would you prefer the program to utilise separate windows?

"Multiple windows would clutter my screen, so I'd prefer one window with all the features laid out concisely. The only use of multiple windows I can think of is a settings window, but that's a stretch."

"I don't mind really. If you can fit it on one window then no."

Review of the interview

Here are some of the key takeaways and new/useful information collected from the interview:

- All the stakeholders (including Kieran) would like the toolbar at the top.
- There needs to be plenty nodes for the user to access and manipulate in the grid so they can build more complex environments and simulations.
- A modern and simple design, theme and UI colour scheme is preferred by both interviewees.
- There shouldn't be many sound effects (and they shouldn't be a priority in development).

- A mouse dragging feature when adding/erasing wall and dense nodes would improve the experience while editing. It would make editing smoother, easier and less frustrating.
- Animation adds to the user experience, but too much will be time consuming and unnecessary.
- The entire program should use only one window to reduce screen clutter.

Proposed Solution

Features

The UI in this program will stay the same throughout and will remain on the same window – there will be no menu/start screen or settings, as everything the user will need will be on the toolbar. The visualisation will occur on a 2-D grid – the graphics will be limited as there won't be any pictures or images, which isn't as necessary in this situation as colours/shapes are good enough representations of different types of nodes.

Sound effects and audio won't feature much in this program. However, there may be a few sound prompts, such as a ding when the shortest path is found, which could help contribute to a more immersive resource and keep the user engaged.

Feature	Description	Explanation
Grid of nodes	There will be a grid of square nodes in which the pathfinding algorithms will be visualised. By default, each node will be 'empty/unvisited'.	Each node will be represented by a label, so that it is visible to the user - the user needs to be able to see the algorithmic processes unfold through these nodes.
Start/Target nodes	The user will be able to determine which node will be the start node, and which node will be the target node – the algorithm will try to find the shortest path from the start node to the target node.	This is essential for making the program interactive – the user needs to be able to manoeuvre these nodes around as they build the environment. Of course, these nodes need to be easily distinguished.
Node state	While the algorithm is running, nodes that have been 'visited' will be distinguished from unvisited nodes; nodes in the shortest path will also be identifiable.	The user needs to be able to tell which nodes have been visited and analysed by the algorithm, and which nodes haven't, as well as the where the shortest path is. The user needs to know which nodes are functional, such as the diversion node.
Path cost	Moving from one node to another will have a 'cost' of one; the route with the lowest cost will be considered the shortest path. If a weighted algorithm is running, 'dense' nodes will have a cost of fifteen.	This is to better represent movement in reality, such as walking or driving between two locations, where a straighter route is usually quicker and more efficient.
Block nodes	The user will be able to allocate certain nodes to be 'block' nodes, which will act like impassable	This is important as it restricts where an algorithm can search, truly testing its capabilities and helping the user fully comprehend how it works by showing its

	obstacles; the algorithm cannot visit these block nodes.	'decision making' in more complex environments.
Dense nodes	The user will also be able to make 'dense' nodes. These nodes will have a cost of fifteen, as opposed to the standard cost of one (or infinity in the case of a wall node).	These nodes could represent traffic or rough terrain in the real world; the algorithm will 'avoid' these nodes where appropriate due to its high movement cost.
Diversion node	There will be an option for the user to have a 'diversion' node, which the shortest path must go through before reaching the target node.	This could have practical applications in programs where, for example, they may have to factor in getting to other locations before reaching the final destination.
Instant path recalculation	Once the algorithm has been visualised, the user will be able to move the start, target and diversion nodes around and the shortest path should be instantly updated and displayed, without all of the individual steps being shown. Nodes that have been visited will also be updated and identified as the user moves these three nodes around.	Some users will want to see the results of a pathfinding algorithm and may not want to keep visualising every time. They will be able to move some of the core nodes (start, target and diversion) around after a simulation is complete to quickly get an overview of how that algorithm would have run in those situations.
Toolbar	A bar at the top of the screen that will contain all the controls, functions and items needed when manipulating the grid and running a simulation.	It is vital that the user can efficiently control the environment and simulation, and so will have access to a range of buttons and drop-down menus on the toolbar.
Algorithm selector	A list in the toolbar which contains all of the pathfinding algorithms that can be visualised.	It is up to the user to decide which algorithm they want to see visualised – each user will be in a different circumstance and will want to try different algorithms, and so there will be a few to choose from.
Unweighted algorithms	It is important to note that some pathfinding algorithms the user may select will be unweighted. This means that the algorithm will not take the costs of dense nodes into account when calculating the shortest path; the shortest path will just be the route from the start to the target node which contains the lowest number of cells.	The user will be made aware of this and won't be able to add dense nodes if the algorithm is unweighted.
Procedurally generate grid	This toolbar will also contain a section where the user can select different methods of automatically generating blocks on the grid, if they don't want to create their own layout, or need a starting point, such as maze generation.	Not all users will want or need to create their own environment, and so it is important that there is a way for them to create an interesting environment without much effort.

Clear grid	From the toolbar, the user will be able to clear aspects of the grid.	The user will have the ability to clear all the nodes on the grid (set them all to empty) except the start and target nodes, which will be reset to their default positions in the grid. They may also want to only clear block and dense nodes, which they should also be able to do.
Run/Stop simulation	There will be a control panel where the user can run and stop/reset the simulation, which will clear all the progress made and reset all the node states.	Of course, the user gets to determine when the visualisation should start. The user will not be able to manipulate the nodes on the grid whilst a simulation is taking place, even when paused; the user can only edit the grid if the simulation has finished or stopped and reset.
Pause/Continue simulation	The user will be able to pause the simulation at any time, using the control panel, and continue it whenever they want.	The user should be able to pause the simulation if they need to analyse the situation further, or if they're leaving their computer and want to continue it later.
Adjust simulation speed	On the control panel, there will be a section to select the simulation speed.	It is important that the user can control the speed of the visualisation, as they will be trying to analyse the algorithms at different paces. For example, they may need to slow down the simulation if they need to see its individual step by step decisions in extra detail.

Limitations

Although it would be ideal to create a very comprehensive and complex program, it isn't realistic in the time I have. This doesn't mean my project can't be effective – I will have limitations, but everything that is necessary or very useful will be included.

I have discussed that the graphics and audio will be limited; it isn't an important factor in an educational resource like this – there will be a few sound effects and animations though to keep the resource engaging and motivating to use.

Limitation	Explanation	Justification
Limited range of algorithms	The user will only be able to select a few pathfinding algorithms which can be visualised.	It would be impractical (and almost impossible) to include every single pathfinding algorithm that has ever been made, and so it is logical that the user will only have a few to choose from – the most popular and common ones.
Only one weighted node (no terrain nodes)	Unlike in the first website I researched, where the user could select 'terrain', which had different weights, I will just have one weighted	Terrain may be distracting or create an environment that is too complex. It may become less obvious to the user how the running algorithm has found the shortest path. One weighted node will be sufficient and clearer, rather than a range of them.

	node which can be placed – the dense node.	
Grid cannot be edited whilst an algorithm is running	The user can only edit the grid when an algorithm isn't running – one hasn't been selected or the simulation has just stopped and reset.	The user and the algorithm may become confused if the environment is edited whilst it is running – it would be way too complicated calculate grid edits as an algorithm is running. For example, the user could (by accident or on purpose) change parts of the grid that have already been visited and considered to be in the shortest path, which could cause confusion and disrupt the visualisation.
Visualisation cannot be reversed.	The simulation will only be able to be paused, slowed down or sped up. It won't be able to be reversed.	It would be very difficult and would require more storage to be able to reverse a visualisation, as data about every node state needs to be recorded at each stage in its simulated history (version history).

Hardware & Software Requirements

To run this program, only a basic general-purpose machine is needed. However, the stress on the processor and memory may increase a little during complex simulations of algorithms – especially during the instant recalculations. For this reason, I recommend RAM of 4-8GB, and a multi-core processor with a clock speed of at least 2GHZ.

The table below shows the hardware and software that is required in order to run and use the program. However, there are other things that the user may have which could be used – such as a speaker (or headphones) to hear sound effects (there won't be many) and a keyboard to have access to keyboard shortcuts, which would increase productivity and ease of use.

Requirement	Type	Purpose and reason
Monitor	**Hardware**	The user needs a medium to view the program and its UI, and interact with the visualiser.
Mouse	**Hardware**	Allows the user to select things in the toolbar and on the grid, such as buttons and nodes.
Storage	**Hardware**	Once developed, the program will be an executable file, which can be distributed via the internet or by physical means (such as optical disks or flash storage – USB stick). This is possible because the executable file will be small (a few megabytes in size), and so not much secondary storage space is needed to store it.
Windows	**Software**	The Windows operating system is required to run the executable file, which will have been compiled for Windows since it will be built on Windows using a Microsoft IDE. Also, Windows is the most popular operating system, and so it will be highly accessible by students and most of my audience.

As well as these requirements for the user, I will need Microsoft Visual Studio IDE with the .NET framework and WinForms installed, where I can build and code my project for my chosen platform – Windows.

Success Criteria

ID	Criteria	Explanation and justification
1	There is 2-D grid of square nodes.	The user should see the visualisation of a pathfinding algorithm in this grid of nodes, where each node helps clearly demonstrate the decisions of the algorithm. There should always be a start and target node on the grid, and they should be easily distinguished from other node types. As discussed in ***thinking abstractly***, I think this is best done using a 2-D grid (rather than graphs).
2	The user can move the start, target and diversion nodes.	For ease of access, and a simpler experience for the user, it should be easy and intuitive to move the start, target and diversion (if 'placed') nodes. This is important – the user needs to determine where the visualisation should start and end. Just like in website number one (***research***), these nodes should be contrasted and noticeable.
3	Nodes will be different colours depending on their states.	It should be clear to the user what the algorithm is doing; this should be communicated through the changing appearance of each node depending on its state – which is best represented by its colour (for example, visited and path nodes should be different colours). As discussed in my ***research***, a matte and simple coloured approach to representing node states is more modern and simpler to understand; it is easier for the eye take in.
4	Each algorithm will attempt to find the shortest path.	I can test by eye whether the shortest path Is consistently found; it should be the path with the lowest 'cost', where each movement to a node will have a cost of one. If a weighted algorithm is running, dense nodes will have a cost of fifteen. I decided this in ***proposed solution***.
5	The user can add and remove block nodes.	This is an important interactive element to the resource (as highlighted in ***proposed solution***). The user should be able to build up the environment as they choose, and these block nodes will play a key role in creating diverse environments for the pathfinding algorithms, as seen in the first website during ***research***.
6	The user can add and remove dense nodes.	As well as block nodes, dense nodes will add more complexity to the gird if the user wants to utilise them. I concluded from my research that I won't include a range of weights like in the first website, just one dense node. I also talked about this in ***limitations***.
7	The user can optionally add a diversion node and can remove it.	This will add another layer of complexity for the user if they choose to add this node. It will help them learn how the algorithms work in more involved scenarios, and help them compare them comprehensively, by evaluating their performances in a wider range of possible situations. I decided this in ***proposed solution***.
8	Shortest path recalculation is almost instant.	An important feature is that the user can move the start or target node once a visualisation is complete, and the program will try to rapidly find the new shortest path and display its results (including indicating visited nodes). This could help the users who may not want to fully visualise a pathfinding algorithm every time. I was inspired by website one using this feature in my ***research***.
9	The toolbar is interactive and contains all the	The toolbar is a key component in the program. It should function well and be intuitive, otherwise the user will have a bad experience in trying to navigate, control and manipulate the program. It should be located at the top of the screen as mentioned in the ***interview***.

	controls and functions.	
10	There is an algorithm selector.	The user should be able to select the pathfinding algorithm they want to see visualised from a drop-down menu of plentiful options.
11	The algorithm selector contains a range of weighted and unweighted pathfinding algorithms.	Some pathfinding algorithms that the user can select will be unweighted. If the algorithm is weighted, then dense nodes will have a cost of fifteen and will affect the calculations in finding the shortest path. I decided to add both weighted an unweighted algorithms in ***proposed solution***.
12	A grid layout can be procedurally generated.	The user will have the ability to either generate a maze using recursive division or generate a totally random layout - both patterns can be built using block or dense nodes. This is important since not all users will want/need to build their own environments, see ***proposed solution***.
13	Specified aspects of the grid can be cleared and reset.	In order to save time, the user should be able to clear the gird entirely (set every node state to 'empty' and reset the start and target nodes to their default positions). Alternatively, the user should be able to just clear specific nodes, such as all the block and dense nodes.
14	The user can start the visualisation.	It is up to the user to decide when to start the simulation – I recalled in my ***research*** that I didn't like the first website's feature of immediately starting the visualisation as soon as the user selected the pathfinding algorithm.
15	The user can stop/abort a simulation.	If the user changes their mind about the algorithm they wish to see, or they need to edit the environment in some way, then they need to stop the algorithm which will set all the nodes states back to unvisited.
16	A simulation can be paused.	If the user becomes preoccupied, or needs to leave the computer, they can choose to pause the simulation; they can come back to it and resume it at any time.
17	A simulation can be resumed.	After the user has paused an algorithm, they should be able to resume from where they left off.
18	The simulation speed can be adjusted at any time.	Users may want to focus on specific parts of the visualisations – they may speed it up/skip to those parts, and then slow it down again to observe it in more detail during those parts.
19	A sound effect once the shortest path is found, and another if it is never found.	Simple sound effects can help keep the user engaged. However, sound effects must be extremely limited, otherwise a mute button will be needed for the user's that start to find the sounds annoying – this was highlighted in the ***interview***.
20	A simple animation that traces out the shortest path once found.	Clean and subtle movement/animation once the shortest path is found may be 'satisfying' for the user and might motivate them to keep running different algorithms and creating new scenarios. Animations were desired by both interviewees during the ***interview***.
21	A dragging feature that allows the user to manipulate multiple nodes.	The user should be able to drag their mouse while holding the left mouse button and add multiple nodes (block or dense nodes) across the grid in one movement, or hold the right mouse button to clear multiple nodes. This dragging feature was mentioned in the ***interview***.
22	Right clicking resets nodes to 'empty'.	Right clicking should 'remove' nodes and set them to 'empty'. The 'dragging feature' should apply here as well.

Design

Solution Structure

In *thinking procedurally*, I split my solution into three main parts: the toolbar, the simulation and the nodes. I did this because each section can be tackled individually to simplify the process of forming the complete solution, and it enables me to 'think ahead' by initially setting up the modules in a way that will accommodate future implementations with less complications – it reduces the probability of me having to revamp the program if I need to add something important, because there will be an effective underlying structure in place that tries to reduce dependence of parts of the program other parts, hence the separate and independently coded modules.

The toolbar will contain most of the necessary controls and components needed to fully utilise the program - such as functions to manipulate the grid and a control bar to govern the visualisation process.

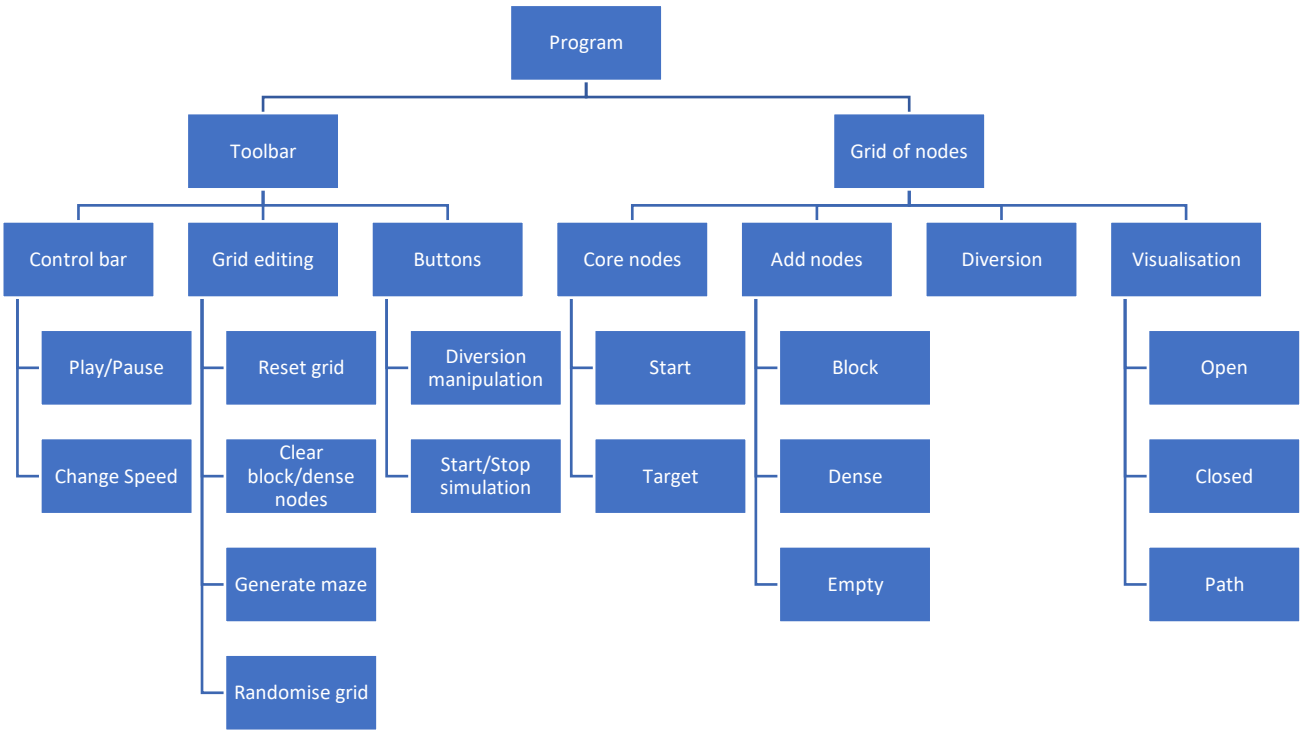
A large part of development will be dedicated to creating the rulesets and conditions that the simulation will run by. Pathfinding algorithms should then be easy to import as their code can be implemented without the need for each algorithm to update UI or check user inputs (such as change in speed). The simulation will deal with updating the grid visuals, once the algorithm is finished, in the order the algorithm took and at the relevant speed; this will make it easier to implement new algorithms if the project expands, as each node is responsible for updating its own appearance and the simulation will coordinate the appearances of affected nodes – the algorithms don't have to factor this in.

Most of the program window will be a grid of nodes, which is where visualisations will take place. Each node will be an object with attributes and ever-changing states; this is important data that the running algorithm will require to interpret the environment and find the shortest path.

There will be two main 'modes' the user will experience while operating the program. The user can either be editing in the grid or running the simulation – they shouldn't be able to do both simultaneously. While in editing mode (the simulation isn't running), the user will be able to edit the grid by adding, replacing and moving nodes. While the simulation is running, the user cannot edit the grid, they can only change the nature of the simulation – for example its speed. I have split the program up in this aspect to make it easier to code and to reduce bugs and errors which may occur if the user is able to manipulate the grid while an algorithm is running. Although the user may find it annoying at first, especially if they want to quickly change only small details in the grid while the simulation is running, it will make the program more concise and accurate and reduces the chances of the algorithm making a mistake (or misinterpreting the grid) as well as reducing the likelihood that the user gets confused if they accidently alter the grid where processes are taking/have taken place, thus providing a clear program flow and producing a more robust product. One of the *success criteria* is 'instant path recalculation', which will allow the user to move the start, target, and diversion nodes around after a visualisation – this is the only time when the user can edit the grid while still technically in 'simulation mode', and it is to reduce the hassle of reloading the simulation if the user only wants to see the final result in similar situations.

Note that this program doesn't contain much 'program flow'. This is because it is more like a tool – the UI doesn't really change, and the visualising grid is the only thing that adapts. The program

doesn't change or progress (like in a game or multi-window app), so a generic flowchart isn't necessary at this stage.



Decomposition

Screen designs and usability features

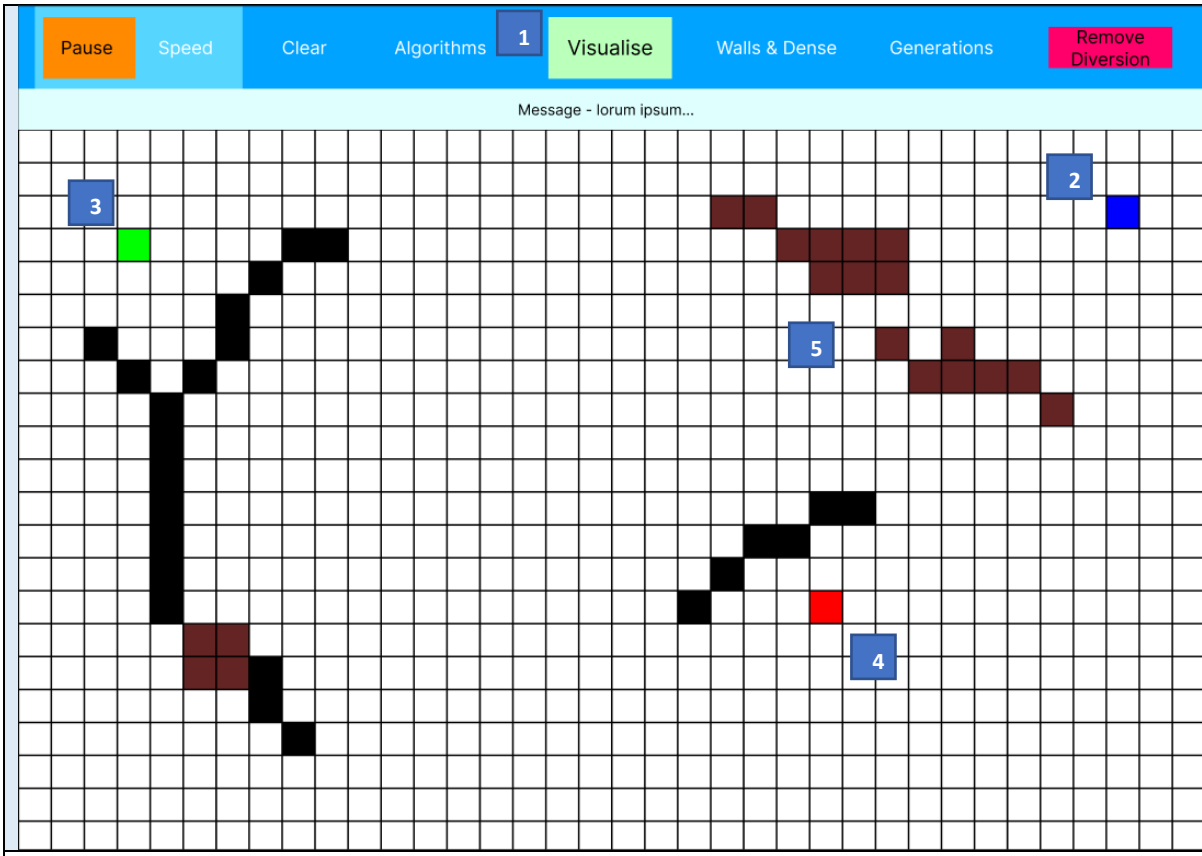
As suggested in the ***interview***: the entire program will be on one window; the toolbar will be at the top and the grid of nodes will take up most of the screen. The interviewees didn't have strong feelings about sound effects, and I concluded that there shouldn't be many in the program. However, a few may be implemented (towards the end of development) in order to provide the user with audio feedback when an important event occurs, such as a ding before a path is drawn.

We also concluded that the UI should be simple and clean, and I decided that graphics should be limited in ***proposed solution*** in order to reduce development time and not distract the user from the functionality of the program. This means that things will be represented by simple 2D geometric shapes and solid colours, more specifically, the visualisation will take place on a grid of square nodes where each node's state or function can be identified by its colour. Small animations may feature (such as gradient path drawing), but only if I have time in development at the end.

It is important that the UI is easy to use and understand so that the user can focus on the visualisation and editing aspects rather than trying to constantly understand how to use the tool, which would be taxing on the user and may demotivate them from using the product. For this reason, the UI will be as consistent and self-explanatory as possible, while also remaining clean and

decluttered. This means that the elements of the toolbar shouldn't move around, and the toolbar shouldn't contain all of the controls at once (as some aren't needed all the time). Since there are two modes (or program states), and different UI elements needed for each, it makes sense to change the function of some buttons on the toolbar depending on the mode. This way, the UI layout stays as consistent as possible while also reducing the space needed for controls. For example, a single button could be used to change modes – it will start the simulation if in editing mode and end the simulation if in simulation mode. It would be the same button, but with different text and background colours depending on the mode and the button's corresponding function. Secondly, related elements/buttons may be grouped to help the user navigate the toolbar more effectively and to make for a cleaner and more organised appearance, improving the user experience and enjoyment. Additionally, there will be a slim message prompt beneath the toolbar which will provide written feedback throughout the program stages in order to guide the user and help troubleshoot. It may also contain some colour codes to help the user understand what each node is.

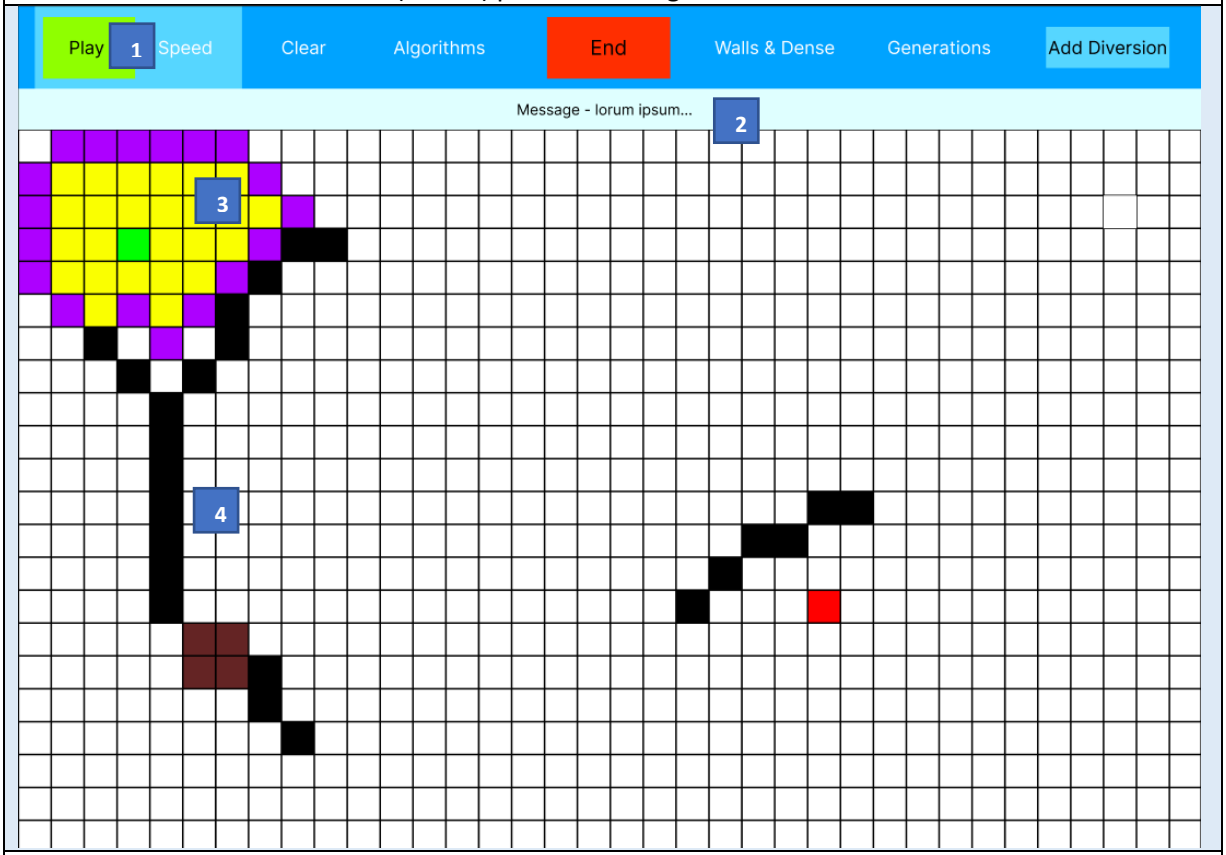
With all this in mind, I've created some mock up screen designs (see table below) which I envision the program's UI to look like at this stage. Although the eventual implementation of the UI will look similar to these designs, there is room for alterations when I code the actual program if I come across limitations, unforeseen drawbacks to the designs or think of improvements after testing how it 'feels' to navigate. For example, the colour scheme may change later on, but the main design concepts should remain.



Editing mode:

1 – The toolbar contains lots of menus and buttons. *Clear* will clear aspects of the grid, *Algorithms* will allow the user to select an algorithm to run, *Walls & Dense* allows the user to select which node they want to place, and *Generations* will provide the user with a selection of methods to procedurally generate a grid layout. In simulation mode, these menus won't have an effect. Clicking *Visualise* will

'exit' editing mode and start the simulation; the button will change colour and display 'End'. Clicking *End* will then end the simulation, and the user will be able to edit the grid again.
2 – This is the diversion node (blue); once placed, the UI will update (*Add Diversion* becomes *Remove Diversion*).
3 – This is the start node (green) which the user can move around.
4 – This is the target node (red) and can be manipulated like the start node.
5 – A collection of dense nodes (brown) placed on the grid.



Simulation mode:
1 – The control panel is where the user can determine the simulation speed and pause/play the simulation. In editing mode, the pause/play button will be inactive (greyed out an unresponsive – it could trigger a message to guide the user if they are confused).
2 – This UI strip will display messages to the user. During editing mode, it might give prompts, and in simulation mode it may display information and data or error messages.
3 – During the simulation, the node states will change to indicate which nodes have been visited (yellow nodes are in the closed set and purple ones are in the open set). The user won't be able to edit the grid while in simulation mode.
4 – These are block nodes (black), which the user has placed on the grid of nodes.

Summary of the key processes and structures

Decomposition is important in making the development of the solution easier to manage and maintain. As well as splitting the program into different modules, as I've previously shown, I intend to decompose each module further and describe how each section might work, expanding on the diagrams I produced in *thinking procedurally*.

I intend to use the Object-Oriented Programming paradigm as this program will contain many objects (nodes) and UI sections (e.g., toolbar). OOP will allow me to take a modular approach to maintain my code more effectively, reuse code to save time and decrease the chances of bugs, and

to be able to test individual parts of the program independently. By employing OOP techniques, I can develop a well written/coded solution that will be easier to read and run. An example of an OOP technique is 'encapsulation', which works by restricting access or control over class attributes by code outside the class (properties and get and set methods is a way to implement encapsulation), only giving the means to read (get) or write (set) an attribute by code outside of the class when absolutely necessary. These get and set methods allow validation to take place so that only the correct data is passed. This reduces unintentional access and alterations of class attributes, which could lead to bugs and unorganised code. Other OOP techniques, such as 'inheritance' and 'polymorphism', probably won't feature in this project, nevertheless, OOP is an effective approach and will still provide many of the benefits mentioned for this project. I plan to create four classes: *Simulation*, *Node*, *Grid* and *Toolbar*. Some classes (such as *Node* and *Toolbar*) will handle events (procedures that are called when the user interacts with the UI in a certain way, such as clicking a button) to flag relevant parts of the program, or react appropriately, in response to user interaction (events are built into the .NET Forms framework). As this project is more like a tool, as opposed to a game, utilising events will be key as user interaction will drive program flow – the program should act appropriately according to user intensions and should be intuitive and helpful to allow for that freedom (to some degree).

Below are some descriptions of the main classes that I will implement in the program. I have made and included some class diagrams which contain most of the fundamental attributes and methods of each class. However, they are not comprehensive, and the final implementations of these classes may differ and will adapt throughout the 'Development Story'. I have left events and constructor methods out of the class diagrams, but they will, of course, be implemented. I will go into further description about the methods/attributes and how they work in the next section (algorithms), but for this section, the names of the methods and attributed in the class diagrams should be enough to give a general idea of what they will do.

Form

This already created class is essential as it initialises the form and important components. It is the main program class that will contain general/miscellaneous code (such as sound) and initialisations. For example, an event will fire when the form (window) loads and is ready to build on, which will run initial code that will setup the UI (e.g., initialising the *Grid* and *Toolbar* classes).

The 'mouse up' event might be needed as the user may stop 'dragging' the mouse when not in contact with a node label, in which case this class shall detect that and flag relevant parts of the program.

Key events:

- ***Form.MouseUp***
- ***Form.Load***

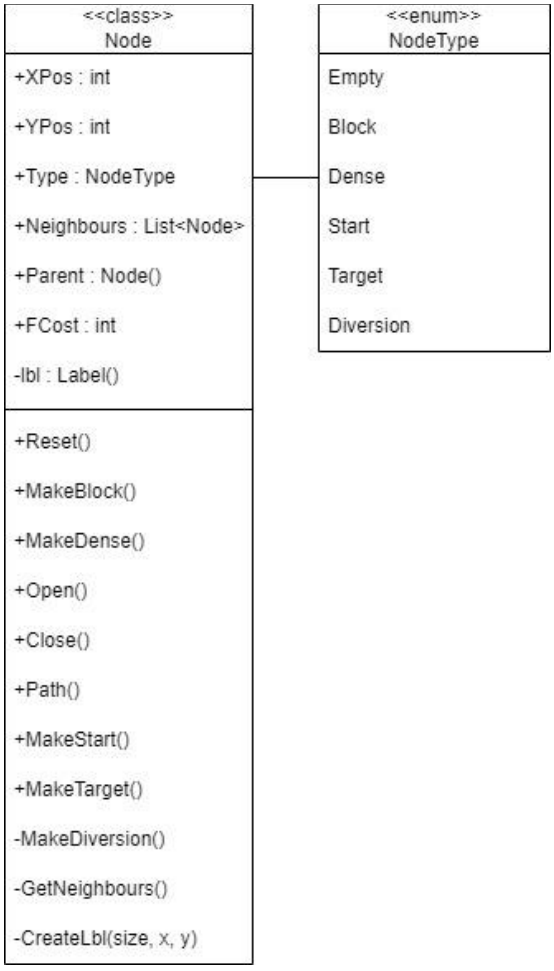
Node

Each node will have different states, types and attributes that will affect the environment and, with many independently changing within a grid, will visualise an algorithm as it searches for the shortest path. Each node will be responsible for its own appearance, and events raised by the label object (*lbl*) in this class will govern node states and appearances (the reason I choose labels to represent nodes is because they don't have any default formatting or functionality like most other controls in WinForms).

This class will contain a selection of public attributes that will allow the simulation to analyse and store information about the node while running, whilst keeping the actual node label object private. It will contain many public methods that will change the node type and appearance, as well as a couple private methods that will be called from within the constructor method to initialise a few attributes.

The *NodeType* Enum won't contain node states such as open, closed and path. This is because these states will only need to be known by the simulation class and algorithms, and the *Open()*, *Close()* and *Path()* methods will purely update the node label's appearance to visualise the algorithmic steps to the user.

To reduce the size of this class diagram, I have left out a list of private constants which may store colours for each node type and state – I have also done this because node colours may change throughout implementation.



Key events:

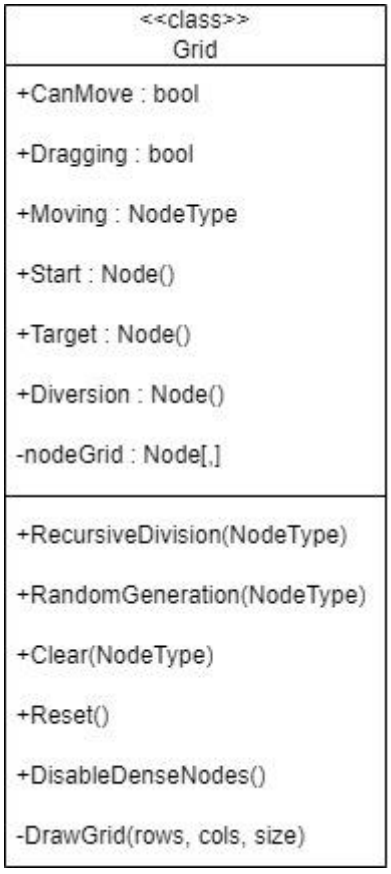
- ***lbl.MouseDown***
- ***lbl.MouseUp***
- ***lbl.MouseEnter***
- ***lbl.MouseLeave***

Grid

The grid of nodes will be dynamically created in this class. The grid will contain all the square nodes needed to visualise the algorithms during simulation.

This class will contain several attributes that will govern the behaviour of the grid in response to user interaction (such as dragging and moving nodes) and keep track of important nodes ready to be referenced during the simulation. It will privately store the grid of node objects to prevent other parts of the program editing it directly without validation. It will contain a few public methods which will change the grid in some way, such as generating a grid layout or clear aspects of the grid. There is a private method that will be called by the constructor to create the grid of nodes.

I will mention what the *DisableDenseNodes()* method does in this class as it is less obvious than the others. If the selected algorithm is 'unweighted' when the simulation starts, the public method can be called to 'disable' all the dense nodes in the grid so that the user knows they won't have an effect. Currently, I'm undecided on how I will do this – I could make each dense node almost transparent or mark it with a different colour as well as displaying a message in the output strip (I don't want to reset each dense node because the user may want to keep the grid layout for future algorithms).



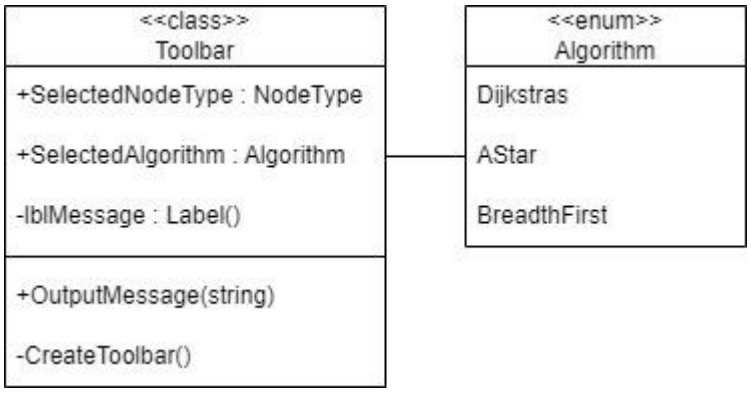
Toolbar

Within this class the toolbar UI segments will be created, using components such as labels and buttons (built-in to the WinForms framework). The toolbar will be a permanent UI element that will allow the user to manipulate the grid and control the simulation. Many events will be raised by user interaction with the toolbar. For example, buttons (*btn*) or dropdown menus (combo boxes - *cbx*) could raise events.

The class will contain a few attributes that will store important data about what the user has selected and intends to simulate, which will change as the user interacts with the UI elements. It will also contain a public method, which can be called throughout the program, that will update the message below the toolbar. The private method, *CreateToolbar()*, will be called from within the constructor when the toolbar object is made – the method will create all the controls on the toolbar and set up event listeners.

The Enum *Algorithm* will make it easier to identify the selected algorithm as the project expands because identifying them with integers or strings may cause inconsistencies or confusion.

The real class will contain several private fields which store each control on the toolbar so they can be referenced and edited from within the class – I have left this out the diagram to reduce its size because there are a lot of UI elements on the toolbar.



Key events:

- ***btnMode.Click***
- ***btnDiversion.Click***
- ***cbxNodeTypes.SelectionChangeCommitted***
- ***cbxGenerations.SelectionChangeCommitted***
- ***cbxAlgorithms.SelectionChangeCommitted***
- ***cbxClear.SelectionChangeCommitted***
- ***cbxSpeed.SelectionChangeCommitted***
- ***btnSimState.Click***

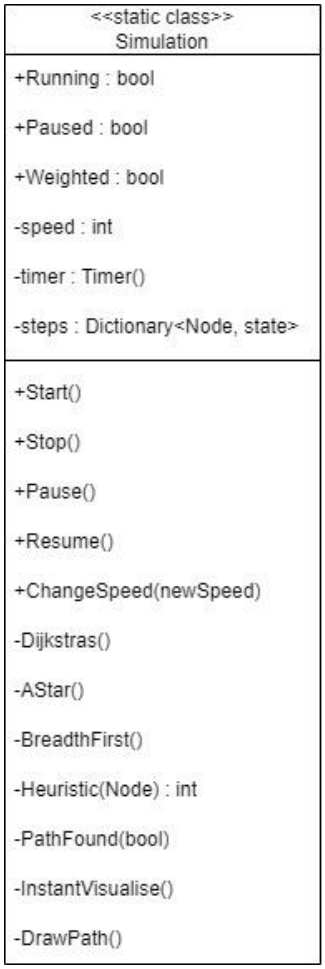
Simulation

The simulation will control the visualisation process (such as pausing/running and its speed). It will contain the pathfinding algorithms and the methods used to visualise its step-by-step processes once the shortest path has or hasn't been found.

It is a static class, which means an object of this class isn't created. It will contain a few Boolean attributes that can be updated as the user interacts with the toolbar and can be read by other classes to check if user inputs are valid, such as not allowing the user to edit a node while it is running *(Running* is true). There will be a few private fields that will contain data structures and objects needed in the visualisation stage of the simulation, which occurs after the algorithm has run. This includes a dictionary of steps taken by the algorithm and a Timer object which will loop through and display the steps at a desired speed.

I have done it this way because if I instead updated the nodes while the algorithm was running, the user wouldn't be able to see the visualisation (step-by-step processes involved) as it would find and visualise the shortest path very quickly, which would make the project pointless. Therefore, I will keep track of every process while the algorithm quickly runs and then display the results afterwards at a desired speed and with the ability to pause and resume the visualisation.

This class will also contain public methods that will edit the nature of the simulation, as well as two separate methods to stop and start it. It will contain many private methods such as the algorithms and methods to start the visualisation process (i.e. *PathFound()*). In most cases, the visualisation will occur in the *timer.Tick* event, which fires each timer 'tick', and will update the nodes one at a time using the information stored in the *steps* data structure. *InstantVisualise()* can be called when 'instant path recalculation' is taking place as it won't start the *timer* but will instead just loop through and display the steps and then call *DrawPath()* (unless the shortest path wasn't found). The private heuristic function will be used by some algorithms (A* Search) to return a 'h-cost' for a given node and known target node (known from *Target* attribute in the *Grid* class). More details about the *Heuristic()* function is given in the algorithms section.



Key events:

- ***timer.Tick***

Algorithms

This table contains pseudocode for most of the important methods that will be contained in this program. The description of each algorithm will be on the right, and the pseudocode on the left. Bear in mind that these algorithms are just ideas at this stage, and the implemented methods will probably be slightly different during development. Some of these algorithms don't fit in their boxes and so have a smaller font size.

Node

procedure MakeBlock()
  Type = NodeType.Block
  lbl.BackColor = "Black"
end procedure
 	
procedure MakeDense()
  Type = NodeType.Dense
  Weight = 15
  lbl.BackColor = "Brown"
end procedure
 	
procedure MakeStart()
  Type = NodeType.Start
  lbl.BackColor = "Green"
end procedure
 	
procedure MakeDiversion()
  Type = NodeType.Diversion
  lbl.BackColor = "Blue"
end procedure
 	
procedure MakeTarget()
  Type = NodeType.Target
  lbl.BackColor = "Red"
end procedure
	These public methods will change the node type of the node and make any corresponding adjustments to the appearance. Other classes will have to call this (such as *Grid* class when randomly generating) because they cannot update the public attribute *NodeType* directly as this wouldn't update the node appearance as well, which means the user wouldn't be aware of each node's type. It will be read-only so that an algorithm can check what node it is. When the user clicks or enters a node label while *Dragging* or *Moving* (both in the *Grid* class) is true, this method will be called if the simulation isn't running, depending on the selected node type (there is an attribute storing this in the *Toolbar* class).

procedure Reset()
  Type = NodeType.Empty
  Weight = 1
  FCost = 0
  Parent = null
  lbl.BackColor = "White"
end procedure
	This public method will reset the node by setting its node type to 'empty', updating the label appearance, and resetting any public attributes that are used by the simulation such as *FCost* and *Parent (Neighbours* won't be reset because that won't change through the program). It will be called by the *MouseDown* event if the user rights clicks, or called when each node in the grid is being reset, or when exiting simulation mode and nodes states need to be cleared (closed, open and path nodes).

procedure CreateLabel(size, x, y)
  lbl = new Label()
  lbl.Height = size
  lbl.Width = size
  lbl.Position.X = x
  lbl.Position.Y = y
	This private method will be called by the constructor when the class is instantiated. It will create the label control and set its properties such as its size and location on the form.

lbl.Border = true Label = lbl Form.Controls.Add(lbl) end procedure	
procedure Open() lbl.BackColor = "PaleGreen" end procedure procedure Close() lbl.BackColor = "Yellow" end procedure procedure Path() lbl.BackColor = "Purple" end procedure	These public methods will update the appearance of the label, which can only be edited from within the node class, to visualise algorithmic processes to the user.

These algorithms form a complete solution to the problem because there will be many instances of the node class, which will make up the bulk of the form, and so these methods will be reused many times whilst only having to write it once. The public methods in this class will be called throughout the program to update the nodes in different situations, showing there is one combined solution to the problem.

Grid

procedure DrawGrid(cols, rows, nodeSize) Array nodeGrid = new Node[][] for i = 0 to cols - 1 for j = 0 to rows - 1 node = new Node(nodeSize, i * nodeSize, j * nodeSize) nodeGrid[i][j] = node next j next i nodeGrid[1][1].MakeStart() nodeGrid[cols - 2][rows - 2].MakeTarget() end procedure	This private method will be called from within the constructor as soon as the *Grid* class has been instantiated. It will 'draw' the grid by creating an instance of each square node and setting the default grid values (such as where the start and target nodes will be initially).

procedure RandomGeneration()
rnd = new Random()
 	
for each node in nodeGrid
ranNum = rnd.Next(3)
 	
if ranNum == 2 then
node.MakeBlock()
else if ranNum == 1 then
node.MakeDense()
else
node.Reset()
end if
next
end procedure
	This public method assigns random grid nodes to be block or dense nodes. It will be called by an event in the *Toolbar* class which detects when the user selects 'Random generation' from a dropdown menu on the toolbar.

procedure DisableDenseNodes() for each node in nodeGrid if node.Type == NodeType.Dense then	This public method, in the grid class, will 'disable' all the dense nodes on the grid by making them semi-transparent. This will be

node.Transparancy = 0.5 end if next end procedure	called when breadth-first search is run, as that pathfinding algorithm is an unweighted algorithm.

procedure Clear(nodeType)
for each node in nodeGrid
if node.Type == nodeType then
node.Reset()
end if
next
end procedure
	When the 'clear grid' option is pressed, an event in the *Toolbar* class is fires which calls this public method if the simulation isn't running. It will remove all block or dense nodes, depending on the passed argument. If the 'reset grid' option is pressed instead, a different public method will be called that will set all node types to empty and set the default positions of the start and target nodes.

These algorithms form a complete solution to the problem because this class incorporates the node class – it contains a private attribute for the grid of nodes. This shows that the complete solution involves these algorithms because they will contribute to the changing of the grid's appearance by editing individual nodes, as well as holding information about the grid and its key components for the simulation class to utilise. Therefore, this class and its methods/attributes will be used everywhere and will have a main part in the complete solution.

Toolbar

procedure OutputMessage(message) lblMessage.Text = message end procedure	This is the only public method in this class. It can be called from anywhere in the program, such as when an event occurs or there is information that the user should know about, and will display the passed in message on the UI strip below the toolbar (*lblMessage*).

These algorithms form a complete solution to the problem because this class will contain most of the program's events, specifically ones involving the user's interaction with the toolbar. These events will then call methods and flag other parts of the program in response, showing that it links different parts of the program together, which forms a complete solution as the user interacts with the tool.

Simulation

procedure ChangeSpeed(newSpeed) if newSpeed == 0 then Speed = 0 timer.Interval = 500 else if newSpeed == 1 then Speed = 1 timer.Interval = 250 else if newSpeed == 2 then Speed = 2 timer.Interval = 100 else if newSpeed == 3 then Speed = 3 timer.Interval = 50 else if newSpeed == 4 then Speed = 4 timer.Interval = 25	This public method is called by an event in the *Toolbar* class which 'listens' for when the user selects a new speed from a combo box on the toolbar. The method will update the simulation speed (timer ticks per second) to the new selected speed.

end procedure	

procedure AStar()
  start = Grid.Start
  target = Grid.Target
  open = new List()
  closed = new List()
 	
  start.GCost = 0
  open.Add(start)
 	
  while true
  current = open[0]
 	
  for each node in open
  current.FCost = current.GCost + Heuristic(current, target)
  node.FCost = node.GCost + Heuristic(node, target)
 	
  if node.FCost < current.FCost then
  current = node
  else if node.FCost == current.FCost then
  if Heuristic(node, target) < Heuristic(current, target) then
  current = node
  end if
  end if
  next
 	
  open.Remove(current)
  closed.Add(current)
  Steps.Add(current, true)
 	
  if current == target then
  return
  end if
 	
  for each neighbour in current.Neighbours
  if not (neighbour.Type == NodeType.Block or
closed.Contains(neighbour)) then
  if not open.Contains(neighbour) or current.GCost +
neighbour.Weight < neighbour.GCost then
  neighbour.GCost = current.GCost + neighbour.Weight
  neighbour.Parent = current
 	
  if not open.Contains(neighbour) then
  open.Add(neighbour)
  Steps.Add(neighbour, false)
  end if
  end if
  end if
  next
  loop
end procedure
	A* Search is one of the pathfinding algorithms that the user can choose to run. It calculates the 'f-cost' of each node by summing its 'g-cost' and 'h-cost'. Although it's similar to Dijkstra's algorithm, the only difference is it generally takes less time to find the shortest path because it calculates the h-cost of each node, which is a heuristic function which returns an estimate for the 'distance' from the destination or importance of 'visiting' that node next – the h-cost is an integer value where the lower the value, the closer it is to the general direction of the destination. Just like Dijkstra's, it will then evaluate each adjacent node's f-cost and moves to the node with the lowest cost. It will be a private method because no other class needs to run it (I struggled to fit this algorithm in the table without it being unreadable).

procedure Start()
  Running = True
 	
  // run algorithms
 	
  timer.Start()
end procedure
	This public method is called by an event in the *Toolbar* class when the main toolbar button is pressed, and the simulation isn't already running. This method will call lots of methods sequentially to govern the visualisation process (such as the method of the selected algorithm) and will analyse and update class attributes (such as setting *Running* to true so that other classes 'know' the user can't edit the grid). If the simulation is already running when the button is pressed, the *Stop()* method will be called instead.

function Heuristic(current, end) colVal = Math.Abs(current.Col - end.Col)	This private function returns an integer value depending on the node passed in and the

rowVal = Math.Abs(current.Row - end.Row) return colVal + rowVal end function	location of the current target node. There are different ways of calculating the heuristic value (h-cost) of a node, but the 'Manhattan distance' is probably the best method for a grid system where each node can go four directions (except at the grid edges) like the grid in this program. Also, the Manhattan distance is an 'admissible' heuristic in this case, which means it guarantees the shortest path will be found.
procedure Pause() timer.Stop() Paused = true end procedure	If the user wishes to pause the simulation, but not end it, they can press the pause button on the toolbar and an event will fire in the *Toolbar* class which will call this public method. This method will then stop the timer to halt the visualisation process. Alternatively, if the simulation is paused and the button *(btnSimState)* is pressed (the button will say "play" on it at this point), the event will call the public method *Resume()*, which will start the timer again.
procedure DrawPath() node = Grid.Target.Parent while node != Grid.Start node.Path() node = node.Parent loop end procedure	This private method will be called once all the 'steps' have been iterated through, but only if a path has been found. It will backtrack from the target node to the start by accessing the *Parents* attribute of each node (starting with the target node), which points to the node it came from. With each node, it will call *Path()* so that the node's appearance can be updated to represent part of the path.

procedure InstantVisualise()
for each step in steps
if step.Value == "close" then
step.Key.Close()
else if step.Value = "open" then
step.Key.Open()
end if
next
 	
DrawPath()
end procedure
	This is another private method which will instantly visualise the algorithm's processes by simply iterating through *steps* and updating relevant nodes (with *Close()* and *Open()*), rather than doing this every timer tick – *DrawPath()* will still be called at the end unless a path wasn't found by the algorithm. This method will be called during 'instant path recalculation' or if the selected speed is 'instant'.

These algorithms form a complete solution to the problem because they are the main algorithms involved in running and visualising a pathfinding algorithm, which is the ultimate purpose of the program. These algorithms will use and reference other classes to display its pathfinding processes to the user (utilising the node and grid class, for example), which shows its important role in the complete solution.

Variables & Validation

Although validation is more relevant in a game or program that involves user inputs, I have compiled a list of events (event names are not final) and variables that will require validation in some way so that the user cannot do something they shouldn't be able to, or the program doesn't break because of incorrect inputs or method parameters.

Name	Type	Validation	Explanation
MouseDown()	**Event handler**	Only edits the clicked label if the simulation isn't running (i.e. *Running* is false in the *Simulation* class).	When the user clicks a label, a method can be called to change the node type and update the label's appearance. However, if the simulation is running, these methods shouldn't be called, and *Dragging* or *Moving* (depending on the node's current type) shouldn't be set to true in the *Grid* class.
MouseEnter()	**Event handler**	Only edits the label if *Dragging* is true or *Moving* is true.	When the cursor enters a label, this event will fire and should only change the node type if *Dragging* is true (i.e. the user is holding down the cursor). If the user originally clicked a start, target or diversion node, then this event should only edit this node if *Moving* is true.
ClearSelected()	**Event handler**	Only clears or resets the grid if *Running* is false.	If the user selects from the 'clear' dropdown on the toolbar, this event method will run. The event method should then only call the appropriate 'clear methods' in the *Grid* class if the simulation isn't running.
AlgorithmSelected()	**Event handler**	*SelectedAlgorithm* in the *Toolbar* class only changes if *Running* is false.	When a new algorithm is selected from the dropdown menu, the change should only be allowed/recognised if the simulation isn't running, since the running algorithm cannot change halfway through a visualisation.
GenerationSelected()	**Event handler**	Only generates a grid layout if *Running* is false.	If the user decides to select a 'generation' from the toolbar, the event fired shouldn't call the appropriate method from the *Grid* class unless the simulation isn't running.
speed	**Integer**	Value is between 0 and 5.	The speed of a simulation is determined by the user and which element of the 'speed' dropdown menu they have selected. The menu should display the speed from slowest (index 0) to fastest (index 5) and the index selected is passed to the *Simulation* class through the *ChangeSpeed()* method which will update the speed (timer 'tick event' frequency). It is important that this passed value is checked so that it is valid, otherwise the speed won't update – this is important if more speeds are added, since the value can then be higher. I was going to use an Enum to represent the

			speeds, however, it can be represented as an integer as it is intuitive that a higher number is a higher speed.

Test Data

Iterative test data

While testing the program during development, I will use the *success criteria* as a guide to inform my test plans. "White box testing" will be used for all iterative testing, which means that the code and internal structure is known to the tester (me, the developer) while testing is taking place. This is so that I can check the code I have written in each prototype works as intended, and can make immediate modifications to the code in response to any errors. White box tests are effective during iterative development because it allows me to trace variables when errors occur using IDE services such as breakpoints and watches.

This section is not a full test plan but contains the main test data that will be used throughout development - I will carry out some form of testing at the end of each prototype. I will also test validation throughout, as considered in the section above.

Test	Expected outcome and explanation	Success Criteria ID
UI elements	The form window contains a grid of square nodes (labels) and a toolbar at the top. Elements on the toolbar (such as buttons and dropdowns) are present and distinguishable.	1, 9
Moving nodes	Start, target and diversion nodes can be identified in the grid and can be moved around using the mouse pointer. The node being moved will follow the mouse as it is dragged across the grid, until the left mouse button is released **[valid].** The grid will always contain a start and target node, even if the user attempts to drag them off the grid **[invalid].** These nodes cannot be moved during the visualisation **[invalid]** but can at the end to activate instant path recalculation **[valid]**.	2
Node states	As a visualisation is taking place, node colours will change to represent node states (such as nodes in the open and closed sets). When entering editing mode, node states disappear – this is so that the user doesn't get confused if they are able to build on top of the results of a simulation.	3
Node types	By left clicking, block and dense nodes can be added **[valid]** and, by right clicking, removed from the grid **[valid]** while in editing mode, and are distinguishable from each other. They serve their respective purposes during a simulation – block nodes cannot be visited, and dense nodes have a higher cost to traverse (cost of fifteen). Block and dense nodes cannot be added or removed during a simulation **[invalid]**.	5, 6
Diversion node	This node can be added or removed from the grid (when a simulation isn't running **[invalid]**) via a button on the toolbar **[valid]**, which will update to represent whether it is being used or not. It will cause the algorithm to find the	7

	shortest path to it first before the target node – the final drawn path will go through the diversion node on its way to the target node. This is so that the user is able to create more complex scenarios for the pathfinding algorithms.	
Algorithms	Multiple algorithms, including weighted and unweighted algorithms, can be selected from the toolbar **[valid]** to be run by the simulation. The selected algorithm should find the shortest path from the start to the target node (via the diversion node if used), once a simulation is started, before its processes are then visualised. If an algorithm is unweighted, dense nodes will not have an effect during the simulation, which will be evident to the user so that they understand dense nodes won't be taken into account (this can be done by changing the appearance of any dense nodes present, for example).	4, 10, 11
Instant path	Once a visualisation is complete, start, target and diversion nodes can be moved around **[valid]** and a new path will be 'instantly' drawn, as well as updating node states. This needs to work effectively so that users can quickly simulate algorithms in many different situations.	8
Grid generations	If the user selects the 'random generation' method from the toolbar dropdown **[valid],** then grid nodes will be randomly assigned to be block or dense nodes (depending on what's selected). If the 'maze generation' method is selected **[valid],** then a maze of block or dense nodes will be generated on the grid using recursive division. This should help users who are struggling with or don't want to make their own layout.	12
Start/Stop simulation	If a simulation isn't taking place and the 'main toolbar button' is selected, a simulation will start **[valid]** - the selected algorithm is run and then the visualisation takes place. If a simulation is running, the visualisation process will stop **[valid]** and all node states will be reset (node types, like block and dense nodes, won't be affected).	14, 15
Pause/Resume	When the pause button is pressed while a simulation is running, the visualisation will halt temporarily **[valid]** (no node states change). If the button is pressed again, the visualisation process will continue as before **[valid],** since it now acts as a 'resume button'.	16, 17
Change speed	If the user selects a new speed from a dropdown on the toolbar, the number of steps per second in the visualisation process will adjust accordingly if a simulation is running **[valid]** – higher speed, more steps per second. The difference between simulation speeds should be noticeable for the user.	18
Sound	When a path is drawn during the simulation, a "DING" sound is made. If a path isn't found, a negative sound is played like a buzzer.	19
Animation	The shortest path is drawn by gradually backtracking from the target to the start node (or to the diversion, then the	20

	start node), editing the appearance of each node along the way. This should make it satisfying to use the tool.	
Dragging	If the user starts 'dragging' (i.e., holding the mouse down) while the cursor is in the grid, all nodes that the mouse then enters will change node type and appearance until the mouse button is released **[valid].** If the mouse button is released outside of the grid, it is important that it stops the 'dragging' process **[invalid].** Of course, if a simulation is running, this feature won't have an effect **[invalid].**	21
Clearing	Right clicking a node on the grid (or dragging with right click) will reset the node type **[valid],** as long as a simulation isn't running **[invalid].** Selecting a node type to clear from a dropdown on the toolbar will remove those node types from the grid **[valid].** If the user selects 'reset' from the dropdown, all nodes will be cleared, the diversion node (if on the grid) will be removed, and the start and target nodes will be set to their default position **[valid].**	13, 22

Post development test data

At the end of development, I will carry out "black box" tests, which are tests where the internal structure of the program isn't known by the tester, or isn't considered. In order to achieve this, I will involve stakeholders as they will not know how the program works and will be able to find mistakes that I overlooked.

The test data to be used here will also be composed of elements in the table above and will go through the *success criteria* to determine if the final project is 'successful'. Validation checks will be carried out in this section, which should be an unbiased process since the stakeholders won't initially know what is and isn't allowed (unlike me).

To test the final product thoroughly, I will use 'valid', 'extreme', and 'invalid' data in the final tests (only valid and extreme data should work). I have created a list of instructions which my stakeholders and testers should follow during post development testing to cover every part of the program (to make sure all the post development test data has been met), followed by a questionnaire – I can use the responses to evaluate the effectiveness and usability of the program.

Instructions

1) Load the program.
2) Add a diversion node.
3) Move the start, target and diversion nodes to new positions.
4) Generate a random grid.
5) Edit the grid by deleting and adding block and dense nodes, using the dragging feature.
6) Clear all block or dense nodes.
7) Reset the grid.
8) Generate a maze.
9) Select an algorithm and start the simulation.
10) Pause the simulation.
11) Resume the simulation.
12) Change the simulation speed (a few times).
13) Move the start, target and diversion nodes around once the visualisation is complete.

14) Stop the simulation.
15) Repeat steps 9-14 for different algorithms.

Questionnaire

- Did you generally enjoy using the program?
- Was the program intuitive and easy to use?
- What did you think of the graphics?
- Did you like the sound effects?
- Was the grid easy to manipulate?
- Was the algorithm visualisation clear?
- Was the program useful?
- Are there any improvements that could have been made?

Sign Off

Before beginning development, I have shown the stakeholders my proposed screen designs and talked them through the important parts of my solution. I have recorded some quick feedback from them (paraphrased).

- Lee generally liked the designs and solution structure, commenting on the effective use of classes to represent parts of the program and events communicate user inputs. He recognised that the screen designs are just mock ups at this stage and said they look "promising". He liked that there are many usability features, giving him an extensive list of tools to use once the program is complete.
- Jude can't really comment on anything other than the screen designs, since he isn't experienced enough to fully understand programming concepts. He really liked the design and said it looks "enticing" and "excites" him to use it in the future.
- Kieran appreciated my solution design and liked the look of the screen designs.

Since all the stakeholders are happy with the design section, and are in agreement that I can start my coded solution, I am ready to move on to the development phase of this project.

Development

Prototype 1 – UI Elements

Implementation

In this initial stage of development, I will dynamically create (code) the UI elements needed for the program – the grid of nodes and the toolbar. I will reference *screen designs and usability features* in the design section to guide my development, using the screen designs as a target that my program should look like.

Initial code

Before I start, I think it's important to mention that *Form1* is a partial class, which means that different parts of the class are written across multiple files. Usually, I will be using and showing the main file, but the screenshot below shows what another file looks like (this partial class already existed when I created the solution/project). This partial class contains a method, *InitialiseComponent* (which is called by the constructor of the class, in the main file), which sets the properties of the form and adds event listeners. I have changed the size and name of the form from within here, as well as added the load event listener – the event handler is located in the main file of the partial class.

```
3 references
partial class Form1
{
    /// <summary>
    ///  Required designer variable.
    /// </summary>
    private System.ComponentModel.IContainer components = null;

    /// <summary>
    ///  Clean up any resources being used.
    /// </summary>
    /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
    0 references
    protected override void Dispose(bool disposing)
    {
        if (disposing && (components != null))
        {
            components.Dispose();
        }
        base.Dispose(disposing);
    }

    #region Windows Form Designer generated code

    /// <summary>
    ///  Required method for Designer support - do not modify
    ///  the contents of this method with the code editor.
    /// </summary>
    1 reference
    private void InitializeComponent()
    {
        SuspendLayout();
        //
        // Form1
        //
        AutoScaleDimensions = new SizeF(8F, 20F);
        AutoScaleMode = AutoScaleMode.Font;
        ClientSize = new Size(1500, 750);
        Name = "Form1";
        Text = "Pathfinding Algorithm Visualiser";
        Load += Form1_Load;
        ResumeLayout(false);
    }
```

Grid of nodes

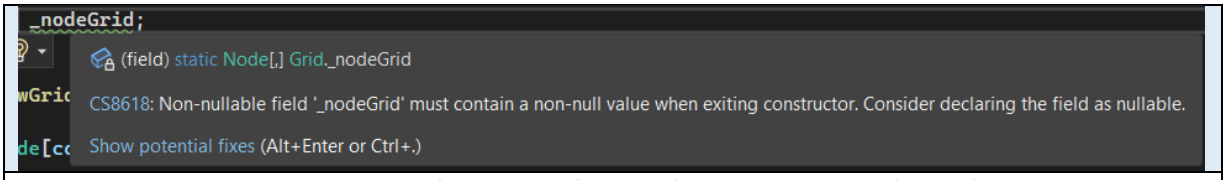
As determined in the design section, I will make a grid class and a node class. I started by coding the node class – I created a private field to store the label object which represents the 'node' on the form. The constructor calls *CreateLbl*, which assigns passed values to the label's properties and adds it to the form. Several parameters are passed to this method (which were passed to the constructor), such as its size, x-position, y-position and the form it should be added to.

```
namespace PathfindingAlgorithmVisualiser
{
    4 references
    public class Node
    {
        private Label _lbl = new();

        1 reference
        public Node(int size, int x, int y, Form form)
        {
            CreateLbl(size, x, y, form);
        }

        1 reference
        private void CreateLbl(int size, int x, int y, Form form)
        {
            // Nodes should be square, so label height and width are the same.
            _lbl.Size = new Size(size, size);
            _lbl.Location = new Point(x, y);
            _lbl.BorderStyle = BorderStyle.FixedSingle;

            // Adds the label to the form so that it's visible to the user.
            form.Controls.Add(_lbl);
        }
    }
}
```

Although I hadn't considered the grid class to be a static class during design, now that I'm coding the solution, I think the grid class should be static because there aren't multiple grid objects, and it would be difficult to access the one grid object since a few classes need access to it, which would mean I'd have to create a global variable to store the grid object, which isn't great programming practice. Having thought about it, I created the static grid class – I made a private field to store the grid of nodes in a two-dimensional array. I then wrote the public method *DrawGrid* (this method now has to be public, since a static class can't have a constructor with parameters), which creates a new node object at each index of the 2D array of nodes. Each node is created with the same size, and its position is a multiple of *i* and *j* so that each node (label) touches another. An offset is applied to the y-position of each node to account for the toolbar at the top of the screen.

```
namespace PathfindingAlgorithmVisualiser
{
    1 reference
    public static class Grid
    {
        // This field is null until 'DrawGrid' is called, so a '?' operator is needed.
        private static Node[,]? _nodeGrid;

        1 reference
        public static void DrawGrid(int cols, int rows, int nodeSize, int offsetY, Form form)
        {
            _nodeGrid = new Node[cols, rows];

            for (int i = 0; i < cols; i++)
            {
                for (int j = 0; j < rows; j++)
                {
                    // Creates a node object at each index in the 2D array 'nodeGrid'.
                    // The y-position of each node is offset to leave space for the toolbar.
                    var node = new Node(nodeSize, i * nodeSize, j * nodeSize + offsetY, form);
                    _nodeGrid[i, j] = node;
                }
            }
        }
    }
}
```

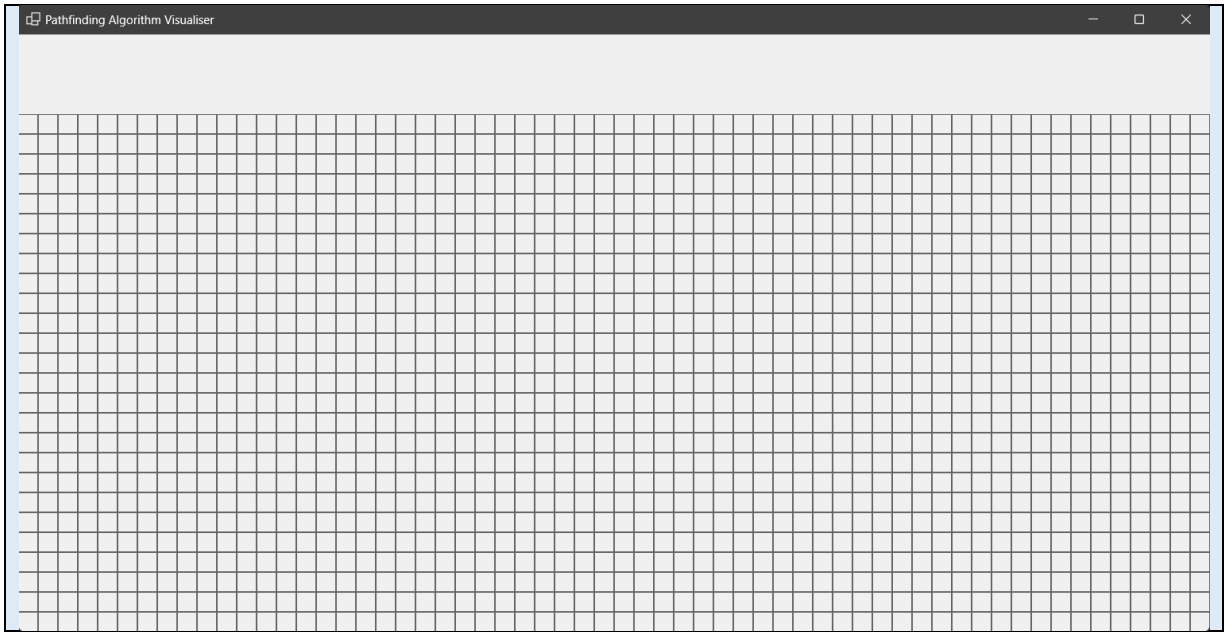
Intellisense (a prediction and suggestion feature built into the Visual Studio IDE) recommended that I made *_nodeGrid* 'nullable' since it doesn't have a value until *DrawGrid* is called. This means that the field will be null until the grid is drawn. The screenshot above shows me using the '?' operator to make the data type nullable. The screenshot below shows the *intellisense* message that appears without the nullable operator. [see **error #1** in appendix]



In order to execute the creation of the grid of nodes (i.e., call *DrawGrid*), the form needs to have loaded to be ready to handle resources and controls. Therefore, the grid is created inside the *Form_Load* event, which executes once the form is ready and functioning. This event contains some constants which store fixed values (such as node size and toolbar height) - I have used constants because it makes it impossible to accidentally edit during run-time and its value can be changed in one place, which reduces the chances of inconsistencies in the code (it also makes the code more readable). The number of grid rows and columns to be made is then calculated by taking the 'client size' of the form and dividing it wholly by the NODE_SIZE (the client height minus the height of the toolbar is used to calculate the number of rows). This means that the grid should be flush will the toolbar and the edges of the window (if the node size is a 'convenient' number).

```
namespace PathfindingAlgorithmVisualiser
{
    3 references
    public partial class Form1 : Form
    {
        1 reference
        public Form1()
        {
            InitializeComponent();
        }

        1 reference
        private void Form1_Load(object sender, EventArgs e)
        {
            const int NODE_SIZE = 25;
            const int TOOLBAR_HEIGHT = 100;

            // Calculates how many columns and rows there should be in the grid.
            int gridCols = ClientSize.Width / NODE_SIZE;
            int gridRows = (ClientSize.Height - TOOLBAR_HEIGHT) / NODE_SIZE;

            Grid.DrawGrid(gridCols, gridRows, NODE_SIZE, TOOLBAR_HEIGHT, this);
        }
    }
}
```

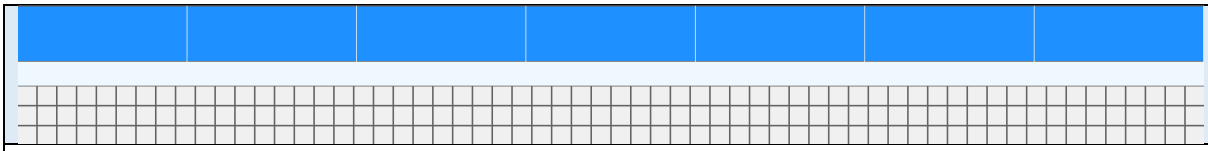
This is what the form looks like currently when I run the program...



Toolbar

For similar reasons as with the grid class, I have realised that a static toolbar class will be more efficient. Apart from that, and the fact that the method to create the toolbar will now be public, most of the features described in *summary of the key processes and structures* for the toolbar should remain the same. To start dynamically creating the toolbar from within the code (which can be quite tough), I have started by creating strips on the toolbar at regular intervals to help me visualise each section of the toolbar and adjust the values of the properties of the controls to fit their designated sections nicely. These strips are temporary – they will be removed once I have constructed the toolbar. I have also created the main bar, which is where all the controls will lie, and the message bar. Here is a preview of this initial setup – the white bars represent the boundaries of a section.

```
int mainBarHeight = height - messageHeight;
int sectionWidth = width / 7;

//TEST
for (int i = 1; i < 8; i++)
{
    var strip = new Label()
    {
        Location = new Point(sectionWidth * i, 0),
        Size = new Size(1, mainBarHeight),
    };

    form.Controls.Add(strip);
}

var mainBar = new Label()
{
    Size = new Size(width, mainBarHeight),
    BackColor = Color.DodgerBlue,
    BorderStyle = BorderStyle.FixedSingle,
};

var message = new Label()
{
    Location = new Point(0, mainBarHeight),
    Size = new Size(width, messageHeight),
    BackColor = Color.AliceBlue,
};
```



I started by creating all the private fields to hold the control objects. I assigned the new objects as well so that the field is never null, otherwise I'd have to use a nullable operator. I made each field 'read only', as suggested by *intellisense*, which means it can't be reassigned. This means I can't accidentally change the object stored in each field. I also went back and made *_lbl* in the node class a read only field, since that was also underlined by *intellisense*.

```
namespace PathfindingAlgorithmVisualiser
{
    1 reference
    public static class Toolbar
    {
        private static readonly Label _messageLbl = new();

        private static readonly ComboBox _speedCbx = new();
        private static readonly ComboBox _clearCbx = new();
        private static readonly ComboBox _algorithmsCbx = new();
        private static readonly ComboBox _nodeTypeCbx = new();
        private static readonly ComboBox _generationsCbx = new();

        private static readonly Button _modeBtn = new();
        private static readonly Button _diversionBtn = new();
        private static readonly Button _simStateBtn = new();
```

I created several methods, with each method creating a new control and adding it to the form, as well as setting its properties – this includes button text and list of items for combo boxes. I called each method from within the *CreateToolbar* method (see next caption), passing in values such as its width, height, x-position, y-position and the form to be added to – these passed values were calculated using the 'measurement variables' (the same variables used in displaying the temporary test strips), so that the controls are equally spaced and centred. The reason I made independent methods to create each control is to reduce the size of the main method, and to ensure the code is readable and well-structured for easier debugging and maintenance. Note that I also made the public method *OutputMessage* which, as described in the design section, will be referenced throughout the program to display messages to the user. I did it at this stage because I was then able to call it to set the initial message to be displayed upon opening the program.

```
1 reference
public static void OutputMessage(string message)
{
    _messageLbl.Text = message;
}

1 reference
private static void CreateMessageLbl(int width, int height, int xPos, int yPos, Form form)
{
    _messageLbl.Location = new Point(xPos, yPos);
    _messageLbl.Size = new Size(width, height);
    _messageLbl.BackColor = Color.AliceBlue;
    _messageLbl.Font = new Font(FontFamily.GenericSansSerif, 10);
    _messageLbl.TextAlign = ContentAlignment.MiddleCenter;

    form.Controls.Add(_messageLbl);
    OutputMessage("Edit the grid, select an algorithm to visualise, and then start a simulation!");
}

1 reference
private static void CreateModeBtn(int width, int height, int xPos, int yPos, Form form)
{
    _modeBtn.Location = new Point(xPos, yPos);
    _modeBtn.Size = new Size(width, height);
    _modeBtn.BackColor = Color.PaleGreen;
    _modeBtn.Font = new Font(FontFamily.GenericSansSerif, 15, FontStyle.Bold);
    _modeBtn.Text = "Visualise";

    form.Controls.Add(_modeBtn);
}
```

```
1 reference
private static void CreateDiversionBtn(int width, int height, int xPos, int yPos, Form form)
{
    _diversionBtn.Location = new Point(xPos, yPos);
    _diversionBtn.Size = new Size(width, height);
    _diversionBtn.BackColor = Color.LightBlue;
    _diversionBtn.Text = "Add Diversion";

    form.Controls.Add(_diversionBtn);
}

1 reference
private static void CreateSimStateBtn(int width, int height, int xPos, int yPos, Form form)
{
    _simStateBtn.Location = new Point(xPos, yPos);
    _simStateBtn.Size = new Size(width, height);
    _simStateBtn.BackColor = Color.Orange;
    _simStateBtn.Text = "Pause";

    form.Controls.Add(_simStateBtn);
}
```

```
1 reference
private static void CreateSpeedCbx(int width, int xPos, int yPos, Form form)
{
    _speedCbx.Location = new Point(xPos, yPos);
    _speedCbx.Width = width;
    _speedCbx.DropDownStyle = ComboBoxStyle.DropDownList;

    _speedCbx.Items.Add("Speed");
    _speedCbx.Items.Add("Very slow");
    _speedCbx.Items.Add("Slow");
    _speedCbx.Items.Add("Medium");
    _speedCbx.Items.Add("Fast");
    _speedCbx.Items.Add("Very fast");
    _speedCbx.Items.Add("Instant");
    _speedCbx.SelectedIndex = 0;

    form.Controls.Add(_speedCbx);
}

1 reference
private static void CreateClearCbx(int width, int xPos, int yPos, Form form)
{
    _clearCbx.Location = new Point(xPos, yPos);
    _clearCbx.Width = width;
    _clearCbx.DropDownStyle = ComboBoxStyle.DropDownList;

    _clearCbx.Items.Add("Clear functions");
    _clearCbx.Items.Add("Clear block nodes");
    _clearCbx.Items.Add("Clear dense nodes");
    _clearCbx.Items.Add("Reset grid");
    _clearCbx.SelectedIndex = 0;

    form.Controls.Add(_clearCbx);
}
```

```
1 reference
private static void CreateAlgorithmsCbx(int width, int xPos, int yPos, Form form)
{
    _algorithmsCbx.Location = new Point(xPos, yPos);
    _algorithmsCbx.Width = width;
    _algorithmsCbx.DropDownStyle = ComboBoxStyle.DropDownList;

    _algorithmsCbx.Items.Add("Select an algorithm");
    _algorithmsCbx.Items.Add("Dijkstra's");
    _algorithmsCbx.Items.Add("A*");
    _algorithmsCbx.Items.Add("Breadth-first");
    _algorithmsCbx.SelectedIndex = 0;

    form.Controls.Add(_algorithmsCbx);
}

1 reference
private static void CreateNodeTypeCbx(int width, int xPos, int yPos, Form form)
{
    _nodeTypeCbx.Location = new Point(xPos, yPos);
    _nodeTypeCbx.Width = width;
    _nodeTypeCbx.DropDownStyle = ComboBoxStyle.DropDownList;

    _nodeTypeCbx.Items.Add("Add node");
    _nodeTypeCbx.Items.Add("Block");
    _nodeTypeCbx.Items.Add("Dense");
    _nodeTypeCbx.SelectedIndex = 0;

    form.Controls.Add(_nodeTypeCbx);
}
```

```
1 reference
private static void CreateGenerationsCbx(int width, int xPos, int yPos, Form form)
{
    _generationsCbx.Location = new Point(xPos, yPos);
    _generationsCbx.Width = width;
    _generationsCbx.DropDownStyle = ComboBoxStyle.DropDownList;

    _generationsCbx.Items.Add("Generations");
    _generationsCbx.Items.Add("Generate block maze");
    _generationsCbx.Items.Add("Generate dense maze");
    _generationsCbx.Items.Add("Random grid layout");
    _generationsCbx.SelectedIndex = 0;

    form.Controls.Add(_generationsCbx);
}
```

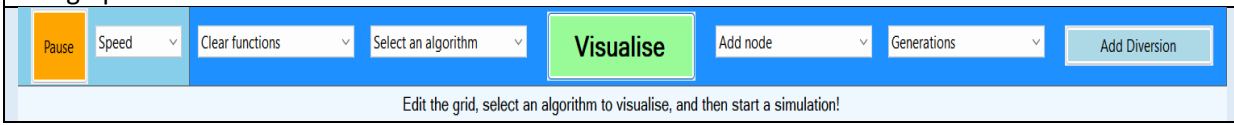
In the *CreateToolbar* method, as seen in the first code screenshot of this table, I calculated some useful measurements based on the passed values, such as the main bar height and max width of each control (width of each 'section'). I also created some labels, that have no functionality, to act as a background for parts of the toolbar - the main bar and simulation control bar. I added the 'aesthetic labels' to the form after calling all the 'creation methods' – it was important I did this last so that they are rendered 'behind' all the other controls.

```
1 reference
public static void CreateToolbar(int width, int height, int messageHeight, Form form)
{
    // Useful values for enabling consistent spacing of controls.
    int mainBarHeight = height - messageHeight;
    int sectionWidth = width / 7;

    // These two labels are just aesthetic and don't have functionality, so they are quickly made here.
    var mainBar = new Label()
    {
        Size = new Size(width, mainBarHeight),
        BackColor = Color.DodgerBlue,
        BorderStyle = BorderStyle.FixedSingle,
    };

    var simControlBar = new Label()
    {
        Size = new Size(sectionWidth, mainBarHeight),
        BackColor = Color.SkyBlue,
        BorderStyle = BorderStyle.FixedSingle,
    };

    // Each method creates a toolbar control.
    CreateMessageLbl(width, messageHeight, 0, mainBarHeight, form);

    CreateModeBtn(sectionWidth - 30, mainBarHeight - 10, (sectionWidth * 3) + 15, 5, form);
    CreateDiversionBtn(sectionWidth - 30, mainBarHeight / 2, (sectionWidth * 6) + 15, mainBarHeight / 4, form);
    CreateSimStateBtn(sectionWidth / 3, mainBarHeight - 4, sectionWidth / 12, 2, form);

    CreateSpeedCbx(sectionWidth / 2, (sectionWidth / 2) - 10, mainBarHeight / 4, form);
    CreateClearCbx(sectionWidth - 20, sectionWidth + 10, mainBarHeight / 4, form);
    CreateAlgorithmsCbx(sectionWidth - 20, (sectionWidth * 2) + 10, mainBarHeight / 4, form);
    CreateNodeTypeCbx(sectionWidth - 20, (sectionWidth * 4) + 10, mainBarHeight / 4, form);
    CreateGenerationsCbx(sectionWidth - 20, (sectionWidth * 5) + 10, mainBarHeight / 4, form);

    // 'simControlBar' is added to the form after all the controls are created so that 'simStateBtn' and 'speedCbx' can be seen.
    form.Controls.Add(simControlBar);
    // 'mainBar' is added to the form last so that it is at the 'back' and all other controls can be seen.
    form.Controls.Add(mainBar);
}
```

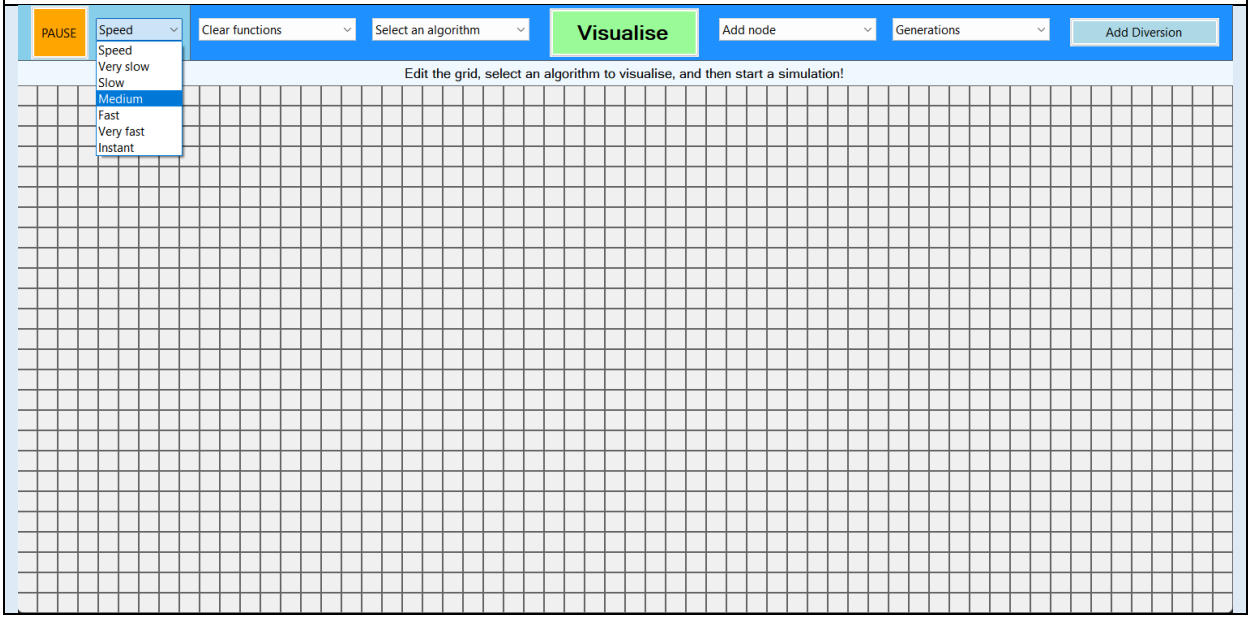
I then had to call this public method from within the *Fom1* class, so that it loads with the grid. As a side note, I also called a method in the constructor to centre the form on the screen when it loads.

```
3 references
public partial class Form1 : Form
{
    1 reference
    public Form1()
    {
        InitializeComponent();
        CenterToScreen();
    }

    1 reference
    private void Form1_Load(object sender, EventArgs e)
    {
        const int NODE_SIZE = 25;
        const int TOOLBAR_HEIGHT = 100;
        const int MESSAGE_HEIGHT = 30;

        // Calculates how many columns and rows there should be in the grid.
        int gridCols = ClientSize.Width / NODE_SIZE;
        int gridRows = (ClientSize.Height - TOOLBAR_HEIGHT) / NODE_SIZE;

        Grid.DrawGrid(gridCols, gridRows, NODE_SIZE, TOOLBAR_HEIGHT, this);
        Toolbar.CreateToolbar(ClientSize.Width, TOOLBAR_HEIGHT, MESSAGE_HEIGHT, this);
    }
}
```

Here is the resulting toolbar, looking quite similar to the original screen designs I made during the design phase.



This is what the form now looks like as a whole, with one of the combo boxes open...



Testing

Test table

Test	Test details	Expected result	Actual result	Success /Fail	Comments
Opening form – Success Criteria: 1, 9	I ran the program.	The grid and toolbar load in.	The grid and toolbar load in.	**Success**	The grid takes a while to load in because there are so many label controls.
Toolbar interaction – Success Criteria: 9	Clicked the three toolbar buttons.	Buttons can be clicked.	Each buttons clicks in.	**Success**	Click animation occurs.
	Expanded each combo box.	Combo boxes display a drop-down list.	Combo boxes display all necessary items.	**Success**	The first index of each combo box is just a description of the list.

Modifications

Despite all the tests being passed, there are a few negatives with this prototype.

One negative is the grid takes a while to load in, and the toolbar doesn't appear until the grid loads and renders. In order to reduce the loading time, I have decided to instead call *CreateToolbar* before *DrawGrid* from within the form class, which does appear to cut down the loading time a bit, since the toolbar now loads in whilst the grid is rendering.

Secondly, I couldn't find a way to display a message on each combo box to describe what items it contains, so I had to make the description one of the combo box items and then select that item from within the code, so that it displays when the toolbar is created. The downside of this is that the user may get confused, as one of the items is just a description. However, the description is always

the first element in the list, and it should be apparent they don't do anything – it's important I get this validation right in the future prototypes.

Review

User feedback

I have asked the main stakeholders (interviewees) some questions about this prototype and recorded their responses.

Q: What are your initial thoughts about the user interface?

- **Lee**: "I'm very happy with the look of the form, it is very similar to the proposed screen designs. It is very appealing and well organised and structured."
- **Jude**: "It looks really cool, and I like that the grid is nice and big. I'm excited to start making some patterns on the grid."

Q: Does this match what you imagined the form to look like?

- **Lee**: "Pretty much. I would've preferred the nodes to be a bit bigger, but we'll see if its finicky in the coming prototypes when you're able to manipulate the grid."
- **Jude**: "Yes."

Q: Is there anything you would change at this stage?

- **Lee**: "No."
- **Jude**: "Nothing. I'm looking forward to the next stages."

I gathered from this that the stakeholders were generally happy with this prototype, so no further alterations need to be made to the UI.

Reflection

I have successfully produced a prototype that displays both the grid of nodes and toolbar upon running the program. The toolbar is interactive, but not functional yet - I will gradually add functionality to each control in further prototypes. The grid is flush with the toolbar, true to the screen designs I created in the design section, and the toolbar controls are aligned, equally spaced, and lightly coloured to create an attractive, yet simple appearance. I also learned more about nullable data types, which I will now utilise throughout development. I think it's logical to move onto implementing node types and adding functionality to the grid next.

Prototype 2 – Node Types & Grid Interaction

Implementation

In this prototype, I will aim to implement all the node types and allow the user to interact with the nodes on the grid. I will use the class diagrams and solution structures I developed in the design section to guide my coding.

Node types

First and foremost, I had to create the *NodeType* Enum to store the possible node types. I then created the *Type* property in the node class to store the node's type. The property can be read anywhere in the program, such as when an algorithm is analysing each node, but can only be set within the node class, which is why it has a 'get' and 'private set' method. I also set the node type to 'empty' in the constructor, as each node will be empty when first created. To note, I added a line of code in the *DrawLabel* method which sets the label's back colour to "White Smoke", which is an off white (to reduce eye strain) - this will be the colour of an empty node.

```
public enum NodeType
{
    Start,
    Target,
    Diversion,
    Block,
    Dense,
    Empty
}
```

```
public class Node
{
    7 references
    public NodeType Type { get; private set; }

    private readonly Label _lbl = new();

    1 reference
    public Node(int size, int x, int y, Form form)
    {
        Type = NodeType.Empty;
        CreateLbl(size, x, y, form);
    }
```

The reason I don't want other classes editing a node's type is because updating the label's appearance comes with that. So, to reduce the likelihood of a node's type changing without the label's back colour updating, I have created public methods that will update the node type and change the label's back colour at the same time. These methods are public because they are needed by the grid class for grid generations or clearing. *MakeDiversion* is private, however, because it is never needed outside the events in this class.

```
0 references
public void Reset()
{
    Type = NodeType.Empty;
    _lbl.BackColor = Color.WhiteSmoke;
}

0 references
public void MakeStart()
{
    Type = NodeType.Start;
    _lbl.BackColor = Color.Green;
}

0 references
public void MakeTarget()
{
    Type = NodeType.Target;
    _lbl.BackColor = Color.Red;
}

0 references
public void MakeBlock()
{
    Type = NodeType.Block;
    _lbl.BackColor = Color.Black;
}

0 references
public void MakeDense()
{
    Type = NodeType.Dense;
    _lbl.BackColor = Color.Brown;
}

0 references
private void MakeDiversion()
{
    Type = NodeType.Diversion;
    _lbl.BackColor = Color.Blue;
}
```

Having tested that these methods work, I then moved over to the grid class to add some code at the end of the *DrawGrid* method that will choose the initial nodes to set as the start and target nodes. It is important that the grid always has one start and one target node on it at all times. I also added

public attributes that store the start, target and diversion nodes, so that classes can keep track of where each of the core nodes are. As mentioned before, this is a static class so its attributes should be nullable since there is no constructor class to set the default values. However, once the grid is drawn, only the diversion attribute can be null thereafter, since the start and target nodes cannot be removed from the grid.

```
7 references
public static class Grid
{
    2 references
    public static Node? Start { get; set; }
    2 references
    public static Node? Target { get; set; }
    2 references
    public static Node? Diversion { get; set; }

    // This field is null until 'DrawGrid' is called, so a '?' operator is needed.
    private static Node[,]? _nodeGrid;

    1 reference
    public static void DrawGrid(int cols, int rows, int nodeSize, int offsetY, Form form)
    {
        _nodeGrid = new Node[cols, rows];

        for (int i = 0; i < cols; i++)
        {
            for (int j = 0; j < rows; j++)
            {
                // Creates a node object at each index in the 2D array 'nodeGrid'.
                // The y-position of each node is offset to leave space for the toolbar.
                var node = new Node(nodeSize, i * nodeSize, j * nodeSize + offsetY, form);
                _nodeGrid[i, j] = node;
            }
        }

        // Setting the initial positions of the start and target nodes.
        _nodeGrid[cols / 6, rows / 2].MakeStart();
        _nodeGrid[(cols / 6) * 5, rows / 2].MakeTarget();
    }
}
```

Now I had to go back to the node class and add in two lines of code in each of *MakeStart*, *MakeTarget*, and *MakeDiversion*. For each of these methods, I had to reset the node of the respective attribute in the grid class, as well as set the value of that attribute to 'this' node, which is being changed. This is so that there is only one of each on the grid at any given time. However, when I did this and ran the program, an exception was thrown. This is because I tried to call *Reset* on the node stored in the start attribute of the grid class, but it was null as the start node hadn't been 'added' to the grid yet. I should have noticed the warning, as seen with the green scribble in the *MakeTarget* method. [see **error #2** in appendix]

```
1 reference
public void MakeStart()
{
    Type = NodeType.Start;
    _lbl.BackColor = Color.Green;

    Grid.Start.Reset();
    Grid.Start = this;
}

1 reference
public void MakeTarget()
{
    Type = NodeType.Targ
    _lbl.BackColor = Col

    Grid.Target.Reset();
    Grid.Target = this;
}

0 references
```

Exception Thrown

System.NullReferenceException: 'Object reference not set to an instance of an object.'

PathfindingAlgorithmVisualiser.Grid.Start.**get** returned null.

Show Call Stack | View Details | Copy Details | Start Live Share session

Exception Settings

Break when this exception type is thrown

Except when thrown from:

PathfindingAlgorithmVisualiser.dll

Open Exception Settings | Edit Conditions

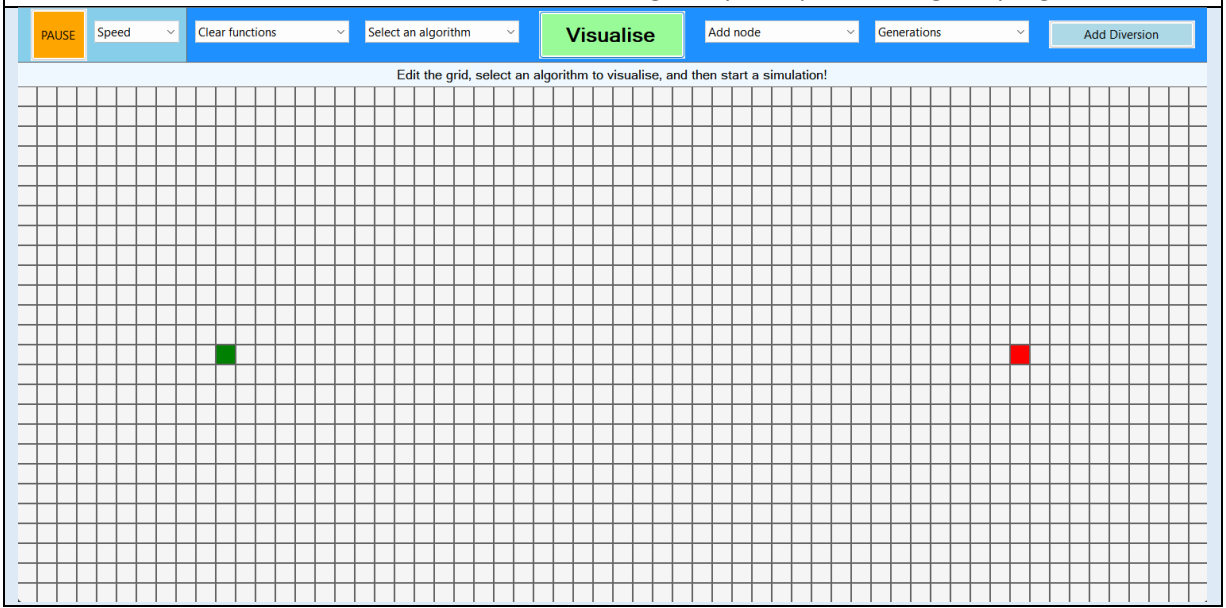
To fix this, I first checked if the grid attribute being changed was null, and if it wasn't I could then reset the node that was stored there. I did this using a 'null-conditional operator' (?.) which evaluates the attribute and only performs the method if it is non-null. Either way, the corresponding attribute will store the new node in each method.

```
1 reference
public void MakeStart()
{
    Type = NodeType.Start;
    _lbl.BackColor = Color.Green;

    Grid.Start?.Reset();
    Grid.Start = this;
}
```

```
1 reference
public void MakeTarget()
{
    Type = NodeType.Target;
    _lbl.BackColor = Color.Red;

    Grid.Target?.Reset();
    Grid.Target = this;
}
```

```
0 references
private void MakeDiversion()
{
    Type = NodeType.Diversion;
    _lbl.BackColor = Color.Blue;

    Grid.Diversion?.Reset();
    Grid.Diversion = this;
}
```

This is what the form now looks like with the default grid layout upon running the program...



Adding and dragging

The next thing to do is implement the label events so that the user can interact with the grid. Before I did this, I added an event to the 'node type combo box' in the toolbar class, which fires when the user selects a new index from the list. I created an event handler which will run when the event fires and will update a public property (which cannot be 'set' from outside the class) that stores the selected node type. If no node is selected, then this property will be null.

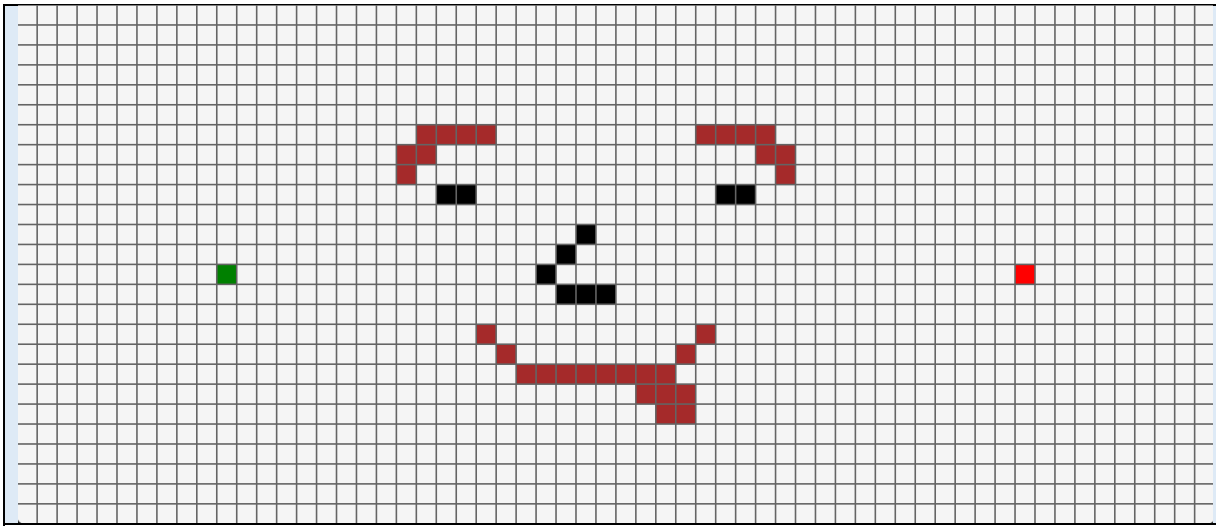
```
3 references
public static class Toolbar
{
    5 references
    public static NodeType? SelectedNodeType { get; private set; }
```

```
1 reference
private static void CreateNodeTypeCbx(int width, int xPos, int yPos, Form form)
{
    _nodeTypeCbx.Location = new Point(xPos, yPos);
    _nodeTypeCbx.Width = width;
    _nodeTypeCbx.DropDownStyle = ComboBoxStyle.DropDownList;

    _nodeTypeCbx.Items.Add("Add node");
    _nodeTypeCbx.Items.Add("Block");
    _nodeTypeCbx.Items.Add("Dense");
    _nodeTypeCbx.SelectedIndex = 0;

    _nodeTypeCbx.SelectionChangeCommitted += NodeTypeCbx_SelectionChangeCommitted;

    form.Controls.Add(_nodeTypeCbx);
}

1 reference
private static void NodeTypeCbx_SelectionChangeCommitted(object? sender, EventArgs e)
{
    // Determines which node the user wants to add to the grid.
    if (_nodeTypeCbx.SelectedIndex == 1)
        SelectedNodeType = NodeType.Block;
    else if (_nodeTypeCbx.SelectedIndex == 2)
        SelectedNodeType = NodeType.Dense;
    else
        SelectedNodeType = null;
}
```

I then created an event handler, in the node class, for the 'mouse down' event which fires when the user clicks a mouse button over the label. Within this handler method, I first check if the node is a start, target or diversion node. If it isn't, I then check which mouse button was clicked using the passed in parameter, *e*. If the left mouse button was clicked, I check the value of the toolbar's *SelectedNodeType* property – if 'block' is selected then *MakeBlock* is called and if 'dense' is selected then *MakeDense* is called. If the right mouse button was clicked instead, then *Reset* is called.

```
1 reference
private void CreateLbl(int size, int x, int y, Form form)
{
    // Nodes should be square, so label height and width are the same.
    _lbl.Size = new Size(size, size);
    _lbl.Location = new Point(x, y);
    _lbl.BorderStyle = BorderStyle.FixedSingle;
    _lbl.BackColor = Color.WhiteSmoke;

    _lbl.MouseDown += Lbl_MouseDown;

    // Adds the label to the form so that it's visible to the user.
    form.Controls.Add(_lbl);
}

1 reference
private void Lbl_MouseDown(object? sender, MouseEventArgs e)
{
    // Only updates the label if it isn't a start, target or diversion node.
    if (Type != NodeType.Start && Type != NodeType.Target && Type != NodeType.Diversion)
    {
        if (e.Button == MouseButtons.Left)
        {
            if (Toolbar.SelectedNodeType == NodeType.Block)
                MakeBlock();
            else if (Toolbar.SelectedNodeType == NodeType.Dense)
                MakeDense();
        }
        else if (e.Button == MouseButtons.Right)
            Reset();
    }
}
```

Although I can only edit one node at a time at this point, I created some art as a quick test. Block nodes are black, and dense nodes are brown.



In order to improve ease of use and to make the manipulation of the grid more intuitive, I will implement a dragging feature that allows the user to add or remove multiple nodes at once, as initially highlighted in the ***interview*** during project analysis and later incorporated into the project design. To start, I added two properties to the grid class – *Dragging* and *ButtonHolding*. *Dragging* is a Boolean which I intend to be true when a mouse button is down, and false otherwise. *ButtonHolding* is a *MouseButtons* Enum which will store which mouse button is being held down – it is null when no mouse button is being pressed.

```
3 references
public static bool Dragging { get; set; }
4 references
public static MouseButtons? ButtonHolding { get; set; }
```

Next, I added more events to the label in the node class – *MouseEnter* and *MouseUp*. I also created a private method (*UpdateLbl*) that will update the label depending on the mouse button passed in – it is basically the same code used in the *Lbl_MouseDown* event handler in the earlier section, but I am able to reuse the code if I put it in a method and then call it within multiple event handlers. So *Lbl_MouseDown* will now call *UpdateLbl*, and then set *Grid.Dragging* to true and Grid.*ButtonHolding* to *e.button*. When the mouse enters a label, the event will call *UpdateLbl* if *Grid.Dragging* is true and pass in the value of *ButtonHolding*, which is the mouse button currently being pressed. When the mouse button is released, the event will set *Grid.Dragging* to false and *Grid.ButtonHolding* to null, which means that the 'Mouse Enter event' won't do anything since the user has stopped 'dragging'. Here is the code for all the label events, event handlers, and the 'label updating method' I described.

```
// Binding label events and handlers.
_lbl.MouseDown += Lbl_MouseDown;
_lbl.MouseEnter += Lbl_MouseEnter;
_lbl.MouseUp += Lbl_MouseUp;
```

```
1 reference
private void Lbl_MouseUp(object? sender, MouseEventArgs e)
{
    // User has stopped dragging.
    Grid.Dragging = false;
    Grid.ButtonHolding = null;
}

1 reference
private void Lbl_MouseEnter(object? sender, EventArgs e)
{
    // The label will only be updated if the user is 'dragging'.
    if (Grid.Dragging)
    {
        if (Grid.ButtonHolding == MouseButtons.Left)
            UpdateLbl(MouseButtons.Left);
        else if (Grid.ButtonHolding == MouseButtons.Right)
            UpdateLbl(MouseButtons.Right);
    }
}

1 reference
private void Lbl_MouseDown(object? sender, MouseEventArgs e)
{
    UpdateLbl(e.Button);

    // Initiates the dragging process.
    Grid.Dragging = true;
    Grid.ButtonHolding = e.Button;
}
```

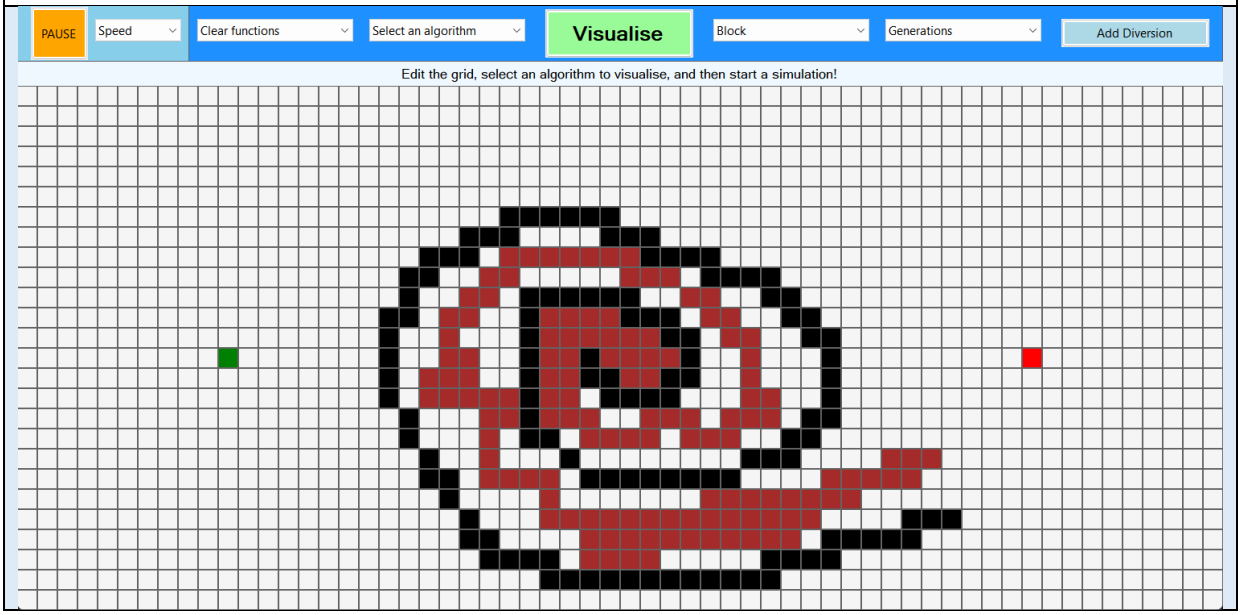
```
3 references
private void UpdateLbl(MouseButtons button)
{
    // Only updates the label if it isn't a start, target or diversion node.
    if (Type != NodeType.Start && Type != NodeType.Target && Type != NodeType.Diversion)
    {
        if (button == MouseButtons.Left)
        {
            if (Toolbar.SelectedNodeType == NodeType.Block)
                MakeBlock();
            else if (Toolbar.SelectedNodeType == NodeType.Dense)
                MakeDense();
        }
        else if (button == MouseButtons.Right)
            Reset();
    }
}
```

At this point I decided to test the dragging feature before moving on. The dragging feature didn't work, which stumped me for a while until I realised the problem didn't lie in my code. After doing some research into the C# documentation, I learned that the *MouseEnter* event doesn't fire while a mouse button is held down because the label that was clicked 'captures' the mouse until the mouse is no longer down, preventing other labels from firing the *MouseEnter* event when the mouse enters them. After looking on Stack Overflow (a developer Q&A site/forum) for inspiration about a possible fix for this 'bug' (or annoying WinForms feature), I managed to find someone who mentioned that you can

manually make the 'capture' property of a control false. I implemented this idea into my project by making the label control's capture property false as soon as the *MouseDown* event fires, which means that *MouseEnter* events should now fire even when the mouse button is held down. In order to do this, I had to convert the *sender* (object which the event applies to) variable of type 'object' to type 'control'. [see **error #3** in appendix]

```
1 reference
private void Lbl_MouseDown(object? sender, MouseEventArgs e)
{
    // This makes it so that other labels can listen for 'MouseEnter' event while the mouse is still pressed.
    if (sender is Control control)
        control.Capture = false;

    UpdateLbl(e.Button);

    // Initiates the dragging process.
    Grid.Dragging = true;
    Grid.ButtonHolding = e.Button;
}
```

The dragging feature now works – I made a chocolate spiral pattern of block and dense nodes.



Moving nodes

It is vital that the user can move the start, target and diversion nodes around the grid, and has been an important feature mentioned throughout the analysis and design sections. To begin, I added a property to the grid class, *Moving*, which stores the node type wanting to be moved by the user – the property is nullable as the user won't always want to move a node.

```
7 references
public static NodeType? Moving { get; set; }
```

Next, I had to add an if-statement into the *MouseDown* event handler in the node class which checked if the node pressed is a start, target or diversion node. If it is, it then sets the *Moving* property of the grid class to the node type of that node, but only if the button pressed is the left mouse button. If it isn't one of the 'core' nodes, it calls *UpdateLbl*, passing the mouse button pressed as a parameter, and then sets the *ButtonHolding* grid property to the button pressed. I removed the check for a start, target or diversion node from the *UpdateLbl* method, since the check has already taken place in the event handler. Within the *MouseEnter* event handler, there is a check if the node is a 'core' node, and if it isn't, a selection process takes place. If *Grid.Moving* is non-null, the relevant 'make' method is called, otherwise *UpdateLabel* is called, passing the value of *Grid.ButtonHolding*. I added a line of code to the *MouseUp* event handler that makes the grid property *Moving* null.

```
1 reference
private void Lbl_MouseDown(object? sender, MouseEventArgs e)
{
    // This makes it so that other labels can listen for 'MouseEnter' event while the mouse is still pressed.
    if (sender is Control control)
        control.Capture = false;

    // Determines if the user is moving or adding a node.
    if (Type == NodeType.Start || Type == NodeType.Target || Type == NodeType.Diversion)
    {
        if (e.Button == MouseButtons.Left)
            Grid.Moving = Type;
    }
    else
    {
        UpdateLbl(e.Button);
        Grid.ButtonHolding = e.Button;
    }

    // Initiates the dragging process.
    Grid.Dragging = true;
}

2 references
private void UpdateLbl(MouseButtons? button)
{
    // Edits node based on which mouse button is pressed, and which node type is selected from the toolbar.
    if (button == MouseButtons.Left)
    {
        if (Toolbar.SelectedNodeType == NodeType.Block)
            MakeBlock();
        else if (Toolbar.SelectedNodeType == NodeType.Dense)
            MakeDense();
    }
    else if (button == MouseButtons.Right)
        Reset();
}
```

```
1 reference
private void Lbl_MouseUp(object? sender, MouseEventArgs e)
{
    // User has stopped dragging.
    Grid.Dragging = false;
    Grid.ButtonHolding = null;
    Grid.Moving = null;
}

1 reference
private void Lbl_MouseEnter(object? sender, EventArgs e)
{
    // The label will only be updated if the user is 'dragging' and not a 'core' node.
    if (Grid.Dragging && Type != NodeType.Start && Type != NodeType.Target && Type != NodeType.Diversion)
    {
        if (Grid.Moving == NodeType.Start)
            MakeStart();
        else if (Grid.Moving == NodeType.Target)
            MakeTarget();
        else if (Grid.Moving == NodeType.Diversion)
            MakeDiversion();
        else
            UpdateLbl(Grid.ButtonHolding);
    }
}
```

The ‘moving system’ worked upon testing – both the start and target nodes move around if you select them and drag (with the left mouse button), and they aren’t deleted when you drag over them. However, if you move the start or target nodes over block or dense nodes and then move it away, those nodes are reset. This effect isn’t desirable, and I could imagine the user getting frustrated if parts of their layout was removed after moving one of these nodes. I didn’t anticipate this during the design phase, but I managed to quite easily come up with a solution. I started by adding a new property in the node class that stores the previous node type. Within the *MakeStart*, *MakeTarget*, and *MakeDiversion* methods, I set the previous node type to be the current node type before changing the current node type. Then I added an if-statement that checks the value of the current core node’s previous node type, and if it was a block or dense node, then the appropriate ‘make

methods' are called to make it back into its previous node type, otherwise it is reset. The respective grid attribute is then updated to point this node, as usual.

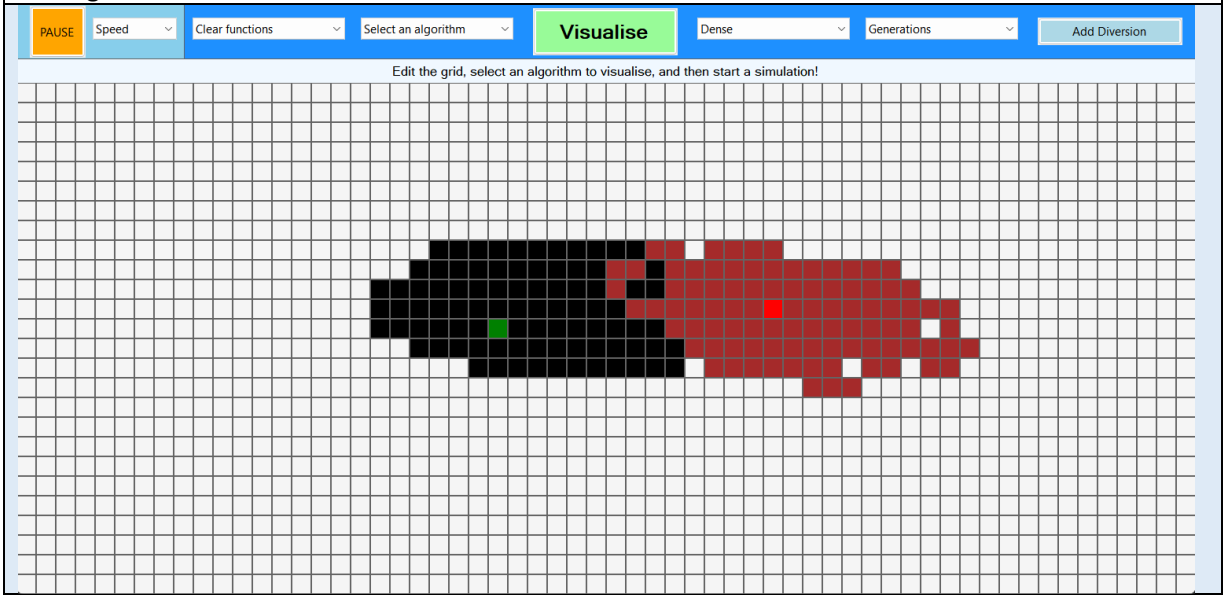
```
9 references
public NodeType PreviousNodeType { get; private set; }
```

```
2 references
public void MakeStart()
{
    PreviousNodeType = Type;
    Type = NodeType.Start;

    _lbl.BackColor = Color.Green;

    if (Grid.Start is not null)
    {
        if (Grid.Start.PreviousNodeType == NodeType.Block)
            Grid.Start.MakeBlock();
        else if (Grid.Start.PreviousNodeType == NodeType.Dense)
            Grid.Start.MakeDense();
        else
            Grid.Start.Reset();
    }

    Grid.Start = this;
}
```

```
2 references
public void MakeTarget()
{
    PreviousNodeType = Type;
    Type = NodeType.Target;

    _lbl.BackColor = Color.Red;

    if (Grid.Target is not null)
    {
        if (Grid.Target.PreviousNodeType == NodeType.Block)
            Grid.Target.MakeBlock();
        else if (Grid.Target.PreviousNodeType == NodeType.Dense)
            Grid.Target.MakeDense();
        else
            Grid.Target.Reset();
    }

    Grid.Target = this;
}
```

```
1 reference
private void MakeDiversion()
{
    PreviousNodeType = Type;
    Type = NodeType.Diversion;

    _lbl.BackColor = Color.Blue;

    if (Grid.Diversion is not null)
    {
        if (Grid.Diversion.PreviousNodeType == NodeType.Block)
            Grid.Diversion.MakeBlock();
        else if (Grid.Diversion.PreviousNodeType == NodeType.Dense)
            Grid.Diversion.MakeDense();
        else
            Grid.Diversion.Reset();
    }

    Grid.Diversion = this;
}
```

Although it's difficult to prove this works using screenshots, it will be shown in the post development testing video.

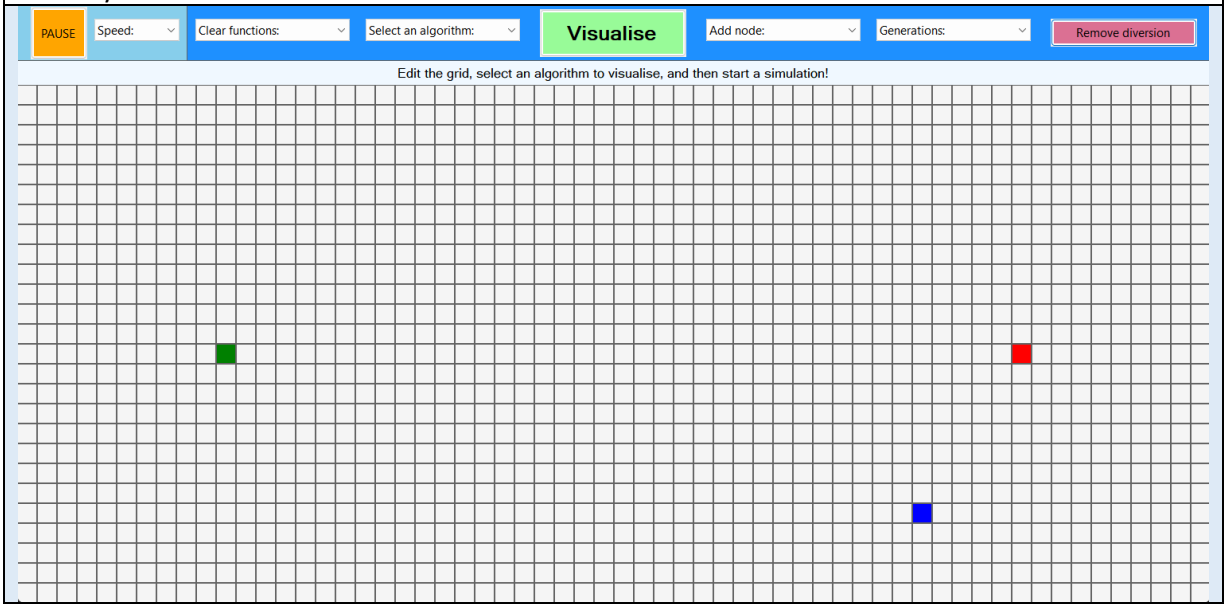


Diversion node

The final part of this prototype is allowing the user to introduce and remove a diversion node. This was quite simple since the code for moving the diversion node has already been written in the previous parts. I added an event to the "Add diversion" button on the toolbar, which executes when the user clicks the button, and first checks if the left mouse button was pressed. If it was, it then checks if the diversion node is on the grid or not. If the diversion node isn't on the grid, it will activate the moving feature by setting the grid attributes *Dragging* to true and *Moving* to *NodeType.Diversion* - when the user clicks a node, the diversion node will stop moving with the mouse, since the *MouseUp* event fires. It will then update the button's text to "Remove diversion" and change its background colour. If a diversion node is on the grid, it will reset that node on the grid; the button text displays "Add diversion" and the back colour changes. Side note: I added colons to the end of the first element in each drop-down list on the toolbar to make it clearer that it is a description element and has no functionality.

```
1 reference
private static void CreateDiversionBtn(int width, int height, int xPos, int yPos, Form form)
{
    _diversionBtn.Location = new Point(xPos, yPos);
    _diversionBtn.Size = new Size(width, height);
    _diversionBtn.BackColor = Color.LightBlue;
    _diversionBtn.Text = "Add Diversion";

    _diversionBtn.MouseClick += DiversionBtn_MouseClick;

    form.Controls.Add(_diversionBtn);
}

1 reference
private static void DiversionBtn_MouseClick(object? sender, MouseEventArgs e)
{
    // Determines whether the user is adding or removing the diversion node.
    if (e.Button == MouseButtons.Left)
    {
        if (Grid.Diversion is null)
        {
            Grid.Moving = NodeType.Diversion;
            Grid.Dragging = true;

            _diversionBtn.Text = "Remove diversion";
            _diversionBtn.BackColor = Color.PaleVioletRed;
        }
        else
        {
            Grid.Diversion.Reset();
            Grid.Diversion = null;

            _diversionBtn.Text = "Add diversion";
            _diversionBtn.BackColor = Color.LightBlue;
        }
    }
}
```

This is what the grid looks like with a diversion node (blue node). Notice how the 'diversion button' now says "Remove diversion" ...



Testing

Test table

Test	Test details	Expected result	Actual result	Success /Fail	Comments
Initial grid layout – Success Criteria: 1	Ran the program and observed the grid.	The grid contains a start and target node.	The grid contains a green start and red target node.	**Success**	The start node is on the left side of the grid, and the target node on the right.
Adding nodes – Success Criteria: 5, 6	Left clicked an empty node with 'block' selected and then 'dense' selected.	When 'block' is selected, the node becomes a block node. When 'dense' is selected, the node becomes a dense node.	The node becomes a block node. The node becomes a dense node.	**Success**	Node types are distinguishable by colour – block nodes are black, dense nodes are brown.
	Left Clicked the start and target nodes with a node selected in the toolbar.	Nothing happens.	Nothing happens.	**Success**	There should always be a start and target node on the grid.
	Left clicked a node with no node selected in the toolbar.	Nothing happens.	Nothing happens.	**Success**	Nodes should only be added if the user select one.

Clearing nodes – Success Criteria: 22	Right clicked a block node and a dense node.	Block node is removed. Dense node is removed.	The block node is reset. The dense node is reset.	**Success**	When a node is reset, it's set to 'empty', and its label colour turns white.
	Right clicked the start and target nodes.	Nothing happens.	Nothing happens.	**Success**	Start and target nodes cannot be deleted.
Dragging – Success Criteria: 21	Dragging across the grid.	When the left mouse button is down, block or dense nodes are added (depending on the selected node on the toolbar) when the cursor enters a label. When the right mouse button is down, nodes are reset when the cursor enters a label.	When 'block' is selected, and the left mouse button is down, nodes the cursor enters become blocks. The same is true when 'dense' is selected, but they become dense nodes. When the right mouse button is down, nodes the cursor enters are reset.	**Success**	It is important this feature works well, otherwise it will be a frustrating experience for the user.
	Dragging over start and target nodes.	The start and target nodes do not change.	The start and target nodes do not change.	**Success**	The user can drag around and over them still.
	Dragging outside the grid and releasing the mouse.	When the user releases the mouse button outside the grid of nodes, dragging stops.	When the user releases the mouse button outside the grid, dragging is still active once re-entering the grid.	**Fail**	See **Modification #1**.
Moving – Success Criteria: 2	Moving the start and target nodes around an empty grid.	The start node follows the mouse whilst the left mouse button is held down. The target node follows the mouse whilst the left mouse	The start node follows the mouse whilst the left mouse button is held down. The target node follows the mouse whilst the left mouse button is held down.	**Success**	The nodes stop moving with the mouse when it is released.

		button is held down.			
	Moving the start and target nodes out of the grid and releasing the mouse.	The start node stays in the grid and stops following the mouse when it re-enters. The target node stays in the grid and stops following the mouse when it re-enters.	The start node stays on the grid when the mouse leaves but carries on following the mouse on re-entry despite the left mouse button no longer being pressed. The target node stays on the grid when the mouse leaves but carries on following the mouse on re-entry despite the left mouse button no longer being pressed.	**Fail**	See **Modification #1**.
	Moving the start and target nodes over block and dense nodes.	The start node doesn't 'clear' any block or dense nodes while its being moved over them. The target node doesn't 'clear' any block or dense nodes while its being moved over them.	The start node doesn't clear any block or dense nodes while its being moved over them. The target node doesn't 'clear' any block or dense nodes while its being moved over them.	**Success**	This problem was solved during implementation.
Diversion node – Success Criteria: 7	Clicked the "Add diversion" button.	The diversion node follows the mouse when it enters the grid.	The blue diversion node follows the mouse when it enters the grid.	**Success**	It follows even though the user isn't holding down the left mouse button.
	Clicked the "Remove diversion" button.	The diversion node is removed from the grid.	The diversion node is removed from the grid but reappears and continues to follow the mouse when it re-enters the grid.	**Fail**	See **Modification #2**.

	Clicked the "Remove diversion" button once the diversion node is fixed on the grid.	The diversion node is removed from the grid.	The diversion node is removed.	**Success**	If the user has already 'placed' the diversion node when they click "Remove diversion", it works.

Modifications

Modification #1: For both dragging and moving nodes, the program doesn't recognise when the user releases the mouse button outside of the grid. This is because the *MouseUp* event is only attached to each label. To fix this problem, I originally tried to add a *MouseUp* event to the form class, but this only works if the user is in full screen mode and doesn't work if the user releases the mouse on the toolbar. Having searched up ways to detect when the user releases the mouse button from anywhere while the program is open, I discovered 'mouse hooks'. However, implementing this required me to install a *NuGet* (package manager for the .NET framework) library, which would've required me to rebuild the project in a different .NET version since it wasn't compatible with mine. Also, having read the library documentation, I concluded that implementing this feature would've taken up too much time. This bug shouldn't affect the user much anyway; if they accidentally release the mouse outside the grid, they can easily stop the dragging/moving by clicking the grid again.

Modification #2: This problem arose because despite me resetting the diversion node using the *Diversion* attribute in the grid class, I didn't set *Dragging* to false or *Moving* to null. This meant that if the user hadn't 'placed' the diversion node and clicked "Remove diversion", the diversion node would follow the mouse again once it entered the grid. I fixed this by disabling 'dragging' (by doing the things mentioned above) within the diversion button event handler in the toolbar class. It's difficult to prove this is now fixed, but I'll show it in the post development testing video.

```
1 reference
private static void DiversionBtn_MouseClick(object? sender, MouseEventArgs e)
{
    // Determines whether the user is adding or removing the diversion node.
    if (e.Button == MouseButtons.Left)
    {
        if (Grid.Diversion is null)
        {
            Grid.Moving = NodeType.Diversion;
            Grid.Dragging = true;

            _diversionBtn.Text = "Remove diversion";
            _diversionBtn.BackColor = Color.PaleVioletRed;
        }
        else
        {
            Grid.Dragging = false;
            Grid.Moving = null;
            Grid.Diversion.Reset();
            Grid.Diversion = null;

            _diversionBtn.Text = "Add diversion";
            _diversionBtn.BackColor = Color.LightBlue;
        }
    }
}
```

Review

User feedback

I have asked the main stakeholders (interviewees) some questions about this prototype and recorded their responses.

Q: What are your initial thoughts on the grid interaction features?

- **Lee**: "The dragging and moving features are very responsive and intuitive, I'm impressed. Sometimes there is a little bit of lag when moving some nodes, but this might be out of your control."
- **Jude**: "I was able to make patterns on the grid easily, using the dragging feature, which I like."

Q: Does this match what you imagined the grid interaction to feel like?

- **Lee**: "Yes. Although I mentioned in the last prototype that the nodes might be too small and finicky, I was wrong – they are just right."
- **Jude**: "Yes. I'm very happy with it and wanted to be able to make layouts easily like this."

Q: Is there anything you would change at this stage?

- **Lee**: "The start and target nodes look a little off centre when the form loads."
- **Jude**: "Personally I would like more node types to be able to add to the grid."

Overall, the stakeholders enjoyed the grid editing experience. Unfortunately, I'm not able to centre the start and target nodes since there is an even number of rows (I don't want to change this since the node sizes would change). I think adding more node types could be a future ambition, but I decided in my *research* that I would only have one weighted node. I responded to the stakeholders with my thoughts on their recommended improvements, and they agreed that no further alterations need to be made at this stage.

Reflection

In this prototype, I've managed to code the different node types and their methods, as well as implement user interaction with the grid, such as dragging block and dense nodes on the grid and moving the start, target and diversion nodes. I have added functionality to the 'diversion button', which now adds or removes the diversion node from the grid when pressed. I managed to fix a logic error which prevented my dragging feature from working, as well as dealt with an exception. I fixed an issue with the diversion button but didn't manage to come up with a simple solution for the problem described in **Modification #1**, which was only a minor issue anyway. Due to these issues, this prototype took slightly longer than expected to develop, but was generally successful. Continuing with grid interaction, I will move onto adding functionality to the toolbar controls that edit the grid in the next prototype.

Prototype 3 – Grid Editing Functions

Implementation

I established in the ***success criteria*** that its important this program contains methods to clear aspects of the grid, as well as quickly generate a grid layout for the user, a feature I liked in the first website of my ***research***. There are two combo boxes on the toolbar that will be used by the user to perform larger scale alterations to the grid. One combo box will provide methods to clear aspects of the grid, and the other provides ways to generate grid layouts. This prototype aims to give function to those combo boxes.

Clearing functions

I decided to start by giving functionality to the 'clearing functions' combo box. To do this, I first had to write the methods involved in clearing aspects of the grid. I created the new methods in the grid class, since it has access to the grid of nodes only. The first method resets all the nodes in the grid that have the node type of the passed parameter. It does this by looping through all the nodes in the grid, and calls *Reset* on the relevant nodes.

```
2 references
public static void Clear(NodeType nodeType)
{
    // Resets the nodes in the grid with the node type specified by the parameter
    if (_nodeGrid is not null)
    {
        foreach (var node in _nodeGrid)
        {
            if (node.Type == nodeType)
                node.Reset();
        }
    }
}
```

The second method resets all the nodes on the grid, and then sets the positions of the start and target node to its initial positions at the start of the program. Both of these methods check if *_nodeGrid* is null first, since *Grid* is a static class and *_nodeGrid* could be null when these methods are called (even though they aren't called before *DrawGrid* gives *_nodeGrid* a value, *intellisense* wanted me to do it, and it does make the code more robust and eliminates the chances of that exception from occurring).

```
1 reference
public static void Reset()
{
    // Clears the entire grid and resets it to its initial layout.
    if (_nodeGrid is not null)
    {
        foreach (var node in _nodeGrid)
            node.Reset();

        _nodeGrid[_nodeGrid.GetLength(0) / 6, _nodeGrid.GetLength(1) / 2].MakeStart();
        _nodeGrid[(_nodeGrid.GetLength(0) / 6) * 5, _nodeGrid.GetLength(1) / 2].MakeTarget();
    }
}
```

Next, I added an event to the 'clearing functions combo box' in the toolbar class which fires when the user selects an item from the list. Depending on what clearing function the user selects, one of the public grid methods I created is called.

```
1 reference
private static void CreateClearCbx(int width, int xPos, int yPos, Form form)
{
    _clearCbx.Location = new Point(xPos, yPos);
    _clearCbx.Width = width;
    _clearCbx.DropDownStyle = ComboBoxStyle.DropDownList;

    _clearCbx.Items.Add("Clear functions:");
    _clearCbx.Items.Add("Clear block nodes");
    _clearCbx.Items.Add("Clear dense nodes");
    _clearCbx.Items.Add("Reset grid");
    _clearCbx.SelectedIndex = 0;

    _clearCbx.SelectionChangeCommitted += ClearCbx_SelectionChangeCommitted;

    form.Controls.Add(_clearCbx);
}

1 reference
private static void ClearCbx_SelectionChangeCommitted(object? sender, EventArgs e)
{
    // Decides which clearing function to call.
    if (_clearCbx.SelectedIndex == 1)
        Grid.Clear(NodeType.Block);
    else if (_clearCbx.SelectedIndex == 2)
        Grid.Clear(NodeType.Dense);
    else if (_clearCbx.SelectedIndex == 3)
        Grid.Reset();
}
```

Although the *Clear* method worked fine and cleared the specified nodes from the grid, the *Reset* method had some issues. Firstly, if you try to reset the grid without moving the start and target nodes first, the start and target nodes are deleted. This happens because within the 'make methods' of the node class, the previous node is reset after the current node and label is changed. This means that if the new start/target node is in the same position as the previous start/target node, it is reset. To fix this, I rearranged the order in the methods so that the previous start/target/diversion node is dealt with first before setting the new one. Another issue is if you reset the grid when the start/target node is 'on top of' a block or dense node, when the nodes are reset and the start and target nodes are 'moved' to their original positions, the 'previous node feature' kicks in and creates a block/dense node at that node. To fix this bug, I edited *MakeStart*, *MakeTarget* and *MakeDiversion* so that they only revert the start/target/diversion node to its previous node if *Moving* is not null. This is now working because when the start/target nodes are reset to their initial positions, the node it's come from (even if it was previously a block or dense node) is always reset because the user isn't moving it themselves.

```
3 references
public void MakeStart()
{
    if (Grid.Start is not null)
    {
        if (Grid.Moving is not null && Grid.Start.PreviousNodeType == NodeType.Block)
            Grid.Start.MakeBlock();
        else if (Grid.Moving is not null && Grid.Start.PreviousNodeType == NodeType.Dense)
            Grid.Start.MakeDense();
        else
            Grid.Start.Reset();
    }

    PreviousNodeType = Type;
    Type = NodeType.Start;

    _lbl.BackColor = Color.Green;

    Grid.Start = this;
}

3 references
public void MakeTarget()
{
    if (Grid.Target is not null)
    {
        if (Grid.Moving is not null && Grid.Target.PreviousNodeType == NodeType.Block)
            Grid.Target.MakeBlock();
        else if (Grid.Moving is not null && Grid.Target.PreviousNodeType == NodeType.Dense)
            Grid.Target.MakeDense();
        else
            Grid.Target.Reset();
    }

    PreviousNodeType = Type;
    Type = NodeType.Target;

    _lbl.BackColor = Color.Red;

    Grid.Target = this;
}
```

```
1 reference
private void MakeDiversion()
{
    if (Grid.Diversion is not null)
    {
        if (Grid.Moving is not null && Grid.Diversion.PreviousNodeType == NodeType.Block)
            Grid.Diversion.MakeBlock();
        else if (Grid.Moving is not null && Grid.Diversion.PreviousNodeType == NodeType.Dense)
            Grid.Diversion.MakeDense();
        else
            Grid.Diversion.Reset();
    }

    PreviousNodeType = Type;
    Type = NodeType.Diversion;

    _lbl.BackColor = Color.Blue;

    Grid.Diversion = this;
}
```

Generations

I started by coding the simpler generation method of the two. I created a public method called *RandomGeneration*, which loops through each node in the grid and either makes it a block or dense node, or resets it, except the start, target and diversion nodes. A random number generator is used to determine what to do to a node, with most of the numbers resulting in resetting the node, which means that it is unlikely there isn't a path from the start to target node as most nodes will be empty.

```
1 reference
public static void RandomGeneration()
{
    if (_nodeGrid is not null)
    {
        var rnd = new Random();

        // Loops through each node in the grid, making most empty.
        foreach (var node in _nodeGrid)
        {
            // Random number between 0 and 9.
            int ranNum = rnd.Next(10);

            if (node.Type != NodeType.Start && node.Type != NodeType.Target && node.Type != NodeType.Diversion)
            {
                if (ranNum == 0)
                    node.MakeBlock();
                else if (ranNum == 1)
                    node.MakeDense();
                else
                    node.Reset();
            }
        }
    }
}
```

The second generation type is a 'recursive maze'. To implement this, I started by creating a *MazeGeneration* method that clears the block and dense nodes in the grid, creates a border of block or dense nodes (depending on the passed node type), and then calls the *Division* method to start the recursive process. This public method will be called to start the maze generation process. I didn't think I would need two methods to generate a maze in the design section, but this first method is needed to set up the initial conditions before starting the recursive division.

```
public static void MazeGeneration(NodeType nodeType)
{
    // Clears the grid first.
    Clear(NodeType.Block);
    Clear(NodeType.Dense);

    // Creates a border around the grid.
    if (_nodeGrid is not null)
    {
        for (int i = 0; i < _nodeGrid.GetLength(0); i++)
        {
            Node topNode = _nodeGrid[i, 0];
            Node bottomNode = _nodeGrid[i, _nodeGrid.GetLength(1) - 1];

            if (nodeType == NodeType.Block)
            {
                if (topNode.Type != NodeType.Start && topNode.Type != NodeType.Target && topNode.Type != NodeType.Diversion)
                    topNode.MakeBlock();

                if (bottomNode.Type != NodeType.Start && bottomNode.Type != NodeType.Target && bottomNode.Type != NodeType.Diversion)
                    bottomNode.MakeBlock();
            }
            else if (nodeType == NodeType.Dense)
            {
                if (topNode.Type != NodeType.Start && topNode.Type != NodeType.Target && topNode.Type != NodeType.Diversion)
                    topNode.MakeDense();

                if (bottomNode.Type != NodeType.Start && bottomNode.Type != NodeType.Target && bottomNode.Type != NodeType.Diversion)
                    bottomNode.MakeDense();
            }
        }
        for (int i = 0; i < _nodeGrid.GetLength(1); i++)
        {
            Node leftNode = _nodeGrid[0, i];
            Node rightNode = _nodeGrid[_nodeGrid.GetLength(0) - 1, i];

            if (nodeType == NodeType.Block)
            {
                if (leftNode.Type != NodeType.Start && leftNode.Type != NodeType.Target && leftNode.Type != NodeType.Diversion)
                    leftNode.MakeBlock();

                if (rightNode.Type != NodeType.Start && rightNode.Type != NodeType.Target && rightNode.Type != NodeType.Diversion)
                    rightNode.MakeBlock();
            }
            else if (nodeType == NodeType.Dense)
            {
                if (leftNode.Type != NodeType.Start && leftNode.Type != NodeType.Target && leftNode.Type != NodeType.Diversion)
                    leftNode.MakeDense();

                if (rightNode.Type != NodeType.Start && rightNode.Type != NodeType.Target && rightNode.Type != NodeType.Diversion)
                    rightNode.MakeDense();
            }
        }

        var rnd = new Random();
        var gaps = new List<(int X, int Y)>();

        // Starts the recursive division algorithm.
        Division(1, 1, _nodeGrid.GetLength(0) - 2, _nodeGrid.GetLength(1) - 2, true, gaps, rnd, nodeType);
    }
}
```

Division is a private method that divides a specified area with a ‘wall’ (row/column of block/dense nodes) which has one gap in it. It then calls itself (recursion) twice, inputting the start and end coordinates (rows/columns) of the two new ‘rooms’ either side of the wall to be divided further. This process continues until a ‘room’ is too small to be divided further. This method has many parameters, including the start and end columns and rows that define the ‘area’ to be divided, the orientation of the previous wall, a list of ‘gap’ coordinates, the R*andom* object, and the node type of the maze. The start of the method contains the return condition, which is met when the width or height of the room is one, or if it's a 2x2 room. Next, the orientation (horizontal or vertical) of the new wall is determined – if the room is wider than it is tall, then a vertical orientation is set, otherwise the orientation is set as ‘horizontal’. If the area is a square, the same orientation as the ‘previous orientation‘ is set, which is to reduce the number of times a small room cannot be divided further due to a gap and surrounding walls. For example, in a 3x3 room with a gap in the middle on the bottom edge, only a horizontal wall can work, which may be the same orientation of the last wall to be made (the bottom edge with the gap) - of course this doesn’t work every time but does reduce the number of larger spaces in the maze. Depending on the determined orientation, a vertical or horizontal wall will then be made. In both cases, this happens by first looping through the possible columns/rows where a wall can be built

and adding them to a list. The wall is possible if it isn't next to another wall and isn't covering a 'gap', which can be checked using the list of gaps already created. If there are no possible places to build a wall, the method is returned. A random index out of the 'possible' list is selected and a wall is built there, with a randomly selected node to be a gap in that wall (which is then added to the list of gaps – list of tuples containing the column and row). At the end, depending on the orientation of the wall, the two new rooms are divided by calling *Division* again twice and passing in the new coordinates. To clarify, the reason I pass in the pre-instantiated *Random* object is so that a new object isn't created at every call, which might make it less random. This method took longer to develop than first anticipated, and required a few iterations and alterations to finally get it to generate a maze as intended - one which always has a route from the start to target node. Here are the screenshots of the two methods – they are quite long so I had to take a few.

```
5 references
private static void Division(int colStart, int rowStart, int colEnd, int rowEnd, bool previousVertical, List<(int X, int Y)> gaps, Random rnd, NodeType nodeType)
{
    // If the area is too small, it stops 'dividing'.
    if (colEnd - colStart == 0 || rowEnd - rowStart == 0 || (colEnd - colStart == 1 && rowEnd - rowStart == 1) || _nodeGrid is null)
        return;

    var possibleWalls = new List<int>();
    bool vertical;

    // Determines which orientation the new 'wall' should be.
    if (colEnd - colStart == rowEnd - rowStart)
        vertical = previousVertical;
    else if (colEnd - colStart > rowEnd - rowStart)
        vertical = true;
    else
        vertical = false;
```

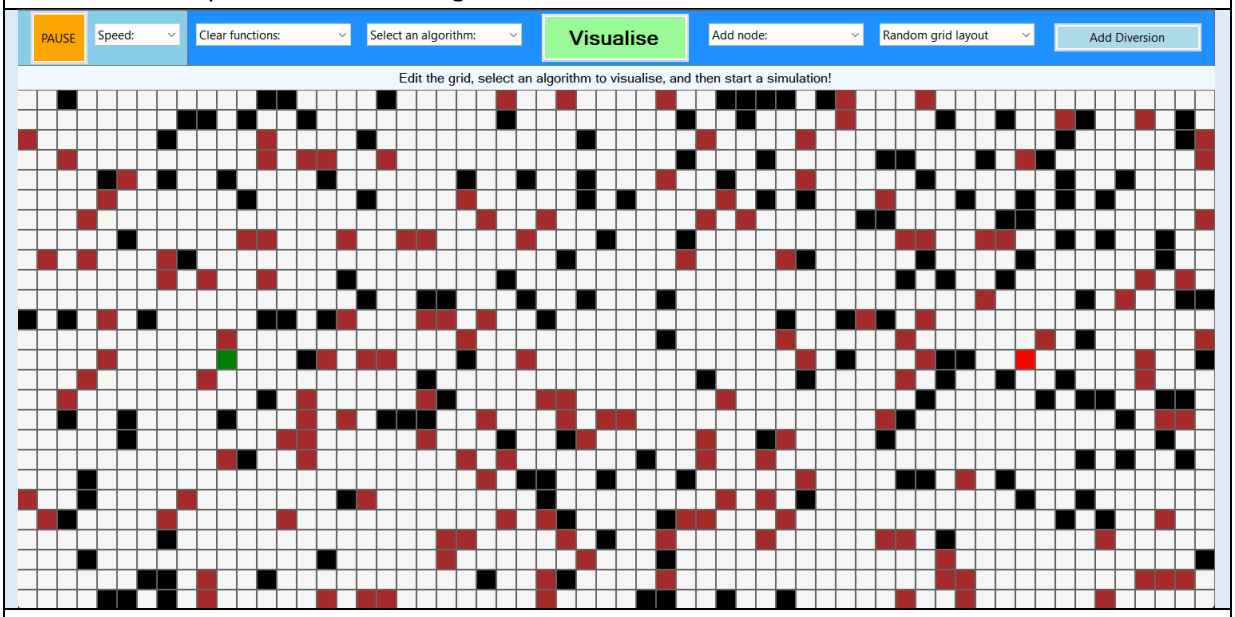
```
// Checks all possible 'wall' placements and randomly chooses one. Creates a random gap in the wall and stores its position. Calls 'Division', inputting the new areas.
// The 'wall' created is dependent on orientation.
if (vertical)
{
    for (int i = colStart + 1; i < colEnd; i++)
    {
        if (!gaps.Contains((i, rowStart - 1)) && !gaps.Contains((i, rowEnd + 1)))
            possibleWalls.Add(i);
    }

    if (possibleWalls.Count == 0)
        return;

    int wallX = possibleWalls[rnd.Next(possibleWalls.Count)];
    int gapY = rnd.Next(rowStart, rowEnd + 1);

    for (int i = rowStart; i <= rowEnd; i++)
    {
        Node node = _nodeGrid[wallX, i];

        if (node.Type != NodeType.Start && node.Type != NodeType.Target && node.Type != NodeType.Diversion)
        {
            if (nodeType == NodeType.Block)
                node.MakeBlock();
            else if (nodeType == NodeType.Dense)
                node.MakeDense();
        }
    }

    Node gap = _nodeGrid[wallX, gapY];

    if (gap.Type != NodeType.Start && gap.Type != NodeType.Target && gap.Type != NodeType.Diversion)
    {
        gaps.Add((wallX, gapY));
        gap.Reset();
    }

    Division(colStart, rowStart, wallX - 1, rowEnd, vertical, gaps, rnd, nodeType);
    Division(wallX + 1, rowStart, colEnd, rowEnd, vertical, gaps, rnd, nodeType);
}
```

```
    else
    {
        for (int i = rowStart + 1; i < rowEnd; i++)
        {
            if (!gaps.Contains((colStart - 1, i)) && !gaps.Contains((colEnd + 1, i)))
                possibleWalls.Add(i);
        }

        if (possibleWalls.Count == 0)
            return;

        int wallY = possibleWalls[rnd.Next(possibleWalls.Count)];
        int gapX = rnd.Next(colStart, colEnd + 1);

        for (int i = colStart; i <= colEnd; i++)
        {
            Node node = _nodeGrid[i, wallY];

            if (node.Type != NodeType.Start && node.Type != NodeType.Target && node.Type != NodeType.Diversion)
            {
                if (nodeType == NodeType.Block)
                    node.MakeBlock();
                else if (nodeType == NodeType.Dense)
                    node.MakeDense();
            }
        }

        Node gap = _nodeGrid[gapX, wallY];

        if (gap.Type != NodeType.Start && gap.Type != NodeType.Target && gap.Type != NodeType.Diversion)
        {
            gaps.Add((gapX, wallY));
            gap.Reset();
        }

        Division(colStart, rowStart, colEnd, wallY - 1, vertical, gaps, rnd, nodeType);
        Division(colStart, wallY + 1, colEnd, rowEnd, vertical, gaps, rnd, nodeType);
    }
}
```

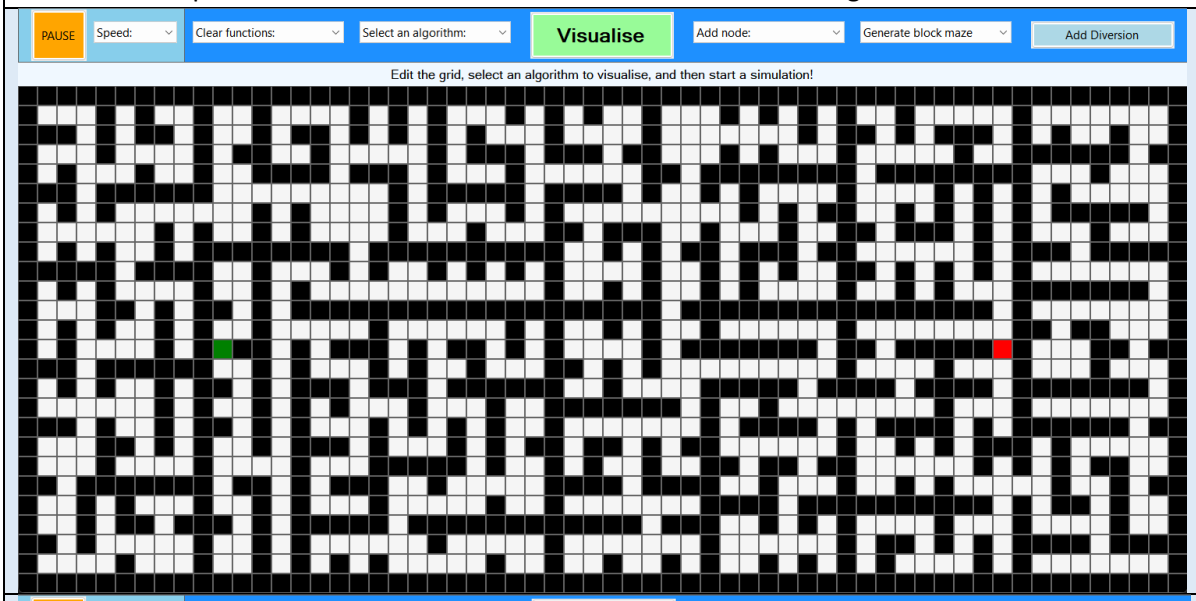
In order for the user to activate these grid generations, I need to give functionality to the 'generations' combo box on the toolbar. The user can select: a random grid, a block maze, or a dense maze.

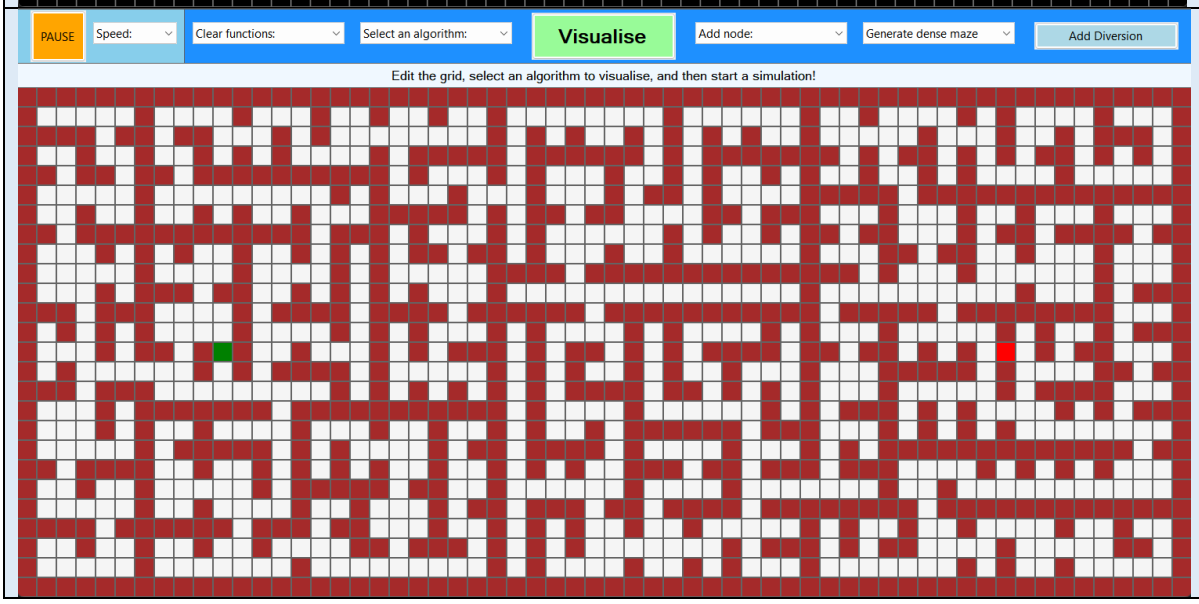
```
1 reference
private static void CreateGenerationsCbx(int width, int xPos, int yPos, Form form)
{
    _generationsCbx.Location = new Point(xPos, yPos);
    _generationsCbx.Width = width;
    _generationsCbx.DropDownStyle = ComboBoxStyle.DropDownList;

    _generationsCbx.Items.Add("Generations:");
    _generationsCbx.Items.Add("Generate block maze");
    _generationsCbx.Items.Add("Generate dense maze");
    _generationsCbx.Items.Add("Random grid layout");
    _generationsCbx.SelectedIndex = 0;

    _generationsCbx.SelectionChangeCommitted += GenerationsCbx_SelectionChangeCommitted;

    form.Controls.Add(_generationsCbx);
}


1 reference
private static void GenerationsCbx_SelectionChangeCommitted(object? sender, EventArgs e)
{
    // Determines which generation layout the user wants.
    if (_generationsCbx.SelectedIndex == 1)
        Grid.MazeGeneration(NodeType.Block);
    else if (_generationsCbx.SelectedIndex == 2)
        Grid.MazeGeneration(NodeType.Dense);
    else if (_generationsCbx.SelectedIndex == 3)
        Grid.RandomGeneration();
}
```

Here is an example of a randomised grid...



Here are examples of a block maze and a dense maze that have been generated...





Testing

Test table

Test	Test details	Expected result	Actual result	Success /Fail	Comments
Clearing functions – Success Criteria: 13	Selected the 'clear block nodes' item and 'clear dense nodes' item with block and dense nodes on the grid.	When 'clear block nodes' is selected, all block nodes are cleared from the grid. When 'clear dense nodes' is selected, all dense nodes are cleared from the grid.	All block nodes are cleared from the grid. All dense nodes are cleared from the grid.	**Success**	Start, target and diversion nodes remain in place.
	Moved the start/target nodes over block/dense nodes, cleared the grid, and then moved the start/target nodes away.	When the start/target nodes are moved over block nodes, then 'clear block nodes' is selected, there are no block nodes on the grid even after the start/target nodes are moved around. When the start/target nodes are moved over dense nodes, then 'clear dense nodes' is selected, there are no dense nodes on the grid even after the start/target nodes are moved around.	When the start/target nodes are moved over block nodes, then 'clear block nodes' is selected, there are block nodes 'underneath' where the start/target nodes were. When the start/target nodes are moved over dense nodes, then 'clear dense nodes' is selected, there are dense nodes 'underneath' where the start/target nodes were.	**Fail**	See **Modification #1**.
	Selected the 'reset grid' item with	All nodes are reset, the diversion node is	All nodes are reset, the diversion node is	**Fail**	See **Modification #2**.

	block/dense nodes and the diversion node on the grid.	removed from the grid and its button is updated, and the start/target nodes are 'moved' back to their initial positions.	removed from the grid, and the start/target nodes are 'moved' back to their initial positions, but the 'diversion button' is not updated.		
Grid generations – Success Criteria: 12	Selected the 'block maze' generation.	A maze of block nodes is generated, which always has a path from the start to target node.	A maze of block nodes is generated, but sometimes the start/target node is surrounded by adjacent blocks.	**Fail**	See **Modification #3**.
	Selected the 'dense maze' generation.	A maze of dense nodes is generated, which always has a path from the start to target node.	A maze of dense nodes is generated, which always has a path from the start to target node.	**Success**	It's okay that sometimes the start/target node is surrounded because dense nodes are traversable.
	Selected the 'random layout' generation.	A random layout of block, dense and empty nodes is generated.	A random layout is generated.	**Success**	The start, target and diversion nodes aren't moved or covered.

Modifications

Modification #1: This happens because when the start/target/diversion nodes are moved, the previous node it was over is recreated. This is an intentional feature that I added in the last prototype but isn't desired in this context. To fix this, I will set all the node's *PreviousNodeType* to *NodeType.Empty*, if that node type is the one being cleared, so that when the start/target/diversion nodes are then moved after the grid is cleared, the node it 'leaves' will be 'empty' if it was 'covering' the node type being cleared. I first had to change this property's 'set method' to public rather than private.

```
4 references
public static void Clear(NodeType nodeType)
{
    // Resets the nodes in the grid with the node type specified by the parameter.
    if (_nodeGrid is not null)
    {
        foreach (var node in _nodeGrid)
        {
            if (node.Type == nodeType)
                node.Reset();
            else if (node.PreviousNodeType == nodeType && (node.Type == NodeType.Start || node.Type == NodeType.Target || node.Type == NodeType.Diversion))
                node.PreviousNodeType = NodeType.Empty;
        }
    }
}
```

Modification #2: To fix this issue, I have to edit the 'diversion button' when the grid is reset. I also realised that I have to set *Grid.Diversion* to null as well, which is an important thing I missed as the simulation will reference this attribute to check if there's a diversion node on the grid. The 'diversion button' cannot be accessed from within the grid class, so I made a public method in the toolbar class

that will update the 'diversion button' and relevant properties when the diversion node is removed. I also call this method from within the 'button click' event in the toolbar class, to reduce duplicate code.

```
2 references
public static void RemoveDiversion()
{
    // Updates the 'diversion button' and removes references to the diversion node.
    if (Grid.Diversion is not null)
    {
        Grid.Dragging = false;
        Grid.Moving = null;
        Grid.Diversion.Reset();
        Grid.Diversion = null;

        _diversionBtn.Text = "Add diversion";
        _diversionBtn.BackColor = Color.LightBlue;
    }
}
```

```
1 reference
private static void DiversionBtn_MouseClick(object? sender, MouseEventArgs e)
{
    // Determines whether the user is adding or removing the diversion node.
    if (e.Button == MouseButtons.Left)
    {
        if (Grid.Diversion is null)
        {
            Grid.Moving = NodeType.Diversion;
            Grid.Dragging = true;

            _diversionBtn.Text = "Remove diversion";
            _diversionBtn.BackColor = Color.PaleVioletRed;
        }
        else
            RemoveDiversion();
    }
}
```

```
1 reference
public static void Reset()
{
    // Clears the entire grid and resets it to its initial layout.
    if (_nodeGrid is not null)
    {
        foreach (var node in _nodeGrid)
            node.Reset();

        Toolbar.RemoveDiversion();

        _nodeGrid[_nodeGrid.GetLength(0) / 6, _nodeGrid.GetLength(1) / 2].MakeStart();
        _nodeGrid[(_nodeGrid.GetLength(0) / 6) * 5, _nodeGrid.GetLength(1) / 2].MakeTarget();
    }
}
```

Modification #3: Although the maze generation was mostly successful, and it always generates a maze with at least one route from one side of the grid to the other, sometimes the start and target nodes (and diversion) are 'blocked in' by the adjacent nodes – this is a problem especially if the maze is a 'block maze', since there will be no possible path to be visualised. I realised that this was occurring because if a 'gap' was supposed to be where a start/target/diversion node is, the gap wouldn't be added to the list of gaps, and so other walls could generate around these nodes and possibly block them in. Therefore, I edited the *Division* method so that it still adds the gap to the list of gaps even if

it is a start/target/diversion node, but just doesn't 'reset' the node. I have taken screenshots of the relevant parts in the *Division* method that have been changed.

```
Node gap = _nodeGrid[wallX, gapY];
gaps.Add((wallX, gapY));

if (gap.Type != NodeType.Start && gap.Type != NodeType.Target && gap.Type != NodeType.Diversion)
    gap.Reset();
```

```
Node gap = _nodeGrid[gapX, wallY];
gaps.Add((gapX, wallY));

if (gap.Type != NodeType.Start && gap.Type != NodeType.Target && gap.Type != NodeType.Diversion)
    gap.Reset();
```

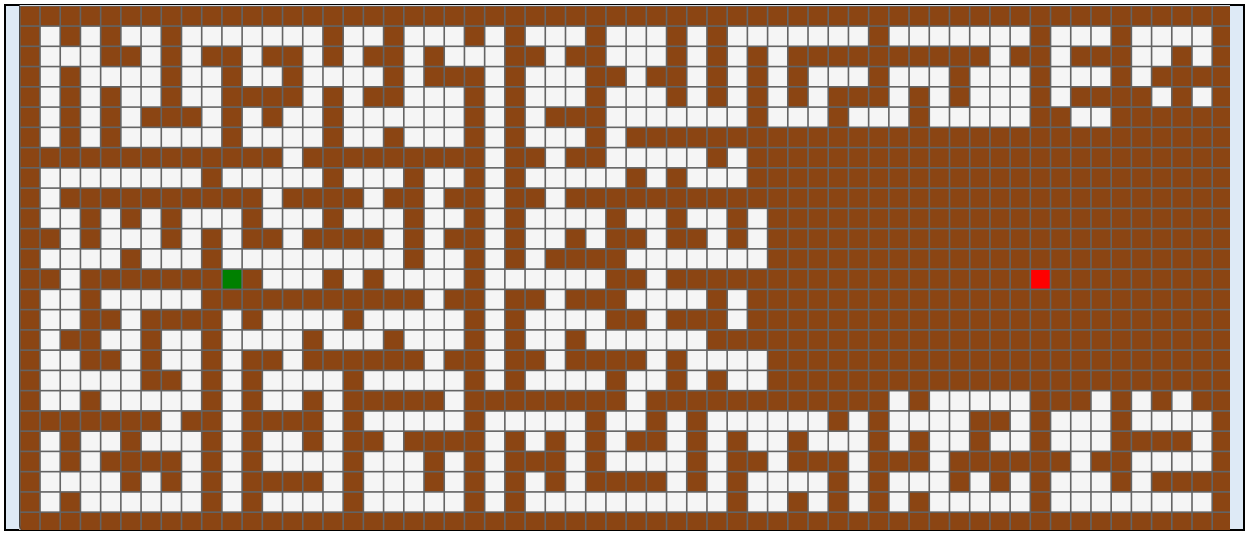
Modification #4: This modification was made in response to Lee's feedback. I set the selected index of each combo box to 0 after the user selects an item.

```
1 reference
private static void ClearCbx_SelectionChangeCommitted(object? sender, EventArgs e)
{
    // Decides which clearing function to call.
    if (_clearCbx.SelectedIndex == 1)
        Grid.Clear(NodeType.Block);
    else if (_clearCbx.SelectedIndex == 2)
        Grid.Clear(NodeType.Dense);
    else if (_clearCbx.SelectedIndex == 3)
        Grid.Reset();

    // Resets the selected index back to the description.
    _clearCbx.SelectedIndex = 0;
}
```

```
1 reference
private static void GenerationsCbx_SelectionChangeCommitted(object? sender, EventArgs e)
{
    // Determines which generation layout the user wants.
    if (_generationsCbx.SelectedIndex == 1)
        Grid.MazeGeneration(NodeType.Block);
    else if (_generationsCbx.SelectedIndex == 2)
        Grid.MazeGeneration(NodeType.Dense);
    else if (_generationsCbx.SelectedIndex == 3)
        Grid.RandomGeneration();

    // Resets the selected index back to the description.
    _generationsCbx.SelectedIndex = 0;
}
```

Modification #5: This modification was made in response to Jude's feedback. I changed the colour of dense nodes to "Saddle Brown", to contrast more with the red target node. As you can see, this colour works better as its less red than the "Brown" colour preset.

```
11 references
public void MakeDense()
{
    Type = NodeType.Dense;
    _lbl.BackColor = Color.SaddleBrown;
}
```



Review

User feedback

I have asked the main stakeholders (interviewees) some questions about this prototype and recorded their responses.

Q: What are your initial thoughts on the clearing functions and grid generations?

- **Lee**: "The clearing functions work well, and the mazes are generated smoothly and quickly – it's a good experience."
- **Jude**: "I love the maze generations."

Q: Does this match what you imagined the grid editing functions to do?

- **Lee**: "Yes."
- **Jude**: "Yes. I'm very happy with it as it allows me to create more interesting grid layouts, which is what I've wanted."

Q: Is there anything you would change at this stage?

- **Lee**: "I think the drop-down menus should reset to the 'description item' after performing a function, so that I can more easily see what each drop-down contains."
- **Jude**: "Sometimes I find it difficult to see the target node if there are a lot of dense nodes around it. I would make the dense nodes are darker colour."

The stakeholders liked the grid editing functions I implemented in this prototype and suggested a few improvements that can be made. I agree with Lee that changing the selected index of the 'clearing functions' and 'generations' combo boxes to the 'description item' once the user has selected a function will help the user identify the combo boxes again [see **Modification #4**]. Like Jude, I also found that the target node is sometimes hard to locate amongst a collection of dense nodes, so I will make the colour of dense nodes a darker brown [see **Modification #5**]. After showing the stakeholders my modifications, they agreed that I should move onto the next prototype.

Reflection

In this prototype, I have added functionality to the 'clearing functions' combo box and 'generations' combo box. The user can now clear specific nodes from the grid, reset the grid to its initial layout,

generate a completely random grid of various node types, generate a random block maze, and generate a random dense maze. I have made many modifications to this prototype after testing it and receiving stakeholder feedback, including the colour of dense nodes, the state of the combo boxes, and the behaviour of the maze generation and clearing functions. I am happy with the recursive division algorithm I created to generate mazes, but it isn't perfect. I don't have time to perfect it now, and it already took long enough to make and adapt to my specific program and needs. For this reason, along with the many modifications made, this prototype took far longer than expected to code. All the editing features are now added, so the next prototypes will focus on the simulation, starting with implementing the pathfinding algorithms themselves.

Prototype 4 – Pathfinding Algorithms & Simulation

Implementation

Of course, a core part of this project is the pathfinding algorithms. As specified in the design section (simulation class diagram) and elsewhere, I will aim to implement three pathfinding algorithms: Dijkstra's algorithm, A* Search, and Breadth First Search (BFS). BFS is the only unweighted algorithm out of the three. Within this section, I will also code some of the rules of the simulation.

Important data

Before coding the algorithms themselves, I created an Enum to store the algorithm names (as strings can sometimes be mistyped and cause errors) and a public property in the toolbar class to store the selected algorithm (can be null if no algorithm is selected). To update this attribute, I added an event to the 'algorithms combo box' which fires when the user selects a new item. Within the event handler, the selected algorithm property updates depending on the selected index.

```
4 references
public enum Algorithm
{
    Dijkstra,
    AStar,
    BFS
}
```

```
3 references
public static Algorithm? SelectedAlgorithm { get; private set; }
```

```
1 reference
private static void CreateAlgorithmsCbx(int width, int xPos, int yPos, Form form)
{
    _algorithmsCbx.Location = new Point(xPos, yPos);
    _algorithmsCbx.Width = width;
    _algorithmsCbx.DropDownStyle = ComboBoxStyle.DropDownList;

    _algorithmsCbx.Items.Add("Select an algorithm:");
    _algorithmsCbx.Items.Add("Dijkstra's");
    _algorithmsCbx.Items.Add("A*");
    _algorithmsCbx.Items.Add("Breadth-first");
    _algorithmsCbx.SelectedIndex = 0;

    _algorithmsCbx.SelectionChangeCommitted += AlgorithmsCbx_SelectionChangeCommitted;

    form.Controls.Add(_algorithmsCbx);
}

1 reference
private static void AlgorithmsCbx_SelectionChangeCommitted(object? sender, EventArgs e)
{
    // Determines which algorithm the user wishes to visualise.
    if (_algorithmsCbx.SelectedIndex == 1)
        SelectedAlgorithm = Algorithm.Dijkstra;
    else if (_algorithmsCbx.SelectedIndex == 2)
        SelectedAlgorithm = Algorithm.AStar;
    else if (_algorithmsCbx.SelectedIndex == 3)
        SelectedAlgorithm = Algorithm.BFS;
    else
        SelectedAlgorithm = null;
}
```

Another thing to do before creating the simulation class and algorithms is to make a *Neighbours* property in the node class. This will make it easier for the algorithms to traverse the grid, as they can access each node's connection to other nodes.

```
14 references
public class Node
{
    36 references
    public NodeType Type { get; private set; }
    11 references
    public NodeType PreviousNodeType { get; set; }
    5 references
    public List<Node> Neighbours { get; set; }

    private readonly Label _lbl = new();

    1 reference
    public Node(int size, int x, int y, Form form)
    {
        Type = NodeType.Empty;
        Neighbours = new List<Node>();

        CreateLbl(size, x, y, form);
    }
```

In the design section, I originally thought that I would create a method in the node class to set the neighbours of that node, but now I've realised that I would need access to the grid of nodes, and so need to do it inside the grid class. I set the neighbours of each node from within the *DrawGrid* method, after the grid of nodes had been created. Most nodes will have four neighbours, but the edge nodes will only have three, and the corners will have two. A series of if-statements are needed to check if the node is a corner or edge node – it can only add a node to the list of neighbours if that node exists in the grid (an "index out of bounds" error message will flag if you try to add a node not in the grid).

```
// Sets the neighbours of each node.
for (int i = 0; i < cols; i++)
{
    for (int j = 0; j < rows; j++)
    {
        Node node = _nodeGrid[i, j];

        if (i > 0)
            node.Neighbours.Add(_nodeGrid[i - 1, j]);

        if (i < _nodeGrid.GetLength(0) - 1)
            node.Neighbours.Add(_nodeGrid[i + 1, j]);

        if (j > 0)
            node.Neighbours.Add(_nodeGrid[i, j - 1]);

        if (j < _nodeGrid.GetLength(1) - 1)
            node.Neighbours.Add(_nodeGrid[i, j + 1]);
    }
}
```

Alongside its neighbours, nodes will also have to store information about their 'g-cost' (this measure will be explained in the algorithm implementation) and 'parent' node (i.e., previous node - used for backtracking), as well as its 'adjacent cost', which is the 'cost' of traversing to that node from a neighbour (cost of one if it's an empty node, and fifteen if it's a dense node). This data will be utilised by the algorithms to find the shortest path – *Parent* is nullable because it will be null until first used.

```
7 references
public int AdjacentCost { get; private set; }
15 references
public int GCost { get; set; }
```

```
5 references
public Node? Parent { get; set; }
```

I then set the 'adjacent cost' of each node from within the 'make methods' - dense nodes have a cost of fifteen, empty nodes have a cost of one. I also set the initial cost of a node to one in the node constructor.

```
10 references
public void Reset()
{
    Type = NodeType.Empty;
    AdjacentCost = 1;

    _lbl.BackColor = Color.WhiteSmoke;
}
```

```
12 references
public void MakeDense()
{
    Type = NodeType.Dense;
    AdjacentCost = 15;

    _lbl.BackColor = Color.SaddleBrown;
}
```

Algorithms

The first pathfinding algorithm I coded was Dijkstra's algorithm. I created the static simulation class to store this private method in. In the design section (class diagrams), I thought this would be a procedure, but I implemented it as a function, because it is then able to return 'true' or 'false' depending on whether the shortest path was found or not; all the pathfinding algorithms will return a Boolean value. I won't go into detail about how each pathfinding algorithm works, but here's a website that goes over the main pathfinding algorithms - Introduction to A* (stanford.edu). You can also read my commented code to understand how it works. Dijkstra's algorithm works using a breadth-first search method (each node's neighbours are visited) and assigns each node a measure of how 'far away' it is from the start node (g-cost), by taking into account previous visited nodes and their individual costs. The algorithm chooses the next node to evaluate by picking the node with the lowest g-cost from the list of visited nodes. This process continues until the evaluated node is the end node (shortest path found), or all the nodes have been searched (no path found).

```
3 references
private static bool Dijkstra(Node start, Node end)
{
    // Nodes to be evaluated.
    var open = new List<Node>();
    // Nodes evaluated.
    var closed = new List<Node>();

    start.GCost = 0;
    open.Add(start);

    // Node in 'open' with the lowest g-cost is evaluated next.
    while (open.Count > 0)
    {
        Node current = open[0];

        foreach (var node in open)
        {
            if (node.GCost < current.GCost)
                current = node;
        }

        // Removes it from 'open' and adds it to the 'closed' set.
        open.Remove(current);
        closed.Add(current);

        // Path found if this node is the 'end node'.
        if (ReferenceEquals(current, end))
            return true;
```

```
        // Loops through the neighbours of the node.
        foreach (var neighbour in current.Neighbours)
        {
            // Skips the neighbour if it's not 'traversable' or has already been evaluated.
            if (neighbour.Type == NodeType.Block || closed.Contains(neighbour))
                continue;

            // If a neighbour is not in the open set, or there is a shorter path to it, it's properties are updated.
            if (!open.Contains(neighbour) || current.GCost + neighbour.AdjacentCost < neighbour.GCost)
            {
                neighbour.GCost = current.GCost + neighbour.AdjacentCost;
                neighbour.Parent = current;

                // If a neighbour is not in the open set, it's added.
                if (!open.Contains(neighbour))
                    open.Add(neighbour);
            }
        }
    }

    return false;
}
```

The next algorithm I implemented was A* Search. Before I could make this algorithm, I needed to make a heuristic function, which I stated in the design section will be 'Manhattan's distance' because of its simplicity and compatibility with a four-way grid. The purpose of this function is to return an estimate for an arbitrary measure of the relative 'distance' from one node to another (usually the node being evaluated to the end node). The heuristic function needs access to the nodes' column and row in the grid, and so I created appropriate properties in the node class to store this. I added parameters to the constructor which allowed the columns and rows to be set when the node is created, which I passed in from the *DrawGrid* method *(i* and *j).*

```
39 references
public class Node
{
    68 references
    public NodeType Type { get; private set; }
    11 references
    public NodeType PreviousNodeType { get; set; }

    8 references
    public List<Node> Neighbours { get; set; }
    5 references
    public Node? Parent { get; set; }

    3 references
    public int Column { get; private set; }
    3 references
    public int Row { get; private set; }
    7 references
    public int AdjacentCost { get; private set; }
    14 references
    public int GCost { get; set; }

    private readonly Label _lbl = new();

    1 reference
    public Node(int size, int x, int y, int col, int row, Form form)
    {
        Type = NodeType.Empty;
        Neighbours = new List<Node>();
        AdjacentCost = 1;
        Column = col;
        Row = row;

        CreateLbl(size, x, y, form);
    }
```

```
1 reference
public static void DrawGrid(int cols, int rows, int nodeSize, int offsetY, Form form)
{
    _nodeGrid = new Node[cols, rows];

    for (int i = 0; i < cols; i++)
    {
        for (int j = 0; j < rows; j++)
        {
            // Creates a node object at each index in the 2D array 'nodeGrid'.
            // The y-position of each node is offset to leave space for the toolbar.
            var node = new Node(nodeSize, i * nodeSize, j * nodeSize + offsetY, i, j, form);
            _nodeGrid[i, j] = node;
        }
    }
```

Here is the simple heuristic function I created in the simulation class, which returns a node's h-cost based on these added node properties.

```
4 references
private static int Heuristic(Node current, Node end)
{
    return Math.Abs(current.Column - end.Column) + Math.Abs(current.Row - end.Row);
}
```

I could now implement A*, which is a variation of Dijkstra's algorithm which also takes into account a node's h-cost (f-cost = g-cost + h-cost) when deciding which node to evaluate next – the node with the lowest f-cost is evaluated next. The heuristic 'guides' the algorithm towards the end node, and can be quicker than Dijkstra's algorithm for this reason, as less nodes are evaluated.

```
3 references
private static bool AStar(Node start, Node end)
{
    // Nodes to be evaluated.
    var open = new List<Node>();
    // Nodes evaluated.
    var closed = new List<Node>();

    start.GCost = 0;
    open.Add(start);

    // Node in 'open' with the lowest f-cost (g-cost + h-cost) is evaluated next.
    // If the f-cost is the same, the node with the lowest h-cost is prioritised.
    while (open.Count > 0)
    {
        Node current = open[0];

        foreach (var node in open)
        {
            int currentFCost = current.GCost + Heuristic(current, end);
            int nodeFCost = node.GCost + Heuristic(node, end);

            if (nodeFCost < currentFCost)
                current = node;
            else if (nodeFCost == currentFCost && Heuristic(node, end) < Heuristic(current, end))
                current = node;
        }

        // Removes it from 'open' and adds it to the 'closed' set.
        open.Remove(current);
        closed.Add(current);

        // Path found if this node is the 'end node'.
        if (ReferenceEquals(current, end))
            return true;
```

```
        // Loops through the neighbours of the node.
        foreach (var neighbour in current.Neighbours)
        {
            // Skips the neighbour if it's not 'traversable' or has already been evaluated.
            if (neighbour.Type == NodeType.Block || closed.Contains(neighbour))
                continue;

            // If a neighbour is not in the open set, or there is a shorter path to it, it's properties are updated.
            if (!open.Contains(neighbour) || current.GCost + neighbour.AdjacentCost < neighbour.GCost)
            {
                neighbour.GCost = current.GCost + neighbour.AdjacentCost;
                neighbour.Parent = current;

                // If a neighbour is not in the open set, it's added.
                if (!open.Contains(neighbour))
                    open.Add(neighbour);
            }
        }
    }

    return false;
}
```

Breadth-First Search (BFS) is the last algorithm I implemented. Unlike the previous two algorithms, it doesn't use cost. This is because it's an unweighted algorithm, and so it treats every node as either 'traversable' or not. This means it won't take dense nodes into account, which I will need to clarify to the user (I do this later). It works by visiting each node's neighbours (starting with the start node), and evaluates them one at a time on a FIFO basis (queue) until the end node is or isn't found.

```
3 references
private static bool BFS(Node start, Node end)
{
    // Queue of nodes to evaluate next.
    var toCheck = new Queue<Node>();
    // List of all visited nodes.
    var visited = new List<Node>();

    toCheck.Enqueue(start);
    visited.Add(start);

    while (toCheck.Count > 0)
    {
        // Next node in the queue to evaluate.
        Node current = toCheck.Dequeue();

        // If it is the end node, the path is found.
        if (ReferenceEquals(current, end))
            return true;

        // Loops through the neighbours of the node.
        foreach (var neighbour in current.Neighbours)
        {
            // If the neighbour is 'traversable' and hasn't been 'visited', it's added to the queue.
            if (neighbour.Type != NodeType.Block && !visited.Contains(neighbour))
            {
                toCheck.Enqueue(neighbour);
                visited.Add(neighbour);
                neighbour.Parent = current;
            }
        }
    }

    return false;
}
```

Simulation

Although I will implement the visualisation 'steps' and processes in the next prototype, I can still create a basic visual to represent a shortest path to test the pathfinding algorithms at this stage. I can also implement the rules for the simulation, such as not allowing grid editing during a simulation, at

this stage. To start, I created a property in the simulation class which will keep track of whether a simulation is running or not.

```
namespace PathfindingAlgorithmVisualiser
{
    9 references
    public static class Simulation
    {
        9 references
        public static bool Running { get; private set; }
```

For this prototype, I will just display the shortest path so that I can test the algorithms work. In order to do this, I need to make a method in the node class called *Path* which will update the label's appearance – the colour of this may change when I implement visualisation in the next prototype.

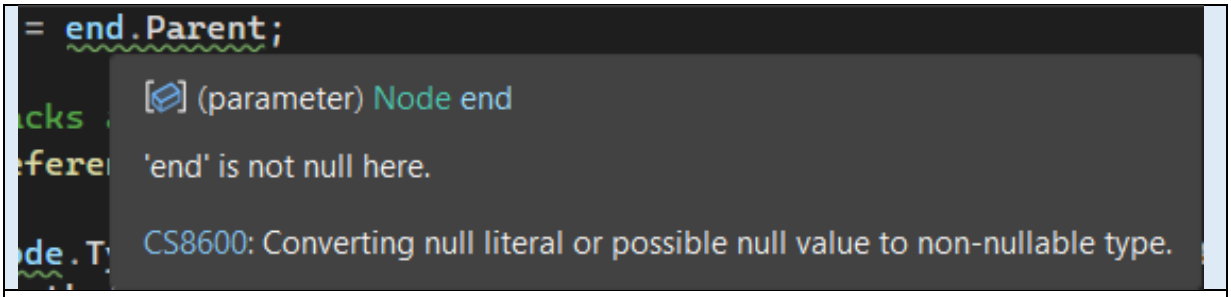
```
1 reference
public void Path()
{
    // Test
    _lbl.BackColor = Color.Fuchsia;
}
```

Originally, I thought I could display the shortest path by backtracking (using the 'parent' property of each node) from the target to the start node. Although this worked when there was no diversion node, it didn't work when there was, since the 'second path' usually interfered with the first and changed node's parent properties. To stop this happening, I created a private field that stores a list of all the nodes in the path, which is updated before the algorithm is run a second time. I created a method, *AddPath*, in the simulation class to perform 'backtracking' between two points and store nodes in the 'path' list.

```
private static List<Node> _path = new();
```

```
9 references
private static void AddPath(Node start, Node end)
{
    Node node = end.Parent;

    // Backtracks and stores the shortest path.
    while (!ReferenceEquals(node, start))
    {
        if (node.Type != NodeType.Start && node.Type != NodeType.Target && node.Type != NodeType.Diversion)
            _path.Add(node);

        node = node.Parent;
    }
}
```

As you can see, this flagged many warnings, since the parent property can be null, and is being stored in a non-nullable data type/object - 'Node'. To fix this, I made the parent property of the node class non-nullable (removed the '?' operator) and assigned its default value to itself in the constructor – it doesn't matter what its initial value is, since the relevant nodes will be updated and used by the pathfinding algorithms when needed. This removed all the warnings. [see **error #4** in appendix]



After that, I created a public method in this class that 'starts' the simulation – this method will change in the next prototype when visualisation is implemented. When a simulation is started, *Running* is set to true and a specific pathfinding algorithm is run depending on which algorithm the user has selected (got from the toolbar class). A check is also needed to determine if the diversion node is on the grid or not. If it isn't, the selected algorithm is only run once (between the start and target node), and if it is, the selected algorithm is run twice (once between the start and diversion node, and once between the diversion and target node). At the end of the method, a path is drawn if the first path is found and the second path isn't applicable (i.e., the diversion isn't on the grid), or if the first path and second path are found. This is so that half a path isn't drawn, for example, from a start to diversion node (or diversion to target), due to only part of the path being found.

```
1 reference
public static void Start()
{
    Running = true;
    _path = new();

    bool firstPathFound = false;
    bool? secondPathFound = null;

    // Calls the necessary pathfinding algorithms and passes their results.
    if (Grid.Start is not null && Grid.Target is not null)
    {
        if (Toolbar.SelectedAlgorithm == Algorithm.Dijkstra)
        {
            if (Grid.Diversion is null)
            {
                firstPathFound = Dijkstra(Grid.Start, Grid.Target);
                if (firstPathFound)
                    AddPath(Grid.Start, Grid.Target);
            }
            else
            {
                firstPathFound = Dijkstra(Grid.Start, Grid.Diversion);
                if (firstPathFound)
                    AddPath(Grid.Start, Grid.Diversion);

                secondPathFound = Dijkstra(Grid.Diversion, Grid.Target);
                if (secondPathFound == true)
                    AddPath(Grid.Diversion, Grid.Target);
            }
        }
```

```
    else if (Toolbar.SelectedAlgorithm == Algorithm.AStar)
    {
        if (Grid.Diversion is null)
        {
            firstPathFound = AStar(Grid.Start, Grid.Target);
            if (firstPathFound)
                AddPath(Grid.Start, Grid.Target);
        }
        else
        {
            firstPathFound = AStar(Grid.Start, Grid.Diversion);
            if (firstPathFound)
                AddPath(Grid.Start, Grid.Diversion);

            secondPathFound = AStar(Grid.Diversion, Grid.Target);
            if (secondPathFound == true)
                AddPath(Grid.Diversion, Grid.Target);
        }
    }
    else if (Toolbar.SelectedAlgorithm == Algorithm.BFS)
    {
        Grid.DisableDenseNodes();

        if (Grid.Diversion is null)
        {
            firstPathFound = BFS(Grid.Start, Grid.Target);
            if (firstPathFound)
                AddPath(Grid.Start, Grid.Target);
        }
        else
        {
            firstPathFound = BFS(Grid.Start, Grid.Diversion);
            if (firstPathFound)
                AddPath(Grid.Start, Grid.Diversion);

            secondPathFound = BFS(Grid.Diversion, Grid.Target);
            if (secondPathFound == true)
                AddPath(Grid.Diversion, Grid.Target);
        }
    }
}
```

```
    // Draws the path only if a 'whole path' has been found.
    if (firstPathFound && (secondPathFound == null || secondPathFound == true))
    {
        foreach (var node in _path)
            node.Path();
    }
}
```

Within the 'breadth-first section' of this method, *DisableDenseNodes* is called (a method which I thought of in the algorithms design section), which is a public method I created in the grid class that makes all the dense nodes mostly transparent to help the user understand that BFS doesn't consider the cost of dense nodes. Before I did this though, I created a method in the node class (called *Disable*) which makes a node partially transparent.

```
1 reference
public static void DisableDenseNodes()
{
    // Makes all dense nodes in the grid semi-transparent.
    if (_nodeGrid is not null)
    {
        foreach (var node in _nodeGrid)
        {
            if (node.Type == NodeType.Dense)
                node.Disable();
        }
    }
}
```

When a simulation is over, the visualisation needs to be cleared, so I made a method in the grid class which clears all the visualisation (this will become more useful later when there is more than just the path). It 'resets' all the 'empty' nodes (which may be different colours due to the visualisation having taken place) by calling the *Clear* method (passing in *NodeType.Empty*), and 'remakes' the dense nodes (which might be 'disabled' or part of a path).

```
1 reference
public void Disable()
{
    // Makes label semi-transparent.
    _lbl.BackColor = Color.FromArgb(50, _lbl.BackColor);
}
```

```
1 reference
public static void ClearVisualisation()
{
    // Clears all visualisations and back colours.
    Clear(NodeType.Empty);

    if (_nodeGrid is not null)
    {
        foreach (var node in _nodeGrid)
        {
            if (node.Type == NodeType.Dense)
                node.MakeDense();
        }
    }
}
```

Now I can create a public method in the simulation class to stop the simulation – this method will be expanded in the next prototype. It calls *ClearVisualisation*, then sets Running to false – it will become apparent later why Running is needed.

```
1 reference
public static void Stop()
{
    // Stops the simulation.
    Grid.ClearVisualisation();
    Running = false;
}
```

Next, I utilised the *Running* property of the simulation class to not allow the user to edit the grid whilst the simulation is running. I did this in the label *MouseDown* event handler in the node class by returning if *Simulation.Running* is true, which prevents all dragging and moving features from being activated.

```
1 reference
private void Lbl_MouseDown(object? sender, MouseEventArgs e)
{
    if (Simulation.Running)
        return;

    // This makes it so that other labels can listen for 'MouseEnter' event while the mouse is still pressed.
    if (sender is Control control)
        control.Capture = false;

    // Determines if the user is moving or adding a node.
    if (Type == NodeType.Start || Type == NodeType.Target || Type == NodeType.Diversion)
    {
        if (e.Button == MouseButtons.Left)
            Grid.Moving = Type;
    }
    else
    {
        UpdateLbl(e.Button);
        Grid.ButtonHolding = e.Button;
    }

    // Initiates the dragging process.
    Grid.Dragging = true;
}
```

I also referenced *Running* from within the toolbar class to restrict the user's access of the toolbar during a simulation. For example, clearing functions and grid generations won't occur if the simulation is running, and the diversion button/node won't update. This reenforces my ideas in *solution structure* where I discussed how the program will have two modes – the user cannot edit during the simulation mode.

```
1 reference
private static void ClearCbx_SelectionChangeCommitted(object? sender, EventArgs e)
{
    // Decides which clearing function to call.
    if (!Simulation.Running)
    {
        if (_clearCbx.SelectedIndex == 1)
            Grid.Clear(NodeType.Block);
        else if (_clearCbx.SelectedIndex == 2)
            Grid.Clear(NodeType.Dense);
        else if (_clearCbx.SelectedIndex == 3)
            Grid.Reset();
    }

    // Resets the selected index back to the description.
    _clearCbx.SelectedIndex = 0;
}
```

```
1 reference
private static void GenerationsCbx_SelectionChangeCommitted(object? sender, EventArgs e)
{
    // Determines which generation layout the user wants.
    if (!Simulation.Running)
    {
        if (_generationsCbx.SelectedIndex == 1)
            Grid.MazeGeneration(NodeType.Block);
        else if (_generationsCbx.SelectedIndex == 2)
            Grid.MazeGeneration(NodeType.Dense);
        else if (_generationsCbx.SelectedIndex == 3)
            Grid.RandomGeneration();
    }

    // Resets the selected index back to the description.
    _generationsCbx.SelectedIndex = 0;
}
```

```
1 reference
private static void DiversionBtn_MouseClick(object? sender, MouseEventArgs e)
{
    // Determines whether the user is adding or removing the diversion node.
    if (e.Button == MouseButtons.Left && !Simulation.Running)
    {
        if (Grid.Diversion is null)
        {
            Grid.Moving = NodeType.Diversion;
            Grid.Dragging = true;

            _diversionBtn.Text = "Remove diversion";
            _diversionBtn.BackColor = Color.PaleVioletRed;
        }
        else
            RemoveDiversion();
    }
}
```

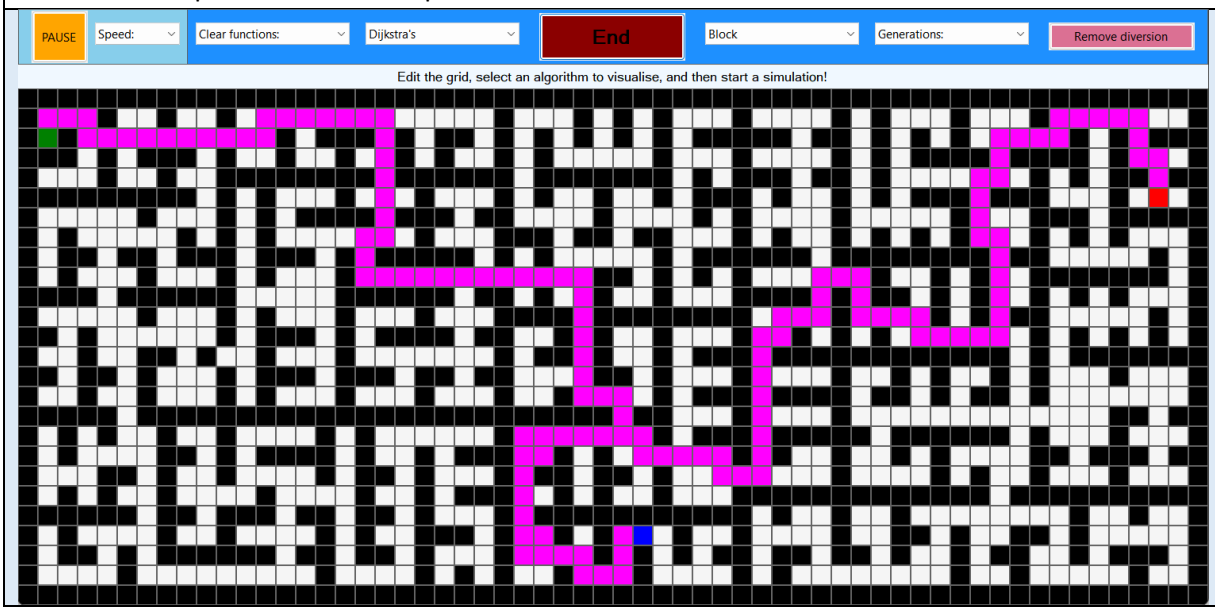
Secondly, I altered the 'algorithm selection combo box' event handler so that the user cannot change the selected index whilst the simulation is running – this is so the user doesn't get confused about which algorithm is actually running later on. To do this, I had to 'cast' the *SelectedAlgorithm* property to an integer and add one, making that the new selected index (which is the selected index when the simulation was started). This works because the Algorithm Enum can be indexed and is ordered in the same way the combo box list is ordered (one is added to ignore the first 'description item').

```
1 reference
private static void AlgorithmsCbx_SelectionChangeCommitted(object? sender, EventArgs e)
{
    // Determines which algorithm the user wishes to visualise.
    if (!Simulation.Running)
    {
        if (_algorithmsCbx.SelectedIndex == 1)
            SelectedAlgorithm = Algorithm.Dijkstra;
        else if (_algorithmsCbx.SelectedIndex == 2)
            SelectedAlgorithm = Algorithm.AStar;
        else if (_algorithmsCbx.SelectedIndex == 3)
            SelectedAlgorithm = Algorithm.BFS;
        else
            SelectedAlgorithm = null;
    }
    else if (SelectedAlgorithm is not null)
        _algorithmsCbx.SelectedIndex = (int)SelectedAlgorithm + 1;
}
```

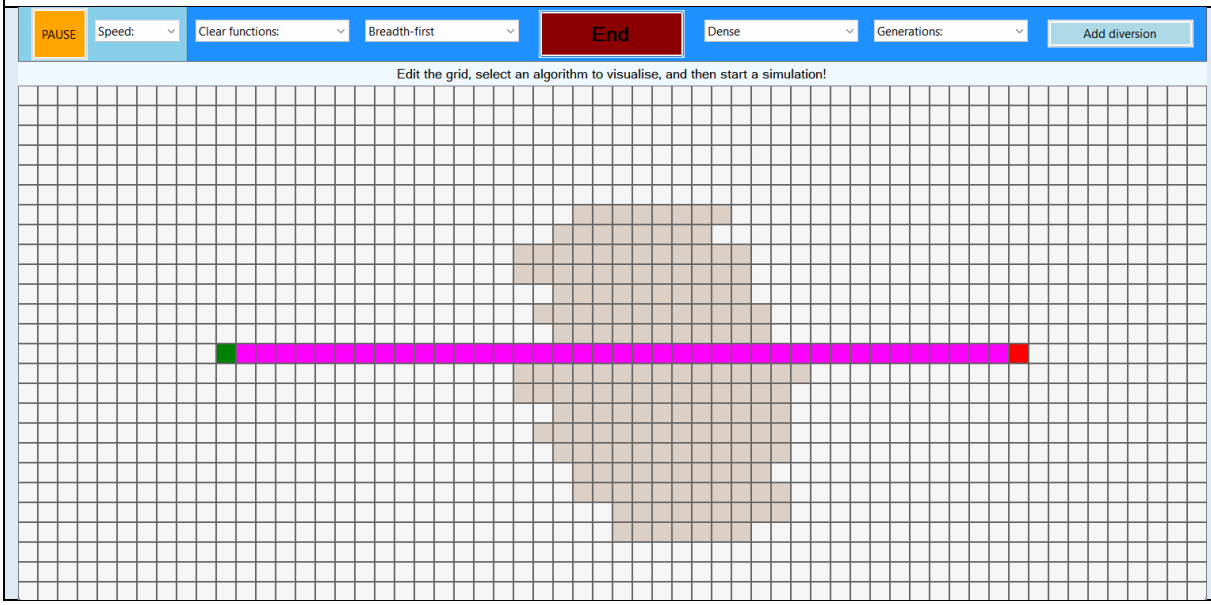
Lastly but importantly, I added an event to the 'mode button' which fires when the user clicks it. If the simulation is running, it calls *Stop* and updates the button's background to green and text to "Visualise". If the simulation isn't running (and an algorithm is selected), it calls *Start* and updates the button's text to "End" and colour to dark red.

```
1 reference
private static void CreateModeBtn(int width, int height, int xPos, int yPos, Form form)
{
    _modeBtn.Location = new Point(xPos, yPos);
    _modeBtn.Size = new Size(width, height);
    _modeBtn.BackColor = Color.PaleGreen;
    _modeBtn.Font = new Font(FontFamily.GenericSansSerif, 15, FontStyle.Bold);
    _modeBtn.Text = "Visualise";

    _modeBtn.MouseClick += ModeBtn_MouseClick;

    form.Controls.Add(_modeBtn);
}

1 reference
private static void ModeBtn_MouseClick(object? sender, MouseEventArgs e)
{
    // Starts or stops the simulation and updates the main button.
    if (e.Button == MouseButtons.Left)
    {
        if (Simulation.Running)
        {
            Simulation.Stop();

            _modeBtn.Text = "Visualise";
            _modeBtn.BackColor = Color.PaleGreen;
        }
        else if (SelectedAlgorithm is not null)
        {
            Simulation.Start();

            _modeBtn.Text = "End";
            _modeBtn.BackColor = Color.DarkRed;
        }
    }
}
```

Here is the output of the shortest path in a block maze...



This shows BFS neglecting dense nodes, which are 'greyed out' here...



Testing

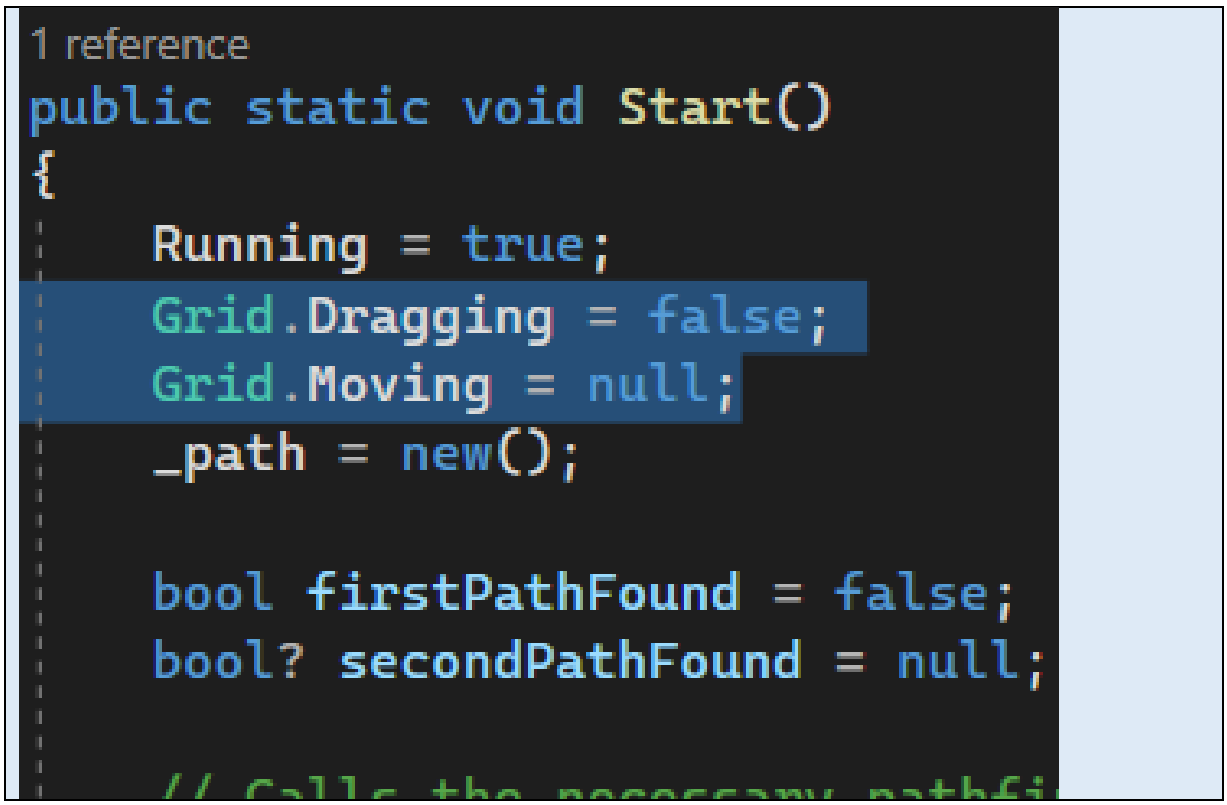
Test table

Test	Test details	Expected result	Actual result	Success /Fail	Comments
Simulation mode – Success Criteria: 14, 15, 4	Ran the simulation with each algorithm selected.	Each algorithm attempts to find the shortest path and displays the path if found.	Each algorithm attempts to find the shortest path and displays it.	**Success**	It is vital that the algorithms work at this stage, as it may be difficult to tell in future prototypes.
	Clicked the 'simulation button' and interacted with the grid and toolbar.	Nodes aren't changed and toolbar functions don't do anything. The simulation button updates and changes state.	Nodes aren't changed and toolbar functions don't do anything. The simulation button updates.	**Success**	It's important the grid cannot be edited by the user during a simulation.
	Clicked the 'simulation button' while dragging and moving.	When the simulation starts, dragging stops. When the simulation starts, the node being moved stops.	Dragging continues upon the mouse's re-entry to the grid. The node continues to move upon the mouse's re-entry to the grid.	**Fail**	See **Modification #1**.
	Clicked the 'simulation button' without	The simulation button doesn't update, and a path isn't drawn.	The simulation button doesn't update, and a path isn't drawn.	**Success**	The user needs to select which pathfinding algorithm they

	selecting an algorithm.				want to see visualised before a simulation can go ahead.
	Clicked the 'simulation button' after the shortest path is displayed.	All visualisation elements (e.g., path nodes) are cleared, the simulation button is updated, and the user can edit the grid.	All visualisation elements are cleared, the simulation button is updated, and the user can edit the grid.	**Success**	This includes 'disabled' dense nodes – they are returned to their original appearance.
	Clicked the 'simulation button' with the diversion node on the grid, and one of the nodes 'blocked in'.	No path is drawn, but the simulation still starts (and the 'simulation button' is updated).	No path is drawn when the simulation starts.	**Success**	Even if half the path is found (e.g., start to diversion), the path shouldn't be drawn since the whole path isn't found.
Algorithm selector – Success Criteria: 10, 11	Selected a different algorithm during a simulation.	The algorithm selector doesn't update.	The same algorithm remains selected.	**Success**	The user can't change the algorithm during the simulation.
	Selected 'breadth-first' and started a simulation.	The dense nodes on the grid turn partially transparent.	The dense nodes appear 'greyed out'.	**Success**	This is to indicate BFS ignores node cost and 'weighting'.

Modifications

Modification #1: To fix this, I set *Grid.Dragging* to false, and *Grid.Moving* to null within the *Start* method of the simulation class.



Review

User feedback

I have asked the main stakeholders (interviewees) some questions about this prototype and recorded their responses.

Q: What are your initial thoughts on the simulation?

- **Lee**: "This is a great start for the visualisation aspect of the program. The pathfinding algorithms seem to be working well."
- **Jude**: "I like making complex paths for the algorithms to find."

Q: Does this match how you imagined the pathfinding algorithms to behave?

- **Lee**: "Yes. We'll see how it behaves once the simulation speed and play/pause are added."
- **Jude**: "I don't really know."

Q: Is there anything you would change at this stage?

- **Lee**: "Not at this stage. I'm looking forward to the visualisation aspects."
- **Jude**: "No."

I don't think there is enough for the stakeholders to see to suggest modifications, since most of this prototype was coding and testing the pathfinding algorithms and setting up the logic for the simulation. I'm now ready to move onto adding visualisation to the simulation.

Reflection

I have successfully implemented the pathfinding algorithms and created some of the initial simulation rules and validations, including restricting the user's editing ability whilst a simulation is

running. I was able to verify the pathfinding algorithms worked by drawing a path on the grid – something that will adapt slightly in the next prototype as other visualisation features are added. No major modifications were needed at this stage, and this prototype didn't take as long as I expected it to. Next, I will implement all the visualisation features and logic.

Prototype 5 – Visualisation

Implementation

This prototype aims to build on top of and change/adapt the previous prototype to implement the visualisation processes, simulation speed and states, and the path recalculation mechanism as mentioned in the *success criteria*. The visualisation aspect will be a core part of this program; it is the whole point of this project and is mentioned numerous times throughout the report thus far.

Visualising steps

The first thing I did was add a private 'list of tuples' field to the simulation class called 'steps', which records all the steps each algorithm takes so that it can be visualised later. The first element of each tuple holds the node to be updated, and the second stores a string which determines whether the node is being 'closed' or 'opened'.

```
private static readonly List<(Node Node, string Action)> _steps = new();
```

With this now created, I added to the Dijkstra's algorithm and A* Search to add a tuple to the list when a node is 'opened' or 'closed'. 'Open' nodes represent nodes that may be evaluated in the future depending on its priority (lower overall cost equals higher priority), and 'closed' nodes represent nodes that have been evaluated (checked if it is the 'end' node and 'opened' all its neighbours).

```
// Removes it from 'open' and adds it to the 'closed' set.
open.Remove(current);
closed.Add(current);
_steps.Add((current, "close"));

// Path found if this node is the 'end node'.
if (ReferenceEquals(current, end))
    return true;

// Loops through the neighbours of the node.
foreach (var neighbour in current.Neighbours)
{
    // Skips the neighbour if it's not 'traversable' or has already been evaluated.
    if (neighbour.Type == NodeType.Block || closed.Contains(neighbour))
        continue;

    // If a neighbour is not in the open set, or there is a shorter path to it, it's properties are updated.
    if (!open.Contains(neighbour) || current.GCost + neighbour.AdjacentCost < neighbour.GCost)
    {
        neighbour.GCost = current.GCost + neighbour.AdjacentCost;
        neighbour.Parent = current;

        // If a neighbour is not in the open set, it's added.
        if (!open.Contains(neighbour))
        {
            open.Add(neighbour);
            _steps.Add((neighbour, "open"));
        }
    }
}
```

I also added to the BFS algorithm to make it store when a node is visited. BFS doesn't use a priority to determine the next node to visit, so there won't be any 'open' nodes added to the 'steps' list.

```
while (toCheck.Count > 0)
{
    // Next node in the queue to evaluate.
    Node current = toCheck.Dequeue();

    // If it is the end node, the path is found.
    if (ReferenceEquals(current, end))
        return true;

    // Loops through the neighbours of the node.
    foreach (var neighbour in current.Neighbours)
    {
        // If the neighbour is 'traversable' and hasn't been 'visited', it's added to the queue.
        if (neighbour.Type != NodeType.Block && !visited.Contains(neighbour))
        {
            toCheck.Enqueue(neighbour);
            visited.Add(neighbour);
            neighbour.Parent = current;
            _steps.Add((neighbour, "close"));
        }
    }
}
```

The next thing to do is create the *Open* and *Close* methods in the node class, which will update the label's back colour.

```
1 reference
public void Open()
{
    _lbl.BackColor = Color.GreenYellow;
}

1 reference
public void Close()
{
    _lbl.BackColor = Color.Yellow;
}

1 reference
public void Path()
{
    _lbl.BackColor = Color.Violet;
}
```

To start the visualisation process, I need to introduce a timer object, which I stored in a private field in the simulation class. I had to 'import' (*'using'* operator) the *Timers* namespace in order to create the timer object in this project. Also, I changed some logic in the *Start* method to clear the list fields instead of creating a new object every time (this meant I could also make them 'read-only'). Additionally, I set the initial speed of the visualiser and added an event handler that will run every 'interval' elapsed.

```
namespace PathfindingAlgorithmVisualiser
{
    using System.Timers;
```

```
    private static readonly List<(Node Node, string Action)> _steps = new();
    private static readonly List<Node> _path = new();
    private static readonly Timer _timer = new();

    1 reference
    public static void Start()
    {
        Running = true;
        Grid.Dragging = false;
        Grid.Moving = null;

        // Initialising and resetting variables.
        _timer.Interval = 100;
        _timer.Elapsed += Timer_Elapsed;
        _path.Clear();
        _steps.Clear();
```

I also added a private Boolean field that stores whether the path was found. After the algorithms are run in the *Start* method, I utilised this new field using the same logic (if statement) as in the previous prototype. Beneath this, I 'start' the timer to begin the visualisation process.

```
    private static bool _pathFound;
```

```
        // The path should only be drawn if the whole path is found.
        if (firstPathFound && (secondPathFound is null || secondPathFound == true))
            _pathFound = true;
        else
            _pathFound = false;

        _timer.Start();
    }
```

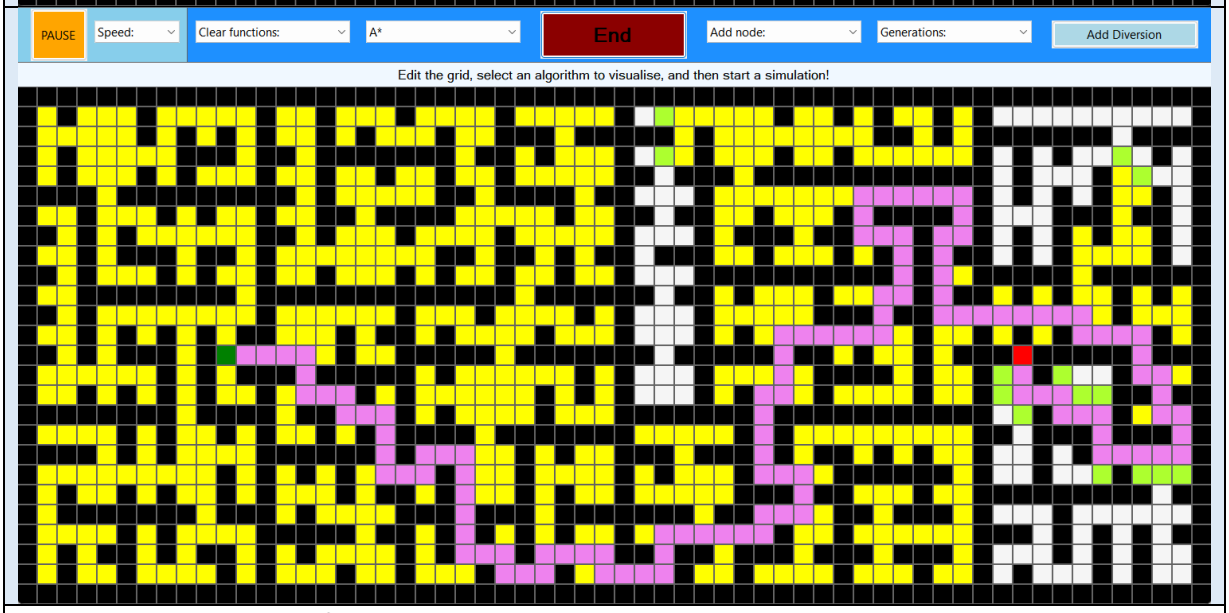
Within the timer event handler that fires every interval, the first element of *steps* is analysed and the relevant action is taken (*Open* or *Closed* is called on the node) depending on the stored 'action' in the tuple. The first element is then removed, so that in the next interval, the first element will be the next one in the list. If *steps* is empty, a path is drawn if *pathFound* is true, and then the timer is stopped as the visualisation is over.

```
    1 reference
    private static void Timer_Elapsed(object? sender, ElapsedEventArgs e)
    {
        if (_steps.Count == 0)
        {
            // Draws the path and stops the timer if all the steps have been shown.
            if (_pathFound)
            {
                foreach (var node in _path)
                    node.Path();
            }

            _timer.Stop();
        }
        else
        {
            // 'Visualises' the next algorithmic step.
            Node node = _steps[0].Node;

            if (node.Type != NodeType.Start && node.Type != NodeType.Target && node.Type != NodeType.Diversion)
            {
                if (_steps[0].Action == "close")
                    node.Close();
                else
                    node.Open();
            }

            _steps.RemoveAt(0);
        }
    }
```

Finally, I added to the *Stop* method to stop the timer in case the simulation is ended (mode button pressed) and the visualisation is still going.

```
1 reference
public static void Stop()
{
    // Stops the simulation.
    _timer.Stop();

    Grid.ClearVisualisation();
    Running = false;
}
```

I ran an A* Search simulation, and the visualisation works.



I noticed that the speed of the visualisation increased every time I started a new simulation. This was because I was adding a new event to the timer every time I called *Start*. This meant that

Timer_Elapsed was being called more and more with every run of the simulation, thus increasing the speed of the visualisation. To prevent this from happening, I created a static constructor and added the *Elapsed* event to the timer from within there instead. A static constructor works differently from regular constructors and is only used to initialise fields – a static class is never instantiated, and so it can't take parameters.

```
0 references
static Simulation()
{
    _timer.Elapsed += Timer_Elapsed;
}
```

Simulation state and speed

Now that the visualisation is working, I need to add functionality to the 'control bar' on the toolbar. First, I added public methods in the simulation class to pause and resume the simulation. I also created a public property, *Paused*, which is a Boolean that will be updated by these methods, and will be referenced in the toolbar class to update the 'play/pause button' appropriately – this is something I thought about a lot in the design section when making the simulation class diagram.

```
4 references
public static bool Paused { get; private set; }
```

```
1 reference
public static void Pause()
{
    _timer.Stop();
    Paused = true;
}

1 reference
public static void Resume()
{
    _timer.Start();
    Paused = false;
}
```

In order to register when the user has clicked the pause/play button, I added a *MouseClick* event to that button in the toolbar class. Within the event handler, the simulation is paused or resumed (depending on whether the simulation is already paused or not) and the label is updated appropriately.

```
1 reference
private static void CreateSimStateBtn(int width, int height, int xPos, int yPos, Form form)
{
    _simStateBtn.Location = new Point(xPos, yPos);
    _simStateBtn.Size = new Size(width, height);
    _simStateBtn.BackColor = Color.Orange;
    _simStateBtn.Text = "PAUSE";

    _simStateBtn.MouseClick += SimStateBtn_MouseClick;

    form.Controls.Add(_simStateBtn);
}

1 reference
private static void SimStateBtn_MouseClick(object? sender, MouseEventArgs e)
{
    // Pauses or resumes the simulation and updates the button's appearance.
    if (e.Button == MouseButtons.Left && Simulation.Running)
    {
        if (Simulation.Paused)
        {
            Simulation.Resume();

            _simStateBtn.Text = "PAUSE";
            _simStateBtn.BackColor = Color.Orange;
        }
        else
        {
            Simulation.Pause();

            _simStateBtn.Text = "PLAY";
            _simStateBtn.BackColor = Color.Lime;
        }
    }
}
```

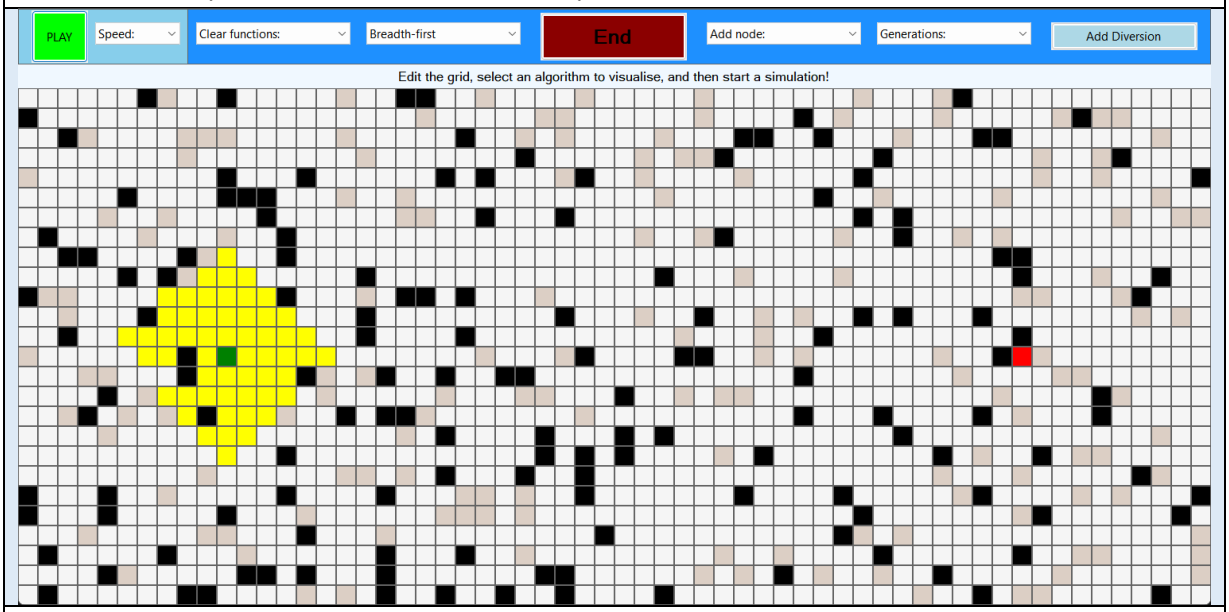
As well as this, I made the button display “Pause” when the simulation ends, so when a new simulation is run, the button isn’t displaying the incorrect text. To achieve this, I updated this label within the *MouseClick* event of the main simulation button to display its default, initial text.

```
1 reference
private static void ModeBtn_MouseClick(object? sender, MouseEventArgs e)
{
    // Starts or stops the simulation and updates the main button.
    if (e.Button == MouseButtons.Left)
    {
        if (Simulation.Running)
        {
            Simulation.Stop();

            _simStateBtn.Text = "PAUSE";
            _simStateBtn.BackColor = Color.Orange;
            _modeBtn.Text = "Visualise";
            _modeBtn.BackColor = Color.PaleGreen;
        }
        else if (SelectedAlgorithm is not null)
        {
            Simulation.Start();

            _modeBtn.Text = "End";
            _modeBtn.BackColor = Color.DarkRed;
        }
    }
}
```

I also set *Paused* to false within the *Start* method of the simulation class.

```
1 reference
public static void Start()
{
    Running = true;
    Paused = false;
    Grid.Dragging = false;
    Grid.Moving = null;
```

Here is an example of the simulation when it's paused...



The other important feature of the simulation, as mentioned many times throughout analysis and design, is the ability to change the visualisation speed. Originally, in my simulation class diagram in the design section, I thought that I would need to store the current simulation speed as a private field, but it's probably best stored as a property, since the toolbar shouldn't need to update the speed if the user selects the same speed again (as this may cause mini pauses in the visualisation) and so needs to be able read the current simulation speed to make that judgement. The *Speed* property will be an integer corresponding to the selected index of the 'speed combo box' on the toolbar.

```
8 references
public static int Speed { get; private set; }
```

I created a *ChangeSpeed* public method in the simulation class that contains a switch-statement to change the timer interval and *Speed* property based on the passed speed parameter.

```
1 reference
public static void ChangeSpeed(int speed)
{
    // Timer interval is changed depending on the selected speed.
    switch (speed)
    {
        case 1:
            Speed = 1;
            _timer.Interval = 1000;
            break;
        case 2:
            Speed = 2;
            _timer.Interval = 100;
            break;
        case 3:
            Speed = 3;
            _timer.Interval = 50;
            break;
        case 4:
            Speed = 4;
            _timer.Interval = 10;
            break;
        case 5:
            Speed = 5;
            _timer.Interval = 1;
            break;
    }
}
```

I then headed over to the toolbar class to give functionality to the speed selector. I bound an event to the combo box to detect when the user selects a new index. In its event handler, the selected index is checked, and if it is 0 (i.e., the description item is selected), the selected index is quickly changed back to the selected speed – this is to prevent the user getting confused if they forgot which speed they selected, and it wasn't being displayed. Otherwise, if the user selects a new speed, the *ChangeSpeed* method is called, passing the selected index as the new speed.

```
1 reference
private static void CreateSpeedCbx(int width, int xPos, int yPos, Form form)
{
    _speedCbx.Location = new Point(xPos, yPos);
    _speedCbx.Width = width;
    _speedCbx.DropDownStyle = ComboBoxStyle.DropDownList;

    _speedCbx.Items.Add("Speed:");
    _speedCbx.Items.Add("Very slow");
    _speedCbx.Items.Add("Slow");
    _speedCbx.Items.Add("Medium");
    _speedCbx.Items.Add("Fast");
    _speedCbx.Items.Add("Very fast");
    _speedCbx.Items.Add("Instant");
    _speedCbx.SelectedIndex = 0;

    _speedCbx.SelectionChangeCommitted += SpeedCbx_SelectionChangeCommitted;

    form.Controls.Add(_speedCbx);
}

1 reference
private static void SpeedCbx_SelectionChangeCommitted(object? sender, EventArgs e)
{
    // Detects when the user selects a new speed.
    if (_speedCbx.SelectedIndex == 0)
        _speedCbx.SelectedIndex = Simulation.Speed;
    else if (_speedCbx.SelectedIndex != Simulation.Speed)
        Simulation.ChangeSpeed(_speedCbx.SelectedIndex);
}
```

I also went back to the 'mode/simulation button' and added another condition to the simulation starting – the simulation speed (i.e., the 'speed combo box' selected index) cannot equal zero, as that means the user hasn't selected a simulation speed yet.

```
1 reference
private static void ModeBtn_MouseClick(object? sender, MouseEventArgs e)
{
    // Starts or stops the simulation and updates the main button.
    if (e.Button == MouseButtons.Left)
    {
        if (Simulation.Running)
        {
            Simulation.Stop();

            _simStateBtn.Text = "PAUSE";
            _simStateBtn.BackColor = Color.Orange;
            _modeBtn.Text = "Visualise";
            _modeBtn.BackColor = Color.PaleGreen;
        }
        else if (SelectedAlgorithm is not null && Simulation.Speed != 0)
        {
            Simulation.Start();

            _modeBtn.Text = "End";
            _modeBtn.BackColor = Color.DarkRed;
        }
    }
}
```

I have missed out one speed - 'instant'. To make an instant visualisation, I will have to make a separate method that does not involve the timer. I created a method that loops through all the steps and then outputs the path if a path has been found.

```
2 references
private static void InstantVisualise()
{
    // Instantly displays visualisation results.
    foreach (var step in _steps)
    {
        if (step.Node.Type != NodeType.Start && step.Node.Type != NodeType.Target && step.Node.Type != NodeType.Diversion)
        {
            if (step.Action == "close")
                step.Node.Close();
            else
                step.Node.Open();
        }
    }

    if (_pathFound)
        DrawPath();
}
```

This method calls *DrawPath*, a small private method I created that calls *Path* on all the nodes in the *path* list. It is the same code used previously in the *TimerElapsed* event handler – I put it in a method since it is now reused in different places.

```
2 references
private static void DrawPath()
{
    // Draws the path.
    foreach (var node in _path)
        node.Path();
}
```

```
1 reference
private static void Timer_Elapsed(
{
    if (_steps.Count == 0)
    {
        // Draws the path and stop
        if (_pathFound)
            DrawPath();

        _timer.Stop();
```

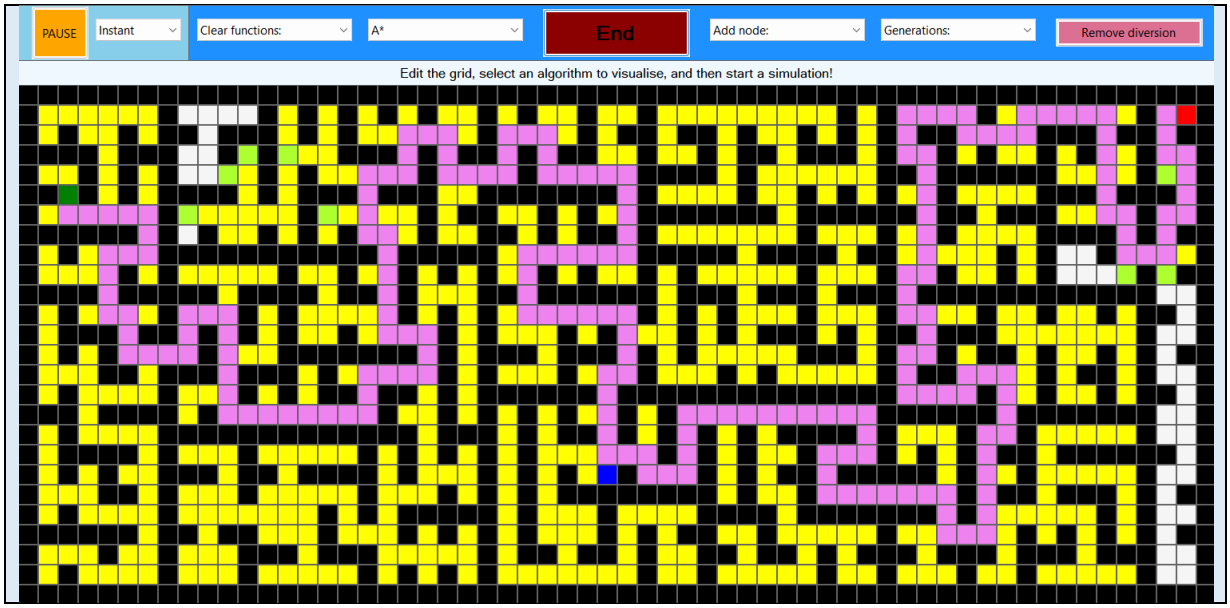
I then added another 'case' to the *ChangeSpeed* method that calls this method after stopping the timer. This is the final case, corresponding to the final speed list item - 'instant'.

```
case 6:
    Speed = 6;

    if (Running)
    {
        _timer.Stop();
        InstantVisualise();
    }
    break;
```

As well as the user being able to change the speed to 'instant' while a simulation is running, they should also be able to select 'instant' before a simulation starts, and then instantly see the results upon starting the simulation. To do this, a check is performed at the end of the *Start* method that determines whether to start the timer, or instantly visualise, depending on the current selected speed.

```
// The path should only be drawn if the whole path is found.
if (firstPathFound && (secondPathFound is null || secondPathFound == true))
    _pathFound = true;
else
    _pathFound = false;

if (Speed == 6)
    InstantVisualise();
else
    _timer.Start();
```

Again, like with many of the features, it's difficult to demonstrate the instant visualisation feature working with only a screenshot, but here it is anyway. Things like this will be shown in the post development testing video.



Instant path recalculation

I originally saw this feature in my *research*; I included it in the *success criteria* and planned my implementation of it throughout the design section. I started by adding a property in the simulation class called *Visualised*. It is a Boolean which will be true when all the 'steps' have been visualised and the path drawn (if a path was found), and the simulation is still 'running'. In the class diagrams of the design section, I discussed a property in the grid class ('CanMove') that would do a similar thing to this, but I think this property in the simulation class makes more sense.

```
5 references
public static bool Visualised { get; private set; }
```

I then created a public method in the simulation class that will run the instant recalculations. It clears the previous visualisations, deletes the contents of the *steps* and *path* lists, runs the algorithm, and then calls *InstantVisualise*.

```
1 reference
public static void InstantRecalculation()
{
    // Instantly shows new visuals and shortest path.
    Grid.ClearVisualisation();

    _path.Clear();
    _steps.Clear();

    RunAlgorithms();
    InstantVisualise();
}
```

RunAlgorithms is a private method I created that contains the same logic as the main bulk of the *Start* method – I moved it into this method because it is called more than once (code reusability). This cut down the size of the *Start* method, and I used this opportunity to move some of the 'resetting logic' into the *Stop* method, as it makes more sense in there. I have only taken a screenshot of the first part of *RunAlgorithms*, since it is a long method and is exactly the same as what was in the *Start*

method before. You can also see it in the code listings at the end of this report. Within the *Stop* method, I set *Visualised* to false.

```
2 references
private static void RunAlgorithms()
{
    bool firstPathFound = false;
    bool? secondPathFound = null;

    // Calls the necessary pathfinding algorithms and stores their results.
    if (Grid.Start is not null && Grid.Target is not null)
    {
        if (Toolbar.SelectedAlgorithm == Algorithm.Dijkstra)
        {
            if (Grid.Diversion is null)
            {
                firstPathFound = Dijkstra(Grid.Start, Grid.Target);
```

```
1 reference
public static void Start()
{
    Running = true;
    Paused = false;
    Grid.Dragging = false;
    Grid.Moving = null;

    // Runs an algorithm then visualises its processes.
    RunAlgorithms();

    if (Speed == 6)
        InstantVisualise();
    else
        _timer.Start();
}

1 reference
public static void Stop()
{
    _timer.Stop();

    // Resetting variables.
    _path.Clear();
    _steps.Clear();

    Visualised = false;
    Grid.ClearVisualisation();
    Running = false;
}
```

To signal when the visualisation has finished, *Visualised* is set to true at the end of the visualisation – within the *TimerElapsed* event handler and *InstantVisualisation*.

```
3 references
private static void InstantVisualise()
{
    // Instantly displays visualisation results.
    foreach (var step in _steps)
    {
        if (step.Node.Type != NodeType.Start && step.Node.Type != NodeType.Target && step.Node.Type != NodeType.Diversion)
        {
            if (step.Action == "close")
                step.Node.Close();
            else
                step.Node.Open();
        }
    }

    if (_pathFound)
        DrawPath();

    Visualised = true;
}

1 reference
private static void Timer_Elapsed(object? sender, ElapsedEventArgs e)
{
    if (_steps.Count == 0)
    {
        // Draws the path and stops the timer if all the steps have been shown.
        if (_pathFound)
            DrawPath();

        _timer.Stop();
        Visualised = true;
    }
```

Now that the visualisation can be tracked, I added some validation to the node class to allow the start, target and diversion nodes to be moved if the visualisation is complete.

```
1 reference
private void Lbl_MouseDown(object? sender, MouseEventArgs e)
{
    if (Simulation.Running && !Simulation.Visualised)
        return;

    // This makes it so that other labels can listen for 'MouseEnter' event while the mouse is still pressed.
    if (sender is Control control)
        control.Capture = false;

    // Determines if the user is moving or adding a node.
    if (Type == NodeType.Start || Type == NodeType.Target || Type == NodeType.Diversion)
    {
        if (e.Button == MouseButtons.Left)
            Grid.Moving = Type;
    }
    else if (!Simulation.Running)
    {
        UpdateLbl(e.Button);
        Grid.ButtonHolding = e.Button;
    }

    // Initiates the dragging process.
    Grid.Dragging = true;
}
```

I then called the instant path recalculation from the inside the *MouseEnter* event handler when the simulation has been visualised and a node moved.

```
1 reference
private void Lbl_MouseEnter(object? sender, EventArgs e)
{
    // The label will only be updated if the user is 'dragging' and not a 'core' node.
    if (Grid.Dragging && Type != NodeType.Start && Type != NodeType.Target && Type != NodeType.Diversion)
    {
        if (Grid.Moving == NodeType.Start)
            MakeStart();
        else if (Grid.Moving == NodeType.Target)
            MakeTarget();
        else if (Grid.Moving == NodeType.Diversion)
            MakeDiversion();
        else
            UpdateLbl(Grid.ButtonHolding);

        if (Simulation.Visualised)
            Simulation.InstantRecalculation();
    }
}
```

Testing

Test table

Test	Test details	Expected result	Actual result	Success /Fail	Comments
Visualisation – Success Criteria: 3	Started a simulation with each of the algorithms and medium speed selected.	Nodes change appearance incrementally until the path is found. Then the path is 'drawn'.	Nodes change appearance incrementally until the path is found and the path is drawn.	**Success**	Closed nodes are yellow, open nodes are green and path nodes are purple.
	Started a simulation with no possible path.	Nodes change appearance incrementally until all the nodes possible have been visited. No path is drawn.	Nodes change appearance incrementally until all the nodes possible have been visited. No path is drawn.	**Success**	It's important visualisation still occurs even if there is no path at the end of it.
	Started a simulation with the "Instant" speed selected.	The final result of the visualisation is shown straight away, along with the path (unless no path was found).	The visualisation is shown straight away, along with the path.	**Success**	The same logic/code is used in the instant recalculation mechanism.
Speed selector – Success Criteria: 18	Selected a new speed during a simulation.	The speed of the visualisation changes.	The speed of the visualisation changes.	**Success**	The 'speed' corresponds to the frequency of timer intervals.
	Selected the "Instant" speed during a simulation.	The incremented visual steps stop and the end result of the visualisation is displayed, along with the shortest path (if found).	The incremented visual steps stop and the end result of the visualisation is displayed, along with the shortest path.	**Success**	This may be useful for some users.
	Selected the "Speed:" list element during a simulation.	The simulation continues at the same speed. The 'speed combo box' doesn't update.	The simulation continues at the same speed. The 'speed combo box' doesn't update.	**Success**	This description element is only there as a placeholder at the beginning.
	Started the simulation without	The visualisation doesn't start. The simulation button doesn't update.	The visualisation doesn't start and the simulation	**Success**	Once the user selects a speed, they are not able to

	selecting a speed.		button doesn't update.		reselect the "Speed:" description element.
Sim state button – Success Criteria: 16, 17	Paused and resumed a simulation by clicking the 'sim state button'.	When the simulation is paused, the visualisation stops and the 'sim state button' updates to display "Play". When the simulation is resumed, the visualisation continues and the 'sim state button' updates to display "Pause".	When the simulation is paused, the visualisation stops and the 'sim state button' updates to display "Play". When the simulation is resumed, the visualisation continues and the 'sim state button' updates to display "Pause".	**Success**	Pausing and resuming a simulation is different from starting and ending one, as the visualisation progress isn't lost.
	Clicked the 'sim state button' when a simulation isn't running.	Nothing happens. The sim state button doesn't update.	Nothing happens.	**Success**	A simulation can only be paused and resumed if it is running.
	Clicked the 'sim state button' when a visualisation is complete, but the simulation is still running.	When a visualisation is complete, pausing and resuming have no effect (except updating the button appearance).	When a visualisation is complete, pausing and resuming have no effect if the selected speed isn't 'instant'. If 'instant' is selected, resuming the simulation will cause the 'steps' to be visualised again over the top of the result (at the previously selected speed).	**Fail**	See **Modification #1**.
	Ended a simulation while it is paused.	The visualisation is cleared (as usual) and the 'sim state button' is updated to display "Pause",	The visualisation is cleared and the 'sim state button' is updated to display "Pause".	**Success**	The sim state button needs to be reset to its initial properties when a

		ready for another simulation.			simulation ends.
Instant recalculation – Success Criteria: 8	Attempted to move the start, target and diversion nodes during visualisation.	Nothing happens.	Nothing happens.	**Success**	Only after a visualisation (and while a simulation is not running, of course) can these nodes be moved.
	Moved the start, target and diversion nodes around after visualisation, while a simulation is still running.	The 'instant recalculation' mechanism quickly displays the results of a visualisation (including the path, if found) with the new start, target and diversion positions.	The instant recalculation mechanism quickly displays the results of a visualisation with the new start, target and diversion positions in the grid.	**Success**	It is a little bit laggy (especially if you move too quickly), but it works well.
	Moved the start, target and diversion nodes around after visualisation and ending the simulation.	'Instant recalculation' doesn't occur – the nodes are moved around as normal.	Instant recalculation doesn't occur.	**Success**	Although the visualisation is complete, the simulation has ended, and so it's back to editing mode where the nodes are moved around as normal.
	Attempted to drag block/dense nodes around the grid after visualisation.	Nothing happens.	Nothing happens.	**Success**	Only moving nodes is allowed after a visualisation.
	Clicked the 'sim state button' at the end of an 'instant visualisation' and then moved nodes around repeatedly.	Pausing and resuming have no effect. 'Instant recalculation' occurs.	Incremental visualisation occurs during the instant recalculation. A random exception occurs if you continue to move nodes around. [see **error #5** in appendix]	**Fail**	See **Modification #1**.

Modifications

Modification #1: A couple tests failed because incremental visualisation started when the 'sim state button' was pressed after an 'instant visualisation' had occurred. This was because the 'steps' list in the simulation class hadn't been incrementally deleted by the 'timer elapsed event handler' since the visualisation was instant, and so when the simulation was then paused and played, the timer was started, which cycled through the not empty 'steps' list. This was a simple fix, as all that was required was to add another condition to the 'sim state button clicked event handler' in the toolbar class that ensured the simulation was only paused/resumed if the visualisation hadn't completed yet.

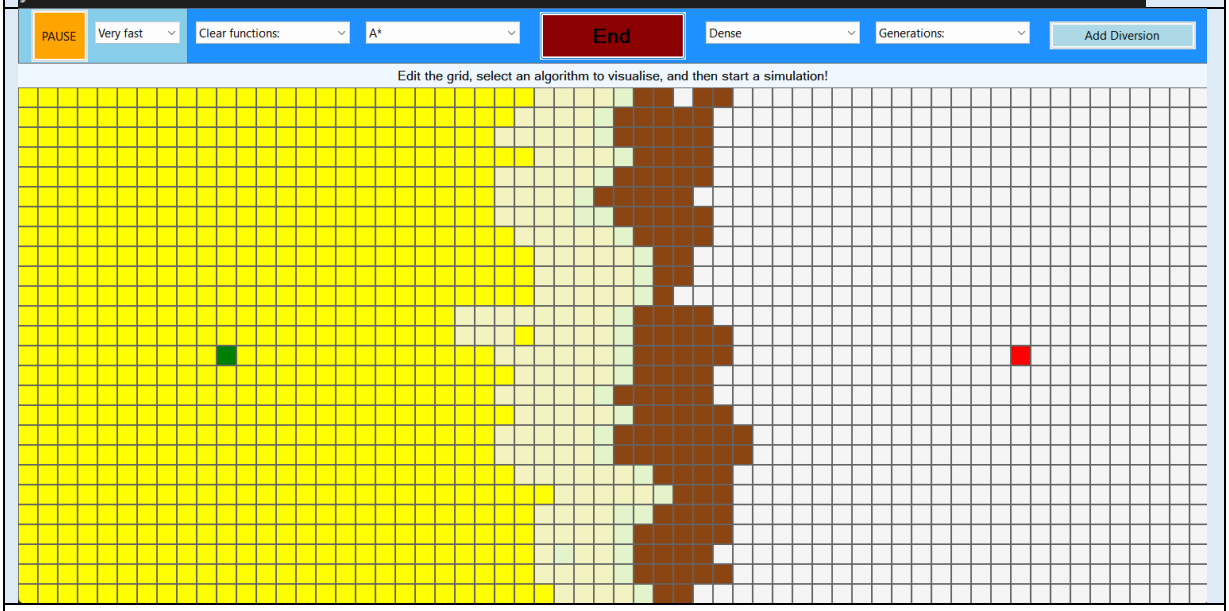
```
1 reference
private static void SimStateBtn_MouseClick(object? sender, MouseEventArgs e)
{
    // Pauses or resumes the simulation and updates the button's appearance.
    if (e.Button == MouseButtons.Left && Simulation.Running && !Simulation.Visualised)
    {
        if (Simulation.Paused)
        {
            Simulation.Resume();

            _simStateBtn.Text = "PAUSE";
            _simStateBtn.BackColor = Color.Orange;
        }
        else
        {
            Simulation.Pause();

            _simStateBtn.Text = "PLAY";
            _simStateBtn.BackColor = Color.Lime;
        }
    }
}
```

Modification #2: To make it easier to see dense nodes 'beneath' visualisations, I reused a method in the node class called *Disable* which I changed to be called *Dim*. I then called this method from within the simulation class on every dense node that was being 'opened' or 'closed', except when BFS was being simulated, since dense nodes will be disabled and not significant anyway. I also had to change the reference to the *Disable* method in the grid class to *Dim* (used in the *DisableDenseNodes* method).

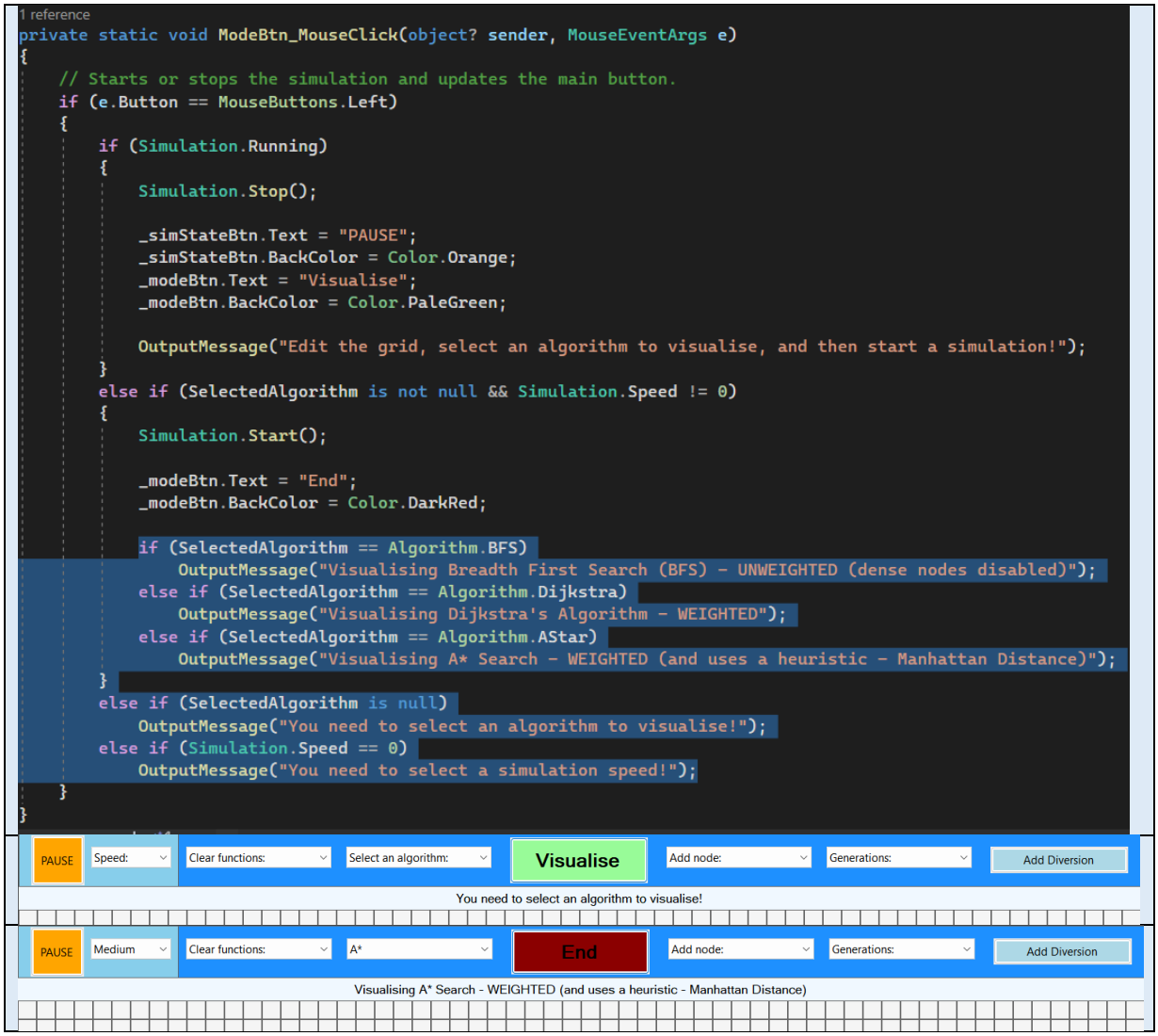
```
3 references
public void Dim()
{
    // Makes label semi-transparent.
    _lbl.BackColor = Color.FromArgb(50, _lbl.BackColor);
}
```

```
3 references
private static void InstantVisualise()
{
    // Instantly displays visualisation results.
    foreach (var step in _steps)
    {
        if (step.Node.Type != NodeType.Start && step.Node.Type != NodeType.Target && step.Node.Type != NodeType.Diversion)
        {
            if (step.Action == "close")
                step.Node.Close();
            else
                step.Node.Open();

            if (Toolbar.SelectedAlgorithm != Algorithm.BFS && step.Node.Type == NodeType.Dense)
                step.Node.Dim();
        }
    }

    if (_pathFound)
        DrawPath();

    Visualised = true;
}
```

```
1 reference
private static void Timer_Elapsed(object? sender, ElapsedEventArgs e)
{
    if (_steps.Count == 0)
    {
        // Draws the path and stops the timer if all the steps have been shown.
        if (_pathFound)
            DrawPath();

        _timer.Stop();
        Visualised = true;
    }
    else
    {
        // 'Visualises' the next algorithmic step.
        Node node = _steps[0].Node;

        if (node.Type != NodeType.Start && node.Type != NodeType.Target && node.Type != NodeType.Diversion)
        {
            if (_steps[0].Action == "close")
                node.Close();
            else
                node.Open();

            if (Toolbar.SelectedAlgorithm != Algorithm.BFS && node.Type == NodeType.Dense)
                node.Dim();
        }

        _steps.RemoveAt(0);
    }
}
```



Modification #3: Jude reminded me about the message output below the toolbar, which I will use to prompt the user and provide information about the pathfinding algorithms. I added prompts to select an algorithm and speed if they tried to click the 'visualise button' without selecting them. I also made the *OutputMessage* method private rather than public, since I don't end up using it outside the toolbar class.



Review

User feedback

I have asked the main stakeholders (interviewees) some questions about this prototype and recorded their responses.

Q: What are your initial thoughts on the visualisation?

- **Lee**: “I like the instant visualisation feature, and the options of simulation speeds.”
- **Jude**: “It’s very interesting to explore the different algorithms and see their processes.”

Q: Does this match how you imagined the visualisation to behave?

- **Lee**: “It goes beyond what I imagined. The different coloured visited nodes add to the depth and detail of the visualisation.”
- **Jude**: “Yes. It’s very helpful and engaging. It is easy to change the speed and pause it if I need to review what it’s done.”

Q: Is there anything you would change at this stage?

- **Lee**: “I think you should make it clearer when the visualisation is over dense nodes, as I can’t remember where they are exactly most of the time.”
- **Jude**: “The text under the toolbar doesn’t do anything. Didn’t you say you were going to make it output different text?”

Having taken the stakeholder’s feedback on board, I have made **Modification #2** in response to Lee’s improvement and **Modification #3** for Jude’s. I intended on updating the toolbar message throughout the code but forgot about it until Jude brought it up now. After showing the stakeholders my solutions, they were fairly happy for me to finish development there.

Reflection

In this final prototype, I have implemented visualisation using the intervals of a timer, simulation speed and state, and instant visualisation (including the instant recalculation mechanism). I have made a few modifications after testing and receiving user feedback, including fixing the pause/resume issue after a visualisation, making it clearer when a visualisation is over a dense node, and implementing messages to guide the user. I am very happy with the instant path recalculation because, although it is a bit laggy (WinForms isn’t great with performance), I thought it would be very difficult to implement, but it didn’t take long at all. This is a very important prototype that actually gives the program a purpose and, for timing reasons, will be the last prototype to seal off the program. I will now move onto the evaluation of the project, where I will first perform post development tests and consult my stakeholders and users.

Evaluation

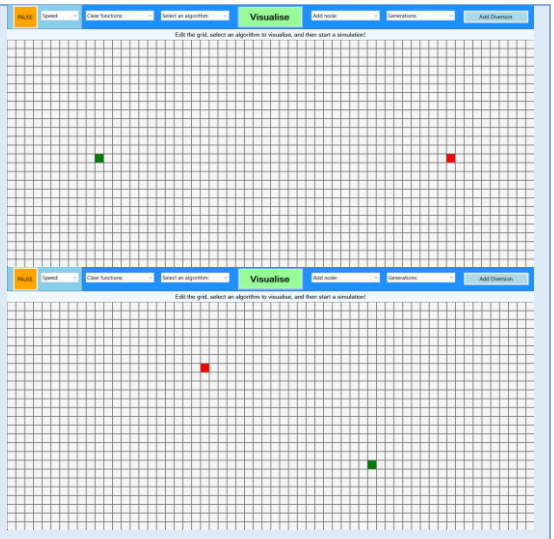
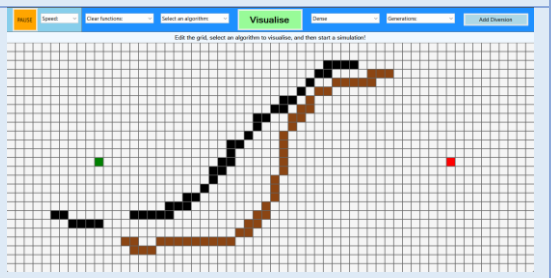
Beta Tests

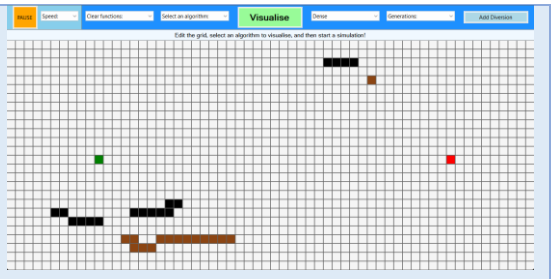
Test report

Now that the program is complete, I will perform final testing and validation checks with the stakeholders and end users. See *post development test data* at the end of the design section for more information.

Grid

Test	Test parameters	Usability testing	Evidence

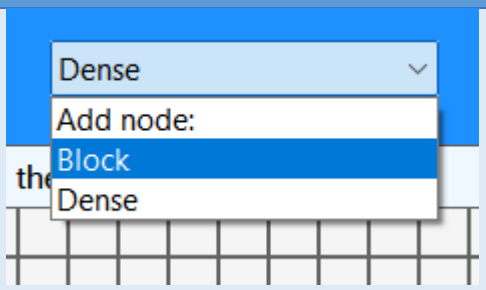
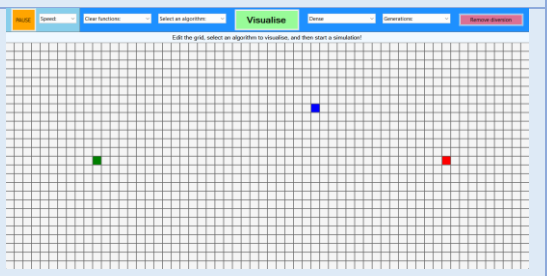
Moving nodes - success	**Valid** – (Left mouse button) User can move the start, target and diversion nodes when the simulation isn't running. **Extreme** – (Left mouse button) User can move the start, target and diversion nodes when a visualisation has ended, even if the simulation is running. **Extreme** – (Left mouse button) User can attempt to move the nodes off the grid, but they remain on the grid edges. **Invalid** – User cannot move the start, target and diversion nodes during a visualisation.	These nodes follow the mouse around the grid smoothly. They cannot be deleted or 'overlapped' on top of each other.	
Dragging - success	**Valid** – (Left mouse button) User can add block and dense nodes if a simulation isn't running. **Invalid** – User cannot add block and dense nodes during a simulation. **Invalid** – User cannot 'replace' the start, target and diversion nodes.	Dragging is smooth and intuitive. The user can select which node they want to add to the grid from the toolbar.	

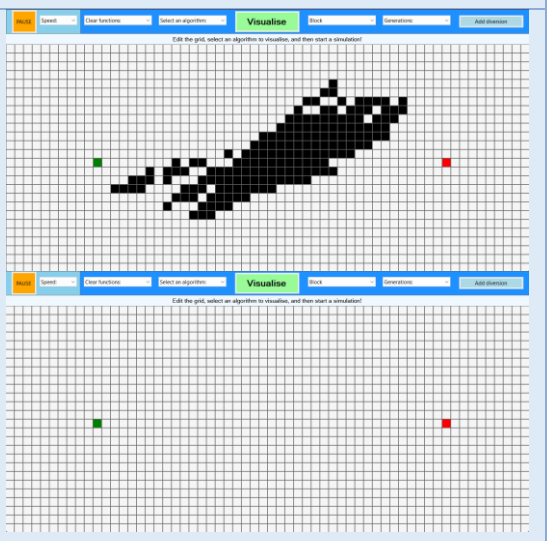
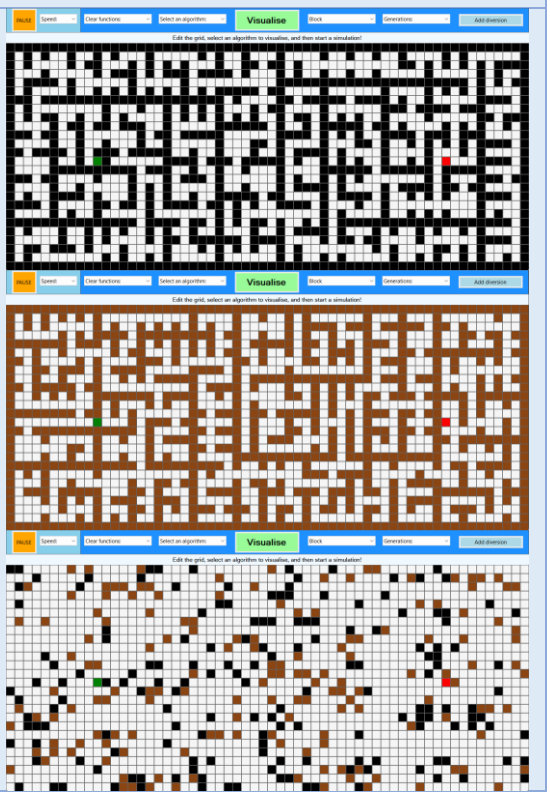
Clearing - success	**Valid** – (Right mouse button) User can remove block and dense nodes when a simulation isn't running (dragging feature also applies). **Invalid** – User cannot remove block and dense nodes during a simulation. **Invalid** – User cannot clear the start, target and diversion nodes.	The user can also clear multiple nodes by dragging.	

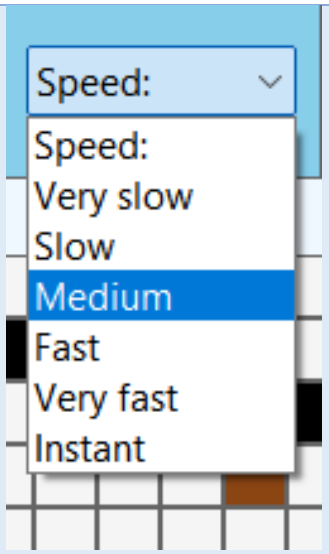
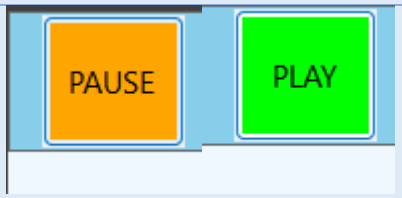
The usability of the grid is of a good standard. The user can drag their mouse to add and remove nodes from the grid, as well as move certain nodes around. This interaction is smooth and responsive.

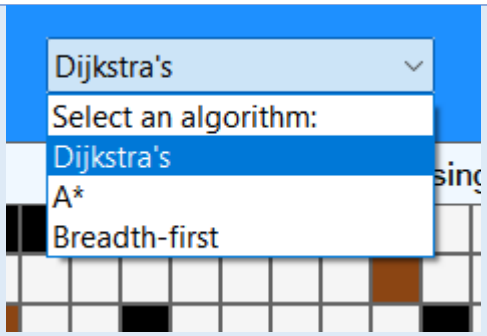
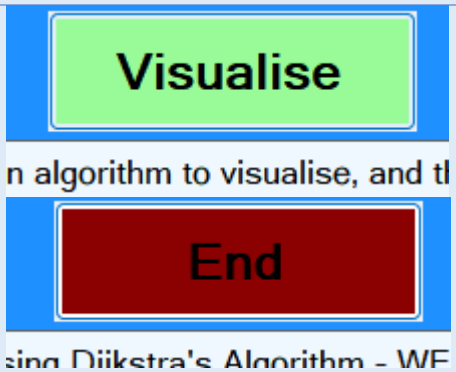
There isn't much validation in this section, but the validation performed was successful. The user cannot add or remove nodes during a simulation, and 'core nodes' cannot be deleted.

Toolbar

Test	Test parameters	Usability testing	Evidence
Node selection - success	**Valid** – User can select a node anytime. **Invalid** – No nodes are placed if the user doesn't select a node (i.e., first 'description element' of the selector is selected).	The node selector contains 'block' or 'dense' nodes. If the simulation isn't running, the user can 'place' the selected node on the grid.	
Diversion node - success	**Valid** – User can click the diversion button and its text updates and the diversion nodes are added to, or removed from, the grid if the simulation isn't running. **Invalid** – User cannot add or remove the	Adding and removing the diversion node is simple to do – the button is responsive and helpful.	

	diversion node from the grid if the simulation is running; the diversion button is unresponsive.		
Clearing functions - success	**Valid** – User can choose one of the clearing functions from the list if the simulation isn't running. The list is then updated to display the description element again. **Invalid** – User cannot choose one of the clearing functions if the simulation is running; selecting an item from the list doesn't do anything.	Performing clearing functions using the list on the toolbar allows the user to clear parts of the grid or reset the entire thing. These functions are helpful and descriptive for the user.	
Grid generations - success	**Valid** – User can choose one of the generations from the list if the simulation isn't running. The list is then updated to display the description element again. **Invalid** – User cannot choose one of the generations if the simulation is running; selecting and item from the list doesn't do anything.	Performing generation functions using the list on the toolbar offers a quick way to create a grid layout.	

Speed selector - success	**Valid** – User can select a simulation speed anytime. If a new speed is selected during a simulation, the visualisation interval changes appropriately. **Extreme** – User can select "Very slow". **Extreme** – User can select "Instant". In this case, the visualisation completes instantly. **Invalid** – User cannot deselect the speed (i.e., they cannot reselect the 'description element').	The speed of a simulation can be changed by the user in real time. The speed selector is segregated on the toolbar to stand out.	
Simulation state - success	**Valid** – User can click the 'sim state button' during a simulation and the button updates (to display "Pause" or "Play") and the simulation is paused or resumed. **Invalid** – User can't click the 'sim state button' if a simulation isn't running or a visualisation has finished; nothing happens if they do.	This button is also segregated from the rest of the toolbar to emphasise its importance in controlling the behaviour of the simulation.	

Algorithm selector - success	**Valid** – User can select an algorithm if the simulation isn't running. **Invalid** – User cannot select an algorithm if the simulation is running; if they do, the list reselects the current algorithm being simulated.	The algorithm list contains weighted and unweighted algorithms so that the user has a varied experience using each one.	
Mode button - success	**Valid** – If a simulation is running, clicking this button ends the simulation and updates the button. If a simulation isn't running, clicking this button starts a simulation and updates the button. **Extreme** – Clicking this button before the visualisation is complete ends the simulation early. **Invalid** – User can't click this button without selecting an algorithm; the user is prompted to select an algorithm first. **Invalid** – User can't click this button without selecting a speed; the user is prompted to select a speed first.	This button is very large and at the centre of the toolbar to get the attention of the user as it indicates whether a simulation is running or not.	

There are a lot of usability features in the toolbar that emphasise certain buttons and drop-down menus to the user, for example, sectioning off the simulation controls ('control bar'). This makes the program easier to use and more intuitive.

Validation was very important here as many functions can't occur whilst the simulation is running. Also, a simulation can only go ahead if an algorithm and speed is selected.

Simulation

Test	Test parameters	Usability testing	Evidence
Instant recalculation - success	**Valid** – User can move the start, target and diversion nodes around the grid if the visualisation is finished (but the simulation still 'running'), and an instant visualisation of the new node positions is shown (including the path if found). **Invalid** – User cannot move the nodes during a visualisation, and 'instant recalculation' doesn't occur after a simulation has ended.	As the user interacts with these nodes after a visualisation, the new visualisation is instantly displayed so the user can quickly see the algorithm's processes in that new situation. This is a very useful feature for some users.	

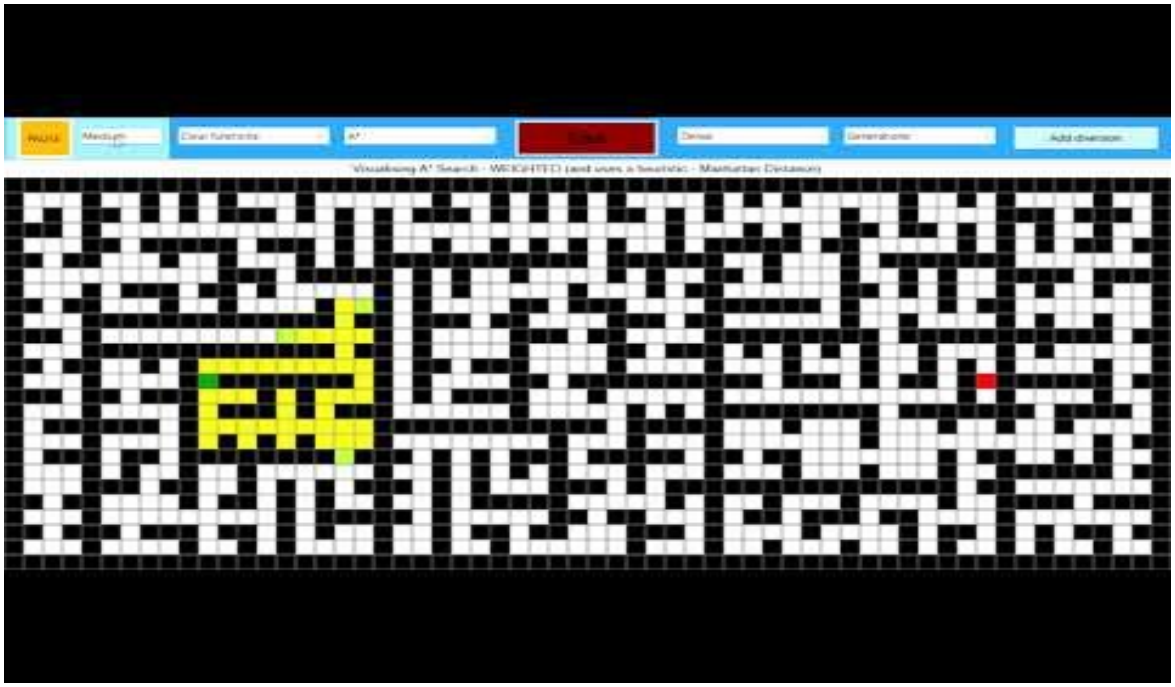
This feature was a success, as it is usable and useful for a lot of users. Also, the validation is tricky to get right here, but I managed to ensure the right restrictions and checks were made when implementing this during development.

Overall, all the post development tests were successful - the program was tested thoroughly and validated. This shows that the final program is robust and stable after development.

Test video

To provide extra evidence that this program works, I have made a test video where I casually run through the program, trying my best to show every feature and include validation checks:

[Pathfinding Algorithm Visualiser - Final Testing. A level coursework]



Stakeholder Review

Throughout development, Jude and Lee have given me important feedback at each development stage. They have also helped with the final/post development tests. Although Kieran has contributed a little bit (more so in the analysis stage), he hasn't really tested or seen much of the program yet. Therefore, I have chosen him, along with another end user (who will remain anonymous), to use the program for the first time so that I can better evaluate the usability and functionality of the program.

I have given these two people a set of instructions to follow that should cover most of the program's features, and a questionnaire to record their thoughts. All of this information, including the instructions and questions, can be found in the *post development test data* of the design section.

Interview

After these testers used the program and followed the instructions I provided, I asked them a series of questions (questionnaire):

Did you generally enjoy using the program?

"Yes, the visualisation was very cool and it piqued my interest."

"Yes, I loved making grid layouts and the maze generation."

Was the program intuitive and easy to use?

"Sometimes. The toolbar is easy to use, but the grid can get quite annoying as it sometimes doesn't detect when I stop dragging the mouse. The program is intuitive overall though."

"The grid was fairly easy to use, but the toolbar got some getting used to before I knew where everything was and what it did. I think some in-game instructions would be a good edition for beginners."

What did you think of the graphics?

"I like the simple approach to the node types and the bold colours of the main nodes to help them stand out. I liked the flow of the simulation and the overall visualisation."

"I thought the visualisation was okay but got confused many times with what the different colours meant. I think a colour code or key would help."

Did you like the sound effects?

N/A

Was the grid easy to manipulate?

"It was pretty good and the dragging feature was very useful. But there's just a few issues if you stop dragging outside the grid as it isn't recognised."

"I thought so. It was very intuitive and responsive to my intentions."

Was the algorithm visualisation clear?

"It was very clear and satisfying to see. I like how the path stands out once it's found."

"In certain situations, yeah. It looks a bit strange and complicated when there are dense nodes involved, and the overall look is quite confusing if you don't know what the nodes represent."

Was the program useful?

"Yes. It was very helpful to see the pathfinding algorithms working in real time. It's helped a little to consolidate my algorithm knowledge for A-level."

"A little bit. It's been an okay introduction to the algorithms, and looks like it could be very useful once you get the hang of it. Nevertheless, it was very fun to use."

Are there any improvements that could have been made?

"I think there should be a better mouse detection mechanic, so that dragging not stopping happens less frequently. It's also a little bit laggy when you try to move the nodes after a visualisation."

"I think there should be some instructions or keys, as well as more prompts and outputs from the toolbar."

Summary and conclusion

Overall, these testers enjoyed using the program and gave a lot of positive feedback, but also a few negatives and areas of improvement. Note that I didn't ask them the question about sound, since I didn't have time to implement it during development.

The users thought that the UI was built well – it was easy to use and navigate. They found the grid controls (mouse controls) intuitive and helpful and enjoyed creating different grid layouts using the toolbar functions along with manually adding/removing nodes. However, one user became frustrated with the grid dragging system, as sometimes it didn't recognise when they had released the mouse and intended to stop dragging. This is an issue that I discussed in *prototype 2*, but didn't find an effective solution to the problem other than importing libraries through *NuGet*. This is something I may consider doing during maintenance.

They generally liked the look of the toolbar, but one user found it overwhelming and a little confusing at first. They recommended I add some instructions that the user can access to help them get introduced to the toolbar functions and controls – this is something I will look into during maintenance. They also wanted more prompts from the message box, which is something I intended to include a lot of during development, but didn't get round to adding many.

Both users found the visualisation satisfying and appealing, but one user didn't really understand what all the nodes meant, especially when dense nodes were involved. They suggested I add a key or colour code. The other user also found the instant recalculation mechanism a bit laggy, which could either mean the mechanism wasn't coded well, or the framework (WinForms) isn't performance efficient in this regard. I will consider both of these suggestions in maintenance.

Success Criteria

Now that the program is finished, it is important I compare it to the *success criteria* in the analysis section to determine if my program was successful, or how much it differs from my original intentions and plans.

Grid

ID	Specification point	Review	Comments	Test reference
1	2-D grid of square nodes.	**Fully met**	Takes up most of the screen; there are plenty of nodes to utilise.	**Prototype 1**
2	The user can move the start, target and diversion nodes.	**Fully met**	This process is smooth and accurate; the nodes follow the mouse around as the user chooses.	**Prototype 2**
3	Nodes are different colours to represent their type and state.	**Fully met**	There is a variety of node types and visualisation states; the 'core nodes' have the boldest colours to stand out.	**Prototype 2** **Prototype 5**
5	The user can add block nodes.	**Fully met**	Left clicking or dragging. Block nodes are black.	**Prototype 2**
6	The user can add dense nodes.	**Fully met**	Left clicking or dragging. Dense nodes are brown.	**Prototype 2**

12	Mazes generations.	**Fully met**	Block or dense mazes do not obstruct core nodes – there is always a path to be found.	**Prototype 3**
13	Resetting the grid.	**Fully met**	When the grid is reset, all nodes are cleared, the diversion node is removed, and the start and target nodes are reset to their initial positions.	**Prototype 3**
21	Dragging feature allows the user to manipulate multiple nodes.	**Fully met**	Block and dense nodes can be added to the grid via dragging. Multiple nodes can be cleared via dragging.	**Prototype 2**
22	Right clicking clears nodes.	**Fully met**	The start, target and diversion nodes cannot be cleared.	**Prototype 2**

I have successfully created an interactive, responsive and detailed grid in my program. I have achieved all aspects of the success criteria relating to the grid, including moving nodes, adding and removing block and dense nodes, and 'deleting' nodes.

Although the dragging feature isn't always reliable, as I realised in development and was also pointed out in the *stakeholder review*, the issue is a problem with the detection system of WinForms. Also, the issue is minor and doesn't affect the user too much (they can simply click the grid again to disable dragging if this happens), and so isn't enough to not meet the criteria. There is a solution though, which I will discuss in maintenance, but it requires other libraries to be imported.

Toolbar

ID	Specification point	Review	Comments	Test reference
9	The toolbar is interactive and contains all the necessary controls and functions.	**Fully met**	The toolbar contains buttons and combo boxes; some controls are sectioned off, such as the 'control bar'.	**Prototype 1**
5	The user can select to add 'block' nodes to the grid.	**Fully met**	This is an option in a list on the toolbar.	**Prototype 2**
6	The user can select to add 'dense' nodes to the grid.	**Fully met**	This is an option in a list on the toolbar.	**Prototype 2**
7	The user can add and remove the diversion node.	**Fully met**	There is a 'diversion button' that adds and removes the diversion node.	**Prototype 2**
10	The toolbar contains an algorithm selector.	**Fully met**	The user needs to select an algorithm before starting a simulation.	**Prototype 1** **Prototype 4**
11	The algorithm selector contains weighted and unweighted pathfinding algorithms.	**Fully met**	There are three algorithms to choose from: Dijkstra's and A* (both weighted), and Breadth First (unweighted).	**Prototype 4**

12	Grid generations can be selected from the toolbar.	**Fully met**	The user can select to generate a block or dense maze, as well as a random grid layout.	**Prototype 3**
13	Grid clearing functions can be selected from the toolbar.	**Fully met**	The user can choose to clear all block or dense nodes, as well as completely reset the grid.	**Prototype 3**
14	The user can start a simulation.	**Fully met**	The main button on the toolbar will start the simulation.	**Prototype 4**
15	The user can end a simulation.	**Fully met**	The main button on the toolbar will end the simulation.	**Prototype 4**
16	The user can pause the simulation.	**Fully met**	The button on the control bar will pause the simulation; this button is unresponsive while a simulation isn't running.	**Prototype 5**
17	The user can resume the simulation.	**Fully met**	The button on the control bar will resume the simulation; this button is unresponsive while a simulation isn't running.	**Prototype 5**
18	The simulation speed can be adjusted.	**Fully met**	There is a list on the control bar that contains a selection of speeds, including an 'instant' speed; the simulation speed will update in real time.	**Prototype 5**

I have successfully created a helpful, engaging toolbar of controls and functions that can be used to manipulate the grid and control the simulation. It contains drop-down menus to generate grid layouts and clear aspects of the grid. It also contains a drop-down menu to equip a specific node type to add to the grid, and a button to add and remove the diversion node. It has an algorithm selector, speed selector, and a button to pause and resume a simulation.

There isn't a criterion for the message bar beneath the toolbar, since I only thought of it during design, but its implementation was kind of successful. I would've liked to use it more in the program but didn't get round to it.

Simulation

ID	Specification point	Review	Comments	Test reference
4	Each algorithm attempts to find the shortest path.	**Fully met**	If a path is found, the shortest path is drawn at the end of the visualisation.	**Prototype 4**
8	'Instant recalculation' is almost instant.	**Partially met**	Although this mechanism works, it is quite laggy, especially if nodes a moved quickly.	**Prototype 5**
3	Visualisation causes nodes to change colours.	**Fully met**	For example, open and closed nodes are different colours.	**Prototype 5**
18	Visualisation is incremental if a 'non-instant' speed is selected.	**Fully met**	The visualisation is smooth and progressive.	**Prototype 5**

The simulation mode was generally successful – it is robust and restrictive where necessary. The visualisation works well and portrays algorithmic processes effectively, and the user can easily manage the simulation's behaviour.

The "instant" recalculation mechanism works, but it is quite laggy, and so isn't really "instant". This could be down to the performance of WinForms but could also be because I didn't implement it in the best way. Either way, this criterion was only partially met - I will look into this further during maintenance. A possible solution to this, for further development, is to involve multiple threads in the code or use "asynchronous programming" techniques, which might help reduce lag as the program can do multiple things at once.

Sound and graphics

ID	Specification point	Review	Comments
19	Sound effects.	**Not met**	I didn't have time to implement this feature.
20	Path animation.	**Not met**	I didn't have time to implement this feature.

Sound effects and path animations were not implemented in this project, despite them being postulated throughout analysis and parts of the design section. This meant that these two criteria were not met. These features are not essential to the program, but would've added to the user experience and engagement with the resource – if I'd had more time, I would've implemented the sound first, and then the animations. I may discuss how these features might be added in the maintenance section. It probably wouldn't have been difficult to implement the sound, as WinForms has built in sound management systems, which I could've used in the form class. The animations would've been trickier though, but I could've made use of some built in transform or paint methods to do this.

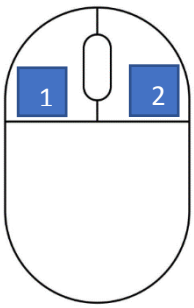
Apart from these features, the general graphics of the program is simple yet appealing. Node types and states are portrayed using colours, something that I decided from early on in the analysis section, where I often discussed how graphics won't be impressive or the focus of this project.

Usability Features

Implemented usability features

Controls

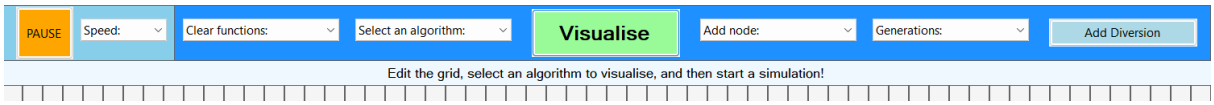
This project does not utilise any keyboard controls, and everything is controlled using the mouse and its buttons.

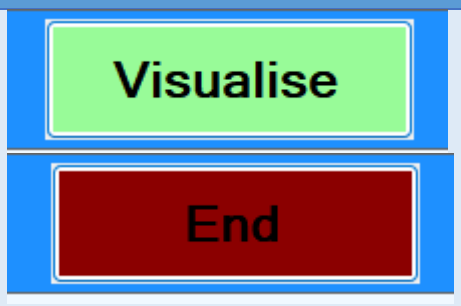
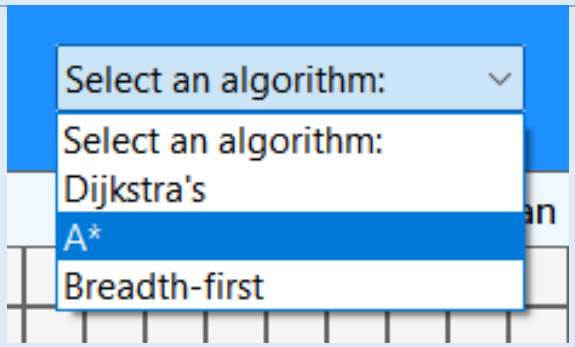
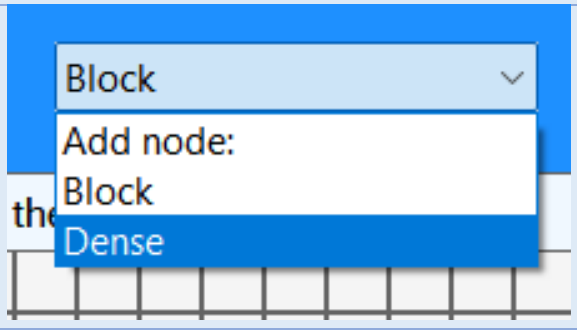


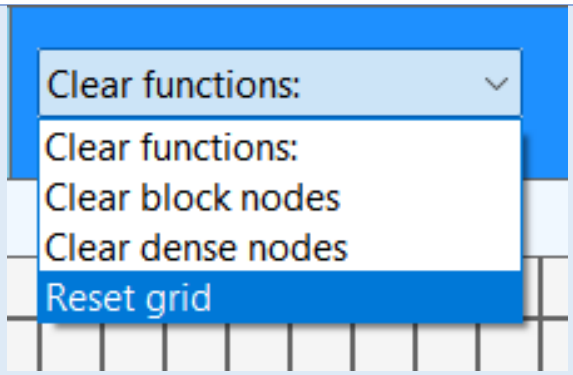
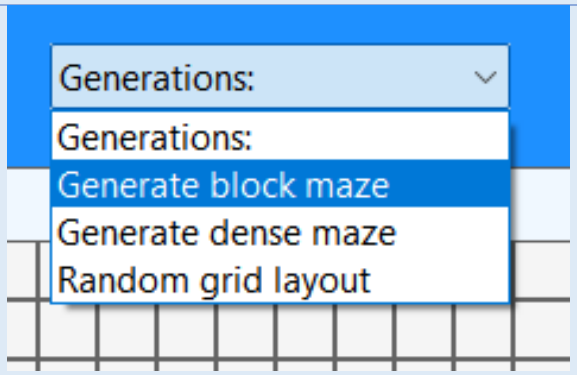
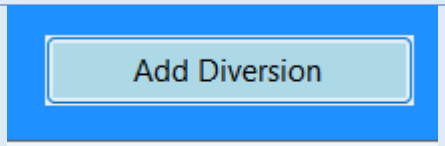
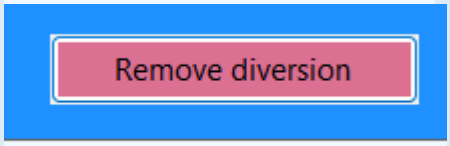
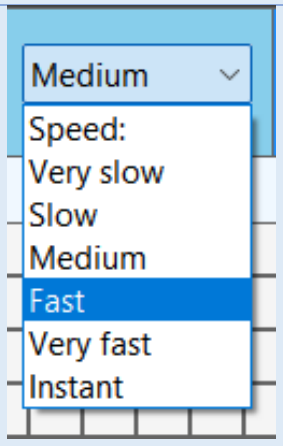
1) **Left mouse button** – most programs and applications (and even operating systems) use the left mouse button as the main button for navigating, selecting and editing. For this reason, I thought it was intuitive to have this button as the one that can move nodes, add nodes, and interact with toolbar buttons and menus.
2) **Right mouse button** – this button is used to clear nodes on the grid.

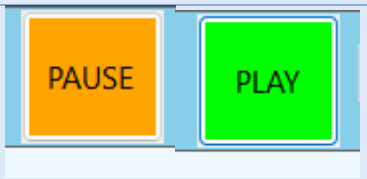
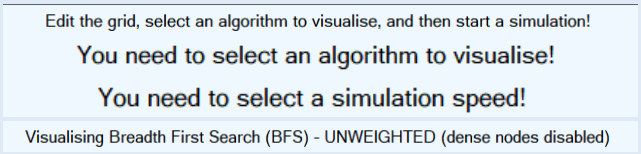
Toolbar

The toolbar is where most of the usability features lie. The algorithms and simulation can be controlled from the toolbar, as well as grid editing functions and node type selections. Overall, the toolbar contains many effective usability features, and is itself an effect UI element. It is fixed at the top of the screen and has a blue background to differentiate it from the grid and message strip below. The control bar (on the left) is a lighter shade of blue to aid the user visually by segregating it from the other controls and functions.



Feature	Effectiveness	Screenshots
Simulation button	This feature was very successful, as the button stands out at the centre of the toolbar to indicate its importance as the main button for switching 'modes'. It changes colour depending on whether the simulation is running or not, to aid the user in quickly seeing what mode it's in.	
Algorithm selector	This feature was fairly successful as it provides the user the multiple pathfinding algorithms to choose from. It was well validated to ensure the user cannot use it inappropriately. If I had a lot more time, or in further development, I would add more pathfinding algorithms to give the user a larger range to choose from.	
Node type selector	This is a successful feature that allows the user to determine which node will be 'placed' when they left click on the grid. The description element, "Add node:", is helpful in identifying what the list does.	

Clearing functions	This is a very successful feature that allows the user to clear specific aspects of the grid or completely reset it. Each function is responsive and the list resets to display the description element “Clear functions:” once a function is selected, so that the user can identify where the list is and what it does again.	
Generations list	This is a very successful feature that allows the user to procedurally generate block or dense mazes, or random layouts. Each function is responsive and the list resets to display the description element “Generations:” once a function is selected, so that the user can identify where the list is and what it does again.	
Diversion button	This is a successful feature that is intuitive and responsive. It allows the user to add and remove the diversion node, which is an important node that is heavily considered during a simulation. It has its own button to help portray its importance, and the changing colour adds to its interactability and usefulness.	 
Speed selector	This is a successful feature that allows the user to update the simulation speed before or during a simulation, as well as instantly ‘finish’ the visualisation process by selecting “instant”. Once a speed is selected, the user cannot reselect “Speed:”, so can’t accidentally click that description element. This control is grouped with the	

	‘sim state button’ to help the user navigate the toolbar and access similar functions.	
Sim state button	Also grouped with the speed selector, this successful feature is interactive and useful for pausing and resuming a simulation without ending it. Its background colour changes to aid the user visually with appropriate colours that are connoted to ‘stop’ and ‘go’.	
Message output	This feature was a partial success because it outputs messages that help the user understand what’s going on. For example, it tells the user to select and algorithm and simulation speed if they tried to start a simulation without doing so. It also displays information about each algorithm as it is visualised. However, it isn’t as useful as it could be. If I had more time, in further development, I would add messages when a visualisation is complete, or to display the final path ‘cost’ (if found), for example.	

Unachieved usability features

As discussed already, one unachieved (or not fully achieved) feature is the message output strip below the toolbar. Although it is working and helpful in many circumstances, it isn’t fully fletched and less helpful than I would’ve liked (as also pointed out in the *stakeholder review*). In further development, it wouldn’t be too difficult to add more triggers where the message box is updated to guide the user even further, and to make the program more user friendly.

Another lacking feature is the sound. There isn’t any sound in the program, which may make it a little boring sometimes. In further development, I could implement a couple sounds, such as when a path is or isn’t found during a simulation, to make the program more engaging and interesting. I originally intended on adding sound to the program (it was in the success criteria), but I ran out of time during development and didn’t want to hold up my stakeholders and end user who weren’t that bothered about sound in the first place anyway.

One feature that I wished I could've implemented better was the description elements of each drop-down menu. In the program, the first element of each list is a description of the list and doesn't have any functionality. I wanted to have the description as text in the combo box, but not its own element in the list; I couldn't find a way to do this though. A better solution, which could be implemented in further development, is to add a label above each menu to describe what it is, so then a list element doesn't have to be created to hold the description.

One user, in the ***stakeholder review***, thought that some instructions should be added to the program, as well as a node state colour key. I agree with this, and it would be a useful thing to implement, in further development, to help guide beginners to the program or to pathfinding algorithms generally (as this program is aimed at students as well as experienced computer scientists). I would create a separate form (window) for the instructions and link it to a button on the toolbar. The colour key could be on the output strip, along with an algorithm's information whilst a simulation is running.

Another user found a problem with the mouse controls and dragging system, which I also found during development. This was an issue with the WinForms mouse detection system, which isn't always accurate. I didn't have time to fix it then, since it required importing and learning other libraries and documentation. I will discuss a possible fix in the next section – maintenance.

Maintenance

This final section will focus on how the program can be maintained, any potential improvements during further development and the limitations that may cause.

Most limitations of my program are due to time management issues and inaccuracies with WinForms. In further development, these limitations can be reduced and it's important the program is maintained as future releases are given to the clients and end users, so that security issues or errors don't arise, and bugs are frequently patched.

Here are some limitations and maintenance that may be considered, along with some ideas for further development:

Limitations and maintenance	Further development
Improve 'mouse up' detection	I didn't fix this issue due to it taking up too much time, but it can be fixed by importing a *NuGet* package that will deal with 'mouse hooks'. This means that the mouse interaction can be detected anywhere in the program, and so there is less chance of a 'mouse up instance' not being detected. This may solve the dragging issue, where sometimes a 'mouse up' isn't recognised.
Instructions and colour codes	I didn't think much about implementing instructions or colour codes, but now that it's been addressed by end users, I think it would benefit a lot of people. The instructions can be held on another form and accessed via the toolbar, and the colour codes (for node states) can be with the instructions or on the toolbar message strip.

Reduce 'instant recalculation' lag	Again, this issue was highlighted by the end users. It could be an issue with the performance of WinForms controls, but could also be an issue with how I have implemented it – I didn't really have time to redesign the mechanism. A possible solution, in further development, could be to incorporate an "asynchronous programming" approach, and utilise multiple threads to increase throughput and multitask during certain parts of the program.
More message outputs	Although this wasn't part of the success criteria, I intended on implementing this since the design section of this project. However, due to time management issues and other priorities during development, I didn't get round to adding much functionality to the output message strip. It would be very easy to add more messages to this control in the future, such as commenting when a visualisation is complete, whether a path is found, and maybe even the path 'cost'.
Sound effects and animation	These criteria were unmet due to time management issues and their lower priority during development. It would be quite easy to implement sound, such as when a path has been drawn or a path hasn't been found, to improve the user experience and engagement. It would be slightly trickier to implement animation, as it requires moving labels smoothly, which is quite tough in WinForms – it could be done though, using certain libraries or built-in classes.
Simulation speed to fast	This issue hasn't been mentioned yet during development or evaluation, and it is a very rare occurrence. Sometimes, when the "Very fast" speed is selected, some labels are missed out by the visualisation because the *TimerElapsed* method is called too frequently, which means some of them don't have time to update to label. This could be a maintenance issue, but can be solved by increasing the timer intervals for those higher speeds, sacrificing speed for accuracy (in fact, I noticed there is little difference between "Fast" and "Very fast", except I've never seen this bug appear when "Fast" is selected).
Different colours for diversion visualisation	This is a future development idea that I have come up with during evaluation of the program. I noticed that it is quite confusing when the diversion node starts visualising, since it sometimes overlaps the 'start to diversion' visualisation and node states, and it can become difficult to see which process was performed at which stage. To fix this, I would create another 'steps' list to record the processes

	from the diversion node to target node, and then output those steps using different label colours, such as light blue and pink, for example. This will make the visualisation more contrasting and easier to understand when complex situations are being processed involving the diversion node.

Appendix

Error Report

Error	Description	Changes made
#1 – Warning	*_nodeGrid* is a non-nullable field, and so it cannot have a null value when "existing the constructor". Since it doesn't have a non-null value until *DrawGrid* is called, this warning is flagged.	Made it nullable using the nullable operator after the data type, so that it can be null until *DrawGrid* is called.
#2 – Exception	*NullReferenceException* is thrown because it tried to call a method on an object reference that wasn't set to an instance of an object.	Used a null-conditional operator to only perform the action if the object reference is non-null.
#3 – Logic	The *MouseEnter* event doesn't fire when another control is 'captured', such as when the mouse button is held down after clicking a label. This meant the 'dragging' feature didn't work.	Within the *MouseDown* event handler, I set the 'capture' property of the *sender* control to false.
#4 – Warning	*Parent* is a nullable property in the node class, and so a non-nullable data type cannot hold the value of this property if it is null.	*Parent* property is no longer nullable – its default value is itself ("*this*"), which is assigned in the constructor.
#5 - Exception	*System.InvalidOperationException: 'Collection was modified; enumeration operation may not execute.'*	Modifications in prototype #5.

Code Listings

```
namespace PathfindingAlgorithmVisualiser
{
    partial class Form1
    {
        private void InitializeComponent()
        {
            SuspendLayout();
            AutoScaleDimensions = new SizeF(8F, 20F);
            AutoScaleMode = AutoScaleMode.Font;
            ClientSize = new Size(1500, 750);
            Name = "Form1";
            Text = "Pathfinding Algorithm Visualiser";
            Load += Form1_Load;
            ResumeLayout(false);
```

```
        }
    }
}
```

```
namespace PathfindingAlgorithmVisualiser
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            CenterToScreen();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            const int NODE_SIZE = 25;
            const int TOOLBAR_HEIGHT = 100;
            const int MESSAGE_HEIGHT = 30;

            // Calculates how many columns and rows there should be in the grid.
            int gridCols = ClientSize.Width / NODE_SIZE;
            int gridRows = (ClientSize.Height - TOOLBAR_HEIGHT) / NODE_SIZE;

            Toolbar.CreateToolbar(ClientSize.Width, TOOLBAR_HEIGHT, MESSAGE_HEIGHT, this);
            Grid.DrawGrid(gridCols, gridRows, NODE_SIZE, TOOLBAR_HEIGHT, this);
        }
    }
}
```

```
namespace PathfindingAlgorithmVisualiser
{
    public class Node
    {
        public NodeType Type { get; private set; }
        public NodeType PreviousNodeType { get; set; }

        public List<Node> Neighbours { get; set; }
        public Node Parent { get; set; }

        public int Column { get; private set; }
        public int Row { get; private set; }
        public int AdjacentCost { get; private set; }
        public int GCost { get; set; }

        private readonly Label _lbl = new();

        public Node(int size, int x, int y, int col, int row, Form form)
        {
            Type = NodeType.Empty;
            Neighbours = new List<Node>();
            AdjacentCost = 1;
            Column = col;
            Row = row;
            Parent = this;

            CreateLbl(size, x, y, form);
        }

        public void Open()
        {
            _lbl.BackColor = Color.GreenYellow;
        }

        public void Close()
        {
            _lbl.BackColor = Color.Yellow;
        }

        public void Path()
        {
            _lbl.BackColor = Color.Violet;
        }

        public void Dim()
        {
```

```
            // Makes label semi-transparent.
            _lbl.BackColor = Color.FromArgb(50, _lbl.BackColor);
        }

        public void Reset()
        {
            Type = NodeType.Empty;
            AdjacentCost = 1;

            _lbl.BackColor = Color.WhiteSmoke;
        }

        public void MakeStart()
        {
            if (Grid.Start is not null)
            {
                if (Grid.Moving is not null && Grid.Start.PreviousNodeType == NodeType.Block)
                    Grid.Start.MakeBlock();
                else if (Grid.Moving is not null && Grid.Start.PreviousNodeType == NodeType.Dense)
                    Grid.Start.MakeDense();
                else
                    Grid.Start.Reset();
            }

            PreviousNodeType = Type;
            Type = NodeType.Start;

            _lbl.BackColor = Color.Green;

            Grid.Start = this;
        }

        public void MakeTarget()
        {
            if (Grid.Target is not null)
            {
                if (Grid.Moving is not null && Grid.Target.PreviousNodeType == NodeType.Block)
                    Grid.Target.MakeBlock();
                else if (Grid.Moving is not null && Grid.Target.PreviousNodeType == NodeType.Dense)
                    Grid.Target.MakeDense();
                else
                    Grid.Target.Reset();
            }

            PreviousNodeType = Type;
            Type = NodeType.Target;

            _lbl.BackColor = Color.Red;

            Grid.Target = this;
        }

        public void MakeBlock()
        {
            Type = NodeType.Block;
            _lbl.BackColor = Color.Black;
        }

        public void MakeDense()
        {
            Type = NodeType.Dense;
            AdjacentCost = 15;

            _lbl.BackColor = Color.SaddleBrown;
        }

        private void MakeDiversion()
        {
            if (Grid.Diversion is not null)
            {
                if (Grid.Moving is not null && Grid.Diversion.PreviousNodeType == NodeType.Block)
                    Grid.Diversion.MakeBlock();
                else if (Grid.Moving is not null && Grid.Diversion.PreviousNodeType ==
NodeType.Dense)
```

```
                    Grid.Diversion.MakeDense();
                else
                    Grid.Diversion.Reset();
            }

            PreviousNodeType = Type;
            Type = NodeType.Diversion;

            _lbl.BackColor = Color.Blue;

            Grid.Diversion = this;
        }

        private void CreateLbl(int size, int x, int y, Form form)
        {
            // Nodes should be square, so label height and width are the same.
            _lbl.Size = new Size(size, size);
            _lbl.Location = new Point(x, y);
            _lbl.BorderStyle = BorderStyle.FixedSingle;
            _lbl.BackColor = Color.WhiteSmoke;

            // Binding label events and handlers.
            _lbl.MouseDown += Lbl_MouseDown;
            _lbl.MouseEnter += Lbl_MouseEnter;
            _lbl.MouseUp += Lbl_MouseUp;

            // Adds the label to the form so that it's visible to the user.
            form.Controls.Add(_lbl);
        }

        private void Lbl_MouseUp(object? sender, MouseEventArgs e)
        {
            // User has stopped dragging.
            Grid.Dragging = false;
            Grid.ButtonHolding = null;
            Grid.Moving = null;
        }

        private void Lbl_MouseEnter(object? sender, EventArgs e)
        {
            // The label will only be updated if the user is 'dragging' and not a 'core' node.
            if (Grid.Dragging && Type != NodeType.Start && Type != NodeType.Target && Type !=
NodeType.Diversion)
            {
                if (Grid.Moving == NodeType.Start)
                    MakeStart();
                else if (Grid.Moving == NodeType.Target)
                    MakeTarget();
                else if (Grid.Moving == NodeType.Diversion)
                    MakeDiversion();
                else
                    UpdateLbl(Grid.ButtonHolding);

                if (Simulation.Visualised)
                    Simulation.InstantRecalculation();
            }
        }

        private void Lbl_MouseDown(object? sender, MouseEventArgs e)
        {
            if (Simulation.Running && !Simulation.Visualised)
                return;

            // This makes it so that other labels can listen for 'MouseEnter' event while the mouse
is still pressed.
            if (sender is Control control)
                control.Capture = false;

            // Determines if the user is moving or adding a node.
            if (Type == NodeType.Start || Type == NodeType.Target || Type == NodeType.Diversion)
            {
                if (e.Button == MouseButtons.Left)
                    Grid.Moving = Type;
            }
```

```
            else if (!Simulation.Running)
            {
                UpdateLbl(e.Button);
                Grid.ButtonHolding = e.Button;
            }

            // Initiates the dragging process.
            Grid.Dragging = true;
        }

        private void UpdateLbl(MouseButtons? button)
        {
            // Edits node based on which mouse button is pressed, and which node type is selected
from the toolbar.
            if (button == MouseButtons.Left)
            {
                if (Toolbar.SelectedNodeType == NodeType.Block)
                    MakeBlock();
                else if (Toolbar.SelectedNodeType == NodeType.Dense)
                    MakeDense();
            }
            else if (button == MouseButtons.Right)
                Reset();
        }
    }
}
```

```
namespace PathfindingAlgorithmVisualiser
{
    public static class Grid
    {
        public static bool Dragging { get; set; }
        public static NodeType? Moving { get; set; }
        public static MouseButtons? ButtonHolding { get; set; }

        public static Node? Start { get; set; }
        public static Node? Target { get; set; }
        public static Node? Diversion { get; set; }

        // This field is null until 'DrawGrid' is called, so a '?' operator is needed.
        private static Node[,]? _nodeGrid;

        public static void ClearVisualisation()
        {
            // Clears all visualisations and back colours.
            Clear(NodeType.Empty);

            if (_nodeGrid is not null)
            {
                foreach (var node in _nodeGrid)
                {
                    if (node.Type == NodeType.Dense)
                        node.MakeDense();
                }
            }
        }

        public static void DisableDenseNodes()
        {
            // Makes all dense nodes in the grid semi-transparent.
            if (_nodeGrid is not null)
            {
                foreach (var node in _nodeGrid)
                {
                    if (node.Type == NodeType.Dense)
                        node.Dim();
                }
            }
        }

        public static void Clear(NodeType nodeType)
        {
            // Resets the nodes in the grid with the node type specified by the parameter.
            if (_nodeGrid is not null)
            {
```

```
                foreach (var node in _nodeGrid)
                {
                    if (node.Type == nodeType)
                        node.Reset();
                    else if (node.PreviousNodeType == nodeType && (node.Type == NodeType.Start ||
node.Type == NodeType.Target || node.Type == NodeType.Diversion))
                        node.PreviousNodeType = NodeType.Empty;
                }
            }
        }

        public static void Reset()
        {
            // Clears the entire grid and resets it to its initial layout.
            if (_nodeGrid is not null)
            {
                foreach (var node in _nodeGrid)
                    node.Reset();

                Toolbar.RemoveDiversion();

                _nodeGrid[_nodeGrid.GetLength(0) / 6, _nodeGrid.GetLength(1) / 2].MakeStart();
                _nodeGrid[(_nodeGrid.GetLength(0) / 6) * 5, _nodeGrid.GetLength(1) / 2].MakeTarget();
            }
        }

        public static void DrawGrid(int cols, int rows, int nodeSize, int offsetY, Form form)
        {
            _nodeGrid = new Node[cols, rows];

            for (int i = 0; i < cols; i++)
            {
                for (int j = 0; j < rows; j++)
                {
                    // Creates a node object at each index in the 2D array 'nodeGrid'.
                    // The y-position of each node is offset to leave space for the toolbar.
                    var node = new Node(nodeSize, i * nodeSize, j * nodeSize + offsetY, i, j, form);
                    _nodeGrid[i, j] = node;
                }
            }

            // Sets the neighbours of each node.
            for (int i = 0; i < cols; i++)
            {
                for (int j = 0; j < rows; j++)
                {
                    Node node = _nodeGrid[i, j];

                    if (i > 0)
                        node.Neighbours.Add(_nodeGrid[i - 1, j]);

                    if (i < _nodeGrid.GetLength(0) - 1)
                        node.Neighbours.Add(_nodeGrid[i + 1, j]);

                    if (j > 0)
                        node.Neighbours.Add(_nodeGrid[i, j - 1]);

                    if (j < _nodeGrid.GetLength(1) - 1)
                        node.Neighbours.Add(_nodeGrid[i, j + 1]);
                }
            }

            // Setting the initial positions of the start and target nodes.
            _nodeGrid[cols / 6, rows / 2].MakeStart();
            _nodeGrid[(cols / 6) * 5, rows / 2].MakeTarget();
        }

        public static void RandomGeneration()
        {
            if (_nodeGrid is not null)
            {
                var rnd = new Random();

                // Loops through each node in the grid, making most empty.
```

```
                foreach (var node in _nodeGrid)
                {
                    // Random number between 0 and 9.
                    int ranNum = rnd.Next(10);

                    if (node.Type != NodeType.Start && node.Type != NodeType.Target && node.Type !=
NodeType.Diversion)
                    {
                        if (ranNum == 0)
                            node.MakeBlock();
                        else if (ranNum == 1)
                            node.MakeDense();
                        else
                            node.Reset();
                    }
                }
            }
        }

        public static void MazeGeneration(NodeType nodeType)
        {
            // Clears the grid first.
            Clear(NodeType.Block);
            Clear(NodeType.Dense);

            // Creates a border around the grid.
            if (_nodeGrid is not null)
            {
                for (int i = 0; i < _nodeGrid.GetLength(0); i++)
                {
                    Node topNode = _nodeGrid[i, 0];
                    Node bottomNode = _nodeGrid[i, _nodeGrid.GetLength(1) - 1];

                    if (nodeType == NodeType.Block)
                    {
                        if (topNode.Type != NodeType.Start && topNode.Type != NodeType.Target &&
topNode.Type != NodeType.Diversion)
                            topNode.MakeBlock();

                        if (bottomNode.Type != NodeType.Start && bottomNode.Type != NodeType.Target
&& bottomNode.Type != NodeType.Diversion)
                            bottomNode.MakeBlock();
                    }
                    else if (nodeType == NodeType.Dense)
                    {
                        if (topNode.Type != NodeType.Start && topNode.Type != NodeType.Target &&
topNode.Type != NodeType.Diversion)
                            topNode.MakeDense();

                        if (bottomNode.Type != NodeType.Start && bottomNode.Type != NodeType.Target
&& bottomNode.Type != NodeType.Diversion)
                            bottomNode.MakeDense();
                    }
                }

                for (int i = 0; i < _nodeGrid.GetLength(1); i++)
                {
                    Node leftNode = _nodeGrid[0, i];
                    Node rightNode = _nodeGrid[_nodeGrid.GetLength(0) - 1, i];

                    if (nodeType == NodeType.Block)
                    {
                        if (leftNode.Type != NodeType.Start && leftNode.Type != NodeType.Target &&
leftNode.Type != NodeType.Diversion)
                            leftNode.MakeBlock();

                        if (rightNode.Type != NodeType.Start && rightNode.Type != NodeType.Target &&
rightNode.Type != NodeType.Diversion)
                            rightNode.MakeBlock();
                    }
                    else if (nodeType == NodeType.Dense)
                    {
                        if (leftNode.Type != NodeType.Start && leftNode.Type != NodeType.Target &&
leftNode.Type != NodeType.Diversion)
```

```
                            leftNode.MakeDense();

                        if (rightNode.Type != NodeType.Start && rightNode.Type != NodeType.Target &&
rightNode.Type != NodeType.Diversion)
                            rightNode.MakeDense();
                    }
                }

                var rnd = new Random();
                var gaps = new List<(int X, int Y)>();

                // Starts the recursive division algorithm.
                Division(1, 1, _nodeGrid.GetLength(0) - 2, _nodeGrid.GetLength(1) - 2, true, gaps,
rnd, nodeType);
            }
        }

        private static void Division(int colStart, int rowStart, int colEnd, int rowEnd, bool
previousVertical, List<(int X, int Y)> gaps, Random rnd, NodeType nodeType)
        {
            // If the area is too small, it stops 'dividing'.
            if (colEnd - colStart == 0 || rowEnd - rowStart == 0 || (colEnd - colStart == 1 && rowEnd
- rowStart == 1) || _nodeGrid is null)
                return;

            var possibleWalls = new List<int>();
            bool vertical;

            // Determines which orientation the new 'wall' should be.
            if (colEnd - colStart == rowEnd - rowStart)
                vertical = previousVertical;
            else if (colEnd - colStart > rowEnd - rowStart)
                vertical = true;
            else
                vertical = false;

            // Checks all possible 'wall' placements and randomly chooses one. Creates a random gap
in the wall and stores its position. Calls 'Division', inputting the new areas.
            // The 'wall' created is dependent on orientation.
            if (vertical)
            {
                for (int i = colStart + 1; i < colEnd; i++)
                {
                    if (!gaps.Contains((i, rowStart - 1)) && !gaps.Contains((i, rowEnd + 1)))
                        possibleWalls.Add(i);
                }

                if (possibleWalls.Count == 0)
                    return;

                int wallX = possibleWalls[rnd.Next(possibleWalls.Count)];
                int gapY = rnd.Next(rowStart, rowEnd + 1);

                for (int i = rowStart; i <= rowEnd; i++)
                {
                    Node node = _nodeGrid[wallX, i];

                    if (node.Type != NodeType.Start && node.Type != NodeType.Target && node.Type !=
NodeType.Diversion)
                    {
                        if (nodeType == NodeType.Block)
                            node.MakeBlock();
                        else if (nodeType == NodeType.Dense)
                            node.MakeDense();
                    }
                }

                Node gap = _nodeGrid[wallX, gapY];
                gaps.Add((wallX, gapY));

                if (gap.Type != NodeType.Start && gap.Type != NodeType.Target && gap.Type !=
NodeType.Diversion)
                    gap.Reset();
```

```
                Division(colStart, rowStart, wallX - 1, rowEnd, vertical, gaps, rnd, nodeType);
                Division(wallX + 1, rowStart, colEnd, rowEnd, vertical, gaps, rnd, nodeType);
            }
            else
            {
                for (int i = rowStart + 1; i < rowEnd; i++)
                {
                    if (!gaps.Contains((colStart - 1, i)) && !gaps.Contains((colEnd + 1, i)))
                        possibleWalls.Add(i);
                }

                if (possibleWalls.Count == 0)
                    return;

                int wallY = possibleWalls[rnd.Next(possibleWalls.Count)];
                int gapX = rnd.Next(colStart, colEnd + 1);

                for (int i = colStart; i <= colEnd; i++)
                {
                    Node node = _nodeGrid[i, wallY];

                    if (node.Type != NodeType.Start && node.Type != NodeType.Target && node.Type !=
NodeType.Diversion)
                    {
                        if (nodeType == NodeType.Block)
                            node.MakeBlock();
                        else if (nodeType == NodeType.Dense)
                            node.MakeDense();
                    }
                }

                Node gap = _nodeGrid[gapX, wallY];
                gaps.Add((gapX, wallY));

                if (gap.Type != NodeType.Start && gap.Type != NodeType.Target && gap.Type !=
NodeType.Diversion)
                    gap.Reset();

                Division(colStart, rowStart, colEnd, wallY - 1, vertical, gaps, rnd, nodeType);
                Division(colStart, wallY + 1, colEnd, rowEnd, vertical, gaps, rnd, nodeType);
            }
        }
    }
}
```

```
namespace PathfindingAlgorithmVisualiser
{
    public static class Toolbar
    {
        public static NodeType? SelectedNodeType { get; private set; }
        public static Algorithm? SelectedAlgorithm { get; private set; }

        private static readonly Label _messageLbl = new();

        private static readonly ComboBox _speedCbx = new();
        private static readonly ComboBox _clearCbx = new();
        private static readonly ComboBox _algorithmsCbx = new();
        private static readonly ComboBox _nodeTypeCbx = new();
        private static readonly ComboBox _generationsCbx = new();

        private static readonly Button _modeBtn = new();
        private static readonly Button _diversionBtn = new();
        private static readonly Button _simStateBtn = new();

        public static void CreateToolbar(int width, int height, int messageHeight, Form form)
        {
            // Useful values for enabling consistent spacing of controls.
            int mainBarHeight = height - messageHeight;
            int sectionWidth = width / 7;

            // These two labels are just aesthetic and don't have functionality, so they are quickly
made here.
            var mainBar = new Label()
            {
                Size = new Size(width, mainBarHeight),
```

```
                BackColor = Color.DodgerBlue,
                BorderStyle = BorderStyle.FixedSingle,
            };

            var simControlBar = new Label()
            {
                Size = new Size(sectionWidth, mainBarHeight),
                BackColor = Color.SkyBlue,
                BorderStyle = BorderStyle.FixedSingle,
            };

            // Each method creates a toolbar control.
            CreateMessageLbl(width, messageHeight, 0, mainBarHeight, form);

            CreateModeBtn(sectionWidth - 30, mainBarHeight - 10, (sectionWidth * 3) + 15, 5, form);
            CreateDiversionBtn(sectionWidth - 30, mainBarHeight / 2, (sectionWidth * 6) + 15,
mainBarHeight / 4, form);
            CreateSimStateBtn(sectionWidth / 3, mainBarHeight - 4, sectionWidth / 12, 2, form);

            CreateSpeedCbx(sectionWidth / 2, (sectionWidth / 2) - 10, mainBarHeight / 4, form);
            CreateClearCbx(sectionWidth - 20, sectionWidth + 10, mainBarHeight / 4, form);
            CreateAlgorithmsCbx(sectionWidth - 20, (sectionWidth * 2) + 10, mainBarHeight / 4, form);
            CreateNodeTypeCbx(sectionWidth - 20, (sectionWidth * 4) + 10, mainBarHeight / 4, form);
            CreateGenerationsCbx(sectionWidth - 20, (sectionWidth * 5) + 10, mainBarHeight / 4,
form);

            // 'simControlBar' is added to the form after all the controls are created so that
'simStateBtn' and 'speedCbx' can be seen.
            form.Controls.Add(simControlBar);
            // 'mainBar' is added to the form last so that it is at the 'back' and all other controls
can be seen.
            form.Controls.Add(mainBar);
        }

        public static void RemoveDiversion()
        {
            // Updates the 'diversion button' and removes references to the diversion node.
            if (Grid.Diversion is not null)
            {
                Grid.Dragging = false;
                Grid.Moving = null;
                Grid.Diversion.Reset();
                Grid.Diversion = null;

                _diversionBtn.Text = "Add diversion";
                _diversionBtn.BackColor = Color.LightBlue;
            }
        }

        private static void OutputMessage(string message)
        {
            _messageLbl.Text = message;
        }

        private static void CreateMessageLbl(int width, int height, int xPos, int yPos, Form form)
        {
            _messageLbl.Location = new Point(xPos, yPos);
            _messageLbl.Size = new Size(width, height);
            _messageLbl.BackColor = Color.AliceBlue;
            _messageLbl.Font = new Font(FontFamily.GenericSansSerif, 10);
            _messageLbl.TextAlign = ContentAlignment.MiddleCenter;

            form.Controls.Add(_messageLbl);
            OutputMessage("Edit the grid, select an algorithm to visualise, and then start a
simulation!");
        }

        private static void CreateModeBtn(int width, int height, int xPos, int yPos, Form form)
        {
            _modeBtn.Location = new Point(xPos, yPos);
            _modeBtn.Size = new Size(width, height);
            _modeBtn.BackColor = Color.PaleGreen;
            _modeBtn.Font = new Font(FontFamily.GenericSansSerif, 15, FontStyle.Bold);
            _modeBtn.Text = "Visualise";
```

```
            _modeBtn.MouseClick += ModeBtn_MouseClick;

            form.Controls.Add(_modeBtn);
        }

        private static void ModeBtn_MouseClick(object? sender, MouseEventArgs e)
        {
            // Starts or stops the simulation and updates the main button.
            if (e.Button == MouseButtons.Left)
            {
                if (Simulation.Running)
                {
                    Simulation.Stop();

                    _simStateBtn.Text = "PAUSE";
                    _simStateBtn.BackColor = Color.Orange;
                    _modeBtn.Text = "Visualise";
                    _modeBtn.BackColor = Color.PaleGreen;

                    OutputMessage("Edit the grid, select an algorithm to visualise, and then start a
simulation!");
                }
                else if (SelectedAlgorithm is not null && Simulation.Speed != 0)
                {
                    Simulation.Start();

                    _modeBtn.Text = "End";
                    _modeBtn.BackColor = Color.DarkRed;

                    if (SelectedAlgorithm == Algorithm.BFS)
                        OutputMessage("Visualising Breadth First Search (BFS) - UNWEIGHTED (dense
nodes disabled)");
                    else if (SelectedAlgorithm == Algorithm.Dijkstra)
                        OutputMessage("Visualising Dijkstra's Algorithm - WEIGHTED");
                    else if (SelectedAlgorithm == Algorithm.AStar)
                        OutputMessage("Visualising A* Search - WEIGHTED (and uses a heuristic -
Manhattan Distance)");
                }
                else if (SelectedAlgorithm is null)
                    OutputMessage("You need to select an algorithm to visualise!");
                else if (Simulation.Speed == 0)
                    OutputMessage("You need to select a simulation speed!");
            }
        }

        private static void CreateDiversionBtn(int width, int height, int xPos, int yPos, Form form)
        {
            _diversionBtn.Location = new Point(xPos, yPos);
            _diversionBtn.Size = new Size(width, height);
            _diversionBtn.BackColor = Color.LightBlue;
            _diversionBtn.Text = "Add Diversion";

            _diversionBtn.MouseClick += DiversionBtn_MouseClick;

            form.Controls.Add(_diversionBtn);
        }

        private static void DiversionBtn_MouseClick(object? sender, MouseEventArgs e)
        {
            // Determines whether the user is adding or removing the diversion node.
            if (e.Button == MouseButtons.Left && !Simulation.Running)
            {
                if (Grid.Diversion is null)
                {
                    Grid.Moving = NodeType.Diversion;
                    Grid.Dragging = true;

                    _diversionBtn.Text = "Remove diversion";
                    _diversionBtn.BackColor = Color.PaleVioletRed;
                }
                else
                    RemoveDiversion();
            }
```

```
        }

        private static void CreateSimStateBtn(int width, int height, int xPos, int yPos, Form form)
        {
            _simStateBtn.Location = new Point(xPos, yPos);
            _simStateBtn.Size = new Size(width, height);
            _simStateBtn.BackColor = Color.Orange;
            _simStateBtn.Text = "PAUSE";

            _simStateBtn.MouseClick += SimStateBtn_MouseClick;

            form.Controls.Add(_simStateBtn);
        }

        private static void SimStateBtn_MouseClick(object? sender, MouseEventArgs e)
        {
            // Pauses or resumes the simulation and updates the button's appearance.
            if (e.Button == MouseButtons.Left && Simulation.Running && !Simulation.Visualised)
            {
                if (Simulation.Paused)
                {
                    Simulation.Resume();

                    _simStateBtn.Text = "PAUSE";
                    _simStateBtn.BackColor = Color.Orange;
                }
                else
                {
                    Simulation.Pause();

                    _simStateBtn.Text = "PLAY";
                    _simStateBtn.BackColor = Color.Lime;
                }
            }
        }

        private static void CreateSpeedCbx(int width, int xPos, int yPos, Form form)
        {
            _speedCbx.Location = new Point(xPos, yPos);
            _speedCbx.Width = width;
            _speedCbx.DropDownStyle = ComboBoxStyle.DropDownList;

            _speedCbx.Items.Add("Speed:");
            _speedCbx.Items.Add("Very slow");
            _speedCbx.Items.Add("Slow");
            _speedCbx.Items.Add("Medium");
            _speedCbx.Items.Add("Fast");
            _speedCbx.Items.Add("Very fast");
            _speedCbx.Items.Add("Instant");
            _speedCbx.SelectedIndex = 0;

            _speedCbx.SelectionChangeCommitted += SpeedCbx_SelectionChangeCommitted;

            form.Controls.Add(_speedCbx);
        }

        private static void SpeedCbx_SelectionChangeCommitted(object? sender, EventArgs e)
        {
            // Detects when the user selects a new speed.
            if (_speedCbx.SelectedIndex == 0)
                _speedCbx.SelectedIndex = Simulation.Speed;
            else if (_speedCbx.SelectedIndex != Simulation.Speed)
                Simulation.ChangeSpeed(_speedCbx.SelectedIndex);
        }

        private static void CreateClearCbx(int width, int xPos, int yPos, Form form)
        {
            _clearCbx.Location = new Point(xPos, yPos);
            _clearCbx.Width = width;
            _clearCbx.DropDownStyle = ComboBoxStyle.DropDownList;

            _clearCbx.Items.Add("Clear functions:");
            _clearCbx.Items.Add("Clear block nodes");
            _clearCbx.Items.Add("Clear dense nodes");
```

```
            _clearCbx.Items.Add("Reset grid");
            _clearCbx.SelectedIndex = 0;

            _clearCbx.SelectionChangeCommitted += ClearCbx_SelectionChangeCommitted;

            form.Controls.Add(_clearCbx);
        }

        private static void ClearCbx_SelectionChangeCommitted(object? sender, EventArgs e)
        {
            // Decides which clearing function to call.
            if (!Simulation.Running)
            {
                if (_clearCbx.SelectedIndex == 1)
                    Grid.Clear(NodeType.Block);
                else if (_clearCbx.SelectedIndex == 2)
                    Grid.Clear(NodeType.Dense);
                else if (_clearCbx.SelectedIndex == 3)
                    Grid.Reset();
            }

            // Resets the selected index back to the description.
            _clearCbx.SelectedIndex = 0;
        }

        private static void CreateAlgorithmsCbx(int width, int xPos, int yPos, Form form)
        {
            _algorithmsCbx.Location = new Point(xPos, yPos);
            _algorithmsCbx.Width = width;
            _algorithmsCbx.DropDownStyle = ComboBoxStyle.DropDownList;

            _algorithmsCbx.Items.Add("Select an algorithm:");
            _algorithmsCbx.Items.Add("Dijkstra's");
            _algorithmsCbx.Items.Add("A*");
            _algorithmsCbx.Items.Add("Breadth-first");
            _algorithmsCbx.SelectedIndex = 0;

            _algorithmsCbx.SelectionChangeCommitted += AlgorithmsCbx_SelectionChangeCommitted;

            form.Controls.Add(_algorithmsCbx);
        }

        private static void AlgorithmsCbx_SelectionChangeCommitted(object? sender, EventArgs e)
        {
            // Determines which algorithm the user wishes to visualise.
            if (!Simulation.Running)
            {
                if (_algorithmsCbx.SelectedIndex == 1)
                    SelectedAlgorithm = Algorithm.Dijkstra;
                else if (_algorithmsCbx.SelectedIndex == 2)
                    SelectedAlgorithm = Algorithm.AStar;
                else if (_algorithmsCbx.SelectedIndex == 3)
                    SelectedAlgorithm = Algorithm.BFS;
                else
                    SelectedAlgorithm = null;
            }
            else if (SelectedAlgorithm is not null)
                _algorithmsCbx.SelectedIndex = (int)SelectedAlgorithm + 1;
        }

        private static void CreateNodeTypeCbx(int width, int xPos, int yPos, Form form)
        {
            _nodeTypeCbx.Location = new Point(xPos, yPos);
            _nodeTypeCbx.Width = width;
            _nodeTypeCbx.DropDownStyle = ComboBoxStyle.DropDownList;

            _nodeTypeCbx.Items.Add("Add node:");
            _nodeTypeCbx.Items.Add("Block");
            _nodeTypeCbx.Items.Add("Dense");
            _nodeTypeCbx.SelectedIndex = 0;

            _nodeTypeCbx.SelectionChangeCommitted += NodeTypeCbx_SelectionChangeCommitted;

            form.Controls.Add(_nodeTypeCbx);
```

```
        }

        private static void NodeTypeCbx_SelectionChangeCommitted(object? sender, EventArgs e)
        {
            // Determines which node the user wants to add to the grid.
            if (_nodeTypeCbx.SelectedIndex == 1)
                SelectedNodeType = NodeType.Block;
            else if (_nodeTypeCbx.SelectedIndex == 2)
                SelectedNodeType = NodeType.Dense;
            else
                SelectedNodeType = null;
        }

        private static void CreateGenerationsCbx(int width, int xPos, int yPos, Form form)
        {
            _generationsCbx.Location = new Point(xPos, yPos);
            _generationsCbx.Width = width;
            _generationsCbx.DropDownStyle = ComboBoxStyle.DropDownList;

            _generationsCbx.Items.Add("Generations:");
            _generationsCbx.Items.Add("Generate block maze");
            _generationsCbx.Items.Add("Generate dense maze");
            _generationsCbx.Items.Add("Random grid layout");
            _generationsCbx.SelectedIndex = 0;

            _generationsCbx.SelectionChangeCommitted += GenerationsCbx_SelectionChangeCommitted;

            form.Controls.Add(_generationsCbx);
        }

        private static void GenerationsCbx_SelectionChangeCommitted(object? sender, EventArgs e)
        {
            // Determines which generation layout the user wants.
            if (!Simulation.Running)
            {
                if (_generationsCbx.SelectedIndex == 1)
                    Grid.MazeGeneration(NodeType.Block);
                else if (_generationsCbx.SelectedIndex == 2)
                    Grid.MazeGeneration(NodeType.Dense);
                else if (_generationsCbx.SelectedIndex == 3)
                    Grid.RandomGeneration();
            }

            // Resets the selected index back to the description.
            _generationsCbx.SelectedIndex = 0;
        }
    }
}
```

```
namespace PathfindingAlgorithmVisualiser
{
    using System.Timers;

    public static class Simulation
    {
        public static bool Running { get; private set; }
        public static bool Paused { get; private set; }
        public static bool Visualised { get; private set; }
        public static int Speed { get; private set; }

        private static bool _pathFound;

        private static readonly List<(Node Node, string Action)> _steps = new();
        private static readonly List<Node> _path = new();
        private static readonly Timer _timer = new();

        static Simulation()
        {
            _timer.Elapsed += Timer_Elapsed;
        }

        public static void Start()
        {
            Running = true;
            Paused = false;
```

```
            Grid.Dragging = false;
            Grid.Moving = null;

            // Runs an algorithm then visualises its processes.
            RunAlgorithms();

            if (Speed == 6)
                InstantVisualise();
            else
                _timer.Start();
        }

        public static void Stop()
        {
            _timer.Stop();

            // Resetting variables.
            _path.Clear();
            _steps.Clear();

            Visualised = false;
            Grid.ClearVisualisation();
            Running = false;
        }

        public static void Pause()
        {
            _timer.Stop();
            Paused = true;
        }

        public static void Resume()
        {
            _timer.Start();
            Paused = false;
        }

        public static void InstantRecalculation()
        {
            // Instantly shows new visuals and shortest path.
            Grid.ClearVisualisation();

            _path.Clear();
            _steps.Clear();

            RunAlgorithms();
            InstantVisualise();
        }

        public static void ChangeSpeed(int speed)
        {
            // Timer interval is changed depending on the selected speed.
            switch (speed)
            {
                case 1:
                    Speed = 1;
                    _timer.Interval = 1000;
                    break;
                case 2:
                    Speed = 2;
                    _timer.Interval = 100;
                    break;
                case 3:
                    Speed = 3;
                    _timer.Interval = 50;
                    break;
                case 4:
                    Speed = 4;
                    _timer.Interval = 10;
                    break;
                case 5:
                    Speed = 5;
                    _timer.Interval = 1;
                    break;
```

```
                case 6:
                    Speed = 6;

                    if (Running)
                    {
                        _timer.Stop();
                        InstantVisualise();
                    }
                    break;
            }
        }

        private static void AddPath(Node start, Node end)
        {
            Node node = end.Parent;

            // Backtracks and stores the shortest path.
            while (!ReferenceEquals(node, start))
            {
                if (node.Type != NodeType.Start && node.Type != NodeType.Target && node.Type !=
NodeType.Diversion)
                    _path.Add(node);

                node = node.Parent;
            }
        }

        private static void RunAlgorithms()
        {
            bool firstPathFound = false;
            bool? secondPathFound = null;

            // Calls the necessary pathfinding algorithms and stores their results.
            if (Grid.Start is not null && Grid.Target is not null)
            {
                if (Toolbar.SelectedAlgorithm == Algorithm.Dijkstra)
                {
                    if (Grid.Diversion is null)
                    {
                        firstPathFound = Dijkstra(Grid.Start, Grid.Target);
                        if (firstPathFound)
                            AddPath(Grid.Start, Grid.Target);
                    }
                    else
                    {
                        firstPathFound = Dijkstra(Grid.Start, Grid.Diversion);
                        if (firstPathFound)
                            AddPath(Grid.Start, Grid.Diversion);

                        secondPathFound = Dijkstra(Grid.Diversion, Grid.Target);
                        if (secondPathFound == true)
                            AddPath(Grid.Diversion, Grid.Target);
                    }
                }
                else if (Toolbar.SelectedAlgorithm == Algorithm.AStar)
                {
                    if (Grid.Diversion is null)
                    {
                        firstPathFound = AStar(Grid.Start, Grid.Target);
                        if (firstPathFound)
                            AddPath(Grid.Start, Grid.Target);
                    }
                    else
                    {
                        firstPathFound = AStar(Grid.Start, Grid.Diversion);
                        if (firstPathFound)
                            AddPath(Grid.Start, Grid.Diversion);

                        secondPathFound = AStar(Grid.Diversion, Grid.Target);
                        if (secondPathFound == true)
                            AddPath(Grid.Diversion, Grid.Target);
                    }
                }
                else if (Toolbar.SelectedAlgorithm == Algorithm.BFS)
```

```
                {
                    Grid.DisableDenseNodes();

                    if (Grid.Diversion is null)
                    {
                        firstPathFound = BFS(Grid.Start, Grid.Target);
                        if (firstPathFound)
                            AddPath(Grid.Start, Grid.Target);
                    }
                    else
                    {
                        firstPathFound = BFS(Grid.Start, Grid.Diversion);
                        if (firstPathFound)
                            AddPath(Grid.Start, Grid.Diversion);

                        secondPathFound = BFS(Grid.Diversion, Grid.Target);
                        if (secondPathFound == true)
                            AddPath(Grid.Diversion, Grid.Target);
                    }
                }
            }

            // The path should only be drawn if the whole path is found.
            if (firstPathFound && (secondPathFound is null || secondPathFound == true))
                _pathFound = true;
            else
                _pathFound = false;
        }

        private static int Heuristic(Node current, Node end)
        {
            return Math.Abs(current.Column - end.Column) + Math.Abs(current.Row - end.Row);
        }

        private static bool Dijkstra(Node start, Node end)
        {
            // Nodes to be evaluated.
            var open = new List<Node>();
            // Nodes evaluated.
            var closed = new List<Node>();

            start.GCost = 0;
            open.Add(start);

            // Node in 'open' with the lowest g-cost is evaluated next.
            while (open.Count > 0)
            {
                Node current = open[0];

                foreach (var node in open)
                {
                    if (node.GCost < current.GCost)
                        current = node;
                }

                // Removes it from 'open' and adds it to the 'closed' set.
                open.Remove(current);
                closed.Add(current);
                _steps.Add((current, "close"));

                // Path found if this node is the 'end node'.
                if (ReferenceEquals(current, end))
                    return true;

                // Loops through the neighbours of the node.
                foreach (var neighbour in current.Neighbours)
                {
                    // Skips the neighbour if it's not 'traversable' or has already been evaluated.
                    if (neighbour.Type == NodeType.Block || closed.Contains(neighbour))
                        continue;

                    // If a neighbour is not in the open set, or there is a shorter path to it, it's
properties are updated.
```

```
                    if (!open.Contains(neighbour) || current.GCost + neighbour.AdjacentCost <
neighbour.GCost)
                    {
                        neighbour.GCost = current.GCost + neighbour.AdjacentCost;
                        neighbour.Parent = current;

                        // If a neighbour is not in the open set, it's added.
                        if (!open.Contains(neighbour))
                        {
                            open.Add(neighbour);
                            _steps.Add((neighbour, "open"));
                        }
                    }
                }
            }

            return false;
        }

        private static bool AStar(Node start, Node end)
        {
            // Nodes to be evaluated.
            var open = new List<Node>();
            // Nodes evaluated.
            var closed = new List<Node>();

            start.GCost = 0;
            open.Add(start);

            // Node in 'open' with the lowest f-cost (g-cost + h-cost) is evaluated next.
            // If the f-cost is the same, the node with the lowest h-cost is prioritised.
            while (open.Count > 0)
            {
                Node current = open[0];

                foreach (var node in open)
                {
                    int currentFCost = current.GCost + Heuristic(current, end);
                    int nodeFCost = node.GCost + Heuristic(node, end);

                    if (nodeFCost < currentFCost)
                        current = node;
                    else if (nodeFCost == currentFCost && Heuristic(node, end) < Heuristic(current,
end))
                        current = node;
                }

                // Removes it from 'open' and adds it to the 'closed' set.
                open.Remove(current);
                closed.Add(current);
                _steps.Add((current, "close"));

                // Path found if this node is the 'end node'.
                if (ReferenceEquals(current, end))
                    return true;

                // Loops through the neighbours of the node.
                foreach (var neighbour in current.Neighbours)
                {
                    // Skips the neighbour if it's not 'traversable' or has already been evaluated.
                    if (neighbour.Type == NodeType.Block || closed.Contains(neighbour))
                        continue;

                    // If a neighbour is not in the open set, or there is a shorter path to it, it's
properties are updated.
                    if (!open.Contains(neighbour) || current.GCost + neighbour.AdjacentCost <
neighbour.GCost)
                    {
                        neighbour.GCost = current.GCost + neighbour.AdjacentCost;
                        neighbour.Parent = current;

                        // If a neighbour is not in the open set, it's added.
                        if (!open.Contains(neighbour))
                        {
```

```
                            open.Add(neighbour);
                            _steps.Add((neighbour, "open"));
                        }
                    }
                }
            }

            return false;
        }

        private static bool BFS(Node start, Node end)
        {
            // Queue of nodes to evaluate next.
            var toCheck = new Queue<Node>();
            // List of all visited nodes.
            var visited = new List<Node>();

            toCheck.Enqueue(start);
            visited.Add(start);

            while (toCheck.Count > 0)
            {
                // Next node in the queue to evaluate.
                Node current = toCheck.Dequeue();

                // If it is the end node, the path is found.
                if (ReferenceEquals(current, end))
                    return true;

                // Loops through the neighbours of the node.
                foreach (var neighbour in current.Neighbours)
                {
                    // If the neighbour is 'traversable' and hasn't been 'visited', it's added to the
queue.
                    if (neighbour.Type != NodeType.Block && !visited.Contains(neighbour))
                    {
                        toCheck.Enqueue(neighbour);
                        visited.Add(neighbour);
                        neighbour.Parent = current;
                        _steps.Add((neighbour, "close"));
                    }
                }
            }

            return false;
        }

        private static void DrawPath()
        {
            // Draws the path.
            foreach (var node in _path)
                node.Path();
        }

        private static void InstantVisualise()
        {
            // Instantly displays visualisation results.
            foreach (var step in _steps)
            {
                if (step.Node.Type != NodeType.Start && step.Node.Type != NodeType.Target &&
step.Node.Type != NodeType.Diversion)
                {
                    if (step.Action == "close")
                        step.Node.Close();
                    else
                        step.Node.Open();

                    if (Toolbar.SelectedAlgorithm != Algorithm.BFS && step.Node.Type ==
NodeType.Dense)
                        step.Node.Dim();
                }
            }

            if (_pathFound)
```

```
                DrawPath();

            Visualised = true;
        }

        private static void Timer_Elapsed(object? sender, ElapsedEventArgs e)
        {
            if (_steps.Count == 0)
            {
                // Draws the path and stops the timer if all the steps have been shown.
                if (_pathFound)
                    DrawPath();

                _timer.Stop();
                Visualised = true;
            }
            else
            {
                // 'Visualises' the next algorithmic step.
                Node node = _steps[0].Node;

                if (node.Type != NodeType.Start && node.Type != NodeType.Target && node.Type !=
NodeType.Diversion)
                {
                    if (_steps[0].Action == "close")
                        node.Close();
                    else
                        node.Open();

                    if (Toolbar.SelectedAlgorithm != Algorithm.BFS && node.Type == NodeType.Dense)
                        node.Dim();
                }

                _steps.RemoveAt(0);
            }
        }
    }
}
```

```
namespace PathfindingAlgorithmVisualiser
{
    public enum NodeType
    {
        Start,
        Target,
        Diversion,
        Block,
        Dense,
        Empty
    }

    public enum Algorithm
    {
        Dijkstra,
        AStar,
        BFS
    }
}
```