

Maze generation & pathfinding

name



Candidate number:

centre number:

# Contents

[Contents 1](#_Toc153802427)

[Analysis 3](#_Toc153802428)

[Problem Description 3](#_Toc153802429)

[Research 3](#_Toc153802430)

[Pathfinding 3](#_Toc153802431)

[Metric to Compare Pathfinding Algorithms 3](#_Toc153802432)

[Maze Generation 4](#_Toc153802433)

[Metric to Compare Maze Generation Algorithms 4](#_Toc153802434)

[Current Systems 6](#_Toc153802435)

[Explanation 6](#_Toc153802436)

[Current Systems 6](#_Toc153802437)

[Summary of findings 6](#_Toc153802438)

[Potential Solutions 7](#_Toc153802439)

[Program 7](#_Toc153802440)

[Pathfinding 8](#_Toc153802441)

[Maze Generation 10](#_Toc153802442)

[User Identification 11](#_Toc153802443)

[Questionnaire 11](#_Toc153802444)

[User Requirements 12](#_Toc153802445)

[Justification for chosen solutions 14](#_Toc153802446)

[Pathfinding 14](#_Toc153802447)

[Maze Generation 15](#_Toc153802448)

[Project Objectives 17](#_Toc153802449)

[Design 20](#_Toc153802450)

[Description 20](#_Toc153802451)

[Interface Design 20](#_Toc153802452)

[Explanation 20](#_Toc153802453)

[Rough Design 21](#_Toc153802454)

[Implementing Final Design 22](#_Toc153802455)

[Controls 22](#_Toc153802456)

[IPSO chart 27](#_Toc153802457)

[Top-Down Diagram 29](#_Toc153802458)

[Class Diagram 30](#_Toc153802459)

[Algorithms 31](#_Toc153802460)

[Maze generation algorithms 31](#_Toc153802461)

[Pathfinding algorithms 35](#_Toc153802462)

[User defined data structures 40](#_Toc153802463)

[Technical Solution 44](#_Toc153802464)

[Code listing 44](#_Toc153802465)

[Main 44](#_Toc153802466)

[Download 48](#_Toc153802467)

[DisplayGrid 50](#_Toc153802468)

[Grid 58](#_Toc153802469)

[MazeGeneration 60](#_Toc153802470)

[PathFinding 74](#_Toc153802471)

[Structures 85](#_Toc153802472)

[Methods 87](#_Toc153802473)

[Enumerals 89](#_Toc153802474)

[Priority Queue 90](#_Toc153802475)

[Techniques used 92](#_Toc153802476)

[Table A 92](#_Toc153802477)

[Table B 92](#_Toc153802478)

[Interface Screenshots 93](#_Toc153802479)

[**Testing** 95](#_Toc153802480)

[Tests 95](#_Toc153802481)

[Fixes 111](#_Toc153802482)

[Evaluation 115](#_Toc153802483)

[Objectives 115](#_Toc153802484)

[Table 115](#_Toc153802485)

[Conclusion 118](#_Toc153802486)

[User Feedback 119](#_Toc153802487)

[Questionnaire 119](#_Toc153802488)

[Results 119](#_Toc153802489)

[Improvements 121](#_Toc153802490)

[1. Quick tutorial 121](#_Toc153802491)

[2. Indication when there is no path 121](#_Toc153802492)

[3. Addition of even more algorithms 122](#_Toc153802493)

# Analysis

## Problem Description

Pathfinding and maze generation are major topics in the field of computer science, having major applications in travelling and logistics optimisation to game development. There exists a variety of different algorithms with a range of different characteristics and uses which can get confusing and overwhelming when having to select the best algorithms for the task.

To solve this problem, I am going to create a program that allows a user to compare different pathfinding and maze generation algorithms. I will need to thoroughly research different metrics and a range of algorithms that will help me complete this task.

## Research

### Pathfinding

In short, pathfinding is the process of finding a path from one point to another. It usually involves analysing and navigation through a set of nodes and connections.

The history of Pathfinding can be traced back to the mid-20th century when many researchers started to try and develop algorithms to solve problems such as shortest paths in transportation networks. Edsger W. Dijkstra, one of the most influential figures in pathfinding, developed Dijkstra's algorithm which found the shortest path in a weighted graph. This later led to developments in pathfinding such as the A\* algorithm, in 1968, which made use of heuristics. And now with the computational advancements in the 21st century much more sophisticated pathfinding algorithms have been developed.

### Metric to Compare Pathfinding Algorithms

Pathfinding algorithms can differ in many different areas:

* Efficiency:

The efficiency of a pathfinding algorithm refers to how quickly it can find a solution. Some algorithms that quickly search in local proximity might be much more efficient in smaller graphs/mazes. Compared to others that can intelligently prioritize nodes more likely to lead to the goal but take longer to compute.

* Use of Heuristics:

Heuristics play a crucial role in guiding pathfinding algorithms. A heuristic is an estimate that provides additional information about the problem. In the case of pathfinding, a heuristic helps to estimate the cost or distance from a given node to the goal node. A good heuristic can guide an algorithm towards the goal node more efficiently, while an overly optimistic heuristic might lead to suboptimal paths.

* Ability to Find the Shortest Path:

Finding the shortest path is a desirable feature in many pathfinding scenarios. Some algorithms guarantee finding the shortest path when applied to graphs by ensuring that the shortest path to each node is found before moving on to the next level. Whereas some do not guarantee finding the shortest path, by prioritizing deeper paths. These may be used in a scenario where finding any path between two nodes is sufficient.

In summary, there are many different characteristics which distinguish different pathing algorithms and these are just an example of a few of them. It will be vital to pick algorithms to use in my program which represent and display a balance of all these features to make it more useful to the end user by giving them an insight into all the available options.

### Maze Generation

Mazes have fascinated people throughout history, taking on various shapes and sizes. The earliest versions of these structures were called labyrinths, which differed from our traditional notion of mazes. Labyrinths were designed with a single winding path, not intended to confuse people, but rather to serve as spiritual journeys, guiding visitors along a meaningful path.

Evidence of these labyrinth structures has been dated to ancient Egypt in the 5th century. As well as being found in Greek mythology with one of the most famous labyrinths being the Labyrinth of Knossos.



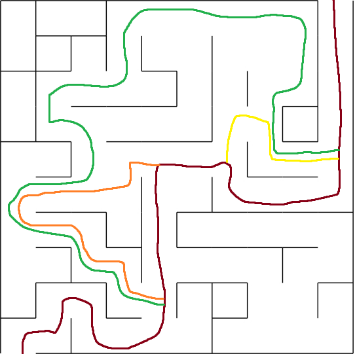
Over time, mazes have evolved more into puzzles made to entertain and confuse people today. An example of this is Hedge mazes which also hold a significant place in the history of mazes. They were often found in grand gardens and estates, serving as both decorative features and sources of entertainment. Some famous examples of these hedge mazes include the Hampton Court Maze in England and the Villa Pisani Maze in Italy.



In addition to physical mazes, mazes have found their way into other forms of entertainment and learning. For example, mazes are now commonly featured on the back of cereal boxes, children's activity books and puzzle magazines. These mazes often have many different levels of difficulty, ranging from simple mazes to more complex paths that test your problem-solving skills. They have been a fun and engaging way for children to develop spatial awareness, logical thinking, and hand-eye coordination skills.

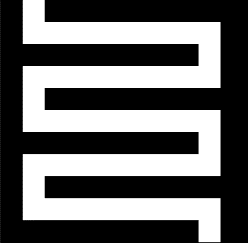
### Metric to Compare Maze Generation Algorithms

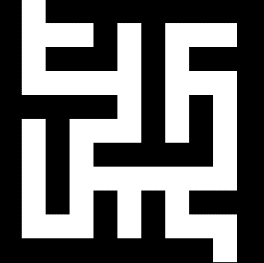
Maze algorithms are a tool to generate these mazes, with each different algorithm having different properties and characteristics. Mazes fall into two main categories, perfect and imperfect. A maze is said to be perfect if it has only one solution and does not contain any loops or inaccessible parts. In other words, it is a maze with a clear and unambiguous path from start to finish.



The absence of loops in a perfect maze ensures that there are no alternative routes or multiple paths leading to the same destination, making the solution unique. As you can see in the maze on the left there are several different paths all leading to the same location which makes the maze imperfect.

As well as this categorization, each maze has its own unique characteristics. Examples of how they can vary are the percentage of dead ends, branching factor, tendency for long corridors, and the presence of "river-like" structures. These can all affect the shortest path length.

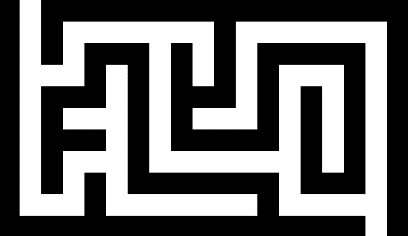
Finding suitable metrics to compare mazes is a challenging task.

For example, the length of the shortest path alone does not always accurately represent the difficulty level. This can be observed by comparing two examples: one with a single clear path (image on left) and another with multiple dead ends (image on right).

In the case of the maze with a single path, while the length of the path may be longer, the solution is relatively straightforward. The main challenge lies in the time required to traverse the entire length of the maze, but there is no ambiguity or need for backtracking.

On the other hand, the maze with many dead ends presents a different type of challenge. Although the length of the shortest path is shorter, the presence of dead ends introduces the possibility of making incorrect turns and needing to backtrack. This adds a layer of complexity as one must carefully navigate through the maze, making correct choices and avoiding dead ends to reach the solution.

A measurement of how often a maze branches off is branching factor. Branching factor is a characteristic which describes the average number of paths branching out from each junction. A maze with a high branching factor leads to less predictable mazes and can lead to the mazes looking “spikey” such as in the image above.



Whereas a low branching factor leads to makes it look “river-like” with long uninterrupted corridors. These mazes can offer advantages in ease of navigation but can lead to longer paths leading away from the desired path towards dead ends. This can be seen in the image on the right.

In summary, maze generation algorithms can generate a range of different mazes that can have a variety of different characteristics. In my program, I will need to select mazes that can show a range of their different characteristics and how they can affect the aesthetic of the maze. I will also need to choose sufficient metrics to compare these mazes by.

## Current Systems

### Explanation

To get a better understanding of existing maze generation and pathfinding systems I have looked into several current solutions.

### Current Systems

#### [Bachelors’ thesis](https://www.divaportal.org/smash/get/diva2:1237178/FULLTEXT02)

When looking for ways to compare mazes I came across a thesis titled "Evaluation of the Complexity of Procedurally Generated Maze Algorithms" by Albin Karlsson, a Bachelor of Science at Blekinge Institute of Technology in Sweden. After a thorough analysis of this paper, I've gained valuable insights into the metrics that I can use to evaluate maze algorithms. Albin Karlsson's approach involved using branch distribution, path length, corridor distribution and completion time, as well as providing a general description of the maze's appearance for evaluation.

#### [Maze Generator](https://www.mazegenerator.net/)

This is a simple website that allows the creation and downloading of mazes. It uniquely allows the selection of triangle, square and hexagonal cells which is something to consider. The creator has chosen to allow a maze size of between 2 and 200 and use a tick box to show or hide the solution. Where the website is lacking is the variety of maze algorithms as they only have the choice of one which isn’t even named anywhere.

As far as the design of the website the walls and space are just black and white with a red line that shows the solution if toggled. They have chosen to make the width of the wall cells much smaller than that of a space which is something I need to consider for my program.

#### [Pathfinding Visualizer](https://clementmihailescu.github.io/Pathfinding-Visualizer/)

This website is more focused on pathfinding and only has the option of recursive division for maze generation. Where it shines is the customisation and placement of walls and other weighted cells or terrain as it calls it, for example, it costs 5 to travel through grass and 50 through water. It doesn’t allow for the resizing of the grid. It visualises the pathfinding step by step showing different colours for open and closed cells as it goes.

Also, rather than a maze it has another option called basic random maze which just “scatters” walls by randomly filling random cells to act as obstacles which I like the idea of.

### Summary of findings

My analysis of current systems has brought up many considerations I need to take into mind such as:

* Metrics for Evaluation: The importance of identifying the necessary metrics for assessing the performance of both pathfinding and maze generation algorithms.
* Maze Size Limit: Determining the maximum maze size, is a key factor in ensuring optimal functionality.
* Algorithm Selection: Choosing the right number of algorithms to include, balancing variety with practicality.
* "Scatter" Feature: The decision to incorporate the random fill "scatter" feature, can enhance user experience and exploration.
* User-Drawn Walls: The consideration of allowing users to draw custom walls, is a feature that will enhance customization and allow for better understanding.
* Terrain and Weighted Walls: Exploring the possibility of introducing varying weights to walls, often referred to as terrain and its potential impact.
* Visualization: Deciding whether to visually demonstrate maze generation and pathfinding processes or simply display the final results, a choice that can significantly affect user engagement.

## Potential Solutions

### Program

There are a range of different applications I could use to achieve my goal:

|  |  |  |
| --- | --- | --- |
| Solution | Pros | Cons |
| Console application | Simplicity: Console applications are straightforward to create and are great for simple interactions.  Lightweight: They are efficient in terms of system resources. | Limited User Interaction: They lack a graphical user interface which may limit user engagement and visualization for maze generation and pathfinding.  Difficulty of inputs: Would need to offer menus and would be overcomplicated to change different options |
| Windows forms application | GUI: Windows Forms applications have a visually appealing and interactive interface which would be ideal for visualizing mazes and pathfinding.  Versatility: Large range of options from text boxes, buttons, sliders, etc. | Windows-Only: Designed for Windows environments, limiting cross-platform compatibility.  Learning Curve: Developing a Windows Forms application may be harder than a console application |
| Mobile application | Accessible: Mobile apps can be used on many different devices, making the program more accessible.  Touch Interaction: Users can interact using touch features enhancing the user experience. | Learning curve: This would require learning platform-specific languages and different developments for iOS and Android.  Limited Screen: Mobile devices have smaller screens, which means I would need to use smaller mazes. |
| Browser application | Accessible: Browser-based applications are accessible from any device with an internet connection.  Easy Updates: Browser-based applications can be updated and maintained on the server side, ensuring all users have access to the latest version without requiring manual updates. | Performance: Might not perform as well as an actual application, especially when handling larger grid sizes and visualizations.  Limited Offline Access: Browser applications rely on an internet connection.  Learning Curve: Developing a browser-based application would involve a significant learning curve. I would need to learn skills in web development as well as server management. |

### Pathfinding

To ensure my solution contains a wide variety of pathfinding algorithms for efficient comparisons of their characteristics and performance, I must consider a range of pathfinding algorithms.

I have picked a range of 12 algorithms to consider and will explore how they work, their use cases and their efficiency.

Here is a short description and use cases of each algorithm:

|  |  |  |
| --- | --- | --- |
| Algorithm | Description | Uses |
| Breadth-First Search (BFS) | Explores all neighbours before deeper levels using a queue. | Shortest path in unweighted graphs, graph traversal. |
| Depth-First Search (DFS) | Explores as deeply as possible along one branch before backtracking using a stack. | Graph traversal, maze solving. |
| Dijkstra's Algorithm | Finds the shortest path in non-negative weighted graphs using a primary queue. | Network routing, GPS navigation. |
| A\* Algorithm | Prioritizes paths with lower cost using a heuristic using a priority queue. | Pathfinding in games, robotics. |
| Bellman-Ford Algorithm | Handles negative edge weights, and finds shortest path. | Network routing, graph analysis. |
| Bidirectional Search | Simultaneously explores from start and goal nodes to meet in the middle using. | Shortest path in large graphs. |
| Floyd-Warshall Algorithm | Finds all shortest paths between all pairs of nodes. | All-pairs shortest path, graph analysis. |
| Greedy Best-First Search | Selects paths that seem closest to the goal based on a heuristic. | Heuristic pathfinding, puzzles. |
| Jump Point Search | Optimized for grid-based paths, reduces unnecessary node exploration. | Grid-based pathfinding. |
| Depth-Limited Search | Limits exploration depth to avoid infinite loops. | Game AI, puzzle-solving. |
| Iterative Deepening DFS | Repeatedly increases depth limit until a solution is found. | Tree and graph traversal. |
| Bidirectional Dijkstra's | Combines Dijkstra's algorithm from both start and goal nodes. | Shortest path in large graphs. |

Here is the time complexity, space complexity and whether or not each of the algorithms finds the optimal path:

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Time Complexity (Big O) | Space Complexity (Big O) | Optimality |
| Breadth-First Search (BFS) | O(V + E) | O(V) | Yes |
| Depth-First Search (DFS) | O(V + E) | O(V) | No |
| Dijkstra's Algorithm | O((V + E) \* log(V)) | O(V) | Yes |
| A\* Algorithm | O((V + E) \* log(V)) | O(V) | Yes |
| Bellman-Ford Algorithm | O(V \* E) | O(V) | Yes |
| Bidirectional Search | O(V + E) | O(V) | Yes |
| Floyd-Warshall Algorithm | O(V3) | O(V2) | Yes |
| Greedy Best-First Search | O(V + E) | O(V) | No |
| Jump Point Search | O(E) | O(V) | No |
| Depth-Limited Search | O(bl) | O(bl) | No |
| Iterative Deepening DFS | O(bd) | O(d) | Yes |
| Bidirectional Dijkstra's | O((V + E) \* log(V)) | O(V) | Yes |

Time and space complexity are represented in terms of the number of vertices (V), edges (E), branching factor (b), and depth (d), as appropriate for each algorithm.

### Maze Generation

There is an abundance of different maze generation algorithms. Each offers distinct characteristics, such as complexity, connectivity, dead ends, corridor length, and branching. By exploring and analysing, I can find the best options to include in my solution to allow a user to understand the different metrics of these algorithms and to test their effects on pathfinding algorithms.

To achieve this, I have picked a range of 12 algorithms to consider and will explore how they work and their efficiency.

Here is a short description of each algorithm:

|  |  |
| --- | --- |
| Algorithm | Description |
| Kruskal's Algorithm | Connects random edges, ensuring no loops, creating a maze with minimal spanning tree properties. |
| Prim's Algorithm | Grows a maze by starting with a single cell and repeatedly adding a random unvisited cell. |
| Recursive Backtracker | A recursive algorithm that creates a maze by exploring as deeply as possible before backtracking using a stack. |
| Aldous-Broder Algorithm | Randomly walks through the maze, creating passages as it goes, repeating until all cells are visited. |
| Growing Tree Algorithm | Maintains a list of unvisited cells and expands the maze from a randomly chosen cell. |
| Hunt-and-Kill Algorithm | Combines a random walk with a systematic search, creating a maze with varying passage lengths. |
| Wilson's Algorithm | Randomly selects a starting point and uses loops-erased random walks to generate the maze. |
| Eller's Algorithm | Creates mazes row by row using set union and partitioning techniques. |
| Recursive Division | Divides the maze recursively into smaller sections, creating a binary tree-like pattern. |
| Sidewinder | Generates a maze row by row, connecting adjacent cells and occasionally forming vertical passages. |
| Binary Tree | Connects cells by creating diagonal passages based on a binary tree structure. |

Here is the time and space complexity of each algorithm:

|  |  |  |
| --- | --- | --- |
| Algorithm | Time Complexity (Big O) | Space Complexity (Big O) |
| Kruskal's Algorithm | O(E log E) | O(V + E) |
| Prim's Algorithm | O(E log E) | O(V + E) |
| Recursive Backtracker | O(V) | O(V) |
| Aldous-Broder Algorithm | O(V2) | O(1) |
| Growing Tree Algorithm | O(E log E) | O(V) |
| Hunt-and-Kill Algorithm | O(V2) | O(1) |
| Wilson's Algorithm | O(V2) | O(V) |
| Eller's Algorithm | O(V2) | O(V) |
| Recursive Division | O(V2) | O(V2) |
| Sidewinder | O(V) | O(1) |
| Binary Tree | O(V) | O(1) |

Time and space complexity are represented in terms of the number of vertices (V) and edges (E) as appropriate for each algorithm.

## User Identification

### Questionnaire

After exploring potential solutions, I decided that surveying my potential users would give me great insight and help me create the best program possible.

I will be surveying 10 different people with a range of interest in the project:

* Teachers who want to use it as an educational tool and to help students visualise different maze and pathfinding algorithms.
* Maze enthusiasts and hobbyists who want to compare different mazes and how they can vary.
* A-level computer science students who want an engaging and fun tool that also supports their learning.

|  |
| --- |
| 1. How many different pathfinding algorithms would you prefer to have available in a pathfinding tool?  * 1-3 * 4-6 * 6+  1. How many different maze algorithms would you prefer to have available in a maze generation tool?  * 1-3 * 4-6 * 6+  1. How much control would you like to have over customizing the grid?  * I'd like the option to choose the position of start and end nodes, as well as the ability to draw walls. * I'd like to be able to place the start/end nodes and have the option for custom weighted walls or "terrain." * No, neither would be necessary for me.  1. What maximum-size grid do you think would best suit a pathfinding and maze-generating program?  * 120x120 * 300x300 * 600x600 * 1200x1200  1. Should the pathfinding process and maze generation be visualized as the algorithms work?  * Yes, I'd like to see the maze and pathfinding in action. * No, I prefer a final result without visualizations.  1. Which application do you prefer for a pathfinding and maze generation program?  * Windows application * A website * An app on your phone * A console application  1. Would you like the option to download the generated mazes, save your customized mazes, or export data related to maze and path metrics?  * Yes, I'd like to download or save mazes. * No, it's not necessary for me.  1. Should the pathfinding process and maze generation be visualized as the algorithms work?  * Yes, I'd like to see the maze and pathfinding in action. * No, I prefer a final result without visualizations.  1. Please select the metrics you would like to view or track during maze exploration and pathfinding (You can select multiple options):  * Length of the shortest path * Number of cells explored during pathfinding * Time taken for pathfinding * Time taken for maze generation * Dead end percentage * Branching Percentage * Average corridor length * Other (please specify): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

### User Requirements

#### How many different pathfinding algorithms would you prefer to have available in a pathfinding tool?

I'm going to be including 4 different pathfinding methods because they give users plenty of choices without being overwhelming or adding an unreasonable amount of time to development to make and test more algorithms. This is supported by the users as the majority chose 1-3 or 4-6 whilst only 1 chose 6+.

#### How many different maze algorithms would you prefer to have available in a maze generation tool?

I have chosen to include 5 maze generation algorithms as the results for the choices are very similar to the pathfinding question. Again, 5 would be a great balance between providing a range of options whilst avoiding excessive complexity.

How much control would you like to have over customizing the grid?

I have chosen to give the option to place starting and ending nodes and create custom walls within the maze. While this feature will require additional development time and add some complexity to the tool, it's highly valuable for certain users, particularly teachers who could especially benefit from demonstrating various pathfinding algorithms on customized grids to their students. As one teacher mentioned in the additional comments, "The ability to create custom walls and showcase a range of pathfinding algorithms would be a fantastic educational asset."

However, I have decided not to implement custom-weighted terrain in the tool. While it could potentially offer some additional features the survey suggests it wouldn't provide significant benefits to the users, making it an unnecessary addition that might increase development without adding value.

#### Which application would you prefer for a pathfinding and maze generation program?

A Windows Forms application is much more suitable for my project than a Console application, Mobile application, or Browser application for a variety of reasons.

Firstly, a Windows Forms application offers a much more user-friendly GUI, making it easier for users to interact with the maze and pathfinding tool compared to a text-based Console application or a more complex Mobile or Browser application.

Secondly, Windows Forms applications are generally better for desktop use, and since maze generation and pathfinding visualization might require a larger screen and better performance, especially for larger grid sizes, it aligns better with the capabilities of a Windows Forms application.

Third, developing a Mobile application would require learning a different programming language and technologies specific to mobile platforms, which would greatly increase the development time. Similarly, a Browser application might require additional web development skills.

Overall, Windows Forms will provide a good balance between usability, performance, and development efficiency for this project, making it the perfect choice.

#### What maximum-size grid do you think would best suit a pathfinding and maze-generating program?

I have chosen to make the maximum grid size 1200 as the results from the survey were inconclusive so I decided to make it the maximum size to account for all users. However, since 1200 will exceed the size of the display box, this will mean the image will need to be zoomed out, which will result in a blurry image and add complexity.

#### Would you like the option to download the generated mazes or save your customized mazes for future use?

I am adding the ability to download the maze as a .png file as most of my users said that it would be a useful feature to add.

#### Should the pathfinding process and maze generation be visualized as the algorithms work?

I will be implementing real-time visualization of the maze generation and pathfinding processes. Visualizing these algorithms as they work will help provide users with a better understanding of how different algorithms operate, making the tool more educational and engaging. In the survey, 8/10 people agreed showing there’s a strong preference towards visualisation from my users.

#### Please select the metrics you would like to view or track during maze exploration and pathfinding:

The survey results overwhelmingly showed that users expressed a strong desire for all these metrics, with every metric having over 50%. These metrics while increasing the time of development and increasing complexity will provide comprehensive insights into the efficiency and characteristics of the algorithms, making them invaluable for users seeking to understand and compare the various pathfinding and maze generation methods. Therefore, ensuring a well-rounded and informative user experience for the tool.

## Justification for chosen solutions

### Pathfinding

While it isn’t possible to implement every pathfinding algorithm due to practical limitations, supported by my questionnaire, I have chosen to focus on 4 algorithms:

* Breadth-first search (BFS) is an excellent choice as it guarantees finding the shortest path in an unweighted graph. By exploring neighbouring nodes level by level, BFS shows the efficiency and completeness of this simple approach. It will serve as a valuable baseline against which I can compare the performance of the other algorithms.
* Depth-first search (DFS) complements BFS by providing a different perspective. DFS explores as deeply as possible along each branch before backtracking. Although it does not guarantee finding the shortest path, DFS enables me to analyse different paths not taken by other algorithms. By adding DFS, I can evaluate the trade-off between completeness and efficiency in certain situations.
* Greedy Depth-First Search (Greedy DFS) will introduce a heuristic element. It prioritizes nodes based on their heuristic values, resulting in more exploration towards the goal. However, it is important to note that Greedy DFS may not always yield the shortest solution. And, by including this algorithm, I can investigate the impact of heuristics on efficiency.
* A\* algorithm is another key addition to my investigation. It combines the advantages of Dijkstra's algorithm and greedy depth-first search by considering both the actual cost from the start node and an estimated cost to the goal node. A\* utilizes a heuristic such as Manhattan distance to prioritize nodes, resulting in efficient and optimized pathfinding. By implementing A\*, I can explore the influence of heuristic estimation on the efficiency and accuracy of finding the shortest path.

### Maze Generation

After carefully considering I have focused on selecting 4 algorithms that would provide a diverse range of maze characteristics, such as complexity, connectivity, dead ends, corridors and branching. By incorporating multiple maze generation algorithms, I can allow users to compare unique traits and observe their impact on pathfinding algorithms:

* Recursive Backtracking algorithm, utilizes depth-first search to carve pathways and backtrack when encountering dead ends. This algorithm often results in mazes with winding corridors and a high percentage of dead ends. It provides an opportunity to examine the impact of dead ends on pathfinding algorithms and assess their efficiency in navigating through intricate and winding paths.
* Kruskal's algorithm, treats the maze as a set of disjointed cells and randomly connects adjacent cells until all cells are connected. This approach yields mazes with a relatively high branching factor, a uniform distribution of paths, and fewer dead ends. By including Kruskal's algorithm, I can explore how this wildly different maze will affect the behaviour of pathfinding algorithms.
* Prim's algorithm starts with a single cell and progressively grows the maze by adding the nearest unvisited cell as a neighbour. This process results in mazes with a tendency for long corridors and fewer dead ends. Including Prim's algorithm allows me to investigate how maze structures with long corridors affect the performance of pathfinding algorithms.
* The Binary Tree algorithm assigns random directions (north or east) to each cell and creates passages accordingly. It has a predictable and flawed pattern, favouring one direction over the other. By including the Binary Tree algorithm, I can explore the challenges posed by this unique pattern and observe how pathfinding algorithms react to such maze structures.

By incorporating these chosen algorithms, I can provide insights into their characteristics and observe how they influence pathfinding algorithms.

## Project Objectives

1. Grid
   1. Grid Representation
      1. Create a custom grid class that can be used to store cell data.
      2. Allow the grid to be accessed using an x and y coordinate or a point
      3. Create a method that allows the user to fill the grid
      4. Create a method that allows the user to set the grid size
   2. Cell Types Enum
      1. Define an enum to represent cell types
         1. Empty
         2. Wall
         3. Open
         4. Closed
         5. Path
         6. Start
         7. Finish
   3. Clear Pathfinding Cells
      1. Provide a user interface button to remove pathfinding-related cells (Open, Closed, Path) from the grid.
   4. Resizable Grid
      1. Allow users to resize the grid using numeric up-down controls and a slider.
      2. Set a maximum grid size of 1200 and a minimum size of 5.
      3. Make the slider use non-linear, scaling steps for grid size selection.
2. Display Grid
   1. Visual Representation
      1. Use unique colours to visually represent different cell types.
         1. Empty (white)
         2. Wall (black)
         3. Open (light blue)
         4. Closed (blue)
         5. Path (dark blue)
         6. Start (green)
         7. Finish (red)
      2. Use a function to return the correct colour from the cell type.
   2. Bitmap Usage
      1. Utilize a bitmap to store the grid.
      2. Draw cells on the bitmap using the Graphics class.
   3. User-Defined Control
      1. Create a user-defined control to handle displaying the grid.
      2. Embed a Picture Box in the user-defined control to display the bitmap.
   4. Adaptive Grid Display
      1. Scale cell sizes on the bitmap to ensure high-quality display when the grid size is smaller than the display.
      2. Use the Picture Box zoom function to compress the bitmap when the grid is larger than the display.
3. Visualization
   1. User-Set Delay
      1. Enable users to set delay using numeric up-down controls and a slider.
      2. Restrict the maximum delay to 200 and the minimum to 0.
   2. Drawing Instructions
      1. Define a custom data structure to store drawing instructions
         1. x coordinate
         2. y locations
         3. colour
         4. size
   3. Timer and Queue
      1. Use a queue to store drawing instructions.
      2. Implement a timer to execute drawing instructions, ensuring responsiveness to user inputs, while simultaneously visualise the different algorithms.
   4. Non-Linear Delay
      1. Customize the delay interval to allow for a wider range of speeds.
      2. Implement delay options, such as drawing all instructions at once (delay 0), drawing multiple instructions at each interval step (delay 1, 2, 3, 4, 5) or taking away 5 from the delay to make it stand from 1 (greater than 5).
4. Pathfinding
   1. Pathfinding Algorithms
      1. Implement a range of pathfinding algorithms
         1. Depth-First Search
         2. Breadth-First Search
         3. Greedy DFS
         4. A\* Search.
   2. Algorithm Selection
      1. Enable users to choose pathfinding algorithms through a combo box.
      2. Use an enum to manage the different algorithm options.
   3. Algorithm Execution
      1. Provide a button to initiate the selected pathfinding algorithm.
   4. Metrics Presentation
      1. Display algorithm execution metrics, including time taken, shortest path, and cells explored.
      2. Use labels to convey all of this information.
5. Maze Generation
   1. Maze Generation Algorithms
      1. Implement a variety of maze generation algorithms
         1. Kruskal's algorithm
         2. Prim's algorithm
         3. Recursive Backtracker
         4. Recursive Division
   2. Algorithm Selection
      1. Enable users to choose maze generation algorithms via a combo box.
      2. Use an enum to manage the different algorithm options.
   3. Algorithm Execution
      1. Provide a button to initiate the selected maze generation algorithm.
   4. Maze Metrics
      1. Display maze generation metrics, including time taken, percentage of dead ends, percentage of branches, and average corridor length.
      2. Use labels to present this data.
6. User Customization
   1. Wall Customization
      1. Enable users to add and remove walls interactively using the MouseMove event.
      2. Add walls when the left mouse button is held, and remove walls when the right mouse button is held.
   2. Start and Finish Placement
      1. Allow users to set the location of the start and finish points using "C" and "V" keys, respectively.
   3. User Guidance
      1. Display instructions using labels to guide users in customizing the grid.
      2. Hide controls related to maze on selection.
7. Download
   1. Separate Window
      1. Implement a separate Windows Forms window for downloading grid images.
   2. File Path Selection
      1. Provide a button to change the file path using the FileBrowserDialog.
   3. Bitmap Download
      1. Allow users to download the grid as a bitmap using a dedicated button and the .save method.

# Design

## Description

The project's design aims to create a versatile and interactive platform for comparing various pathfinding and maze generation algorithms. The application is constructed using the VB.Net programming language and leverages the Windows Forms framework to create a user-friendly graphical user interface. An object-oriented approach organises the codebase for efficiency and maintainability. To manage the grid's visual representation, a custom user-defined control is introduced, featuring a Picture Box that serves as the canvas for grid display. The user interface incorporates interactive controls, including buttons, numeric up-and-down boxes, sliders, and labels, enabling user customization of grid parameters, algorithm selection, and seamless interaction with the application.

The application offers a comprehensive collection of pathfinding algorithms such as Depth-First Search, Breadth-First Search, Greedy DFS, and A\* Search, each designed to find optimal paths through the grid and offer diverse problem-solving strategies. Detailed metrics, which include time taken, shortest path, and cells explored, are calculated and displayed using labels to enrich the user experience.

Users can further expand their testing of pathfinding algorithms or compare different maze generation algorithms, ranging from Kruskal's Algorithm, Prim's Algorithm, Recursive Backtracker, and Recursive Division. In depth metrics that measure factors such as time taken, percentage of dead ends, percentage of branches, and average corridor length facilitate this.

On top of this, the application allows users to interactively add or remove walls on the grid using mouse events and key commands (e.g., "C" for start point and "V" for the finish point) while labels provide instructions and facilitate user interactions. Additionally, users can download grid images via a separate window, where they can choose the file path and save grid images as bitmaps.

## Interface Design

### Explanation

In my user interface, the left side will feature the grid user-defined control with an embedded picture box for grid display. And on the right side, a range of interactive controls, ranging from buttons to sliders, will allow user interaction.

### Rough Design

|  |  |
| --- | --- |
| Main Form | Download |
|  |  |

|  |  |  |
| --- | --- | --- |
| 1 |  | This provides an illustrative preview of a visually represented grid, showcasing distinct grid types through varying colours: Empty (white), Wall (black), Open (light blue), Closed (blue), Path (dark blue), Start (green), and Finish (red). It also showcases what users could create within the customization mode. |
| 2 |  | In this section, users can choose from various pathfinding algorithms using a combobox. The interface provides two buttons, one for initiating the pathfinding process and the other for clearing pathfinding-related cells (Open, Closed, Path) from the grid. Three informative labels are displayed, presenting metrics for time taken, shortest path, and cells explored. |
| 3 |  | Grid customisation had 3 different interfaces dependant on selection. This view is presented when a maze generation algorithm is chosen. The user can use the combobox to select the desired algorithm or opt for user customization. A "Generate" button initiates the maze creation process, and four labels are displayed to convey crucial metrics: time taken, percentage of dead ends, percentage of branches, and average corridor length. |
|  | Upon selecting User Customization, the interface presents a different set of options. Instead of sliders or maze metrics, a key is displayed. This key provides clear instructions to the user, guiding them on how to set the start and finish points, as well as add or remove walls from the grid. |
| 4 |  | Within this section, users have the ability to fine-tune program settings, including grid size and delay. These adjustments are made using a combination of a slider and a numeric up-and-down box. Additionally, a "download" button is available, allowing users to initiate a process that opens a separate form application for downloading grid images. |
| 5 |  | This section, shows the download form. It includes two crucial buttons: one for selecting the desired file path and the other to initiate the download process to the chosen file path. |

### Implementing Final Design

Here, I've translated my initial design into the graphical user interface (GUI) of a Windows Form application.

|  |
| --- |
| This section will have 2 different interfaces dependant on selection:   * Maze generation Algorithms * Grid Customisation     Here is the download  form: |

### Controls

#### Forms

|  |  |
| --- | --- |
| Name: Main.vb  Type: Form  Size: 780, 650  FormBorderStyle: FixedToolWindow  Text: “Pathfinding & Maze Generation”  Font: Microsoft Sans Serif, Black (#000000), 8.25pt  Back Colour: ControlDarkDark (#696969) | Name: Download.vb  Type: Form  Size: 280, 110  FormBorderStyle: FixedToolWindow  Text: “Download”  Font: Microsoft Sans Serif, Black (#000000), 8.25pt  Back Colour: ControlDarkDark (#696969) |

#### DisplayGrid

|  |
| --- |
| Name: Grid  Type: DisplayGrid  Size: 600, 600  Location: 5, 5  Back Colour: ControlDarkDark (#696969) |

#### Pathfinding

|  |  |
| --- | --- |
| Name: lblPathFinding  Type: Label  Size: 150, 20  Location: 610, 5  TextAllign: MiddleCenter  Text: “Pathfinding”  Font: Tahoma, White (#FFFFFF), 12.5pt  Back Colour: Transparent | Name: cmbSelectedPath  Type: ComboBox  Size: 150, 20  Location: 610, 30  Font: Microsoft Sans Serif, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF)  Items: [Breadth-first search, A\* search, Depth-first search, GreedyDFS] |
| Name: btnSolve  Type: Button  Size: 73, 40  Location: 610, 55  Text: “Solve”  Font: Microsoft Sans Serif, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) | Name: btnClear  Type: Button  Size: 73, 40  Location: 687, 55  Text: “Clear”  Font: Microsoft Sans Serif, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) |
| Name: lblShortestPath  Type: Label  Size: 100, 20  Location: 610, 100  Text: “Shortest path:”  Font: Tahoma, White (#FFFFFF), 8.25pt  Back Colour: Transparent | Name: txtShortestPath  Type: Label  Size: 55, 20  Location: 705, 100  Text: “”  Font: Tahoma, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) |
| Name: lblCellsExplored  Type: Label  Size: 100, 20  Location: 610, 125  Text: “Explored:”  Font: Tahoma, White (#FFFFFF), 8.25pt  Back Colour: Transparent | Name: txtCellsExplored  Type: Label  Size: 55, 20  Location: 705, 125  Text: “”  Font: Tahoma, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) |
| Name: lblTimeTakenPath  Type: Label  Size: 100, 20  Location: 610, 150  Text: “Time taken:”  Font: Tahoma, White (#FFFFFF), 8.25pt  Back Colour: Transparent | Name: txtTimeTakenPath  Type: Label  Size: 55, 20  Location: 705, 150  Text: “”  Font: Tahoma, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) |

#### Grid Customisation

|  |  |
| --- | --- |
| Name: lblGridCustomisation  Type: Label  Size: 150, 20  Location: 610, 205  TextAllign: MiddleCenter  Text: “Grid Customisation”  Font: Tahoma, White (#FFFFFF), 12.5pt  Back Colour: Transparent | Name: cmbSelectedMaze  Type: ComboBox  Size: 150, 20  Location: 610, 230  Font: Microsoft Sans Serif, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF)  Items: [Recursive Backtracking, Recursive Division, Kruskals, Binary Tree, Prims, User Customisation] |
| Name: btnGenerate  Type: Button  Size: 150, 40  Location: 610, 255  Text: “Generate”  Font: Microsoft Sans Serif, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) | Name: btnGenerateSolve  Type: Button  Size: 150, 40  Location: 610, 405  Text: “Generate and Solve”  Font: Microsoft Sans Serif, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) |
| Name: pnlCustomisation  Type: Label  Size: 159, 95  Location: 610, 300  Text: “Start – C Finish – V Wall - LMB  Empty – RMB”  Font: Tahoma, White (#FFFFFF), 8.25pt  Back Colour: Transparent |  |
| Name: pnlMazeAlgorithm  Type: Panel  Size: 150, 95  Location: 610, 300  Back Colour: Transparent   |  |  | | --- | --- | | Name: lblTimeTakenGrid  Type: Label  Size: 100, 20  Location: 0, 0  Text: “Time taken:”  Font: Tahoma, White (#FFFFFF), 8.25pt  Back Colour: Transparent | Name: txtTimeTakenGrid  Type: Label  Size: 55, 20  Location: 95, 0  Text: “”  Font: Tahoma, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) | | Name: lblDeadends  Type: Label  Size: 100, 20  Location: 0, 25  Text: “Deadends:”  Font: Tahoma, White (#FFFFFF), 8.25pt  Back Colour: Transparent | Name: txtDeadends  Type: Label  Size: 55, 20  Location: 95, 35  Text: “”  Font: Tahoma, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) | | Name: lblBranches  Type: Label  Size: 100, 20  Location: 0, 50  Text: “Branches:”  Font: Tahoma, White (#FFFFFF), 8.25pt  Back Colour: Transparent | Name: txtBranches  Type: Label  Size: 55, 20  Location: 95, 50  Text: “”  Font: Tahoma, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) | | Name: lblAverageLength  Type: Label  Size: 100, 20  Location: 0, 75  Text: “Avg Length:”  Font: Tahoma, White (#FFFFFF), 8.25pt  Back Colour: Transparent | Name: txtAverageLength  Type: Label  Size: 55, 20  Location: 95, 75  Text: “”  Font: Tahoma, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) | | |

#### Settings

|  |  |
| --- | --- |
| Name: lblSettings  Type: Label  Size: 150, 20  Location: 610, 485  TextAllign: MiddleCenter  Text: “Settings”  Font: Tahoma, White (#FFFFFF), 12.5pt  Back Colour: Transparent | Name: btnDownload  Type: Button  Size: 150, 40  Location: 610, 565  Text: “Download”  Font: Microsoft Sans Serif, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) |
| Name: lblDelay  Type: Label  Size: 55, 20  Location: 605, 505  Text: “Delay:”  Font: Tahoma, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) | Name: trbDelay  Type: TrackBar  Size: 69, 45  Location: 650, 505  Maximum: 200  Minimum: 0  Value: 100 |
| Name: nudDelay  Type: NumericUpDown  Size: 50, 20  Location: 710, 510  Maximum: 200  Minimum: 0  Value: 100 | Name: lblSize  Type: Label  Size: 55, 20  Location: 605, 540  Text: “Size:”  Font: Tahoma, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) |
| Name: trbSize  Type: TrackBar  Size: 69, 45  Location: 650, 535  Maximum: 18  Minimum: 0  Value: 6 | Name: nudSize  Type: NumericUpDown  Size: 50, 20  Location: 710, 540  Maximum: 1200  Minimum: 5  Value: 25 |

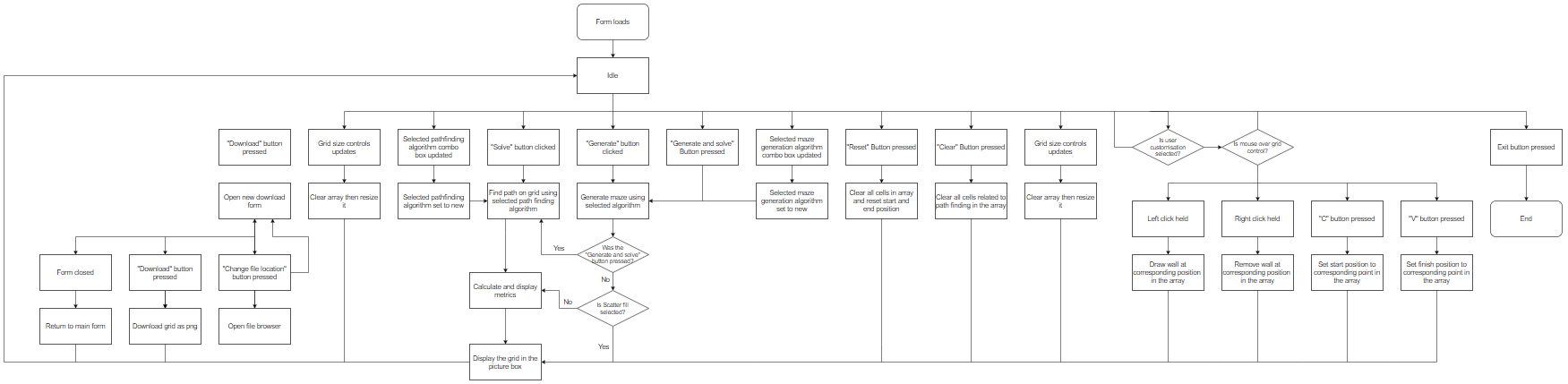
#### Download

|  |  |
| --- | --- |
| Name: btnChangeFileLocation  Type: Button  Size: 120, 50  Location: 10, 10  Text: “Change File Location”  Font: Microsoft Sans Serif, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) | Name: btnDownload  Type: Button  Size: 120, 50  Location: 140, 10  Text: “Download”  Font: Microsoft Sans Serif, Black (#000000), 8.25pt  Back Colour: White (#FFFFFF) |

## IPSO chart

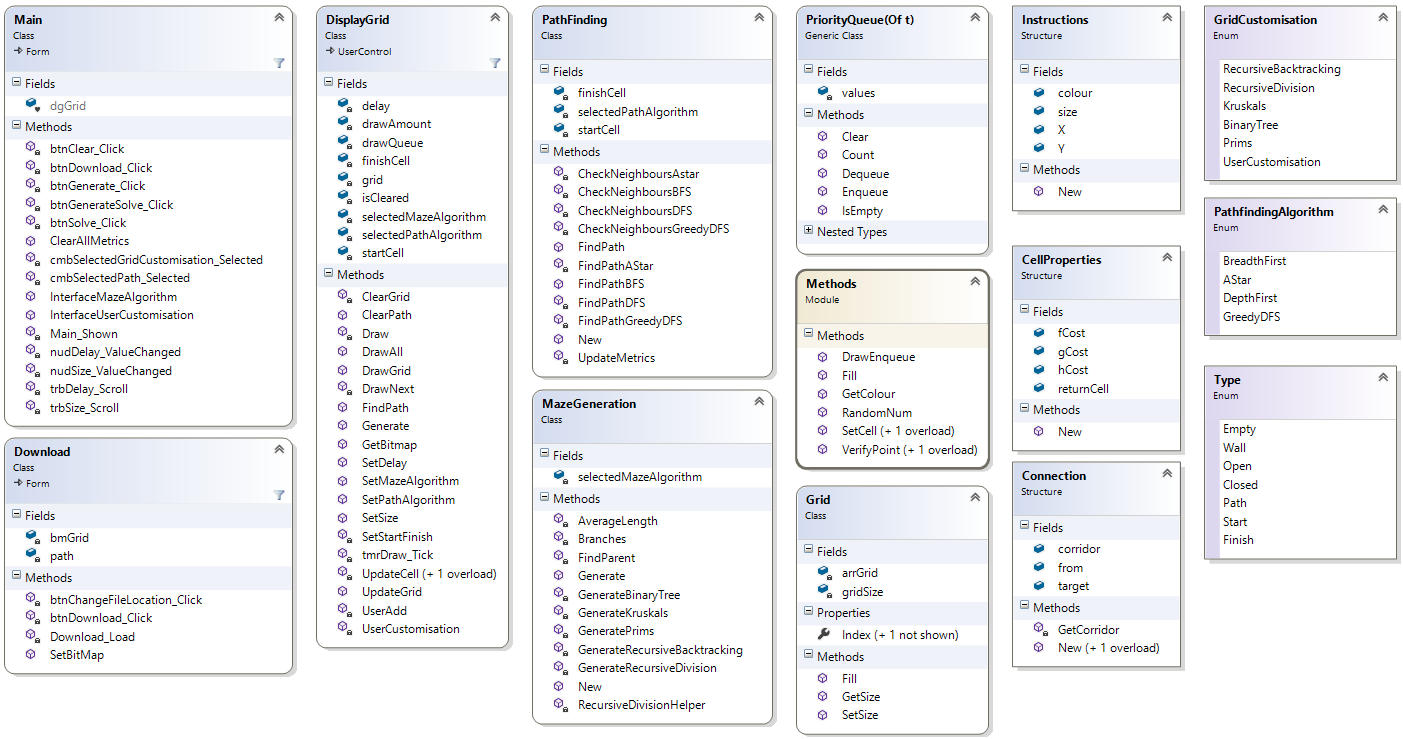
|  |  |  |  |
| --- | --- | --- | --- |
| Input | Process | Storage | Output |
| "Generate" button pressed | - User initiates maze generation algorithm  - Use the selected maze generation algorithm to solve the grid.  - Update the grid array with the new maze.  - Store a drawing queue for grid recreation | - Updated grid array with pathfinding data  - Drawing queue for visualization  - Metrics | Display the updated grid in the Picture Box embedded in the grid control. Display the metrics of the maze generation algorithm, including time taken, percentage of dead ends, percentage of branches, and average corridor length. |
| Grid customisation combobox changed to a maze generation algorithm | - Hide controls related to user customisation  - Show controls for maze generation metrics  - Store the maze generation algorithm as the selected grid customisation option. | - Current grid customisation option | Display maze generation metrics, change combobox selected index and stored grid customisation option to the maze generation algorithm. |
| Grid customisation combobox changed to a maze generation algorithm | - Hide controls related to maze generation metrics or scatter fill  - Show controls for user customisation  - Store user customisation as the selected grid customisation option. | - Current grid customisation option | Display user customisation controls, change combobox selected index and stored grid customisation option to user customisation. |
| User customizes grid (adds/removes walls) | - Allow users to interactively add or remove walls.  - Respond to mouse input to add or remove walls. | - Update grid array | Update the grid to reflect added or removed walls. |
| User customizes grid (sets start or finish locations) | - Enable users to set the start and finish points using "C" and "V" keys, respectively. | - Start and finish locations | Update the grid with the new start and finish locations. |
| "Solve" button pressed | - Use the selected pathfinding algorithm to solve the grid.  - Update the grid array with pathfinding cells, e.g., the path and explored cells.  - Store a drawing queue for grid recreation. | - Updated grid array with pathfinding data  - Drawing queue for visualization  - Metrics | Display the updated grid in the Picture Box embedded in the grid control. Display the metrics of the pathfinding algorithm, including time taken, shortest path, and cells explored. |
| Path finding combobox changed | - Update the selected pathfinding algorithm  - Store the selected algorithm as current pathfinding algorithm | - Current pathfinding algorithm | Change combobox selected index and current pathfinding algorithm option to the selected maze generation algorithm. |
| User sets delay | - Allow users to specify a delay for visualizations.  - Configure drawing instructions based on the delay settings.  - Customize the delay interval. | - Delay settings | Adjust the timing for drawing visualizations according to the user's specified delay. |
| User resizes grid | - Adjust the grid size based on the user's input.  - Resize the bitmap and array used to store the grid. | - Grid size settings  - Resize bitmap and array | Redraw the resized grid in the Picture Box embedded in the grid control. Resize the size of the array and the bitmap. |
| User opens download window | - Open a separate window for downloading grid images. | - Download window | Allow users to select the file path for downloading grid images. |
| “Change file location” button pressed | - Opens a FolderBrowserDialog  - Takes input of a file path from user | - Selected file path | Set the selected file path as the download location. |
| “Download” button pressed | - Downloads bitmap of grid to selected file path | - Bitmap in file location | Outputs bitmap to storage location. |

## Top-Down Diagram



## Class Diagram

­­­­



## Algorithms

### Maze generation algorithms

#### Kruskal's Algorithm

|  |
| --- |
| Subroutine GenerateKruskals():  For X = 0 To GridSize - 1 Step 2  For Y = 0 To GridSize - 1 Step 2  If VerifyPoint(X + 2, Y) Then  connections.Add(New Connection(X, Y, X + 2, Y))  End If  If VerifyPoint(X, Y + 2) Then  connections.Add(New Connection(X, Y, X, Y + 2))  End If  Next  Next  connections = connections.OrderBy(random)  For Each connection In connections:  fromParent = FindParent(connection.From, connections)  targetParent = FindParent(connection.Target, connections)  If fromParent <> targetParent Then  parents(fromParent) = targetParent  SetCell(connection.From, Type.Empty)  SetCell(connection.Corridor, Type.Empty)  SetCell(connection.Target, Type.Empty)  End If  Next  End Subroutine  Private Function FindParent(cell, parents)  If Not parents.ContainsKey(cell) Then  parents(cell) = cell  ElseIf parents(cell) <> cell Then  parents(cell) = FindParent(parents(cell), parents)  End If  Return parents(cell)  End Function |

This is my pseudocode for the random kruskals algorithm.

Kruskal's algorithm is a technique for constructing a minimal spanning tree within a weighted graph. Unlike the traditional Kruskal's algorithm, which prioritizes edges based on their weights, the Randomized Kruskal's algorithm introduces a random element. This alteration results in the generation of a random perfect maze, particularly useful in scenarios involving grid-based structures.

The core process of the algorithm involves placing all potential edges into a list and then randomly selecting an edge. Upon selecting an edge, the algorithm examines the two nodes it connects. If these nodes are already connected, the edge is discarded. On the other hand, if the nodes are not connected, the algorithm establishes the connection.

To support this approach, the algorithm relies on a helper method named FindParent(). This recursive function plays a crucial role in verifying whether two cells belong to the same set. Additionally, the algorithm utilizes a custom-defined data type called Connection. This data type captures two points, stores them, and facilitates the calculation of the corridor cell between them.

In the context of my solution, this algorithm contributes to achieving Objective 5.1.1 by expanding the array of implemented algorithms, thereby enhancing the overall variety of solutions.

#### Prim's Algorithm

|  |
| --- |
| Subroutine GeneratePrims():  randomStartingCell = (random)  frontierList.Add(randomStartingCell)  visited.Add(randomStartingCell)  While frontierList.Count > 0  currentConnection = frontierList(random)  currentCell = currentConnection.Target  frontierList.Remove(CurrentConnection)  SetCell(currentConnection.Corridor, Type.Empty)  SetCell(currentCell, Type.Empty)  directions = {(-2, 0), (2, 0), (0, -2), (0, 2)}  For Each direction In directions  neighbour = (currentCell + direction)  If VerifyPoint(neighbour) And Not visited.Contains(neighbour) Then  visited.Add(neighbour)  frontierList.Add(New Connection(currentCell, neighbour))  End If  Next  End While  End Subroutine |

This is my psedocode for Prim’s algorithm.

Prim's Algorithm is a maze generation technique that focuses on creating a minimal spanning tree within a graph. Unlike Kruskal's approach, Prim's Algorithm starts with a single, randomly chosen cell and incrementally grows the maze by adding passages to adjacent cells. This method leads to the formation of a perfect maze with a randomized pattern.

The algorithm initiates by selecting a random starting cell, marking it as part of the maze, and adding its neighboring walls to a list. It then iteratively selects a wall from the list, checking if the cells on either side belong to the maze or not. If one of the cells is part of the maze and the other is not, the wall becomes a passage, and the non-maze cell is incorporated into the maze. The walls of this new cell are then added to the list.

Prim's Algorithm employs a specialized 'Connection' type to manage relationships between cells, storing two points and calculating the connecting corridor cell

By incorporating Prim's Algorithm into the solution, Objective 5.1.1 is fulfilled as it introduces a distinctive maze generation method, contributing to the overall diversity of algorithms implemented in the solution.

#### Recursive Backtracker

|  |
| --- |
| Subroutine GenerateRecursiveBacktracking():  SetCell(CurrentCell, Type.Empty)  visitedStack.Push(currentCell)  Do Until visitedStack.Count = 0  directions = {(-2, 0), (2, 0), (0, -2), (0, 2)}  For Each direction In directions  neighbouringCell = (currentCell + direction)  If VerifyPoint(neighbouringCell) Then  If grid(neighbouringCell) = Type.Wall Then  neighbours.Add(neighbouringCell)  End If  End If  Next  If neighbours.Count > 0 Then  neighbourCell neighbours(random)  corridorCell = ((neighbourCell + currentCell) \ 2)  visitedStack.Push(neighbourCell)  SetCell(corridorCell, Type.Empty)  SetCell(neighbourCell, Type.Empty)  currentCell = neighbourCell  Else  currentCell = visitedStack.Pop  End If  Loop  End Subroutine |

This is my pseudocode for the Recursive Backtracking algorithm.

Recursive Backtracking is a maze generation technique that operates by exploring a random path and backtracking when it encounters a dead-end. Unlike other algorithms, Recursive Backtracking creates mazes with long, winding passages and open areas.

The algorithm starts by selecting a random starting cell and marks it as part of the maze. It then explores a random neighboring cell, repeating the process recursively until it reaches a dead-end. Upon backtracking, it continues exploring other unvisited paths. This cycle continues until all cells are part of the maze.

By integrating Recursive Backtracking into the solution, Objective 5.1.1 is achieved, enhancing the variety of maze generation algorithms in the overall solution.

#### Recursive Division

|  |
| --- |
| Subroutine GenerateRecursiveDivision():  boundary = grid.GetSize - 2 + grid.GetSize Mod 2  RecursiveDivisionHelper( (0, 0), (boundary, boundary))  End Subroutine  Subroutine RecursiveDivisionHelper(tL, bR):  If (bR.X - tL.X) > 1 And (bR.Y - tL.Y) > 1 Then  divideHorizontally = False  If (bR.X - tL.X) = 0 Then  divideHorizontally = True  ElseIf (bR.Y - tL.Y) = 0 Then  divideHorizontally = False  Else  divideHorizontally = RandomNum(0, 1)  End If  If divideHorizontally Then  wallPosition = (tL.Y + 1) + RandomNum(0, Math.Floor((bR.Y - tL.Y - 1) / 2)) \* 2  For X = tL.X To bR.X  SetCell(X, wallPosition, Type.Wall)  Next  Else  wallPosition = (tL.X + 1) + RandomNum(0, Math.Floor((bR.X - tL.X - 1) / 2)) \* 2  For Y = tL.Y To bR.Y  SetCell(wallPosition, Y, Type.Wall)  Next  End If  openingPosition = 0  If divideHorizontally Then  openingPosition = tL.X + RandomNum(0, Math.Floor((bR.X - tL.X) / 2)) \* 2  SetCell(openingPosition, wallPosition, Type.Empty)  Else  openingPosition = tL.Y + RandomNum(0, Math.Floor((bR.Y - tL.Y) / 2)) \* 2  SetCell(wallPosition, openingPosition, Type.Empty)  End If  RecursiveDivisionHelper(tL, (bR.X, wallPosition - 1))  RecursiveDivisionHelper((tL.X, wallPosition + 1), bR)  End If  End Subroutine |

This is my pseudocode for the Recursive Division algorithm.

Recursive Division is a maze generation technique that divides the maze into smaller sections through recursive splitting. It creates mazes with long, straight passages and a distinctive pattern.

The algorithm begins by establishing the boundary of the maze. It then recursively selects a random wall within the current section and removes it, creating two smaller sections. This process continues until the sections are reduced to a certain size. Additionally, openings are introduced in the walls to ensure connectivity.

By incorporating Recursive Division into the solution, Objective 5.1.1 is fulfilled as it introduces a unique approach to maze generation, contributing to the overall diversity of algorithms implemented in the solution.

#### Binary Tree

|  |
| --- |
| Subroutine GenerateBinaryTree():  For X = 0 To grid.GetSize - 1 Step 2  For Y = 0 To grid.GetSize - 1 Step 2  direction = RandomNum(1, 2)  If (direction = 1 Or X = 0) And VerifyPoint(X, Y - 1, grid.GetSize) Then  SetCell(X, Y - 1, Type.Empty)  End If  If (direction = 2 Or Y = 0) And VerifyPoint(X - 1, Y, grid.GetSize) Then  SetCell(X - 1, Y, Type.Empty)  End If  SetCell(X, Y, Type.Empty)  Next  Next  End Subroutine |

This is my pseudocode for the Binary Tree algorithm.

Binary Tree is a maze generation technique that creates mazes with distinct diagonal passages and a regular pattern.

The algorithm iterates through each cell in the grid, selecting a random direction (down or right). If the chosen direction is down or the cell is on the left border, it removes the cell below. If the direction is right or the cell is on the top border, it removes the cell to the right. This process continues until all cells are visited.

By incorporating Binary Tree into the solution, Objective 5.1.1 is fulfilled, as it introduces a specific maze generation method, contributing to the overall diversity of algorithms implemented in the solution.

### Pathfinding algorithms

#### Breadth-First Search (BFS)

|  |
| --- |
| Subroutine FindPathBFS():  shortestPath = 1  cellsExplored = 1  openQueue.Enqueue(startCell)  Do Until openQueue.Count = 0 Or pathFound = True  currentCell = openQueue.Dequeue()  returnCell = currentCell  If grid(currentCell) < Type.Start Then  SetCell(currentCell, Type.Closed)  cellsExplored += 1  End If  directions = {(0, -1), (1, 0), (0, 1), (-1, 0)}  For Each direction In directions  CheckNeighboursBFS(currentCell, returnCells, openQueue, pathFound, direction)  Next  Loop  If pathFound Then  Do Until returnCell = startCell  shortestPath += 1  SetCell(returnCell, Type.Path)  returnCell = returnCells(returnCell)  Loop  End If  End Subroutine  Subroutine CheckNeighboursBFS(currentCell, returnCells, openQueue, pathFound, direction):  neighbour = Point(currentCell + direction)  If VerifyPoint(neighbour, grid.GetSize) Then  If grid(neighbour) = Type.Empty Then  openQueue.Enqueue(neighbour)  SetCell(neighbour, Type.Open)  returnCells(neighbour) = currentCell  ElseIf grid(neighbour) = Type.Finish Then  pathFound = True  End If  End If  End Subroutine |

This is my pseudocode for the Breadth-First Search (BFS) algorithm.

Breadth-First Search is a pathfinding algorithm that explores the grid layer by layer, prioritizing nodes closer to the starting point for traversal.

The algorithm begins by enqueuing the start cell and iteratively dequeuing cells while enqueueing their unexplored neighbors. It continues until the destination cell is reached or all accessible cells are explored.

By incorporating Breadth-First Search into the solution, Objective 4.1.1 is met, as it introduces a specific pathfinding algorithm, contributing to the overall variety of algorithms implemented in the solution.

#### Depth-First Search (DFS)

|  |
| --- |
| Subroutine FindPathDFS():  shortestPath = 1  cellsExplored = 1  openStack.Push(startCell)  Do Until openStack.Count = 0 Or pathFound = True  currentCell = openStack.Pop()  returnCell = currentCell  If grid(currentCell) < Type.Start Then  cellsExplored += 1  SetCell(currentCell, Type.Closed)  End If  directions = {(0, -1), (-1, 0), (0, 1), (1, 0)}  For Each direction In directions  CheckNeighboursDFS(currentCell, returnCells, openStack, pathFound, direction)  Next  Loop  If pathFound Then  Do Until returnCell = startCell  shortestPath += 1  SetCell(returnCell, Type.Path)  returnCell = returnCells(returnCell)  Loop  End If  End Subroutine  Subroutine CheckNeighboursDFS(currentCell, returnCells, openStack, pathFound, direction):  neighbour = Point(currentCell + direction)  If VerifyPoint(neighbour, grid.GetSize) Then  If grid(neighbour) = Type.Empty Then  openStack.Push(neighbour)  SetCell(neighbour, Type.Open)  returnCells(neighbour) = currentCell  ElseIf grid(neighbour) = Type.Finish Then  pathFound = True  End If  End If  End Subroutine |

This is my pseudocode for the Depth-First Search (DFS) algorithm.

Depth-First Search is a pathfinding algorithm that explores the grid by traversing as far as possible along one branch before backtracking.

The algorithm initiates by pushing the start cell onto a stack and then iteratively popping cells, exploring their unvisited neighbors. It continues until the destination cell is reached or all accessible cells are explored.

By incorporating Depth-First Search into the solution, Objective 4.1.1 is met, as it introduces a distinct pathfinding algorithm, contributing to the overall diversity of algorithms implemented in the solution.

#### Greedy Best-First Search

|  |
| --- |
| Subroutine FindPathGreedyDFS():  shortestPath = 1  cellsExplored = 1  openQueue = New PriorityQueue(Of Point)  cellProperties = New Dictionary(Of Point, CellProperties)  openQueue.Enqueue(startCell, 0) ' Assuming 0 as the initial heuristic cost for the startCell  cellProperties(startCell) = New CellProperties  Do Until openQueue.IsEmpty Or pathFound = True  currentCell = openQueue.Dequeue()  If grid(currentCell) <> Type.Closed Then  returnCell = currentCell  If grid(currentCell) < Type.Start Then  SetCell(currentCell, Type.Closed)  cellsExplored += 1  End If  directions = {(0, -1), (1, 0), (0, 1), (-1, 0)}  For Each direction In directions  CheckNeighboursGreedyDFS(grid, currentCell, cellProperties, openQueue, pathFound, direction)  Next  End If  Loop  If pathFound Then  Do Until returnCell = startCell  shortestPath += 1  SetCell(returnCell, Type.Path)  returnCell = cellProperties(returnCell).ReturnCell  Loop  UpdateMetrics(shortestPath, cellsExplored)  End If  End Subroutine  Subroutine CheckNeighboursGreedyDFS(grid, currentCell, cellProperties, openQueue, pathFound, direction):  neighbour = Point(currentCell + direction)  If VerifyPoint(neighbour, grid.GetSize) Then  If grid(neighbour) = Type.Empty Then  newProperties = New CellProperties(neighbour, currentCell, finishCell, cellProperties)  cellProperties(neighbour) = newProperties  openQueue.Enqueue(neighbour, newProperties.hCost)  SetCell(neighbour, Type.Open)  ElseIf grid(neighbour) = Type.Finish Then  pathFound = True  End If  End If  End Subroutine |

This is my pseudocode for the Greedy Best-First Search algorithm.

Greedy Best-First Search is a heuristic-based pathfinding algorithm that prioritizes nodes based on their estimated distance to the goal without considering the actual cost of reaching them.

The algorithm initiates by enqueueing the start cell with an initial heuristic estimate to the goal. It then iteratively dequeues cells, evaluating their neighbors and enqueuing them based on their heuristic estimates. The algorithm selects the cell that appears most promising according to the heuristic, progressing towards the goal.

By incorporating Greedy Best-First Search into the solution, Objective 4.1.1 is met, as it introduces an efficient heuristic-based pathfinding algorithm, contributing to the overall diversity of algorithms implemented in the solution.

#### A\* Algorithm

|  |
| --- |
| Subroutine FindPathAStar():  shortestPath = 1  cellsExplored = 1  openQueue.Enqueue(startCell, 0, 0)  Do Until openQueue.IsEmpty Or pathFound = True  currentCell = openQueue.Dequeue()  If grid(currentCell) <> Type.Closed Then  returnCell = currentCell  If grid(currentCell) < Type.Start Then  SetCell(currentCell, Type.Closed)  cellsExplored += 1  End If  directions = {(0, -1), (1, 0), (0, 1), (-1, 0)}  For Each direction In directions  CheckNeighboursAstar(currentCell, cellProperties, openQueue, pathFound, direction)  Next  End If  Loop  If pathFound Then  Do Until returnCell = startCell  shortestPath += 1  SetCell(returnCell, Type.Path)  returnCell = cellProperties(returnCell).ReturnCell  Loop  End If  End Subroutine  Subroutine CheckNeighboursAstar(currentCell, cellProperties, openQueue, pathFound, direction):  neighbour = (currentCell + direction)  If VerifyPoint(neighbour, grid.GetSize) Then  If {Type.Empty, Type.Open, Type.Closed}.Contains(grid(neighbour)) Then  newProperties = (neighbour, currentCell, finishCell, cellProperties)  If grid(neighbour) = Type.Empty Then  cellProperties(neighbour) = newProperties  openQueue.Enqueue(neighbour, newProperties.fCost, newProperties.hCost)  SetCell(neighbour, Type.Open)  ElseIf cellProperties.ContainsKey(neighbour) AndAlso newProperties.fCost < cellProperties(neighbour).fCost Then  cellProperties(neighbour) = newProperties  openQueue.Enqueue(neighbour, newProperties.fCost, newProperties.hCost)  End If  ElseIf grid(neighbour) = Type.Finish Then  pathFound = True  End If  End If  End Subroutine |

This is my pseudocode for the A\* Search algorithm.

A\* Search is a pathfinding algorithm that combines the principles of Dijkstra's algorithm and Greedy Best-First Search, using a heuristic to estimate the cost of the cheapest path from the start cell to the goal.

The algorithm begins by enqueueing the start cell with an initial cost of 0 and iteratively dequeues cells, evaluating their neighbors. It calculates the total cost for each neighbor based on the actual cost from the start cell and a heuristic estimate to the goal. The cell with the lowest total cost is selected for exploration.

By incorporating A\* Search into the solution, Objective 4.1.1 is met, as it introduces a sophisticated pathfinding algorithm, contributing to the overall diversity of algorithms implemented in the solution.

### User defined data structures

#### Cell properties

|  |
| --- |
| Structure CellProperties  returnPt As Point  hCost As Integer  gCost As Integer  fCost As Integer  Subroutine CalculateHCost(currentPt, finishPt):  hCost = Math.Abs(finishPt.X - currentPt.X) + Math.Abs(finishPt.Y - currentPt.Y)  End Subroutine  Subroutine CalculateGCost(cellProperties, parentPt):  gCost = CellProperties(parentPt).GCost + 1  End Subroutine  Subroutine CalculateFCost():  fCost = hCost + gCost  End Subroutine  Subroutine SetReturnPt(newReturnPt):  returnPt = newReturnPt  End Subroutine  End Structure |

This is my pseudocode for the CellProperties structure used in A\* Search and Greedy Best-First Search.

The CellProperties structure encapsulates essential information for each cell during pathfinding.

In the A\* Search algorithm, CellProperties aids in computing and updating costs during exploration, prioritizing cells based on their total cost (GCost + HCost). This structure supports efficient path reconstruction after reaching the destination.

For Greedy Best-First Search, the CellProperties structure focuses solely on heuristic estimates (HCost). During exploration, cells are enqueued based on their heuristic values, directing the algorithm towards the goal. Path reconstruction relies on the ReturnPt attribute to trace the most promising path.

By integrating the CellProperties structure into both A\* Search and Greedy Best-First Search, the solution achieves Objective 4.1.1 by implementing versatile pathfinding algorithms, enriching the algorithmic diversity in the overall solution.

#### Maze Algorithm

|  |
| --- |
| Public Enum MazeAlgorithm  RecursiveBacktracking  RecursiveDivision  Kruskals  BinaryTree  Prims  ScatterFill  UserCustomisation  End Enum |

The MazeAlgorithm enum supports easy selection of maze generation algorithms in the solution, aligning with Objective 5.2. Each enum value corresponds directly to a ComboBox index, facilitating user-friendly customization. This design promotes modularity, enabling the addition of new algorithms while ensuring a consistent and accessible interface for users.

#### Path Algorithm

|  |
| --- |
| Public Enum PathAlgorithm  BreadthFirst  AStar  DepthFirst  GreedyDFS  End Enum |

The PathAlgorithm enum streamlines the selection of pathfinding algorithms in the solution, aligning with Objective 4.2. Each enum value corresponds directly to a ComboBox index, simplifying user interaction. This design promotes modularity, making it easy to add new algorithms while maintaining a consistent and user-friendly interface.

#### Priority Queue

|  |
| --- |
| Class PriorityQueue(Of t):  Structure Data:  Subroutine New(newItem, newPriority1, newPriority2):  item = newItem  priority1 = newPriority1  priority2 = newPriority2  End Subroutine  End Structure  Function Count():  Return values.Count  End Function  Function IsEmpty():  Return values.Count = 0  End Function  Subroutine Clear():  values.Clear()  End Sub  Subroutine Enqueue(newIte, newPriority, Optional newPriority2 = 0):  newData = New Data(newItem, newPriority1, newPriority2)  index = 0  For i = 0 To values.Count - 1:  If newData.priority1 > values(i).priority1 Or (newData.priority1 = values(i).priority1 And newData.priority2 > values(i).priority2):  index += 1  Else Exit For  End If  Next  values.Insert(index, newData)  End Sub  Function Dequeue():  dequeueData = values(0)  values.RemoveAt(0)  Return dequeueData.item  End Function  End Class |

The provided pseudocode outlines a versatile PriorityQueue class, accommodating any data type with one or two priorities. This structure is integral to A\* Search and Greedy Best-First Search, offering flexibility in prioritizing nodes based on cost or heuristic estimates. The class meets Objective 4.1 by providing a foundational and adaptable data structure for diverse algorithm implementations.

#### Type

|  |
| --- |
| Public Enum Type  Empty  Wall  Open  Closed  Path  Start  Finish  End Enum |

The provided enumeration Type enhances code readability by using descriptive labels for different cell types in the grid. This approach contributes to Objective 1.2.1 by promoting clear and understandable code, as developers can easily discern the purpose and state of each cell in the grid based on the expressive labels.

#### Instructions

|  |
| --- |
| Structure Instructions  x As Integer  y As Integer  colour As Color  size As Integer  Subroutine New(newx, newy, newColour, Optional newSize = 1):  x = newx  y = newy  colour = newColour  size = newSize  End Sub  End Structure |

The provided code introduces a structure named Instructions designed to encapsulate information necessary for drawing on a grid. This structure includes attributes such as coordinates (x and y), color (colour), and an optional size (size). The New subroutine serves as a constructor for initializing these attributes.

Below is an example of how it could be used to store information in a queue and then display a drawing step-by-step using a timer:

|  |
| --- |
| drawQueue As Queue(Of Instructions)  Private Subroutine tmrDraw\_Tick() Handles tmrDraw.Tick  If drawQueue.Count = 0 Then  tmrDraw.Stop()  Else  DrawNext()  End If  End Subroutine  Private Subroutine DrawNext()  current = drawQueue.Dequeue  Draw(current.X, current.Y, current.colour, current.size)  End Subroutine  Private Subroutine Draw(x, y, colour, Optional size = 1)  bm.FillRectangle((colour), x, y, size, size)  End Subroutine |

Here, a drawing queue, denoted as drawQueue, is declared as a queue of Instructions. The purpose of this queue is to store a sequence of drawing instructions, enabling a step-by-step visualization of the drawing process.

A timer, labeled tmrDraw, triggers the tmrDraw\_Tick subroutine, responsible for handling the drawing process. If the drawing queue (drawQueue) is not empty, the DrawNext subroutine is invoked. Otherwise, the timer is halted.

Within the DrawNext subroutine, the next drawing instruction (current) is dequeued from the queue, and the Draw subroutine is called with the relevant parameters.

The Draw subroutine takes coordinates (x and y), color (colour), and an optional size (size) as parameters. It utilizes these parameters to fill a rectangle on the bitmap (bm) using the FillRectangle method for drawing.

In terms of meeting objectives, this implementation satisfies Objective 3.2 and fulfills Objective 3.3 by providing a dynamic and interactive mechanism for rendering multiple drawing steps, enhancing user engagement with the grid.

# Technical Solution

## Code listing

### Main

|  |
| --- |
| ' Form that acts as main page that has all controls and the display grid  Public Class Main  ' Subroutine that's ran when the form is shown  Private Sub Main\_Shown(sender As Object, e As EventArgs) Handles Me.Shown  ' Sets the default selected maze and path finding indexes in the combo box  cmbSelectedGridCustomisation.SelectedIndex = GridCustomisation.RecursiveBacktracking  cmbSelectedPath.SelectedIndex = PathfindingAlgorithm.BreadthFirst  pnlCustomisation.BringToFront()  ' Generates and displays the selected maze algorithm  dgGrid.Generate()  dgGrid.DrawAll()  dgGrid.UpdateGrid()  End Sub  #Region "Controls"  ' Subroutine that sets the selected maze generation algorithm when the combo box is changed  Private Sub cmbSelectedGridCustomisation\_Selected(sender As Object, e As EventArgs) Handles cmbSelectedGridCustomisation.SelectedIndexChanged  Dim selectedGridCustomisation As GridCustomisation = cmbSelectedGridCustomisation.SelectedIndex  ' Sets the interface to grid customisation if it is the selected grid customisation option  If selectedGridCustomisation = GridCustomisation.UserCustomisation Then  InterfaceUserCustomisation()  Else  InterfaceMazeAlgorithm()  End If  dgGrid.SetMazeAlgorithm(selectedGridCustomisation)  End Sub  ' Subroutine that sets the selected path finding algorithm when the combo box is changed  Private Sub cmbSelectedPath\_Selected(sender As Object, e As EventArgs) Handles cmbSelectedPath.SelectedIndexChanged  Dim selectedPathAlgorithm As PathfindingAlgorithm = cmbSelectedPath.SelectedIndex  dgGrid.SetPathAlgorithm(selectedPathAlgorithm)  End Sub  ' Subroutine that sets the delay of the grid and updates the numeric up down box to match  Private Sub trbDelay\_Scroll(sender As Object, e As EventArgs) Handles trbDelay.Scroll  Dim newDelay As Integer = trbDelay.Value  ' If the numeric up down box value doesn't match the new delay updates it  If nudDelay.Value <> newDelay Then  nudDelay.Value = newDelay  End If  End Sub  ' Subroutine that sets the delay of the grid and updates the numeric up down box to match  Private Sub nudDelay\_ValueChanged(sender As Object, e As EventArgs) Handles nudDelay.ValueChanged  Dim newDelay As Integer = nudDelay.Value  ' If the track bar value doesn't match the new delay updates it  If trbDelay.Value <> newDelay Then  trbDelay.Value = newDelay  End If  dgGrid.SetDelay(newDelay)  End Sub  ' Subroutine that sets the size of the grid and updates the numeric up down box to match  Private Sub trbSize\_Scroll(sender As Object, e As EventArgs) Handles trbSize.Scroll  ' Creates an array that represents the custom scaling of the track bar  Dim sizeList() As Integer = {5, 8, 10, 12, 15, 20, 25, 30, 40, 50, 60, 75, 100, 120, 150, 200, 300, 600, 1200}  ' Sets the new size as the index in the array  Dim newSize As Integer = sizeList(trbSize.Value)  ' If the numeric up down box value doesn't match the new size updates it  If nudSize.Value <> newSize Then  nudSize.Value = newSize  End If  End Sub  ' Subroutine that sets the size of the grid and updates the track bar to match  Private Sub nudSize\_ValueChanged(sender As Object, e As EventArgs) Handles nudSize.ValueChanged  Dim newSize As Integer = nudSize.Value  ' If the track bar value doesn't match the new size updates it  If trbSize.Value <> newSize Then  ' Creates an array that represents the custom scaling of the track bar  Dim sizeList() As Integer = {5, 8, 10, 12, 15, 20, 25, 30, 40, 50, 60, 75, 100, 120, 150, 200, 300, 600, 1200}  Dim index As Integer  ' Works out the track bar index of the new size  Do Until newSize <= sizeList(index)  index += 1  Loop  trbSize.Value = index  End If  dgGrid.SetSize(newSize)  End Sub  ' Subroutine that generates the selected grid customisation algorithm  Private Sub btnGenerate\_Click(sender As Object, e As EventArgs) Handles btnGenerate.Click  dgGrid.Generate()  dgGrid.DrawGrid()  End Sub  ' Subroutine that solves the grid using the selected path finding algorithm  Private Sub btnSolve\_Click(sender As Object, e As EventArgs) Handles btnSolve.Click  dgGrid.ClearPath()  dgGrid.FindPath()  dgGrid.DrawGrid()  End Sub  ' Subroutine that clears all path finding cells on the grid  Private Sub btnClear\_Click(sender As Object, e As EventArgs) Handles btnClear.Click  dgGrid.ClearPath()  dgGrid.UpdateGrid()  End Sub  ' Subroutine that generate and solve button a maze using the selected maze and path finding algorithms  Private Sub btnGenerateSolve\_Click(sender As Object, e As EventArgs) Handles btnGenerateSolve.Click  dgGrid.Generate()  dgGrid.FindPath()  dgGrid.DrawGrid()  End Sub  ' Subroutine that open the download form and sets the bitmap  Private Sub btnDownload\_Click(sender As Object, e As EventArgs) Handles btnDownload.Click  Download.Show()  Download.BringToFront()  Download.Location = New Point(Location.X + 200, Location.Y + 250)  ' Draws all so the bitmap passed to download is the completed grid  dgGrid.DrawAll()  Download.SetBitMap(dgGrid.GetBitmap)  End Sub  #End Region  #Region "Interface"  ' Subroutine that shows the controls related to maze generation  Public Sub InterfaceMazeAlgorithm()  pnlCustomisation.Visible = False  btnGenerate.Text = "Generate"  End Sub  ' Subroutine that shows the controls related to user customisation  Public Sub InterfaceUserCustomisation()  pnlCustomisation.Visible = True  btnGenerate.Text = "Reset"  End Sub  ' Subroutine that resets all text boxes showing information about metrics  Public Sub ClearAllMetrics()  ' Path finding  txtShortestPath.Text = ""  txtCellsExplored.Text = ""  txtTimeTakenPath.Text = ""  ' Maze generation  txtDeadends.Text = ""  txtBranches.Text = ""  txtAverageLength.Text = ""  txtTimeTakenGrid.Text = ""  End Sub  #End Region  End Class |

### Download

|  |
| --- |
| ' Form that acts as a pop up that allows the user to download the currently generated grid  Public Class Download  ' Stores the path as a string and the bitmap to download  Dim path As String = Application.StartupPath.Substring(0, Application.StartupPath.Length - 9) & "Images"  Dim bmGrid As Bitmap  ' Subroutine that sets the bitmap to download  Public Sub SetBitMap(newBMGrid As Bitmap)  bmGrid = newBMGrid  End Sub  ' Subroutine that sets the path for the folder browser  Private Sub Download\_Load(sender As Object, e As EventArgs) Handles MyBase.Load  fbdChooseFileLocation.SelectedPath = path  End Sub  ' Subroutine that opens a file browser and allows user to select path  Private Sub btnChangeFileLocation\_Click(sender As Object, e As EventArgs) Handles btnChangeFileLocation.Click  fbdChooseFileLocation.ShowDialog()  path = fbdChooseFileLocation.SelectedPath  End Sub  ' Subroutine that downloads the bitmap to the selected file path  Private Sub btnDownload\_Click(sender As Object, e As EventArgs) Handles btnDownload.Click  Dim stepper As Integer  Dim saved As Boolean  ' Loops through until it finds a file name that isn't taken  Do Until saved  Dim newPath As String = path & "\Grid" & stepper & ".png"  ' If the file doesn't exist then save the bitmap  If Not IO.File.Exists(newPath) Then  bmGrid.Save(newPath)  saved = True  End If  stepper += 1  Loop  ' Hides the form  Hide()  End Sub  End Class |

### DisplayGrid

|  |
| --- |
| ' User control that allows for displaying of grid  Public Class DisplayGrid  ' Function that returns whether or not a key is currently presses  Private Declare Function GetAsyncKeyState Lib "user32" (vKey As Integer) As Integer  ' Variables to the current grid customisation and path finding algorithms  Private selectedMazeAlgorithm As GridCustomisation  Private selectedPathAlgorithm As PathfindingAlgorithm  Private delay As Integer = 100  Private drawAmount As Integer = 1  ' Array that stores the types of all cells on the grid  Private grid As New Grid  ' Boolean that stores whether or not the path has been clear which is needed for user customisation  Private isCleared As Boolean  ' Queue to store the instructions to draw the array  Private bmGrid As New Bitmap(600, 600)  Private drawQueue As New Queue(Of Instructions)  ' Variables to store the StartCell and FinishCell coordinates  Private startCell As Point  Private finishCell As Point  ' Subroutine that sets the selected path finding algorithm  Public Sub SetPathAlgorithm(newPathAlgorithm As PathfindingAlgorithm)  ' Updates the variable  selectedPathAlgorithm = newPathAlgorithm  ' Clears the path then solves the maze  ClearPath()  FindPath()  DrawGrid()  End Sub  ' Subroutine that sets the selected maze algorithm  Public Sub SetMazeAlgorithm(newMazeAlgorithm As GridCustomisation)  ' Updates the variable  selectedMazeAlgorithm = newMazeAlgorithm  ' Generates new maze  Generate()  DrawGrid()  End Sub  ' Subroutine that sets the new delay for drawing the grid  Public Sub SetDelay(newDelay As Integer)  ' Updates the variable  delay = newDelay  tmrDraw.Interval = 1  Select Case delay  Case <= 5  drawAmount = 7 - delay  Case > 5  drawAmount = 1  tmrDraw.Interval = delay - 5  End Select  End Sub  ' Subroutine that sets the new size of the grid  Public Sub SetSize(newSize As Integer)  ' Updates the variable and resizes the array  grid.SetSize(newSize)  ' Sets the size of the bitmap  If grid.GetSize >= 600 Then  bmGrid = New Bitmap(grid.GetSize, grid.GetSize)  Else  bmGrid = New Bitmap(600, 600)  End If  ' If the grid customisation is user customisation reset  If selectedMazeAlgorithm = GridCustomisation.UserCustomisation Then  Generate()  ' If it's a maze generation algorithm stop drawing  Else  drawQueue.Clear()  End If  End Sub  ' Subroutine that sets the start and finish cell dependent on the grid customisation option  Private Sub SetStartFinish()  If selectedMazeAlgorithm = GridCustomisation.UserCustomisation Then  startCell = New Point(0, 0)  finishCell = New Point(grid.GetSize - 1, grid.GetSize - 1)  Else  Dim boundary As Integer = grid.GetSize - 2 + grid.GetSize Mod 2  startCell = New Point(0, 0)  finishCell = New Point(boundary, boundary)  End If  End Sub  ' Function that returns the bitmap  Public Function GetBitmap()  Return bmGrid  End Function  ' Subroutine that stops the timer then sets all the Closed, Open and Path cells to empty  Public Sub ClearPath()  If Not isCleared Then  isCleared = True  ' Sets all the Closed, Open and Path cells to empty  DrawAll()  For Y = 0 To grid.GetSize - 1  For X = 0 To grid.GetSize - 1  If {Type.Path, Type.Open, Type.Closed}.Contains(grid(X, Y)) Then  UpdateCell(X, Y, Type.Empty)  End If  Next  Next  End If  End Sub  ' Fills the grid with 1 colour  Private Sub ClearGrid()  grid.Fill(Type.Empty)  Draw(0, 0, GetColour(Type.Empty), grid.GetSize)  End Sub  ' Subroutine that updates the cell and draws the update  Private Sub UpdateCell(X As Integer, Y As Integer, cellType As Type)  grid(X, Y) = cellType  Draw(X, Y, GetColour(cellType))  End Sub  Private Sub UpdateCell(Point As Point, cellType As Type)  UpdateCell(Point.X, Point.Y, cellType)  End Sub  #Region "Main"  ' Subroutine that allows the user to customise the job  Private Sub UserCustomisation(sender As Object, e As MouseEventArgs) Handles picGrid.MouseMove, picGrid.MouseDown  If selectedMazeAlgorithm = GridCustomisation.UserCustomisation Then  ' Turn the location of the cursor on the grid into the corresponding coordinates in the array  Dim X As Integer = Math.Floor(e.Location.X / (600 / grid.GetSize))  Dim Y As Integer = Math.Floor(e.Location.Y / (600 / grid.GetSize))  Dim newPT As New Point(X, Y)  ' Check if the point is inside the bounds of the array and that it isnt the start or finish cell  'If VerifyPoint(newPT, grid.GetSize) AndAlso grid(newPT) < Type.Start Then  If VerifyPoint(newPT, grid.GetSize) Then  ' Sets to wall when LMB is pressed  If e.Button = MouseButtons.Left Then  UserAdd(newPT, Type.Wall)  ' Sets to empty when RMB is pressed  ElseIf e.Button = MouseButtons.Right Then  UserAdd(newPT, Type.Empty)  ' Sets to StartCell when the C key is pressed  ElseIf GetAsyncKeyState(Keys.C) Then  UpdateCell(startCell, Type.Empty)  startCell = newPT  UserAdd(newPT, Type.Start)  ' Sets to FinishCell when the V key is pressed  ElseIf GetAsyncKeyState(Keys.V) Then  UpdateCell(finishCell, Type.Empty)  finishCell = newPT  UserAdd(newPT, Type.Finish)  End If  End If  End If  End Sub  ' Subroutine used by UserCustomisation that updates the cell then clears the path  Private Sub UserAdd(pt As Point, cellType As Type)  UpdateCell(pt, cellType)  ClearPath()  UpdateGrid()  End Sub  ' Subroutine that generates a maze using the selected maze algorithm  Public Sub Generate()  ' Clears the metrics and drawing queue  Main.ClearAllMetrics()  drawQueue.Clear()  ' Sets the start and finish cells  SetStartFinish()  ' If user customisation is selected clear the grid and set the start and finish point  If selectedMazeAlgorithm = GridCustomisation.UserCustomisation Then  ClearGrid()  UpdateCell(startCell, Type.Start)  UpdateCell(finishCell, Type.Finish)  UpdateGrid()  ' If a maze generation algorithm is selected generate a maze using that algorithm  Else  Dim mazeGeneration As New MazeGeneration(selectedMazeAlgorithm)  Randomize()  mazeGeneration.Generate(grid, drawQueue)  End If  End Sub  ' Subroutine that solves the grid using the selected path finding algorithm  Public Sub FindPath()  Dim pathFinding As New PathFinding(selectedPathAlgorithm, startCell, finishCell)  pathFinding.FindPath(grid, drawQueue)  isCleared = False  End Sub  #End Region  #Region "Grid Drawing"  ' Subroutine that draws a cell on the grid using graphics  Private Sub Draw(newx As Integer, newy As Integer, newColour As Color, Optional newSize As Integer = 1)  Dim gridScale As Integer = Math.Floor(600 / grid.GetSize)  Dim bmGraphics As Graphics = Graphics.FromImage(bmGrid)  bmGraphics.FillRectangle(New SolidBrush(newColour), gridScale \* newx, gridScale \* newy, gridScale \* newSize, gridScale \* newSize)  End Sub  ' Subroutine that dequeues and draws an instruction from the drawing queue  Private Sub DrawNext()  Dim current As Instructions = drawQueue.Dequeue  Draw(current.X, current.Y, current.colour, current.size)  End Sub  ' Subroutine that finishes drawing and empties queue  Public Sub DrawAll()  Do Until drawQueue.Count = 0  DrawNext()  Loop  End Sub  ' Subroutine that draws the grid  Public Sub DrawGrid()  ' Completes all the instructions in the drawing queue if there is no delay  If delay = 0 Then  DrawAll()  UpdateGrid()  ' Starts the timer which displays the grid with a interval delay  Else  tmrDraw.Start()  End If  End Sub  ' Subroutine that is ran every tick of the drawing timer  Private Sub tmrDraw\_Tick() Handles tmrDraw.Tick  ' If the drawing queue is empty stop the timer  If drawQueue.Count = 0 Then  tmrDraw.Stop()  ' If the delay is 0 then draw all instructions  ElseIf delay = 0 Then  DrawAll()  tmrDraw.Stop()  UpdateGrid()  ' Draws drawing amount of drawing instructions  Else  For i = 1 To drawAmount  If drawQueue.Count <> 0 Then  DrawNext()  Else  i = drawAmount  End If  Next  UpdateGrid()  End If  End Sub  ' Subroutine that updates the bitmap that is used to display the grid  Public Sub UpdateGrid()  picGrid.BackgroundImage = bmGrid  picGrid.Refresh()  End Sub  #End Region  End Class |

### Grid

|  |
| --- |
| ' Defines a class to act as the Grid 2d array  Public Class Grid  Private gridSize As Integer  Private arrGrid(,) As Integer  ' Uses the Default property to allow the class to be accessed as an array using either the x and y coordinates or a point  Default Public Property Index(X As Integer, Y As Integer)  Get  Return arrGrid(X, Y)  End Get  Set(value)  arrGrid(X, Y) = value  End Set  End Property  Default Public Property Index(pt As Point)  Get  Return arrGrid(pt.X, pt.Y)  End Get  Set(value)  arrGrid(pt.X, pt.Y) = value  End Set  End Property  ' Subroutine that sets the grid size and redefines the array  Public Sub SetSize(newSize As Integer)  gridSize = newSize  ReDim arrGrid(gridSize - 1, gridSize - 1)  End Sub  ' Function that returns the size of the grid  Public Function GetSize() As Integer  Return gridSize  End Function  ' Subroutine that fills the grid with CellType  Public Sub Fill(cellType As Integer)  For X = 0 To gridSize - 1  For Y = 0 To gridSize - 1  ' If not already CellType then set to CellType  If arrGrid(X, Y) <> cellType Then  arrGrid(X, Y) = cellType  End If  Next  Next  End Sub  End Class |

### MazeGeneration

|  |
| --- |
| ' Class that stores maze generation algorithms  Public Class MazeGeneration  ' Stores properties needed for maze generation  ' ReadOnly means it acts like a constant but it can be assigned in the constructor  ReadOnly selectedMazeAlgorithm As GridCustomisation  ' Constructor that sets the properties needed for maze generation  Public Sub New(newSelectedMazeAlgorithm As GridCustomisation)  selectedMazeAlgorithm = newSelectedMazeAlgorithm  End Sub  ' Subroutine that generates a maze using or the selected maze algorithm  Public Sub Generate(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions))  Select Case selectedMazeAlgorithm  Case GridCustomisation.RecursiveBacktracking  GenerateRecursiveBacktracking(grid, drawQueue)  Case GridCustomisation.RecursiveDivision  GenerateRecursiveDivision(grid, drawQueue)  Case GridCustomisation.Kruskals  GenerateKruskals(grid, drawQueue)  Case GridCustomisation.BinaryTree  GenerateBinaryTree(grid, drawQueue)  Case GridCustomisation.Prims  GeneratePrims(grid, drawQueue)  End Select  ' Calculates the metrics for the generated maze  AverageLength(grid)  Branches(grid)  End Sub  ' Subroutine that generates a maze using the recursive backtracking maze generation algorithm  Private Sub GenerateRecursiveBacktracking(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions))  ' Initialize a variable to store the time taken and a stop watch to record the time  Dim timeTaken As Decimal  Dim sw As New Stopwatch  ' Sets all cells in grid to walls  Fill(Type.Wall, grid, drawQueue)  ' Start the stopwatch  sw.Start()  ' Creates a stack to store the visited cells to backtrack  Dim visitedStack As New Stack(Of Point)  ' Chooses a random Cell to start generating the maze from  Dim currentCell As New Point(RandomNum(0, (grid.GetSize) / 2 - 1) \* 2,  RandomNum(0, (grid.GetSize) / 2 - 1) \* 2)  ' Sets the CurrentCell to empty and pushes it onto stack  SetCell(CurrentCell, Type.Empty, grid, drawQueue)  visitedStack.Push(currentCell)  ' Loop until the stack is empty  Do Until visitedStack.Count = 0  Dim neighbours As New List(Of Point)  ' Define possible movement directions  Dim directions As New List(Of Point) From {New Point(-2, 0), New Point(2, 0),  New Point(0, -2), New Point(0, 2)}  ' Checks all 4 neighbouring cells to CurrentCell  For Each direction In directions  ' Creates a variable to store the NeighbouringCell  Dim neighbouringCell As New Point(currentCell.X + direction.X, currentCell.Y + direction.Y)  ' Checks if the NewCell is in the bounds of the array and is a wall  If VerifyPoint(neighbouringCell, grid.GetSize) Then  If grid(neighbouringCell) = Type.Wall Then  ' Adds NewCell to the list of neighbours  neighbours.Add(neighbouringCell)  End If  End If  Next  ' If there is any neighbours  If neighbours.Count > 0 Then  ' Chooses random neighbour and calculates the CorridorCell  Dim neighbourCell As Point = neighbours(RandomNum(0, neighbours.Count - 1))  Dim corridorCell As New Point((neighbourCell.X + currentCell.X) \ 2, (neighbourCell.Y + currentCell.Y) \ 2)  ' Pushes NeighbourCell to the stack  visitedStack.Push(neighbourCell)  ' Sets the NeighbourCell and CorridorCell to Empty  SetCell(corridorCell, Type.Empty, grid, drawQueue)  SetCell(neighbourCell, Type.Empty, grid, drawQueue)  ' Sets NeighbouringCell as the CurrentCell  currentCell = neighbourCell  Else  ' If the CurrentCell has no neighbours then backtrack  currentCell = visitedStack.Pop  End If  Loop  ' Stop the stopwatch and calculate the TimeTaken  sw.Stop()  timeTaken = Math.Round(sw.ElapsedTicks / Stopwatch.Frequency \* 1000, 3)  ' Updates the controls on the form with the new metric  Main.txtTimeTakenGrid.Text = timeTaken  End Sub  ' Subroutine that generates a maze using the random Kruskals maze generation algorithm  Private Sub GenerateKruskals(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions))  ' Initialize a variable to store the time taken and a stop watch to record the time  Dim timeTaken As Decimal  Dim sw As New Stopwatch  ' Sets all cells in grid to walls  Fill(Type.Wall, grid, drawQueue)  ' Start the stopwatch  sw.Start()  ' Creates list to store all edges in the grid  Dim connections As New List(Of Connection)  ' Create a dictionary to keep track of connected cells  Dim parents As New Dictionary(Of Point, Point)  ' Add all edges to the list  For X = 0 To grid.GetSize - 1 Step 2  For Y = 0 To grid.GetSize - 1 Step 2  ' Right from the cell  If VerifyPoint(X + 2, Y, grid.GetSize) Then  connections.Add(New Connection(X, Y, X + 2, Y))  End If  ' Down from the cell  If VerifyPoint(X, Y + 2, grid.GetSize) Then  connections.Add(New Connection(X, Y, X, Y + 2))  End If  Next  Next  ' Shuffle the edges list  Dim r As New Random  connections = connections.OrderBy(Function(i) r.Next()).ToList  ' Iterate through all the connections in random order  For Each connection In connections  ' Find the parent of each cell in the connection  Dim fromParent As Point = FindParent(connection.From, parents)  Dim targetParent As Point = FindParent(connection.Target, parents)  ' If the two cells have different parents then they belong to different sets so connect them  If fromParent <> targetParent Then  ' Update the parents of the from cell  parents(fromParent) = targetParent  ' If the FromCell isn't empty then set it to empty  If grid(connection.From) <> Type.Empty Then  SetCell(connection.From, Type.Empty, grid, drawQueue)  End If  ' Set the CorridorCell to empty  SetCell(connection.Corridor, Type.Empty, grid, drawQueue)  ' If the TargetCell isn't empty then set it to empty  If grid(connection.Target) <> Type.Empty Then  SetCell(connection.Target, Type.Empty, grid, drawQueue)  End If  End If  Next  ' Stop the stopwatch and calculate the TimeTaken  sw.Stop()  timeTaken = Math.Round(sw.ElapsedTicks / Stopwatch.Frequency \* 1000, 3)  ' Updates the controls on the form with the new metric  Main.txtTimeTakenGrid.Text = timeTaken  End Sub  ' Helper Function for Kruskals to find the parent of a cell  Private Function FindParent(cell As Point, parents As Dictionary(Of Point, Point)) As Point  ' If the cell isn't in the dictionary add it with itself as the value  If Not parents.ContainsKey(cell) Then  parents(cell) = cell  ' If the CurrentCell isn't equal to the ParentCell then run this subroutine recursively until it is  ElseIf parents(cell) <> cell Then  parents(cell) = FindParent(parents(cell), parents)  End If  ' Return the ParentCell  Return parents(cell)  End Function  ' Subroutine that generates a maze using the Binary Tree maze generation algorithm  Private Sub GenerateBinaryTree(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions))  ' Initialize a variable to store the time taken and a stop watch to record the time  Dim timeTaken As Decimal  Dim sw As New Stopwatch  ' Sets all cells in grid to walls  Fill(Type.Wall, grid, drawQueue)  ' Start the stopwatch  sw.Start()  ' Iterate through all cells in the grid  For X = 0 To grid.GetSize - 1 Step 2  For Y = 0 To grid.GetSize - 1 Step 2  ' Choose a random direction (1=down or 2-right)  Dim direction As Integer = RandomNum(1, 2)  ' If the direction is down or on the left border remove the cell down  If (direction = 1 Or X = 0) And VerifyPoint(X, Y - 1, grid.GetSize) Then  SetCell(X, Y - 1, Type.Empty, grid, drawQueue)  End If  ' If the direction is right or on the top border remove the cell to the right  If (direction = 2 Or Y = 0) And VerifyPoint(X - 1, Y, grid.GetSize) Then  SetCell(X - 1, Y, Type.Empty, grid, drawQueue)  End If  ' Set the CurrentCell to Empty  SetCell(X, Y, Type.Empty, grid, drawQueue)  Next  Next  ' Stop the stopwatch and calculate the TimeTaken  sw.Stop()  timeTaken = Math.Round(sw.ElapsedTicks / Stopwatch.Frequency \* 1000, 3)  ' Updates the controls on the form with the new metric  Main.txtTimeTakenGrid.Text = timeTaken  End Sub  ' Subroutine that generates a maze using Prims maze generation algorithm  Private Sub GeneratePrims(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions))  ' Initialize variables to record metrics of the maze generation algorithm  Dim timeTaken As Decimal  Dim sw As New Stopwatch  ' Sets all cells in grid to walls  Fill(Type.Wall, grid, drawQueue)  ' Start the stopwatch  sw.Start()  ' Choose a random starting cell  Dim randomStartingCell As New Point(RandomNum(0, (grid.GetSize) / 2 - 1) \* 2,  RandomNum(0, (grid.GetSize) / 2 - 1) \* 2)  ' Creates a list to store frontier and visited cells  Dim frontierList As New List(Of Connection)  Dim visited As New Hashset(Of Point)  ' Define possible movement directions  Dim directions As New List(Of Point) From {New Point(-2, 0), New Point(2, 0),  New Point(0, -2), New Point(0, 2)}  ' Adds the RandomStartingCell to the list of frontier cells  visited.Add(randomStartingCell)  frontierList.Add(New Connection(randomStartingCell, randomStartingCell))  ' Loop until no FrountierCells left  While frontierList.Count > 0  ' Pick a random cell from the frontier list  Dim currentConnection As Connection = frontierList(RandomNum(0, frontierList.Count - 1))  frontierList.Remove(CurrentConnection)  Dim currentCell As Point = currentConnection.Target  SetCell(currentConnection.Corridor, Type.Empty, grid, drawQueue)  SetCell(currentCell, Type.Empty, grid, drawQueue)  For Each direction In directions  Dim neighbour As New Point(currentCell.X + direction.X, currentCell.Y + direction.Y)  If VerifyPoint(neighbour, grid.GetSize) AndAlso Not visited.Contains(neighbour) Then  visited.Add(neighbour)  frontierList.Add(New Connection(currentCell, neighbour))  End If  Next  End While  ' Stop the stopwatch and calculate the TimeTaken  sw.Stop()  timeTaken = Math.Round(sw.ElapsedTicks / Stopwatch.Frequency \* 1000, 3)  ' Updates the controls on the form with the new metric  Main.txtTimeTakenGrid.Text = timeTaken  End Sub  ' Subroutine that generates a maze using Recursive Division maze generation algorithm  Private Sub GenerateRecursiveDivision(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions))  ' Initialize a variable to store the time taken and a stop watch to record the time  Dim timeTaken As Decimal  Dim sw As New Stopwatch  ' Sets all cells in grid to walls  Fill(Type.Empty, grid, drawQueue)  ' Start the stopwatch  sw.Start()  Dim boundary As Integer = grid.GetSize - 2 + grid.GetSize Mod 2  ' Call the recursive division function to generate the maze  RecursiveDivisionHelper(New Point(0, 0), New Point(boundary, boundary), grid, drawQueue)  ' Stop the stopwatch and calculate the TimeTaken  sw.Stop()  timeTaken = Math.Round(sw.ElapsedTicks / Stopwatch.Frequency \* 1000, 3)  ' Updates the controls on the form with the new metric  Main.txtTimeTakenGrid.Text = timeTaken  End Sub  ' Helper subroutine used for recursive division  Private Sub RecursiveDivisionHelper(tL As Point, bR As Point, grid As Grid, ByRef drawQueue As Queue(Of Instructions))  ' TL is TopLeft, BR is BottomRight  ' Base case: If the bounds are too small, stop dividing  If (bR.X - tL.X) > 1 And (bR.Y - tL.Y) > 1 Then  ' Choose whether to divide horizontally or vertically  Dim divideHorizontally As Boolean  If (bR.X - tL.X) = 0 Then  DivideHorizontally = True  ElseIf (bR.Y - tL.Y) = 0 Then  DivideHorizontally = False  Else  divideHorizontally = RandomNum(0, 1)  End If  ' Choose a random cell for the wall  Dim wallPosition As Integer  If divideHorizontally Then  ' Divide horizontally  wallPosition = (tL.Y + 1) + RandomNum(0, Math.Floor((bR.Y - tL.Y - 1) / 2)) \* 2  ' Create a horizontal wall  For X = tL.X To bR.X  SetCell(X, wallPosition, Type.Wall, grid, drawQueue)  Next  Else  ' Divide vertically  wallPosition = (tL.X + 1) + RandomNum(0, Math.Floor((bR.X - tL.X - 1) / 2)) \* 2  ' Create a vertical wall  For Y = tL.Y To bR.Y  SetCell(wallPosition, Y, Type.Wall, grid, drawQueue)  Next  End If  ' Create openings in the wall  Dim openingPosition As Integer  If DivideHorizontally Then  openingPosition = tL.X + RandomNum(0, Math.Floor((bR.X - tL.X) / 2)) \* 2  ' Create an opening in the wall horizontally  SetCell(openingPosition, wallPosition, Type.Empty, grid, drawQueue)  Else  openingPosition = tL.Y + RandomNum(0, Math.Floor((bR.Y - tL.Y) / 2)) \* 2  ' Create an opening in the wall vertically  SetCell(wallPosition, openingPosition, Type.Empty, grid, drawQueue)  End If  ' Recursive calls to divide the sub-regions  If divideHorizontally Then  RecursiveDivisionHelper(tL, New Point(bR.X, wallPosition - 1), grid, drawQueue)  RecursiveDivisionHelper(New Point(tL.X, wallPosition + 1), bR, grid, drawQueue)  Else  RecursiveDivisionHelper(tL, New Point(wallPosition - 1, bR.Y), grid, drawQueue)  RecursiveDivisionHelper(New Point(wallPosition + 1, tL.Y), bR, grid, drawQueue)  End If  End If  End Sub  ' Subroutine that works out the average path length in a maze  Private Sub AverageLength(ByRef grid As Grid)  ' Variable to store current length whilst calculating  Dim currentLength As Integer  ' Dictionary to store all the path lengths  Dim lengths As New Dictionary(Of Integer, Integer)  ' Go through all horizontals and add their lengths to the dictionary  For Y = 0 To grid.GetSize - 1 Step 2  For X = 0 To grid.GetSize - 1  ' If the cell is empty increase the current length  If grid(X, Y) = Type.Empty Then  currentLength += 1  End If  ' If its the final cell or wall of a path then add the length to the dictionary  If grid(X, Y) = Type.Wall Or X = grid.GetSize - 1 Then  If currentLength > 1 Then  If lengths.ContainsKey(currentLength) Then  lengths(currentLength) += 1  Else  lengths.Add(currentLength, 1)  End If  End If  currentLength = 0  End If  Next  currentLength = 0  Next  ' Go through all verticals and add their lengths to the dictionary  For X = 0 To grid.GetSize - 1 Step 2  For Y = 0 To grid.GetSize - 1  ' If the cell is empty increase the current length  If grid(X, Y) = Type.Empty Then  currentLength += 1  End If  ' If its the final cell or wall of a path then add the length to the dictionary  If grid(X, Y) = Type.Wall Or Y = grid.GetSize - 1 Then  If currentLength > 1 Then  If lengths.ContainsKey(currentLength) Then  lengths(currentLength) += 1  Else  lengths.Add(currentLength, 1)  End If  End If  currentLength = 0  End If  Next  currentLength = 0  Next  ' Creates variables to store the total path length and number of paths  Dim totalLength As Integer  Dim totalPaths As Integer  Dim averageLength As Decimal  ' Goes through each length in the dictionary and works out total length and number of paths  For Each length In lengths  totalLength += length.Key \* length.Value  totalPaths += length.Value  Next  ' Work out the average length  averageLength = Math.Round(totalLength / totalPaths \* 100) / 100  ' Updates the controls on the form with the new metric  Main.txtAverageLength.Text = averageLength  End Sub  ' Works out the branch and dead-end percentage in a maze  Private Sub Branches(ByRef grid As Grid)  ' Creates a list of potential branch directions  Dim directions As New List(Of Point) From {New Point(0, -1), New Point(1, 0), New Point(0, 1), New Point(-1, 0)}  ' Creates a dictionary to store the number of branches at each point  Dim branches As New Dictionary(Of Integer, Integer)  ' Iterate through each cell to see how many branches it has  For X = 0 To grid.GetSize - 1 Step 2  For Y = 0 To grid.GetSize - 1 Step 2  Dim branchNumber As Integer = 0  ' Check all 4 directions and if its open increase the branch number  For Each direction In directions  Dim newPT As New Point(X + direction.X, Y + direction.Y)  ' Adds cell as a branch if it is empty  If VerifyPoint(newPT, grid.GetSize) AndAlso grid(newPT) = Type.Empty Then  branchNumber += 1  End If  Next  ' Add the branch number to the dictionary or increment the number  If branches.ContainsKey(branchNumber) Then  branches(branchNumber) += 1  Else  branches.Add(branchNumber, 1)  End If  Next  Next  ' Creates variables to store the total branches and branch numbers  Dim totalBranches As Integer  Dim totalBranchNumber As Integer  Dim differentBranches As Integer  Dim percentageOfDeadends As Decimal  Dim percentageOfBranches As Decimal  ' Works out the total number of branches and number of cells with branches  For Each branch In branches  totalBranchNumber += branch.Key \* branch.Value  totalBranches += branch.Value  Next  ' Work out percentage of dead ends which is cells with 1 branch  percentageOfDeadends = Math.Round(branches(1) / totalBranches \* 1000) / 10  ' Works out percentage of cells which branch out which means they have 3 or 4 branches from a cell  If branches.ContainsKey(3) Then differentBranches += branches(3)  If branches.ContainsKey(4) Then differentBranches += branches(4)  percentageOfBranches = Math.Round(differentBranches / totalBranches \* 1000) / 10  ' Updates the controls on the form with the new metrics  Main.txtDeadends.Text = percentageOfDeadends & "%"  Main.txtBranches.Text = percentageOfBranches & "%"  End Sub  End Class |

### PathFinding

|  |
| --- |
| ' Class that stores path finding algorithms  Public Class PathFinding  ' Stores properties needed for path finding  ' ReadOnly means it acts like a constant but it can be assigned in the constructor  ReadOnly selectedPathAlgorithm As PathfindingAlgorithm  ReadOnly startCell As Point  ReadOnly finishCell As Point  ' Constructor that sets the properties needed for maze generation  Public Sub New(newSelectedPathAlgorithm As PathfindingAlgorithm, newStartCell As Point, newFinishCell As Point)  selectedPathAlgorithm = newSelectedPathAlgorithm  startCell = newStartCell  finishCell = newFinishCell  End Sub  ' Subroutine that solves a grid using the selected path finding algorithm  Public Sub FindPath(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions))  ' Sets the start and finish cell  SetCell(startCell, Type.Start, grid, drawQueue)  SetCell(finishCell, Type.Finish, grid, drawQueue)  Select Case selectedPathAlgorithm  Case PathfindingAlgorithm.BreadthFirst  FindPathBFS(grid, drawQueue)  Case PathfindingAlgorithm.AStar  FindPathAStar(grid, drawQueue)  Case PathfindingAlgorithm.DepthFirst  FindPathDFS(grid, drawQueue)  Case PathfindingAlgorithm.GreedyDFS  FindPathGreedyDFS(grid, drawQueue)  End Select  End Sub  ' Subroutine that solves the grid using the Breath-First Search algorithm  Public Sub FindPathBFS(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions))  ' Initialises variables to record metrics of the path finding algorithm  Dim shortestPath As Integer = 1  Dim cellsExplored As Integer = 1  Dim timeTaken As Decimal  Dim sw As New Stopwatch  ' Starts the stop watch  sw.Start()  ' Create a queue to keep track of cells for BFS  Dim openQueue As New Queue(Of Point)  ' Creates a dictionary to keep track of the parent cells for each cell explored  Dim returnCells As New Dictionary(Of Point, Point)  ' Store the current return cell to reconstruct the path to the start cell  Dim returnCell As Point  ' Keeps a boolean that shows whether the FinishCell has been found or not  Dim pathFound As Boolean  ' Enqueue the start cell to the OpenQueue  openQueue.Enqueue(startCell)  ' Loop until the list is empty or a path has been found from StartCell to FinishCell  Do Until openQueue.Count = 0 Or pathFound = True  ' Dequeues from OpenQueue and sets it to CurrentCell  Dim currentCell As Point = openQueue.Dequeue()  ' Updates coordinates of the ReturnCell  returnCell = currentCell  ' Sets the CurrentCell to closed if it isn't the start or finish cell  If grid(currentCell) < Type.Start Then  SetCell(currentCell, Type.Closed, grid, drawQueue)  cellsExplored += 1  End If  ' Define a list of possible movement directions  Dim directions As New List(Of Point) From {New Point(0, -1), New Point(1, 0), New Point(0, 1), New Point(-1, 0)}  ' Checks all 4 neighbouring cells to CurrentCell  For Each direction In directions  CheckNeighboursBFS(grid, drawQueue, currentCell, returnCells, openQueue, pathFound, direction)  Next  Loop  ' Reconstructs the path if the FinishCell is found  If pathFound Then  ' Loops until StartCell is found  Do Until returnCell = startCell  shortestPath += 1  SetCell(returnCell, Type.Path, grid, drawQueue)  returnCell = returnCells(returnCell)  Loop  ' Stops the stopwatch and calculates the TimeTaken  sw.Stop()  timeTaken = Math.Round(sw.ElapsedTicks / Stopwatch.Frequency \* 1000, 3)  ' Updates all the controls on the form with recorded metrics  UpdateMetrics(shortestPath, cellsExplored, timeTaken)  End If  End Sub  ' Helper subroutine that checks the neighbouring cells for BFS  Private Sub CheckNeighboursBFS(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions), currentCell As Point, ByRef returnCells As Dictionary(Of Point, Point),  ByRef openQueue As Queue(Of Point), ByRef pathFound As Boolean, direction As Point)  ' Initialises a variable to store coordinates of the neighbour cell  Dim neighbour As New Point(currentCell.X + direction.X, currentCell.Y + direction.Y)  ' Checks if the neighbour is within the bounds of the array  If VerifyPoint(neighbour, grid.GetSize) Then  ' If the neighbour is empty enqueues it and sets it to open  If grid(neighbour) = Type.Empty Then  openQueue.Enqueue(neighbour)  SetCell(neighbour, Type.Open, grid, drawQueue)  returnCells(neighbour) = currentCell  ' If the neighbour is the FinishCell the path is found  ElseIf grid(neighbour) = Type.Finish Then  pathFound = True  End If  End If  End Sub  ' Subroutine that solves the grid using the Depth-First Search algorithm  Private Sub FindPathDFS(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions))  ' Initialise variables to record metrics of the path finding algorithm  Dim shortestPath As Integer = 1  Dim cellsExplored As Integer = 1  Dim timeTaken As Decimal  Dim sw As New Stopwatch  ' Start the stopwatch  sw.Start()  ' Create a stack to keep track of cells for DFS  Dim openStack As New Stack(Of Point)  ' Create a dictionary to keep track of the parent cell for each cell explored  Dim returnCells As New Dictionary(Of Point, Point)  ' Store the current ReturnCell to reconstruct the path to the StartCell  Dim returnCell As Point  ' Keep a boolean that shows whether the FinishCell has been found or not  Dim pathFound As Boolean  ' Push the StartCell to the OpenStack  openStack.Push(startCell)  ' Loop until the stack is empty or a path has been found from StartCell to FinishCell  Do Until openStack.Count = 0 Or pathFound = True  ' Pop a cell from the stack  Dim currentCell As Point = openStack.Pop()  ' Update coordinates of the ReturnCell  returnCell = currentCell  ' Set the CurrentCell to closed if it isn't the StartCell  If grid(currentCell) < Type.Start Then  cellsExplored += 1  SetCell(currentCell, Type.Closed, grid, drawQueue)  End If  ' Define possible movement directions  Dim directions As New List(Of Point) From {New Point(0, -1), New Point(-1, 0), New Point(0, 1), New Point(1, 0)}  ' Checks all 4 neighbouring cells to CurrentCell  For Each direction In directions  CheckNeighboursDFS(grid, drawQueue, currentCell, returnCells, openStack, pathFound, direction)  Next  Loop  ' Reconstruct the path if the FinishCell is found  If pathFound Then  ' Loop until StartCell is found  Do Until returnCell = startCell  shortestPath += 1  SetCell(returnCell, Type.Path, grid, drawQueue)  returnCell = returnCells(returnCell)  Loop  ' Stops the stopwatch and calculates the TimeTaken  sw.Stop()  timeTaken = Math.Round(sw.ElapsedTicks / Stopwatch.Frequency \* 1000, 3)  ' Updates all the controls on the form with the new metrics  UpdateMetrics(shortestPath, cellsExplored, timeTaken)  End If  End Sub  ' Helper subroutine that checks the neighbouring cells for DFS  Private Sub CheckNeighboursDFS(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions), currentCell As Point, ByRef returnCells As Dictionary(Of Point, Point),  ByRef openStack As Stack(Of Point), ByRef pathFound As Boolean, direction As Point)  ' Initialises a variable to store coordinates of the neighbour cell  Dim neighbour As New Point(currentCell.X + direction.X, currentCell.Y + direction.Y)  ' Checks if the neighbour is within the bounds of the array  If VerifyPoint(neighbour, grid.GetSize) Then  ' If the neighbour is empty set it to open  If grid(neighbour) = Type.Empty Then  openStack.Push(neighbour)  SetCell(neighbour, Type.Open, grid, drawQueue)  returnCells(neighbour) = currentCell  ' If the neighbour is the FinishCell the path is found  ElseIf grid(neighbour) = Type.Finish Then  pathFound = True  End If  End If  End Sub  ' Subroutine that solves the grid using the A\* search algorithm  Private Sub FindPathAStar(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions))  ' Initialise variables to record metrics of the path finding algorithm  Dim shortestPath As Integer = 1  Dim cellsExplored As Integer = 1  Dim timeTaken As Decimal  Dim sw As New Stopwatch  ' Start the stopwatch  sw.Start()  ' Create a priority queue to keep track of cells to be searched  Dim openQueue As New PriorityQueue(Of Point)  ' Create a dictionary to store all of the cell properties  Dim cellProperties As New Dictionary(Of Point, CellProperties)  ' Store the current ReturnCell to reconstruct the path to the startCell  Dim returnCell As Point  ' Keep a boolean that shows whether the FinishCell has been found or not  Dim pathFound As Boolean  ' Add the StartCell to the queue  openQueue.Enqueue(startCell, 0, 0)  cellProperties(startCell) = New CellProperties  ' Loop until the list is empty or a path has been found from StartCell to FinishCell  Do Until openQueue.IsEmpty Or pathFound = True  ' Get the first item in the queue as CurrentCell  Dim currentCell As Point = openQueue.Dequeue()  ' If the CurrentCell isn't closed  If grid(currentCell) <> Type.Closed Then  ' Update ReturnCell, acting as the starting cell to reconstruct  returnCell = currentCell  ' Set the CurrentCell to closed if it isn't the StartCell  If grid(currentCell) < Type.Start Then  SetCell(currentCell, Type.Closed, grid, drawQueue)  cellsExplored += 1  End If  ' Define possible movement directions  Dim directions As New List(Of Point) From {New Point(0, -1), New Point(1, 0), New Point(0, 1), New Point(-1, 0)}  ' Checks all 4 neighbouring cells to CurrentCell  For Each direction In directions  CheckNeighboursAstar(grid, drawQueue, currentCell, cellProperties, openQueue, pathFound, direction)  Next  End If  Loop  ' Reconstruct the path if FinishCell is found  If pathFound Then  ' Loop until StartCell is found  Do Until returnCell = startCell  shortestPath += 1  SetCell(returnCell, Type.Path, grid, drawQueue)  returnCell = cellProperties(returnCell).ReturnCell  Loop  ' Stops the stopwatch and calculates the TimeTaken  sw.Stop()  timeTaken = Math.Round(sw.ElapsedTicks / Stopwatch.Frequency \* 1000, 3)  ' Updates all the controls on the form with new metrics  UpdateMetrics(shortestPath, cellsExplored, timeTaken)  End If  End Sub  ' Helper subroutine that checks the neighbouring cells for A\* search algorithm  Private Sub CheckNeighboursAstar(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions), currentCell As Point, direction As Point,  ByRef cellProperties As Dictionary(Of Point, CellProperties), ByRef openlist As PriorityQueue(Of Point), ByRef pathFound As Boolean)  ' Initialises variables to store coordinates and properties of new neighbour cell  Dim neighbour As New Point(currentCell.X + direction.X, currentCell.Y + direction.Y)  ' Checks if the x and y coordinates are within the bounds of the array  If VerifyPoint(neighbour, grid.GetSize) Then  If {Type.Empty, Type.Open, Type.Closed}.Contains(grid(neighbour)) Then  ' Calculate all the properties and set the ReturnCell of the NewCell  Dim newProperties As New CellProperties(neighbour, currentCell, finishCell, cellProperties)  ' If the NewCell is empty then update the properties, enqueue the cell and set it to open  If grid(neighbour) = Type.Empty Then  cellProperties(neighbour) = newProperties  openlist.Enqueue(neighbour, newProperties.fCost, newProperties.hCost)  SetCell(neighbour, Type.Open, grid, drawQueue)  ' If the cell isn't empty and has a lower FCost then update the properties and enqueue the cell  ElseIf cellProperties.ContainsKey(neighbour) AndAlso newProperties.fCost < cellProperties(neighbour).fCost Then  cellProperties(neighbour) = newProperties  openlist.Enqueue(neighbour, newProperties.fCost, newProperties.hCost)  End If  ' If the neighbour is the FinishCell the path is found  ElseIf grid(neighbour) = Type.Finish Then  pathFound = True  End If  End If  End Sub  ' Subroutine that solves the grid using the Greedy Best-First Search algorithm  Private Sub FindPathGreedyDFS(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions))  ' Initialise variables to record metrics of the path finding algorithm  Dim shortestPath As Integer = 1  Dim cellsExplored As Integer = 1  Dim timeTaken As Decimal  Dim sw As New Stopwatch  ' Start the stopwatch  sw.Start()  ' Use a priority queue to keep track of cells to be searched  Dim openQueue As New PriorityQueue(Of Point)  ' Create a dictionary to store all of the cells properties  Dim cellProperties As New Dictionary(Of Point, CellProperties)  ' Store the current ReturnCell to reconstruct the path to the StartCell  Dim returnCell As Point  ' Keep a boolean that shows whether the FinishCell has been found or not  Dim pathFound As Boolean  ' Add the StartCell to the queue  cellProperties(startCell) = New CellProperties  OpenQueue.Enqueue(startCell, 0, 0)  ' Loop until the list is empty or a path has been found from StartCell to FinishCell  Do Until openQueue.IsEmpty Or pathFound = True  ' Set the CurrentCell to the first item in the queue  Dim currentCell As Point = openQueue.Dequeue()  If grid(currentCell) <> Type.Closed Then  ' Update the coordinates of the ReturnCell  returnCell = currentCell  ' Set the CurrentCell to closed if it isn't the StartCell  If grid(currentCell) < Type.Start Then  SetCell(currentCell, Type.Closed, grid, drawQueue)  cellsExplored += 1  End If  ' Define possible movement directions  Dim directions As New List(Of Point) From {New Point(0, -1), New Point(1, 0), New Point(0, 1), New Point(-1, 0)}  ' Checks all 4 neighbouring cells to CurrentCell  For Each direction In directions  CheckNeighboursGreedyDFS(grid, drawQueue, currentCell, cellProperties, openQueue, pathFound, direction)  Next  End If  Loop  ' Reconstruct the path if FinishCell is found  If pathFound Then  ' Loop until StartCell is found  Do Until returnCell = startCell  shortestPath += 1  SetCell(returnCell, Type.Path, grid, drawQueue)  returnCell = cellProperties(returnCell).ReturnCell  Loop  ' Stops the stopwatch and calculates the TimeTaken  sw.Stop()  timeTaken = Math.Round(sw.ElapsedTicks / Stopwatch.Frequency \* 1000, 3)  ' Updates all the controls on the form with new metrics  UpdateMetrics(shortestPath, cellsExplored, timeTaken)  End If  End Sub  ' Helper subroutine that checks the neighbouring cells for Greedy Best-First Search algorithm  Private Sub CheckNeighboursGreedyDFS(ByRef grid As Grid, ByRef drawQueue As Queue(Of Instructions), currentCell As Point, direction As Point,  ByRef cellProperties As Dictionary(Of Point, CellProperties), ByRef openlist As PriorityQueue(Of Point), pathFound As Boolean)  ' Initialises variables to store coordinates and properties of the new neighbour cell  Dim neighbour As New Point(currentCell.X + direction.X, currentCell.Y + direction.Y)  ' Checks if the x and y coordinates are within the bounds of the array  If VerifyPoint(neighbour, grid.GetSize) Then  ' If it's empty set it to open  If grid(neighbour) = Type.Empty Then  ' Calculate all the properties and set the ReturnCell of the NewCell  Dim newProperties As New CellProperties(neighbour, currentCell, finishCell, cellProperties)  ' Add the NewCells properties to the CellProperties dictionary  cellProperties(neighbour) = newProperties  ' Enqueue the new cell in the priority queue using the H-cost as the priority  openlist.Enqueue(neighbour, newProperties.hCost)  ' Set the NewCell to Open  SetCell(neighbour, Type.Open, grid, drawQueue)  ' If the neighbour is the FinishCell the path is found  ElseIf grid(neighbour) = Type.Finish Then  pathFound = True  End If  End If  End Sub  ' Subroutine updates the controls on the form with the new metrics  Private Sub UpdateMetrics(shortestPath As Integer, cellsExplored As Integer, timeTaken As Decimal)  Main.txtShortestPath.Text = shortestPath  Main.txtCellsExplored.Text = cellsExplored  Main.txtTimeTakenPath.Text = timeTaken  End Sub  End Class |

### Structures

|  |
| --- |
| ' Structure used in Kruskals and Prims maze generation algorithm to store connections between two cells  Public Structure Connection  ' Creates 3 variables to store the from, target and corridor cells  ReadOnly from As Point  ReadOnly target As Point  Dim corridor As Point  ' Constructor that takes the points or coordinates of two cells and calculates the corridor cell  Public Sub New(newFromX As Integer, newFromY As Integer, newTargetX As Integer, newTargetY As Integer)  from = New Point(newFromX, newFromY)  target = New Point(newTargetX, newTargetY)  corridor = GetCorridor()  End Sub  Public Sub New(newFrom As Point, newTarget As Point)  From = newFrom  Target = newTarget  Corridor = GetCorridor()  End Sub  ' Function that returns the corridor cell calculated using the FromCell and TargetCell  Private Function GetCorridor()  Dim newCorridor As New Point((From.X + Target.X) / 2, (From.Y + Target.Y) / 2)  Return newCorridor  End Function  End Structure  ' Structure used to represent instructions for drawing on the grid  Public Structure Instructions  ' Creates 4 variables to store the information needed to draw a cell  Dim X As Integer  Dim Y As Integer  Dim colour As Color  Dim size As Integer  ' Constructor for creating an instruction with specified coordinates, colour, and size  ' Size is optional so it can be set to grid size and be used to fill the entire grid  Sub New(newX As Integer, newY As Integer, newColour As Color, Optional newSize As Integer = 1)  X = newX  Y = newY  colour = newColour  size = newSize  End Sub  End Structure  ' Structure used in A\* Search and GreedyDFS which stores heuristics used in path finding  Public Structure CellProperties  ' Store the CurrentCell's ParentCell  Dim returnCell As Point  ' Store the H-Cost an estimation of the distance to the FinishCell  Dim hCost As Integer  ' Store the G-Cost the total distance to reach the CurrentCell from the StartCell  Dim gCost As Integer  ' Store the F-cost the sum of H-cost and G-cost  Dim fCost As Integer  ' Subroutine that calculates the sets the return cell and calculated the costs  Sub New(currentCell As Point, parentCell As Point, finishCell As Point, cellProperties As Dictionary(Of Point, CellProperties))  returnCell = parentCell  hCost = Math.Abs(finishCell.X - currentCell.X) + Math.Abs(finishCell.Y - currentCell.Y)  gCost = cellProperties(returnCell).gCost + 1  fCost = hCost + gCost  End Sub  End Structure |

### Methods

|  |
| --- |
| ' Module that stores methods that are used across the solution  Module Methods  ' Function that returns the colour of a CellType  Public Function GetColour(cellType As Type)  ' Defines the colours for each Type  Dim Empty As Color = Color.FromArgb(255, 255, 255, 255)  Dim Wall As Color = Color.FromArgb(255, 90, 90, 90)  Dim Open As Color = Color.FromArgb(255, 0, 220, 170)  Dim Closed As Color = Color.FromArgb(255, 0, 220, 220)  Dim Path As Color = Color.FromArgb(255, 0, 170, 220)  Dim Start As Color = Color.FromArgb(255, 0, 220, 0)  Dim Finish As Color = Color.FromArgb(255, 220, 0, 0)  ' Creates an array of the Colours  Dim colours() As Color = {Empty, Wall, Open, Closed, Path, Start, Finish}  ' Returns the colour of CellType  Return colours(cellType)  End Function  ' Function that verifies the X and Y coordinates or a point are within the bounds of the array  Public Function VerifyPoint(X As Integer, Y As Integer, gridSize As Integer) As Boolean  If X >= 0 And X < gridSize And Y >= 0 And Y < gridSize Then  Return True  Else Return False  End If  End Function  Public Function VerifyPoint(pt As Point, gridSize As Integer) As Boolean  Return VerifyPoint(pt.X, pt.Y, gridSize)  End Function  ' Function that generates a random number between an upper and lower bound  Public Function RandomNum(lBound As Integer, uBound As Integer) As Integer  Return Math.Floor(((uBound - lBound + 1) \* Rnd()) + lBound)  End Function  ' Subroutine that sets a Cell to type in the array and enqueues the instruction to draw this change  Public Sub SetCell(X As Integer, Y As Integer, cellType As Type, grid As Grid, ByRef drawQueue As Queue(Of Instructions))  grid(X, Y) = cellType  DrawEnqueue(drawQueue, X, Y, GetColour(cellType))  End Sub  Public Sub SetCell(pt As Point, cellType As Type, grid As Grid, ByRef drawQueue As Queue(Of Instructions))  SetCell(pt.X, pt.Y, cellType, grid, drawQueue)  End Sub  ' Subroutine that enqueues a new drawing instruction using the parameters  Public Sub DrawEnqueue(drawQueue As Queue(Of Instructions), X As Integer, Y As Integer, newColour As Color, Optional newSize As Integer = 1)  drawQueue.Enqueue(New Instructions(X, Y, newColour, newSize))  End Sub  ' Fills the grid with one type  Public Sub Fill(cellType As Type, grid As Grid, ByRef drawQueue As Queue(Of Instructions))  ' Sets all cells in the array to CellType  grid.Fill(cellType)  ' Makes the entire grid CellType  DrawEnqueue(drawQueue, 0, 0, GetColour(cellType), grid.GetSize)  End Sub  End Module |

### Enumerals

|  |
| --- |
| ' Define an Enum to represent the different cells in the grid  Public Enum Type  Empty  Wall  Open  Closed  Path  Start  Finish  End Enum  ' Define an Enum to represent the different path finding algorithms  Public Enum PathfindingAlgorithm  BreadthFirst  AStar  DepthFirst  GreedyDFS  End Enum  ' Define an Enum to represent the different grid customisation options  Public Enum GridCustomisation  RecursiveBacktracking  RecursiveDivision  Kruskals  BinaryTree  Prims  UserCustomisation  End Enum |

### Priority Queue

|  |
| --- |
| ' Class that defines a priority queue data structure that can store any data type as t and has 2 priorities  Public Class PriorityQueue(Of t)  ' Structure used to store an item and its priority  Structure Data  Dim item As t  Dim priority1 As Integer  Dim priority2 As Integer  Sub New(newItem As t, newPriority1 As Integer, newPriority2 As Integer)  item = newItem  priority1 = newPriority1  priority2 = newPriority2  End Sub  End Structure  ' List that stores the data items in the priority queue  Private values As New List(Of Data)  ' Function that returns the number of items in the priority queue  Public Function Count()  Return values.Count  End Function  ' Function that returns True if the priority queue is empty and False otherwise  Public Function IsEmpty()  If values.Count = 0 Then  Return True  Else Return False  End If  End Function  ' Subroutine that clears all the items in the priority queue  Public Sub Clear()  values.Clear()  End Sub  ' Subroutine that adds a new item to the priority queue, with the specified two priorities  Public Sub Enqueue(newItem As t, newPriority1 As Integer, Optional newPriority2 As Integer = 0)  ' Create a new data item with the specified item value and priorities  Dim newData As New Data(newItem, newPriority1, newPriority2)  ' Find the index in the list where the new item should be inserted, based on its priorities  Dim index As Integer  For i = 0 To values.Count - 1  ' Compare the new item's priorities with the priorities of the existing items in the list  ' If the new item has higher or equal priorities, move to the next item in the list  If newData.priority1 > values(i).priority1 Or (newData.priority1 = values(i).priority1 And newData.priority2 > values(i).priority2) Then  index += 1  Else Exit For  End If  Next  ' Insert the new item into the list at the calculated index  values.Insert(index, newData)  End Sub  ' Function that removes and returns the item with the highest priorities in the priority queue  Public Function Dequeue()  ' Get the data item at the front of the list (with the highest priorities)  Dim dequeueData As Data = values(0)  ' Remove the data item from the list  values.RemoveAt(0)  ' Return the item value  Return dequeueData.item  End Function  End Class |

TODO: Add textboxes over technique table?

## Techniques used

### Table A

|  |  |  |
| --- | --- | --- |
| Algorithm | Description | Page Reference |
| Recursive Backtracking | Stack, List, Graph Traversal, Complex user-defined algorithm | Pg. 51 |
| Kruskals | Lists, Dictionary, Complex user-defined algorithm | Pg. 52 |
| Binary tree | Complex user-defined algorithm | Pg. 53 |
| Prims | List, Graph traversal, Complex user-defined algorithm | Pg. 54 |
| Recursive Division | List, Complex user-defined algorithm | Pg. 55 |
| Breath First Search | Queue, Graph traversal | Pg. 59 |
| Depth First Search | Stack, Graph traversal | Pg. 60 |
| A\* Search | Priority Queue, Dictionary, List, Graph traversal, Complex user-defined algorithm | Pg. 62 |
| Greedy Depth First Search | Priority Queue, Dictionary, List, Graph traversal, Complex user-defined algorithm | Pg. 63 |

### Table B

|  |  |  |
| --- | --- | --- |
| Algorithm | Description | Page Reference |
| Writing and reading from files | Download grid as bitmap to selected file path | Pg. 45 |

## Interface Screenshots

|  |
| --- |
| Screenshot of blank grid with user customisation selected: |
|  |
| Example of custom draw grid with start and end points moved, solved: |
|  |
| Examples of differently sized mazes generated and solved: |
|  |
| Examples of half generated and half solved mazes |
|  |
| Download form: |
|  |
| Close up of how the metrics look: |
|  |

**Testing**

## Tests

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Description | Expected | Pass |
| 1.1 | Grid Representation: Create a grid structure using a 2D array to store cell data. Allow access using points or x and y coordinates. | To validate the functionality of the grid structure, a test scenario was executed. A 3x3 grid was instantiated, and its size was set to 3. Subsequently, values were inserted using both direct coordinates and a Point object. The test successfully demonstrated the correct functioning of the grid:  Dim grid As New Grid  grid.SetSize(3)  grid(1, 0) = 2  grid(0, 1) = 4  Dim pt As New Point(2, 2)  grid(pt) = 8  Debug.Print("Size: " & grid.GetSize)    The output, indicates that the grid accurately returns its size. This test affirms the creation of a 2D array of size 3x3, successful population of the grid with the value 1, and the correct assignment of values 2, 4, and 8 at specified coordinates. |  |
| 1.2 | Cell Types Enum: Define an enum to represent cell types - Empty, Wall, Open, Closed, Path, Start, Finish. | To assess the effectiveness of the enums I've created, I've generated a 3x3 grid. I've then assigned values to their respective enums within the grid and will now verify if the grid accurately reflects the corresponding integers assigned to each enum. This test ensures that the enum values are properly set and appropriately represented in the designated positions within the grid. As you can see below this test was successful.  Dim grid As New Grid  grid.SetSize(3)  grid(0, 0) = Type.Start  grid(0, 1) = Type.Wall  grid(1, 1) = Type.Wall  grid(0, 2) = Type.Finish |  |
| 1.3 | Clear Pathfinding Cells: Provide a button to remove pathfinding-related cells (Open, Closed, Path) from the grid. | In the following demonstration, I've established a grid, and generated the shortest path, and will now illustrate the functionality of the "Clear" button. This button is designed to selectively remove pathfinding-related cells (Open, Closed, Path) while preserving the walls and the start/finish points on the grid. This showcases shows how it effectively does this leaving only the walls and the start/finish points. |  |
| 1.4 | Resizable Grid: Allow users to resize the grid using controls and a slider. Set max size to 1200, min size to 5. Use non-linear, scaling steps for grid size selection. | To assess the Resizable Grid objective, I used the numeric up-down box and slider to resize the grid dynamically, showcasing sizes at both the minimum (5) and maximum (1200) values. Meeting the objective's non-linear scaling requirement, I set the grid size to 200, emphasizing its alignment with the 80% mark on the slider, despite  being one-sixth of the total range.  Moreover, while the slider operates in increments of 600, users can easily input custom numeric values within the specified range. As demonstrated below, I entered 89 in the numeric up-down box.  The numeric up-down box handles the validation. It restricts the entry of non-numeric characters, rounds decimal numbers to the nearest whole number, enforces a minimum value of 5 for numbers below the threshold, caps numbers exceeding 1200 to 1200, and handles invalid inputs like "67..4..3224." by reverting to the previous valid value. |  |
| 2.1 | Display Grid - Visual Representation: Use unique colours to represent different cell types - Empty (white), Wall (black), Open (light blue), Closed (blue), Path (dark blue), Start (green), Finish (red). Create a method to return the correct colour from the cell type. | In the demonstration below, I'm showcasing the retrieval of the correct RGB colour using the GetColour method for the cell types Finish and Closed:   |  |  | | --- | --- | | Dim type As Type = Type.Finish  Dim colour As Color = GetColour(type) | Dim type As Type = Type.Closed  Dim colour As Color = GetColour(type) |   This illustrates how the method successfully returns the corresponding colours for the Finish (red) and Closed (blue) cell types. |  |
| 2.2, 2.3 | Bitmap Usage in the User-Defined Control: Create a user-defined control to handle displaying the grid. Embed a Picture Box in the user-defined control to display the bitmap. Draw cells on the bitmap using the Graphics class. | Here, I am illustrating the usage of a user-defined control to handle grid display. The control embeds a PictureBox for bitmap presentation and utilizes the Graphics class to draw cells. In the examples below, you can see the instruction and how it is correctly carried out on the bitmap:  Dim bmGraphics As Graphics = Graphics.FromImage(bmGrid)  bmGraphics.FillRectangle(New SolidBrush(colour), x, y, size, size)   |  |  |  | | --- | --- | --- | |  |  |  | |  |
| 2.4 | Adaptive Grid Display: Scale cell sizes on the bitmap for high-quality display. Use the Picture Box zoom function to compress the bitmap when the grid is larger than the display. | In this test, I varied the grid sizes to observe how the display box utilizes the Picture Box for presenting generated mazes. For standard maze sizes, the display box functions correctly, ensuring a proper representation:  However, two issues emerged during testing:   * 1. Scaling for Larger Grids: For grid sizes surpassing 600, the display fails to scale appropriately, preventing the entire maze from being displayed.      * 1. Grid Scale Calculation Issue: There's a problem in calculating the grid scale, resulting in incomplete displays for sizes that are not factors of 600, cutting off the right and bottom sides.   s  These issues indicate a failure to fully scale and display the entire generated maze in either example, thereby not meeting the objective. |  |
| 3.1 | Visualization - User-Set Delay: Enable users to set delay using controls and a slider. Restrict max delay to 200 and min to 0. | To evaluate the Delay objective, I employed the numeric up-down box and slider to adjust the delay dynamically, demonstrating functionality at both the minimum (0) and maximum (200) values.    The numeric up-down box effectively handles validation, preventing non-numeric entries and rounding decimal numbers to the nearest whole number. It enforces a minimum value of 0, caps numbers at 200, and manages invalid inputs like ".868….6.6." by reverting to the previous valid value. |  |
| 3.2, 3.3 | Instructions structure, timer and queue for drawing maze: Define a custom data structure for drawing instructions - x coordinate, y locations, colour, size.  Timer and Queue: Use a queue to store drawing instructions. Implement a timer to execute drawing instructions for algorithm visualization. | To evaluate this functionality, I'll initiate the generation of a 10x10 grid using recursive backtracking.    Above is the draw queue, currently comprising 50 instructions. The first three instructions have been shown and will be demonstrated below   |  |  |  | | --- | --- | --- | | Here you can see the first instruction sets every cell to a wall. | Now the starting cell is enqueued with a position of (8, 6). | The coordinates of (8, 5) are correctly drawn one below the starting cell. | |  |
| 3.4 | Non-Linear Delay: Customize the delay interval for various speeds. Implement delay options for drawing instructions. | delay = newDelay  tmrDraw.Interval = 1  Select Case delay  Case <= 5  drawAmount = 7 - delay  Case > 5  drawAmount = 1  tmrDraw.Interval = delay - 5  End Select  For delay values ranging from 1 to 5, the time interval is set to 1, and it draws 7 – delay cells at once. This adjustment is necessary for larger mazes where a time delay of 1 may still be too slow.  In an example with a delay of 1, the draw amount is correctly calculated as  7 - 1 = 6.    As you can see above this functions properly and draws 6 cells within 1 tick.  In an example with a delay of 50 the interval is appropriately set to 45 (50 - 5), ensuring a suitable pacing for larger delays.  This testing and verification indicate that the time delays are functioning as intended, successfully passing the test scenarios. |  |
| 4.1 | Pathfinding - Pathfinding Algorithms: Implement various pathfinding algorithms - Depth-First Search, Breadth-First Search, Greedy DFS, A\* Search. | To test all 4 algorithms and to see if they were preforming as expected I ran them on the same custom drawn grid and recorded the resulting path and metrics.   |  |  | | --- | --- | | Breadth-First Search | Depth-First Search | | Greedy DFS | A\* Search. |   As demonstrated above, all algorithms performed as anticipated. Both A\* and BFS successfully identified the shortest path, with BFS exploring more cells compared to A\*. Notably, DFS and Greedy DFS identified non-optimal paths, as expected. However, Greedy DFS exhibited slightly better performance, aligning with predictions. Furthermore, both Greedy DFS and DFS exhibited quicker runtimes due to their exploration of significantly fewer cells during the algorithmic process. This testing shows all pathfinding algorithms working as expected.  On top of this I ran these algorithms each 20 times on different mazes of size 10, 30, 200 and 600 to verify that there were no errors. |  |
| 4.2 | Algorithm Selection: Enable users to choose pathfinding algorithms through a combo box. Use an enum to manage the different algorithm options. | To test this, I selected A\* and stepped through my program seeing if the code works and the selected path algorithm is set.  Dim selectedPathAlgorithm As PathfindingAlgorithm = cmbSelectedPath.SelectedIndex |  |
| 4.3 | Algorithm Execution: Provide a button to initiate the selected pathfinding algorithm. | To test this, I pressed the solve button with BFS selected and stepped through the  program making sure that the correct algorithm was ran. |  |
| 4.4 | Metrics Presentation: Display algorithm execution metrics - time taken, shortest path, and cells explored. Use labels to convey information. | In testing the pathfinding metrics, I generated a maze using recursive backtracking with a size of 10 and executed an A\* search on it.    As depicted in the example above, the shortest path is 20, evident by counting the length of the dark blue cells. Additionally, the explored cells amount to 24, discernible by counting both the dark and light blue cells. This demonstrates the successful presentation of algorithm execution metrics through clear and informative labels. |  |
| 5.1 | Maze Generation - Maze Generation Algorithms: Implement maze generation algorithms - Kruskal's, Prim's, Binary tree, Recursive Backtracker and Recursive Division. | To test all 5 maze generation algorithms and to see if they were generating mazes as expected, I ran them all on a grid of size 25 and recorder the metrics.   |  |  | | --- | --- | | Recursive Division | Recursive Backtracker | | Binary tree | Prim's | | Kruskal's |   As shown above each maze-generating algorithm produces distinct and visually identifiable patterns. The recursive backtracker, for instance, crafts mazes reminiscent of rivers, featuring intricate and winding corridors. In contrast, the recursive backtracking algorithm yields mazes with elongated horizontal and vertical pathways.  Binary tree generation results in mazes characterized by straight corridors along the left and top, forming a distinct binary tree-like pattern. Both Prim's and Kruskal's algorithms tend to produce more jagged mazes with a higher percentage of dead ends compared to other methods. While the mazes from these two algorithms appear similar, it's possible to discern the extension pattern in Prim's maze as it radiates outward from a single point.  On top of this I generated each maze 20 times for grid sizes of 10, 30, 200 and 600 to verify that there were no errors.  However, whilst generating using Prims algorithm I ran into the error:    This means the test failed as I have failed to implement a prims algorithm that works for all sizes and variants. |  |
| 5.2 | Algorithm Selection: Enable users to choose maze generation algorithms via a combo box. Use an enum to manage the different algorithm options. | In testing, I seleced Kruskal's algorithm and stepped through my program. This involved assessing whether the code functions accurately set the chosen path algorithm. The feature successfully performed as expected, confirming the proper functionality and selection of the specified maze generation algorithm, passing the test.  Dim selectedGridCustomisation As GridCustomisation = cmbSelectedGridCustomisation.SelectedIndex |  |
| 5.3 | Algorithm Execution: Provide a button to initiate the selected maze generation algorithm. | To test this, I pressed the solve button with Recursive Backtracking selected and stepped through the program making sure that the correct algorithm was ran. |  |
| 5.4 | Maze Metrics: Display maze generation metrics - time taken, percentage of dead ends, percentage of branches, average corridor length. | For testing maze generation metrics, I created a maze using recursive backtracking with a size of 10:    The example above showcases the accurate presentation of metrics, including the percentage of dead ends, percentage of branches, and the average corridor length. This passes the test. |  |
| 6.1 | User Customization - Wall Customization: Enable users to add and remove walls interactively using the MouseMove event. Add walls on left button, remove on right button. | To test this objective, I have selected user customisation with a grid of size 25 and customised it.  As you can see in the examples below, here is a few ways you could customise the grid by, adding and removing walls.   |  |  |  | | --- | --- | --- | |  |  |  |   It passes the test however the drawing was quite slow on larger grid sizes and could lead to the walls skipping and missing all the mouse locations leaving a dotted trail if drawing too fast. And while it does still pass the test, this could be optimised. |  |
| 6.2 | Start and Finish Placement: Allow users to set the location of the start and finish points using "C" and "V" keys, respectively. | As evident below, users can dynamically set the start and finish points at different locations, showcasing functionality beyond the default top-left and bottom-right positions.   |  |  |  | | --- | --- | --- | |  |  |  |   However, a notable observation surfaces when trying to position both points at the exact same spot as it will place the points on top of each other, as shown below.   |  |  |  | | --- | --- | --- | |  |  |  |   While this occurrence doesn't break the program and users can still adjust the points away from each other, implementing a prevention measure for this overlap would further enhance the user experience and refine the overall functionality. |  |
| 6.3 | User Guidance: Display instructions using labels to guide users in customizing the grid. Hide controls related to the maze on selection. | On selection of user customization, the form conceals the metrics and displays the user controls guide, as shown on the the right. This successfully meets the test criteria, ensuring a streamlined and user-friendly experience when engaging in grid customization. |  |
| 7.1 | Download - Separate Window: Implement a separate Windows Forms window for downloading grid images. | On the right is my download form, which is opened by pressing the download button. Closing the download form or pressing the download button returns you to the main form. This setup enhances user navigation and provides a dedicated space for handling grid image downloads, contributing to a seamless and intuitive user experience. |  |
| 7.2 | File Path Selection: Provide a button to change the file path using the FileBrowserDialog. | Demonstrated in the example, upon pressing the "Change File Location" button, a file browser dialog opens, providing accurate functionality for users to selectively choose a path location for saving the grid. This achieves the objective. |  |
| 7.3 | Bitmap Download: Allow users to download the grid as a bitmap using a dedicated button and the save method. | Upon pressing the download button, the grid is successfully saved as a PNG file in the selected file location, as illustrated on the right where grid 0 to 3 have been saved. This passes the test. |  |

## Fixes

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Issue | Solution | Resolved |
| 2.4, part 1 | With grid sizes larger than 600, which is the size of the display box, the grid isn’t correctly scaling to show the maze in its entirety. | For Values such as 1200, only the upper left 600 pixels of the grid is displayed:    This issue can be addressed by adjusting the assignment of the grid image to the PictureBox. Instead of setting the image directly, utilize the BackgroundImage property:  picGrid.Image = bmGrid -> picGrid.BackgroundImage = bmGrid  This adjustment is crucial because the property responsible for scaling down the image to fit the PictureBox only works for the BackgroundImage, ensuring the entire maze is displayed regardless of its size. |  |
| 2.4, part 2 | There's an issue in calculating the grid scale, resulting in sizes that are not factors of 600 being inadequately displayed on the grid, cutting off the right and bottom sides. | Dim gridScale As Integer = Math.Ceiling(600 / grid.GetSize)  For values such as 55:  600 / 55 = 10.9 -> 11.  11 \* 55 = 605  605 > 600  As shown above math.ceiling led to incomplete grid displays for sizes not perfectly divisible by 600, as evident by this maze where the sides are cut off:    Recognizing this, the calculation was changed to:  Dim gridScale As Integer = Math.Floor(600 / grid.GetSize)  However, this introduced a new problem for grid sizes exceeding 600:  600 / 1200 = 0.5 -> 0  0 \* 1200 = 0  Leading to nothing being drawn on the grid:    To resolve this, the formula was further adjusted to:  Dim gridScale As Integer = Math.Max(1, Math.Floor(600 / grid.GetSize))  Ensuring that the grid scale value is always a minimum of 1. This fix ensures that the grid displays correctly for various sizes, meeting the objective and passing the test. |  |
| 5.1 | Prims algorithms crashing – index outside the array. | Upon investigation, I observed that this issue occurred more frequently with smaller maze sizes, and only when generating mazes of even sizes.  Dim RndStartingCell As New Point(RandomNum(0, (GridSize - GridSize Mod 2) / 2) \* 2,  RandomNum(0, (GridSize - GridSize Mod 2) / 2) \* 2)    The problem stemmed from the calculation when the grid size was, for example, 6:  6 – 6 Mod 2 / 2 \* 2 =  6 / 2 \* 2 =  6  This resulted in an index that exceeded the valid range, as a grid size of 6 should have indexes ranging from 0 to 5.  This explains why it more commonly occurred for smaller sizes, as the chances of it being the maximum value increases, and only with even sizes, as sizes of for example 5 would return the correct value of 4.  Dim randomStartingCell As New Point(RandomNum(0, (grid.GetSize) / 2 - 1) \* 2,  RandomNum(0, (grid.GetSize) / 2 - 1) \* 2)  The issue has been resolved with the following corrected code, which is not only more readable but also addresses the problem. |  |
| 6.1 | On large grids when drawing with left click and quickly dragging the walls would skip a dotted trail if drawing too fast. | Whenever moving the start / finish point or adding / removing walls the method ClearPath() is ran. This method iterates through each cell and removes cells relating to pathfinding. This is a problem on larger mazes as this process takes along time.  Private isCleared As Boolean  If Not isCleared Then  To fix it needing to run this everytime, I added a global variable called isCleared and and an if condition inside of ClearPath() which returns whether or not the path has any pathfinding cells to remove. Whenever the method FindPath() is ran the isCleared Boolean is set to false.  This drastically increases the speed that you can draw on the grid and avoids unessessary clearing of the grid when there is no path finding cells. |  |
| 6.2 | The start and finish point can be set on the same location leading to werid scenarios with pathfinding. | To address the issue of setting the start and finish points at the same location, additional conditions have been added to the respective if statements. For instance, when setting the finish cell:  ElseIf GetAsyncKeyState(Keys.V) And grid(newPT) <> Type.Start Then  To enhance code readability and minimize redundancy, this could be done in a single line when verifying the point as placing a wall, empty, start, or finish cell shouldn’t be done on a start or finish cell:  If VerifyPoint(newPT, grid.GetSize) AndAlso grid(newPT) < Type.Start Then  The use of AndAlso is crucial to ensure a thorough check within boundaries before proceeding, minimizing errors and improving the overall clarity of the code. These improvements meet my objectives and improve the user experience. |  |

# Evaluation

## Objectives

### Table

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Description | Comment | Met |
| 1.1.1 | Create a custom grid class that can be used to store cell data. | Custom grid class implemented. |  |
| 1.1.2 | Allow the grid to be accessed using an x and y coordinate or a point | Grid access via x, y coordinates or point enabled. |  |
| 1.1.3 | Create a method that allows the user to fill the grid | Method implemented for grid filling. |  |
| 1.1.4 | Create a method that allows the user to set the grid size | Method for setting grid size created. |  |
| 1.2.1 | Define an enum to represent cell types | Enum established for cell types. |  |
| 1.3.1 | Provide a user interface button to remove pathfinding-related cells (Open, Closed, Path) from the grid | UI button for removing pathfinding-related cells added. |  |
| 1.4.1 | Allow users to resize the grid using numeric up-down controls and a slider | Grid resizing enabled with controls and slider. |  |
| 1.4.2 | Set a maximum grid size of 1200 and a minimum size of 5 | Grid size constraints set successfully. |  |
| 1.4.3 | Make the slider use non-linear, scaling steps for grid size selection | Non-linear scaling implemented for the slider. |  |
| 2.1.1 | Use unique colours to visually represent different cell types | Unique colors applied for cell type representation. |  |
| 2.1.2 | Use a function to return the correct colour from the cell type | Function implemented for returning cell color. |  |
| 2.2.1 | Utilize a bitmap to store the grid | Bitmap used for grid storage. |  |
| 2.2.2 | Draw cells on the bitmap using the Graphics class | Cells drawn on the bitmap using Graphics. |  |
| 2.3.1 | Create a user-defined control to handle displaying the grid | User-defined control created for grid display. |  |
| 2.3.2 | Embed a Picture Box in the user-defined control to display the bitmap | Picture Box embedded for bitmap display. |  |
| 2.4.1 | Scale cell sizes on the bitmap to ensure high-quality display when the grid size is smaller than the display | Bitmap cell sizes scaled for optimal display.  Faced an issue with generating the grid scale.  Which was resolved by using Math.Floor instead of Math.Ceiling and ensuring a minimum size of 1 for the calculated value. |  |
| 2.4.2 | Use the Picture Box zoom function to compress the bitmap when the grid is larger than the display | Picture Box zoom used to compress bitmap for larger grids.  Faced an issue where the zoom function wasn’t working and it meant for grid sizes larger than 600 the entire grid wasn’t shown.  I fixed this by setting the .backgroundimage of the picture box rather than the .image. |  |
| 3.1.1 | Enable users to set delay using numeric up-down controls and a slider | User-enabled delay setting implemented. |  |
| 3.1.2 | Restrict the maximum delay to 200 and the minimum to 0 | Delay limits successfully set. |  |
| 3.2.1 | Define a custom data structure to store drawing instructions | Custom data structure defined for drawing instructions. |  |
| 3.3.1 | Use a queue to store drawing instructions | Queue utilized for storing drawing instructions. |  |
| 3.3.2 | Implement a timer to execute drawing instructions, ensuring responsiveness to user inputs, while simultaneously visualize the different algorithms | Timer used for executing drawing instructions. |  |
| 3.4.1 | Customize the delay interval to allow for a wider range of speeds | Delay interval customization implemented. |  |
| 3.4.2 | Implement delay options, such as drawing all instructions at once (delay 0), drawing multiple instructions at each interval step (delay 1, 2, 3, 4, 5) or taking away 5 from the delay to make it stand from 1 (greater than 5) | Delay options incorporated successfully. |  |
| 4.1.1 | Implement a range of pathfinding algorithms | Multiple pathfinding algorithms implemented. |  |
| 4.2.1 | Enable users to choose pathfinding algorithms through a combo box | User-enabled algorithm selection via combo box. |  |
| 4.2.2 | Use an enum to manage the different algorithm options | Enum used for managing algorithm options. |  |
| 4.3.1 | Provide a button to initiate the selected pathfinding algorithm | Button added for initiating pathfinding algorithms. |  |
| 4.4.1 | Display algorithm execution metrics, including time taken, shortest path, and cells explored | Metrics displayed accurately with labels. |  |
| 4.4.2 | Use labels to convey all of this information | Labels employed for conveying algorithm metrics. |  |
| 5.1.1 | Implement a variety of maze generation algorithms | Multiple maze generation algorithms implemented.  Faced an issue with Prims algorithm, where was getting an out of bounds error caused by my generation of a random starting point.  I fixed this by improving this selection to make sure it only selects a position inside the boundaries of the array. |  |
| 5.2.1 | Enable users to choose maze generation algorithms via a combo box | User-enabled algorithm selection for maze generation. |  |
| 5.2.2 | Use an enum to manage the different algorithm options | Enum used for managing maze generation algorithm options. |  |
| 5.3.1 | Provide a button to initiate the selected maze generation algorithm | Button added for initiating maze generation algorithms. |  |
| 5.4.1 | Display maze generation metrics, including time taken, percentage of dead ends, percentage of branches, and average corridor length | Metrics displayed accurately with labels. |  |
| 5.4.2 | Use labels to present this data | Labels employed for presenting maze generation metrics. |  |
| 6.1.1 | Enable users to add and remove walls interactively using the MouseMove event | Walls added and removed interactively with MouseMove event. |  |
| 6.1.2 | Add walls when the left mouse button is held, and remove walls when the right mouse button is held | Walls added and removed according to mouse button events.  Faced a problem, with the efficiency of drawing where quickly dragging would create a dotted trail.  Fixed this by improving the ClearPath() method and creating a variable to store whether or not there are path finding cells on the grid. |  |
| 6.2.1 | Allow users to set the location of the start and finish points using "C" and "V" keys, respectively | Start and finish points set using designated keys.  Faced an issue where the start and finish point could be set on the same location leading to werid scenarios with pathfinding.  I fixed this by adding a condition that would only allow you to change the positions if the cell wasn’t a start or finish cell. |  |
| 6.3.1 | Display instructions using labels to guide users in customizing the grid | Instructions displayed with labels for user guidance. |  |
| 6.3.2 | Hide controls related to maze on selection | Controls related to maze hidden as intended. |  |
| 7.1.1 | Implement a separate Windows Forms window for downloading grid images | Separate window implemented for grid image downloading. |  |
| 7.2.1 | Provide a button to change the file path using the FileBrowserDialog | Button added for changing file path using FileBrowserDialog. |  |
| 7.3.1 | Allow users to download the grid as a bitmap using a dedicated button and the .save method | Grid downloadable as a bitmap using dedicated button. |  |

### Conclusion

I have sufficiently fulfilled all the outlined objectives established in my analysis.

## User Feedback

### Questionnaire

|  |
| --- |
|  |

### Results

|  |  |
| --- | --- |
|  | |
| Role: | A-level computing teacher |
| Ease of use: | 7/10 – Said how choosing the pathfinding and maze generation algorithm was intuitive but could explain the delay and size setting better. |
| Effectiveness: | 10/10 – Works quickly and is a nice tool for visualising pathfinding and maze generation algorithms. |
| Improvements: | * Add a quick tutorial or a hints prompt to help users discover the capibilities of the app. * Suggest possible combinations for algorithms that work well to show differences. |

|  |  |
| --- | --- |
|  | |
| Role: | A-level computing student |
| Ease of use: | 10/10 – It had a very simple UI and that it was self-explanitory |
| Effectiveness: | 10/10 – It worked for all of his tests that he preformed |
| Improvements: | * Indication of when a path isn’t found. |

|  |  |
| --- | --- |
|  | |
| Role: | A-level computing student |
| Ease of use: | 9/10 – Very user friendly and effective. |
| Effectiveness: | 10/10 – All the calculations and varients of the algorithms worked well, especially enjoyed the maze customisation option. |
| Improvements: | * Add more pathfinding and maze generation algorithms for more option when testing. |

## Improvements

My evaluation so far has highlighted 3 main improvements I could make to my solution:

### 1. Quick tutorial

The results from my questionnaire concluded that I could increase my ease of use by implementing a quick tutorial / hints prompt.

This could be implemented by loading the program to a starting form that had a quick explaintion of the setting and how you can set the delay/size.

This would improve it by allowing my user to more easily understand the options I have provided in my program.

### 2. Indication when there is no path

One of my users suggested I could implement a feature that indicates when there is no path.

An example of how this could be implemented is by use of either a message box or I could set all the path finding cells to red.

This would improve it by showing the user when they have created an impossible grid with no results rather than them thinking the program has stopped when in reality it has explored as much as it could.

### 3. Addition of even more algorithms

Add more pathfinding and maze generation algorithms.

This could be achieved by adding additional algorithms such as bidirectional search, random walk algorithm, jump point search, Wilson's algorithm or Aldous-Broder algorithm.

Give the user even more option for testing and compairing a wider range of pathfinding and maze generation algorithms.