# **Maze Generation**

## **Analysis**

The primary end user for the maze generation and pathfinding project will be my classmate Ibraheem, a college student who’s interest in computer science is insurmountable. Ibraheem has shown interest in questions regarding algorithms and their practical applications worldwide. This makes him a prime candidate for this topic. As a student, he has a basic grasp of programming principles but has yet to gain experience with more complex algorithms in maze generation and pathfinding. This lab will provide Ibraheem with an intuitive and entertaining way to learn those algorithms and, ultimately, a richer learning experience. Knowledge from Ibraheem as a peer will go a long way in making the application inclusive and exciting for other students. His perspective will help fine-tune the project into an excellent educational tool for bridging the gap between theory and practice in applying those algorithms.

A maze is a puzzling configuration of barriers and passages that can confuse people trying to get from one point to another. Research on the generation of mazes has its roots in graph theory and computer science, which date back to the 1960s and 1970s. Computers became common during this time, and scholars and researchers began investigating algorithmic ways to generate mazes.

In the 1980s and 1990s, researchers began exploring algorithms for generating mazes using random numbers. Among the most successful algorithms worked out was Depth-First Search (DFS), wherein a random choice is made between unvisited adjacent cells during the passage through the graph. Such an algorithm obtained wide popularity due to its simplicity of implementation. Prim's and Kruskal’s algorithms are other essential techniques primed from this basis. Further down, a description is made of how a few methods work, including DFS: This algorithm of Prim's is quite simple; it selects one vertex arbitrarily and keeps adding edges with minimum weight to the result until the final output is achieved. Prim's algorithm can be implemented using the following steps: Construct an MST taking any arbitrary vertex; after that, all the edges will be determined to connect the tree generated in the above step to the new vertices. Then, it picks the minimum of those edges and adds that to the tree. Repeat step 2 to get the minimum spanning tree.

Kruskal's algorithm starts by choosing the minor weighted edges and keeps adding more edges to the result until it finally arrives at the result. The steps to perform Kruskal's algorithm procedurally are highlighted: Sort all the edges in increasing order of their weight. Pick the minimum weighted edge and add it to the spanning tree. If a cycle forms while adding that edge, neglect it. Choose edges until they connect all the vertices. Here, the minimum spanning tree will be obtained.

The Binary Tree Maze Generator from the 2000s is the best-remembered exception to the random process. It is an exact memory-less maze-making algorithm; there is no limit on making mazes since it constructs perfect mazes but saves no state. In recent years, maze creation has encompassed increasingly complex algorithms and visualisations. Various methods tried by researchers include elaborating optimum mazes using MST, creating labyrinths as a graph traversal problem using DFS and BFS algorithms, and animation or visualisation while attempting to show the development of labyrinths and labyrinth exploration. Maze construction today is a domain that has considerably evolved, with many algorithms and procedures that enable obtaining mazes of different natures and complexities. New technologies like machine learning and generative modelling open new horizons for constructing more realistic and playful mazes.

Maze generation and pathfinding represent very standard topics in computer science, spanning the range from the most straightforward game development, like Legend of Zelda, to the most complex robotics, such as self-driving cars. But often, such subjects get shrouded in a cloak of mystery for a student or an enthusiast if theoretical explanations are given. My project tries to solve this problem by developing an interactive, visualisation-based application. Such a tool would allow the user to observe in real-time how a maze is being generated and how its solution is found for different maze generation methodologies and pathfinding algorithms, respectively. Such a strategy for experiential learning would enable the user to play with the varied parameters, evaluate efficiencies, and obtain practical experience with algorithmic performance and complexity. The project connects theory and practice, enabling the student or the enthusiast to learn the basic concepts of computer science in a friendly and effortless manner.

## **End-user**

The primary end user for the maze generation and pathfinding project will be my classmate Ibraheem, a college student whose interested in computer science is insurmountable. Ibraheem has shown interest in questions regarding algorithms and their practical applications worldwide. This makes him a prime candidate for this topic. As a student, he has a basic grasp of programming principles but has yet to gain experience with more complex algorithms in maze generation and pathfinding. This lab will provide Ibraheem with an intuitive and entertaining way to learn those algorithms and, ultimately, a richer learning experience. Knowledge from Ibraheem as a peer will go a long way in making the application inclusive and exciting for other students. His perspective will help fine-tune the project into an excellent educational tool for bridging the gap between theory and practice in applying those algorithms.

## **Questionnaire:**

**1.** **What's your current understanding of maze generation and pathfinding algorithms?**

My understanding of maze generation and pathfinding is relatively simple. In summary, maze generation generates a complicated path with obstacles and dead ends, whereas finding an optimal path in the maze is known as pathfinding. Theoretically, the discussion of depth-first and breadth-first search has been in-depth in class, but I need clarification on how these two work in practice. I know that algorithms could include A\* and Dijkstra's, though I am incredibly confident in explaining the difference between the two or when to use which.

**2.** **What aspects of these algorithms do you find most challenging to grasp?**

It is difficult to understand how such algorithms make decisions at each step while traversing complicated mazes. For example, how does the algorithm decide to place a wall while generating the labyrinth so that there will be a valid path? It is also difficult to understand how pathfinding algorithms like A\* balance exploration and convergence toward the goal. It is also not easy to know how such algorithms handle the presence of dead ends or multiple paths.

**3.** **How do you prefer to learn new concepts in computer science**

It is difficult to understand how such algorithms make decisions at each step, passing through complicated mazes. For example, how does the algorithm decide to place a wall while generating the labyrinth so that there will be a valid path? How do pathfinding algorithms like A\* balance exploration and convergence toward the goal? It is also not easy to understand how such algorithms handle the presence of dead ends or multiple paths.

**4.** **What features would you find most helpful in a tool for learning about these algorithms?**

An exciting aid would be an animated visualisation of the progressive growth of the maze and its solution.

**5.** **Would you prefer to see the algorithms working in real-time or step through them at your own pace?**

It is worth having both available. Real-time watching is perfect for an overview and comparisons of speed in various ways it can be done. However, allowing yourself to stop and go, step by step, manually is essential if someone wants to understand which exact decision has been taken at any given point in time. Could this be done through this slider, introducing an explicit button that would progress one step at a time?

**6.** **How important is it for you to customise parameters like maze size?**

I would also like to change parameters dynamically. For example, I could change the maze size and see how the algorithms run on smaller and larger mazes. The possibility of changing starting and final points regarding pathfinding would allow me to understand how algorithms will adapt.

**7.** **Are you interested in comparing the performance of different algorithms?**

I am very interested in comparing performances. It would be great to see whether one of those algorithms is faster or more effective for every maze. It would also be great to add an option to run several algorithms on the same maze and then compare their execution times and the length of the paths they create. I want to learn more about how these algorithms will behave under different conditions, such as giant mazes or mazes with many dead ends.

**8.** **How important is the user interface design to you? What would make it most intuitive for you?**

What matters to me most is how the UI is. Intuitive, clean, with easy switching between different algorithms and any visualisations available, precise controls should be present to start, stop, and make step-by-step movements of the algorithms. Ideally, the perfect UI should be divided into three logical zones: a maze visualisation zone, one zone for algorithm control, and a statistics or code presentation zone. Colour coding would be of great help, too, in distinguishing several elements, for example, walls, paths, and where precisely the algorithm is.

**9.** **Do you hope to achieve specific learning outcomes using this tool?**

My main goal is to acquire an overall idea of how these algorithms work, theoretically and practically. I want to visualise all the procedural activity in my mind while I code or discuss such algorithms. I also want to know where to apply each algorithm, hence learning all its advantages and disadvantages. Ultimately, this will give me the confidence to implement such an implementation independently in future projects.

**10.** **Do you have any concerns or potential barriers preventing you from using such a tool effectively?**

I worry that working with the tool will be over complicated or the user will need prior knowledge. I would like the number of options to be manageable, an overloaded interface, or something that might overwhelm them. Also, explanations may need more technical terminology, which may cause misunderstandings. A user tutorial or aid section would help guide users through the available features.

## **Project Brief:**

The developed project will aim to design an interactive maze generation and pathfinder visualisation suitable for computer science students, including Ibraheem. It will include a few maze generation algorithms such as Depth-First Search, Prime's, Kruskal's, and pathfinding algorithms like A\*, Dijkstra's and Breadth-First Search. It should be made available using an easy-to-use, intuitive visual interface. The new tool will enable real-time interaction and step-by-step execution, allowing Ibraheem to learn using his preferred method. The system will offer users pause, rewind, and fast-forward during an algorithm's execution. It will consist of a separate workspace: maze representation, algorithm manipulation controls, and code visualisation, using distinct colours to distinguish the various parts in each. Also, one can change the size of a maze, the difficulty level, and the speeds of algorithms to see many other outcomes.

## **Research**

**Which Programming language should I use C# or Java?**

**C#:**

C# is a high-level language that was created by Anders Hejlsberg in 1999 for Microsoft. It is an object-oriented, type-safe programming language. The programming Language was originally named “Cool” and was set to run on Microsoft .NET framework.

*Pros*: C# is object oriented making it flexible and efficient, it is Statically typed which makes it easy to read and understand saving time. It also offers seamless integration into .NET and backward compatibility.

*Cons*: Codes must be compiled when a change has been made. The speed is not as comparable as other compiled languages, and it cannot perform low-level utility such as directly interacting with hardware or firmware.

**JavaScript:**

JavaScript was created in 1995 by James Gosling for Sun Microsystems, later becoming one of the most popular programming languages to date. Java is used for many things such as games, web development, cloud computing and AI (artificial intelligence).

*Pros:* Java is easy to learn, and it is also object-oriented just like C#. This makes it so you can create code that you can reuse. It is also Multithreading – Running more than one thread at a time so you can maximise CPU utilisation.

*Cons:* The amount of memory that Java uses is a lot, and it is a lot slower than other programming languages and compared to the statically typed C# java tends to be harder to read since it is mostly verbose.

**Python:**

Python was one such language developed by Guido van Rossum in 1991, and at this very moment, it has emerged as one of the most in-demand languages. The usage of Python ranges over a wide horizon, from the development of web pages, data analysis, and artificial intelligence to scientific computing, among many more.

*Pros*: Readability and ease in Python syntax make it the first choice for beginners. Because of the large number of libraries and frameworks, complex applications can be built using Python. An active community means there is a lot of tutorials, discussion forums, and information resources. The Python language supports multiple platforms: Windows, Macintosh, and Linux.

*Cons*: Python is slower compared to other languages such as Java and C++ because it is an interpreted language. Python applications can consume a lot of memory, especially when dealing with applications handling large datasets. Python can be used for this purpose, though a little less powerful, unlike Swift or Kotlin.

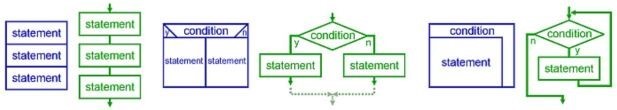
**Final thoughts:**

After weighing all the pros and cons, I will be using JavaScript instead of using either Python or C#. JavaScript is very good at dealing with the complexity associated with generating dynamic mazes due to its simplicity and readability, as well as some of the libraries that come along with it.

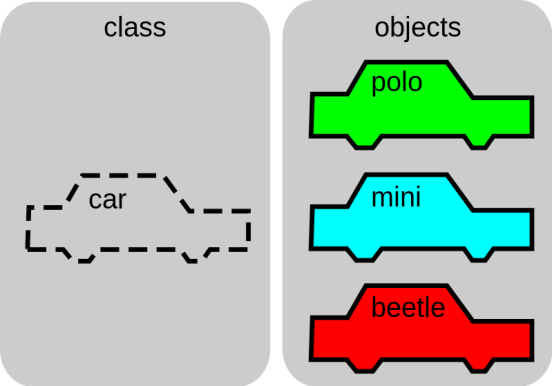
**OOP (Object Oriented Programming) VS Structured Programming**

**Structured Programming:**

Structured Programming, also called modular programming, divides a program into modules and functions. The statements are enclosed in curly brackets and each function performs a subtask. C# is an example of a structured programming language but can also be used for object-oriented programming. When using C#, the programmer can create a function which can call other functions. If a variable is called inside a function, it is called a local variable and can only be used inside the function. Global variables can be used by any function.



**Object Oriented Programming:**



Object oriented programming makes use of objects so that the programmer can represent real world scenarios. Any entity with states and behaviours is an object. While methods indicate an object's behaviours, states represent an object's characteristics or data. Books, Employees, Students, and so on are examples of objects. These objects pass messages to other objects by interacting with each other. A class is a blueprint for creating other objects. To create a Book object, you need a Book class.

Apart from classes and objects, OOP is supported by four main pillars. These include:

**Encapsulation –** Bundling data and methods that work on that data within one unit.

**Polymorphism –** Can access objects of different types through the same interface.

**Abstraction –** Hides the unnecessary information and shows only the important information.

**Inheritance –** Allows a new class to be able to gain the properties and methods of an existing class.

|  |  |
| --- | --- |
| **Structured Programming** | **Object Oriented Programming** |
| Data is not secured | Data is secure |
| Difficult to reuse code | Easy to reuse code |
| There are no access specifiers | There are access specifiers such as private, public and protected |
| Difficult to modify structured programs | Easier to modify structured programs |
| Divides the code into modules or function | Contains data in the form of fields know as attributes and code in the form of procedures know as methods |
| Focuses on dividing the program into a set of functions in which each function works as a subprogram | Focuses on representing a program using a set of objects which encapsulates data and objects |
| It gives more importance of code | It gives more importance to data |

### **1. Maze Generation Techniques**

**Prim's Algorithm**

The most common approach to find the minimum spanning trees is Prim's algorithm, which can be adapted for maze generation. It starts with an initial grid filled with walls, picks up a random starting cell, and from that cell, it adds walls into a priority queue, selecting and removing for further expansion the least expensive wall. This will keep the method ensuring that the creation of the paths is done through the continuously closest adjacent walls.

**Randomised Prim's Algorithm**

An enhanced form of the basic Prim's algorithm, this treats the grid as a graph of nodes. It first instantiates a field completely walled and then arbitrarily selects a cell to start with. Then, this process entails choosing iteratively a minimum-wall cost from a randomly chosen connected component. This will keep forming a path inside the maze: It assures connectivity among all cells with minimum cost.

**Kruskal's Algorithm**

Kruskal's algorithm, much like Prim's, can be modified for mazes by considering each cell a node within a graph. Each node starts out in isolation. It removes random walls, but does not allow any loops and, in essence, it uses a disjoint set to keep track of the various "chunks" of connected cells. The outcome is a spanning tree where any two points will have one and only one path.

**Depth-first search (recursive backtracking)**

In this algorithm, a path is cut through, using a stack to perform a depth-first traversal. Starting from an arbitrary position, it moves in one of the directions to a new unvisited adjacent cell, backtracking as needed until all cells are visited. This ensures that the maze will be loop-free with clear solutions.

**Eller's Algorithm**

Eller's algorithm generates mazes iteratively, one row at a time. From left to right, it connects the cells of the current row with each other and updates the appropriate sets. Moving to the next row, it decides which cells will connect vertically and merges the sets accordingly to keep the maze path uninterrupted.

**Recursive division**

The above-mentioned top-down approach involves repeatedly dividing the grid into smaller sections, using the walls that block them, allowing passage between partitions. If this process continues, it will systematically create complex mazes of separated regions and channels.

**Comparison of Techniques**

The Maze Complexity, Appearance and navigational Solvability are affected by the algorithm choice:

**Complexity**: Recursive backtracking and depth-first search tend to build intricate, narrow pathways, while Eller's and Kruskal's tend to build more regular and balanced mazes with mixed pathway widths.

**Appearance**: Recursive division creates sharply divided mazes with open areas whereas Prim's and recursive backtracking are more path-filled and tight.

**Solvability**: Mazes generated by both recursive division and Kruskal's algorithm tend to allow passage with more open paths and smoother progress compared to the deep recursion of backtracking, which generates a unique and sometimes very complex puzzle.

**Grid Structures**

The grid structure defines the main characteristics of the maze:

**Quadratic Lattices**

Square lattices are common in maze generation since they are comparatively easy and simple to implement. It gives predictable orthogonal navigation and hence simpler implementation of algorithms.

**Hexagonal Grids**

Hexagonal grids add a little more complexity, with six possible directions of movement, and they can be aesthetically cute-providing many challenging pathfinding scenarios. They prevent "straight-line" navigation, which enhances the difficulty and creative appeal of mazes.

**Triangular Grids**

Triangle grids are less common but allow for varied types of movement and can create some interesting maze patterns. They allow much more diagonal movement, making for some surprisingly different maze resolutions.

**Weighted Grids**

Introduction of weights within the grids will also allow generating mazes of varied difficulty. The weights on paths can signify some form of terrain difficulties that could impact a pathfinding algorithm and strategy.

### **2. Pathfinding Algorithms**

**Breadth-First Search (BFS)**

Breadth-First Search is particularly adapted to mazes without weights since its very nature consists of going through all sibling nodes of the current depth before entering nodes strictly on the next depth level. Because of this property, when a node is encountered for the first time, it is reached through the shortest path, meaning a path with the fewest number of edges. Because it is done on the front, BFS is perfectly suited for unweighted mazes where every step or move between cells has an equal "cost" or weight, since it can be guaranteed the shortest path in terms of steps will be found.

**Dijkstra's Algorithm**

Dijkstra's algorithm is important in finding the shortest path in graphs, which have different costs for the paths. It systematically explores paths from the source node, summing the path costs and expanding the least costly node first. Hence, the algorithm is useful, especially in the weighted maze where the movement can have different costs regarding different terrains or obstacles that require more effort to cross. Unlike in BFS, Dijkstra takes these weights into consideration, hence it provides the truly shortest path when several paths are different in cost.

**A\* Algorithm**

A\* Algorithm improves the efficiency of pathfinding by embedding strengths of Dijkstra's accumulation of path costs and heuristics for estimating the cost to reach the goal. The heuristic better orients the search in the direction of the goal and may thus drastically reduce the time of search, provided the heuristic has been engineered correctly.

Algorithm A is applied wherever starting and goal positions are known, and finds applications in computer games and route-finding applications. It averages between optimality and performance: besides the cost coming from the path cost to reach a node, it includes an estimation of the cost needed to reach the goal.

**Comparison**

**Time Complexity**

It works in a time complexity of O(V + E), where V is the total number of vertices and E is the total number of edges. Hence, its regular cost expansion makes it good for traversing unweighted graphs.

Dijkstra's Algorithm uses a priority queue; thus, it has the best time complexity of O((V+ E) log V) by using a binary heap. It applies to weighted graphs; however, it is computationally more expensive compared to BFS on unweighted graphs.

A\* Algorithm: The time complexity of A\* is identical to that of Dijkstra's, i.e., O((V + E) log V), but is heavily dependent upon the underlying heuristic. With a good heuristic, A\* could substantially reduce the average running time with it having a time complexity of O(b^d)

**Real-World Performance**

Breadth-First Search is quite an efficient and simple algorithm; however, it applies only within very basic and unweighted contexts.

Dijkstra's algorithm gives the shortest path for every weighted scenario with perfect accuracy but takes more computation time in return compared to BFS.

A\* Algorithms find applications in contexts when fast, heuristic-based searches are needed. A\* strikes a particularly advantageous balance of efficiency and accuracy when the heuristic function is near the true cost to the target, normally outperforming BFS and Dijkstra's algorithm in real-world, speed-sensitive applications.

### **3. Programming Techniques**

**The role of data structures in path finding and maze generation.**

**Stacks**

Depth-First Search (DFS): This is a major method in conducting DFS within maze building. Such a method relies on a stack, where every time it comes across a dead-end, it backtracks, thus traversing different paths until it constructs the whole maze or finds its way out.

**Queues**

Breadth-First Search: Queues play an important role in how BFS works. Pathfinding algorithms find the paths in an unweighted maze. The queue will permit the algorithm to check all sibling nodes that exist at the present depth before moving to the nodes at the next depth level.

**Priority Queues**

Both Dijkstra's algorithm and A\* use a priority queue, which is so important that its priority for every node depends on the cost of the path leading to it. They are pathfinding heuristics, which can concentrate their explorations on the most promising paths in an effective manner.

**Optimization Techniques for Large Mazes and Real-Time Usage**

A\* Algorithm-Heuristics Techniques: Heuristics will help the algorithm of A\* to reduce useless explorations. A good heuristic function can decrease computation time drastically.

Bidirectional Search: Perform the search from both the starting point and the goal simultaneously, possibly cutting the search space in half.

Path Compression: Use path compression in the union-find data structures while optimising for checking connectivity, which is an essential operation in any efficient maze creation and solving.

Dynamic programming and caching involve the preservation of previously calculated paths or segments of a maze for subsequent use in analogous scenarios, thereby minimising unnecessary computations.

Multithreading and Parallel Processing: Utilise parallel algorithms for the segmentation of the maze into parts whose operations can easily be parallelized to optimise computational time.

### **4. Applications**

**Practical Applications of Maze Generation and Pathfinding**

Maze generation and pathfinding algorithms find their applications across many industries and domains, both practically and otherwise. Some of the key areas wherein such algorithms are applied include:

**1. Robotics**

Autonomous Navigation: The A\* and Dijkstra's pathfinding algorithms, among others, are crucial in robotics navigation to plan paths around obstructive objects in dynamic environments.

SLAM: Robots make use of maze-solving algorithms for the purpose of mapping and locating themselves in an unknown environment.

**2. Gaming**

Level Design: Maze-generating algorithms are one of the usual ways to automatically generate challenging levels with diverse obstacles and routes.

NPC Movement: In-game pathfinding is required when controlling NPCs and needs them to be able to get from one point in the game to another as quickly as possible.

**3. Navigation Systems**

GPS and Mapping Services: The pathfinding algorithm is used as the backbone in GPS-related applications and helps provide optimal routes incorporating real-time data from traffic APIs.

Indoor Navigation Systems: Algorithms for indoor navigation can be adapted to support wayfinding within large facilities, such as airports and malls.

**4. Logistics and Supply Chain**

Route Optimization: In logistics, route optimization for deliveries can be achieved using the pathfinder to greatly reduce time and cost.

Warehouse Automation: Maze generation helps design efficient paths for autonomous vehicles in warehouses, improving order-picking efficiency.

**5. Telecommunications**

Network Routing: Pathfinding algorithms are utilised to determine optimal paths for data packet transmission, improving network efficiency and bandwidth usage.

## **Numbered objectives:**

**1. Implement and utilize the Cell class to represent individual cells of the maze grid and provide functions for handling and visual representation of such cells.**

SMART Objective: The goal is to create and implement a complete `Cell` class that properly models individual cells in the maze and includes functions for cell attribute management in addition to visualization.

Performance Criteria:

* The `Cell` class has to keep track of each cell's status as a wall, pathway, visited node, start point, or end point.
* The class will provide methods for accessing and updating cell attributes, such as but not limited to `isWall()`, `setPath()`, `setVisited()`, etc.
* The class is also intended to observe the walls that surround the cell (for example, `hasNorthWall()`, `removeEastWall()`)
* The class will provide methods to draw the cell on the GUI, including handling different cell states and colours.
* The class will store the cell's position within the maze grid (x, y coordinates).
* The class will provide methods for gaining access to neighbouring cells and ensure effective implementation of boundary conditions.
* The class will be designed to improve efficiency by lowering both processing overhead and memory utilization.
* The class will be well-documented, with clear explanations of its properties and methods.

**2. Implement and use the Grid class to represent the maze structure with functions to manipulate the grid and interact with maze creation and pathfinding algorithms.**

Performance Criteria:

SMART Objective: To create and implement a flexible `Grid` class that best represents the maze layout to support functions for maze modifications and to provide easy integration with maze creation routines and pathfinding routines.

Performance criteria:

* The `Grid` class will use a 2D array or similar data structure to store `Cell` objects, representing the maze.
* The instructional manual will provide methods for designing and initializing the grid to set up the basic walls of the maze.
* The class will provide methods to access and modify individual cells within the grid.
* The class will cover methods for calculating the size of the grid with respect to width and height.
* The class will provide methods for determining whether a cell is within a given set of grid boundaries.
* The class will cover methods for drawing the grid on the graphical user interface and cells and obstacles.
* The class must be designed to maximize efficiency and enable easy and quick access to cells and grids for manipulation.
* The educational material will be carefully documented, including detailed explanations of its qualities and functional capacities.

**3. Implement DFS**

SMART Objective: To implement the DFS algorithm to generate mazes of varying sizes, with a focus on clear visualization of the generation process and efficient algorithm execution.

Performance Criteria:

* The Depth-First Search (DFS) algorithm can generate a valid maze, one that has at least one path from start to finish, in 100% of the tests run for maze sizes ranging from 25x25 to 99x99.
* The building of a 25x25 maze will be complete in under 0.5 seconds based on exact time measurement tools.
* The visual representation of maze generation will employ distinct color-coding for visited cells, the current cell, and final maze walls, with smooth transitions between states.
* The algorithm is crafted to correctly handle edge cases, such as preventing out-of-bounds access and ensuring all cells are always connected.
* The implementation will use a 2D array or grid data structure to represent the maze, with clear definitions for cell states (WALL, PATH, VISITED).
* The algorithm implementation will be done using a recursive strategy, leveraging the call stack to enable the maze traversal.
* The generated mazes will exhibit the characteristics of DFS mazes, such as long, winding corridors and fewer loops.
* The algorithm enables us to identify a starting cell for building the maze.
* The design remains in a modular structure to accommodate the inclusion of other maze generation methods in the future versions.

**4. Implement Prim's algorithm**

SMART Objective: To utilize Prim's algorithm for generating mazes of different sizes with special focus on visual representation of maze development and improving the efficiency of the algorithm.

Performance Criteria:

* Prim's algorithm must always generate a valid maze—a maze with at least one path possible from the start to the finish—in 100% of test runs for maze sizes ranging from 25x25 to 99x99.
* The creation of a 25x25 maze will take a time of less than 0.5 seconds using precision timing techniques.
* The visual representation of maze generation will employ distinct color-coding for visited cells, the current cell, and final maze walls, with smooth transitions between states.
* The algorithm must be written to correctly handle edge cases, such as preventing out-of-bounds accesses and ensuring all cells remain connected.
* The implementation will use a 2D array or grid data structure to represent the maze, with clear definitions for cell states (WALL, PATH, VISITED).
* The resulting mazes will exhibit characteristics typical of Prim's algorithm, such as multiple passages and shorter dead-end sections.
* The program allows for the customization of the cell used in generating the maze.
* The implementation will take on a module-based design to enable easy replacement with different maze creation routines in future developments.

**5. Implement Hunt and Kill algorithm**

SMART Objective: To utilize Hunt and Kill’s algorithm for the creation of mazes of different sizes, making both visualization and algorithm running time efficient.

Performance Criteria:

* Hunt and Kill’s algorithm always generates a valid maze, one with at least one path from the start to the finish, in 100% of test runs for maze sizes ranging from 25x25 to 99x99.
* The creation of a 25x25 maze will be done in less than 0.5 seconds by measurement with high-resolution timing equipment.
* The graphical representation of maze generation will use precise color-coding to differentiate between visited cells, the current cell, and the final walls of the maze, with smooth transitions between states.
* The program is intended to deal with unusual situations properly, including preventing looping and ensuring that all cells are consistently connected to each other.
* The implementation will use a 2D array or grid data structure to represent the maze, with clear definitions for cell states (WALL, PATH, VISITED).
* The generated mazes will illustrate the properties of Hunt and Kill’s algorithm, which include a greater number of possible paths and a smoother movement.
* The algorithm ensures that there exists exactly one exclusive and separate route between each set of points in the maze.
* The implementation shall follow a modular approach that enables easy support for different maze generation algorithms.

**6. Implement A\* search to enable pathfinding and a corresponding visual representation of pathfinding operations and exact navigation through maze structures with different degrees of difficulty.**

SMART Objective: Utilizing the A\* algorithm to successfully find the shortest path in mazes of varying complexity, while also stressing the importance of a proper visualization of the search method.

Performance Criteria:

* The A\* algorithm will find the shortest path from the start point to the end point in 100% of test runs for mazes of varying sizes and complexities, including mazes with dead ends and multiple paths.
* The algorithm's performance will be measured by the number of nodes expanded and the execution time, with results compared against other pathfinding algorithms (Dijkstra's, BFS).
* The graphical representation of the pathfinding process will clearly display the open set, closed set, current node, and final path via the use of distinct color-coding and animations.
* The implementation of this process will allow users to set the starting and ending coordinates of the pathfinding process.
* The algorithm will use a heuristic function to guide the search, and the success of the heuristic will be measured and documented systematically.
* The implementation will handle edge cases such as mazes with no solution and with invalid entry and exit points.
* The effectiveness of the algorithm will be tested against both weighted and unweighted maze structures as needed.
* The implementation will be modular, allowing for easy integration of other pathfinding algorithms.

**7. Implement Dijkstra's algorithm for pathfinding and provide a detailed visual representation of the pathfinding process with guaranteed accurate navigation through mazes of varied complexities.**

SMART Objective: To use Dijkstra's algorithm to find the shortest path through mazes and to consider special emphasis on visualization and accuracy for variable levels of maze difficulty.

Performance Criteria:

* The Dijkstra's algorithm will find the shortest path from the start point to the end point in 100% of test runs for mazes of varying sizes and complexities.
* The performance of the algorithm will be evaluated by measuring nodes expanded and execution time, with the results compared against those of other pathfinding algorithms like A\* and BFS.
* The visual representation of the pathfinding algorithm will differentiate between open set, closed set, and current node using a specific colour-coding and animations and finally represent the final path.
* The implementation of this system will allow users to specify the starting and ending points of the pathfinding process.
* The algorithm can efficiently handle weighted mazes, where applicable, where different paths are linked with varying expenses.
* The implementation will handle edge cases, such as mazes with no solution or invalid start/end points.
* The algorithm's performance will be evaluated in both unweighted and weighted mazes.
* The framework will utilize a modular framework to enable easy addition of future pathfinding methods.

**8. Implement the Breadth-First Search (BFS) algorithm for pathfinding to provide an exhaustive visual representation of the pathfinding process while ensuring accurate navigation through unweighted mazes.**

SMART Objective: To implement the BFS algorithm to find the shortest path in unweighted mazes, with a focus on clear visualization of the search process.

Performance Criteria:

* The BFS algorithm will find the shortest path from the start point to the end point in 100% of test runs for unweighted mazes of varying sizes and complexities.
* The efficiency of the algorithm will be tested on the number of nodes explored and execution time and compared with those of other pathfinding algorithms like A\* and Dijkstra.
* The graphical representation of the pathfinding process will clearly show the visited nodes, the node being considered, and the final path, using precise colour differentiation and animations.
* The implementation will
* allow users to set initial and ending points in the pathfinding process.
* The algorithm shows the ability to solve unweighted mazes, where all paths have the same cost.
* The implementation will cover exceptional cases such as mazes with no solution or with invalid start and exit points.
* The implementation design will utilize a modular framework to make it easier to integrate different pathfinding methods.

**9. Create and design a modular Graphical User Interface (GUI) to support user interaction with pathfinding and maze functions.**

SMART Objective: To design and implement a user-friendly GUI that allows users to easily select maze generation and pathfinding algorithms, adjust maze parameters, and visualize the algorithms in action, with a modular design facilitating future expansion.

Performance Criteria:

* The interface will be compartmentalized into individual areas for maze visualization, algorithm management, statistical data and code output display and user tutorials that follow defined design practices.
* The user interface architecture is built in a modular design to support painless incorporation of future features and algorithms without having to make large changes to the codebase.
* The interface is constructed to provide clear visual feedback to the user, such as the lighting up of the cell being currently processed, the path being traced, and the final solution path.
* The UI will include input validation to prevent errors, such as ensuring maze dimensions are within acceptable ranges and start/end points are within the maze boundaries.
* The user interface shall be responsive and efficient to maintain smooth animations and updating even with increasing maze size.
* The user interface will integrate functionalities that promote accessibility by having keyboard navigation and sufficient colour contrast to support those with disabilities.
* The interface must support a debugging log that gives instant insight into how the algorithm is performing and hence make development and debugging easier.
* The interface will allow the adjustment of the speed of animation for the algorithms, thus enabling users to view the process at their preferred pace.
* The UI will provide options to reset the maze and export the generated maze or solution path.

**10. Create a statistical framework that offers dynamic performance measurement for maze navigation and pathfinding-related algorithms.**

SMART Objective: Developing a statistical module to precisely monitor and represent efficiency and effectiveness-related performance metrics for maze creation and pathfinding methods with the aim of producing useful insights.

Performance Criteria:

* The module has been designed to report and display execution time for maze and pathfinding-related algorithms with accuracy using exact timing mechanisms.
* The module is intended to observe how many nodes pathfinding routines go through and provide a measurement for gauging their search efficiency.
* The module is made to calculate and display path length through different pathfinding solutions and make it easier to compare different algorithms.
* The module will make it possible to clear the statistics and then export them to a file for future reference.
* The module will be incorporated into the GUI, presenting the statistics in an understandable format, for example, through the use of tables or graphs.
* The module will not significantly impact the overall performance of the maze generation and pathfinding algorithms.
* The module will cover unusual situations, especially those that arise with a pathfinding algorithm that is not capable of producing a resolution.
* The module has been designed with some flexibility in order to make it easy to incorporate additional performance criteria in future iterations.

**11. Ensure that the application is resilient and efficient in error management to provide a consistent and reliable user experience.**

SMART Objective: Develop strong error management practices in the application to prevent system crashes, adequately deal with unexpected situations, and provide useful feedback to users.

Performance criteria:

* The application will handle invalid user inputs gracefully, such as incorrect maze dimensions or invalid start/end points, without crashing.
* The program is designed to provide users with informative error messages that explain what is causing the errors and offer possible remediation.
* The program is made to catch exceptions and unexpected situations such as file input/output failures or memory space failures without causing a program crash.
* The program will utilize input validation mechanisms to prevent faults by checking that the maze dimensions are within specified acceptable ranges.
* The program will log errors and exceptions to a file, thus making debugging and troubleshooting easier.
* The program will be tested with a varied set of inputs and circumstances to identify and correct any possible bugs.
* The program should be resilient to unexpected incidents such as system failures or lack of resources.
* The application's error handling mechanisms will be well-documented, explaining how errors are detected and handled.

## **Design**

**System Overview**

The pathfinding and maze generation application is structured into several distinct sections in order to make effective construction, visualisation, and interaction with the mazes easier.

The main component is the Maze Generation Module. It consists of various algorithms, including Depth-First Search, Prim's, and Kruskal's, which allow for the generation of mazes with varying complexity levels. All the algorithms work on a grid data structure, modifying cells and walls to form complex paths.

Next, the generated maze is used as input to the Pathfinding Module. This module uses several different algorithms, namely A\*, Dijkstra's algorithm, and Breadth-First Search, to find and depict the shortest path between two points in the maze.

One of the core components of the system is the Visualisation Engine. The engine efficiently renders the maze, the generation process of the maze, and the pathfinding solutions in a logical and understandable manner. It provides a visual depiction of the maze, highlighting the paths, the walls, and the progression of the algorithms.

The User Interface (UI) serves as the chief means of user interaction. It allows users to select algorithms for pathfinding and maze generation, set the parameters (i.e., dimensions and complexity of the maze), and control visualisation aspects (i.e., start, pause, and step-through execution). In addition, the UI provides features to evaluate performance differences between different algorithms.

At the centre of these modules is a Data Management Component responsible for storing and managing maze data, algorithm states, and user settings.

The system was designed to provide an interactive and instructional experience, allowing users to acquire a better understanding of pathfinding algorithmics and maze generation algorithmics through dynamic visualisation and interactive experience.

**A Modular Configuration System Framework**

The architecture of the pathfinding and maze generation software is based on a modular architecture framework. This approach goes beyond choosing an architecture style; instead, it represents a strategic plan that addresses the underlying issues of the project specifically, as well as simultaneously meeting the needs of a sustainable, flexible, and robust application.

Essentially, the system architecture is a collection of independent modules, each responsible for handling a specific aspect of the application's behaviour. While those modules do interact with each other, what goes on internally within each is largely independent.

**Key Modules:**

**Maze Generation Module:** This module covers the core ideas and techniques relevant to the generation of mazes.

It takes parameters such as maze dimensions, generation algorithm choice (e.g., Depth-First Search, Prim's, Kruskal's), and grid type (e.g., square, hexagonal) as inputs. Internally, it is possible to break it into sub-modules, possibly one per algorithm, keeping the codebase organized and well-structured.

It supports the integration of new generation algorithm designs without affecting the other components of the system.

**Pathfinding Module**: It is responsible for running a multitude of pathfinding algorithmic routines.

It receives the maze data (likely a grid representation) and start/end points as input.

Similar to the Maze Generation Module, this system is also divided into several distinct sub-modules, each responsible for a specific algorithm for example A\*, Dijkstra's, or Breadth-First Search. A modular architecture allows it to incorporate new pathfinding methods or alter existing methods.

**The Visualization Module:** Responsible for controlling the visual representation of the maze and the pathfinding process.

The program takes maze information and navigation steps as input, then prints them to the screen.

The independence of the underlying principles used to build and navigate a maze means that modifications or refinements to the visualization itself—such as changes to the colour scheme or adding animation—are possible without requiring changes to the underlying algorithms.

**User Interface (UI) Module:** The module provides the user with several interfaces via which to interact with the application.

It processes user input in the forms of algorithm choice, parameter setting of the maze, and generation or pathfinding process activation. By keeping the UI separate, the application's core functionality remains independent of the specific interface, which proves useful for potential future modifications such as supporting different input methods or platforms.

**Data Management Module**: It is responsible for managing the data structures being used within the application.

This includes the representation of the maze (e.g., grids, graphs), path information, and algorithm state.

A well-designed data module provides data consistency and is a single point of access to other modules.

**Advantages of Modularity**

**Maintainability**: Modular architecture makes it much easier, easier to debug, and maintain the codebase. Dividing responsibilities into specific modules makes it easier to find and correct issues. When a defect is discovered, it is easier to pinpoint the problem to a specific module, instead of having to search completely through a large, monolithic codebase.

**Extensibility**: The ability to add new functionalities or algorithms to the system with ease. One is able to add new modules or modify existing ones without, at the same time, affecting other parts of the system. For example, when incorporating a new algorithm to generate a maze, a person would create a new module that conforms to the interface defined for generating a maze.

**Reusability**: An important aspect of modules, allowing them to be used in different projects or environments. For instance, the pathfinding module can be independent and used in a separate application that requires pathfinding functionality.

**Testability**: The modular design pattern's implementation helps to improve unit testability. Every module is tested independently, hence verifying it works appropriately on its own. This makes the resulting software more reliable and stable.

**Collaborative work:** Modular architecture allows simultaneous work on several modules by many developers, reducing chances of conflicting work, which improves the overall effectiveness of the process of development.

**Scalability**: The system is highly scalable by supporting the addition of new modules or the enhancement of current ones when the need arises. This is a particularly important feature considering the growing complexity of the application, especially in terms of running more complex algorithms or accommodating larger sizes of the maze.

**Execution in JavaScript**

JavaScript is well-suited for implementing a modular design

**Functions**: Modularity is supported by JavaScript through the concept of function. A module is a collection of function offers different functionalities.

**Objects:** Provide a mechanism to encapsulate associated data and methods, providing an effective method of building a module. This method is particularly useful where modules need to maintain state.

**Classes (ES6 modules):** Enrols a systematic method of defining modules in the JavaScript programming language. The use of import and export statements enhances the availability of the components of a module to other modules through export. This method promotes encapsulation and reduces the possibility of name collusions.

**Module Patterns (IIFEs):** Immediately Invoked Function Expressions (IIFEs) is a time-tested way of building modules in JavaScript. It creates a closure, thus allowing variables and functions to be encapsulated in the module, making it private.

**System Architecture Diagram**

A diagram of a software system

AI-generated content may be incorrect.

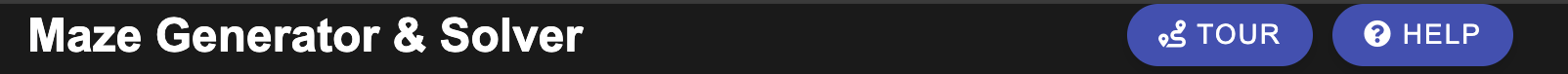
**Key Interactions:**

1. The UI sends user inputs (algorithm choice, parameters) to the Control Logic.
2. The Control Logic triggers the module that handles Pathfinding or Maze Generation.
3. The Visualization Engine receives data from both modules prior to drawing the maze.
4. The Data Management module maintains the grid and cell states of all the components.

**User Interface (UI) Design**

**UI Layout and Navigation**

**Layout Structure:**

 A screenshot of a game

AI-generated content may be incorrect. A screenshot of a game

AI-generated content may be incorrect.

Top Section: Header with title ("Maze Generator & Solver"), help buttons (guided tour, help menu).

A screenshot of a game

AI-generated content may be incorrect. A screen shot of a computer

AI-generated content may be incorrect. A blue and white rectangular sign with white text

AI-generated content may be incorrect.

Control Panel:

* Left: Maze generation algorithm dropdown (Recursive Backtrack, Prim's, Hunt and Kill) and "Generate Maze" button.
* Right: Pathfinding algorithm dropdown (A\*, Dijkstra's, DFS) and "Solve Maze" button.
* Middle: Maze size selector (preset or custom dimensions) and start/end point inputs.

A black grid with red and green squares

AI-generated content may be incorrect.

Visualisation Canvas: Central area displaying the maze grid (25x25 by default) with colour-coded cells (start=green, end=red, path=blue).

A screenshot of a video game

AI-generated content may be incorrect.

Bottom Panel:

* Statistics: Time elapsed, steps, completion percentage, and status.
* Speed Control: Slider for animation speed.
* Debug Log: Real-time logs for tracking algorithm execution.
* Utility Buttons: "Instant," "Reset," and "Export PNG" for workflow control.

**Navigation Flow:**

1. Select maze size and generation algorithm → Generate maze.
2. Set start/end points (optional) → Choose pathfinding algorithm → Solve maze.
3. Adjust speed or use "Instant" to skip animations.
4. Reset or export results.

**Design Choices:**

Modular layout separates controls, visualisation, and diagnostics for clarity.

Real-time updates (e.g., step counter, completion percentage) enhance user feedback.

**UI Wireframes/Mock-ups**

* Top Bar: It is the bar of pushbuttons for the algorithm selection and actions which make changes instantaneously.
* Central Canvas: Grid-based maze with dark grey walls, light grey for paths, and highlighted cells.
* Input Fields: There are numerical inputs for the "Start Row/Col" and "End Row/Col" options. Just bear in mind that the IDs you are entering are numerical in format.
* Status Panel: Time, steps, and a debug log at the bottom to accomplish the task.

**User Interaction Design**

**Input Methods:**

* A drop-down menu to choose an algorithm.
* Numeric input method for creating the grid with custom start/end points.
* A slider for animation speed adjustment.
* Buttons that allow you to create/solve mazes, reset the board, and export.

**Feedback Mechanisms:**

**A screenshot of a computer

AI-generated content may be incorrect.**

* Visual clues: Multiple colours denote visited cells, current cell (dark blue), and path.
* Text feedback: Debug log shows entries (e.g., "Initialising grid").
* Progress Bars: The number of completed tasks will be shown as a percentage, and the counter will be increased by 1.

**Error Handling:**

Input-validation method [e.g., allowing a maze of maximum dimensions between 5 and 100 cells]

Warnings for invalid start/end points (e.g., placing on walls).

Accessibility Considerations

* Keyboard Navigation: Support for tabbing through controls and Enter key activation.
* Screen Reader Compatibility: ARIA labels for buttons and inputs (partially implemented; could be expanded).
* Contrast Ratios: High-contrast colours (e.g., green/red on dark background) for visibility.
* Scalable UI: Responsive design for varying screen sizes (needs enhancement for mobile).

**Colour Palette and Styling**

**Colour Scheme:**

* Primary: Dark theme (#1a1a1a background, #2a2a2a panels) to reduce eye strain.
* Accents: Blue (#2196F3) for buttons and path visualisation.
* Red (#f44336) for end point, green (#4CAF50) for start point.
* Grey (#4a4a4a) for walls and text.

**Fonts:**

Sans-serif (Arial) for readability.

**Styling Justification:**

* Dark theme improves focus on maze visualisation.
* Colour coding simplifies tracking algorithm progress (e.g., blue path vs. grey walls).
* Rounded corners and subtle shadows (box-shadow) modernise the interface.

**Algorithm Design**

# **Designing Depth-First Search (DFS)**

Depth-First Search (DFS) is a basic algorithm that has been utilized for searching or traversing graph and tree data structures. In the context of maze generation, DFS presents an efficient and relatively simple method for creating the complex, interconnected paths. The main idea that DFS is based upon is to completely search the current branch before backtracking is started.

## **1. Algorithm Overview**

The Depth-First Search algorithm starts at a given node, in this case, meaning a cell in the maze. From there, it goes forward as per the following methodology:

1. Mark the current node as visited. This action helps prevent the algorithm from visiting the same cell again, thus avoiding the creation of a loop.
2. Explore a neighbour that has not been visited yet. If more unvisited neighbour nodes exist, the algorithm usually picks one randomly, to ensure that the maze has varying structures.
3. The process requires recursive repetition, setting the chosen neighbour node as the active node. This implies that the algorithm continues deeper in the maze along the set path.
4. The algorithm must backtrack upon finding a node that has no unvisited neighbour nodes. This involves returning to the previous node on the path and trying its other unvisited neighbours.
5. Continue the backtracking process until the start node has been revisited and all the neighbour nodes are visited.

## **2. Data Structures**

DFS for maze generation can be implemented using a simple 2D array (or grid) to represent the maze. Each cell in the grid can have one of several states:

* WALL: Represents a blocked path.
* PATH: Represents an open path.
* VISITED: This indicates that the algorithm has visited this specific cell.

Aside from the grid layout, a stack is also implicitly used in the recursive call to track the path that is being explored. This role is served by the call stack in the program.

## **3. Recursive Approach**

The most common implementation of DFS is recursive:

DFS(cell):

Mark cell as VISITED

While cell has unvisited neighbours:

Choose a random unvisited neighbour, nextCell

Knock down the wall between cell and nextCell // Create a path

DFS(nextCell)

## **4. Handling of Edge Cases**

The algorithm must be invoked from a valid cell that lies within the boundaries of the maze grid.

Boundary Cells: The algorithm should ensure that it does not try to access cells that are outside the defined boundaries of the grid. This can be done by checking the validity of the neighbouring cells' coordinates before trying to read them.

Isolated Cells: The depth-first nature of Depth-First Search ensures that all parts of the maze remain in contact with the starting cell, avoiding the isolation of any area.

Boundaries of the Maze: While creating an external boundary for the maze, the cells that lie along the grid boundary are assigned as WALLs.

**5. Detailed Explanation with Example**

Initial grid:

WALL WALL WALL

WALL START WALL

WALL WALL WALL

1. Begin at the 'START' cell and mark it as 'VISITED'.
2. The algorithm randomly chooses to go 'RIGHT'.
3. Remove the Wall. The current alignment has the following configuration:

WALL WALL WALL

WALL START PATH

WALL WALL WALL

1. Call DFS recursively on the 'PATH' cell. Mark it as 'VISITED'.
2. It can choose to go in the 'DOWN' direction from the 'PATH' cell.
3. Remove the Wall. The structure is represented below:

WALL WALL WALL

WALL START PATH

WALL WALL PATH

1. Call DFS on the new 'PATH' cell.
2. This process continues until a single cell has no neighbouring, unvisited cells. Then the algorithm backtracks.

**6. Depth first search Pseudocode**

FUNCTION DFS\_Maze\_Generation(maze, currentCell):

// maze: 2D array representing the maze grid

// currentCell: The current cell being visited (a coordinate pair)

// Mark the current cell as visited

maze[currentCell.row, currentCell.column] = VISITED

// Get the unvisited neighbours of the current cell

unvisitedNeighbours = GET\_UNVISITED\_NEIGHBOURS(maze, currentCell)

// While there are unvisited neighbours

WHILE unvisitedNeighbours is not empty:

// Choose a random unvisited neighbour

randomIndex = RANDOM(0, LENGTH(unvisitedNeighbours) - 1)

nextCell = unvisitedNeighbours[randomIndex]

// Create a path between the current cell and the next cell

REMOVE\_WALL(maze, currentCell, nextCell)

// Recursively call DFS on the next cell

DFS\_Maze\_Generation(maze, nextCell)

//Re-calculate unvisited neighbours after recursion.

unvisitedNeighbours = GET\_UNVISITED\_NEIGHBOURS(maze, currentCell)

ENDWHILE

ENDFUNCTION

**7. Get Unvisited Neighbours Pseudocode**

FUNCTION GET\_UNVISITED\_NEIGHBOURS(maze, cell):

// maze: 2D array representing the maze grid

// cell: The cell to get neighbours of

neighbours = [] // List to store the unvisited neighbours

//Check the cell to the 'NORTH'

IF cell.row - 1 >= 0 AND maze[cell.row - 1, cell.column] == WALL:

// Create a new cell object

northCell = Cell(cell.row - 1, cell.column)

APPEND northCell TO neighbours

ENDIF

//Check the cell to the 'EAST'

IF cell.column + 1 < LENGTH(maze[0]) AND maze[cell.row, cell.column + 1] == WALL:

eastCell = Cell(cell.row, cell.column + 1)

APPEND eastCell TO neighbours

ENDIF

//Check the cell to the 'SOUTH'

IF cell.row + 1 < LENGTH(maze) AND maze[cell.row + 1, cell.column] == WALL:

southCell = Cell(cell.row + 1, cell.column)

APPEND southCell TO neighbours

ENDIF

//Check the cell to the 'WEST'

IF cell.column - 1 >= 0 AND maze[cell.row, cell.column - 1] == WALL:

westCell = Cell(cell.row, cell.column - 1)

APPEND westCell TO neighbours

ENDIF

RETURN neighbours

ENDFUNCTION

**8. Remove Wall Pseudocode**

FUNCTION REMOVE\_WALL(maze, cell1, cell2):

// maze: 2D array representing the maze

// cell1, cell2: The two adjacent cells between which the wall should be removed

// Determine the relative position of the cells

IF cell1.row == cell2.row: // Cells are in the same row

IF cell1.column < cell2.column:

// cell2 is to the right of cell1

maze[cell1.row, (cell1.column + cell2.column) / 2] = PATH // Remove wall between

ELSE

// cell1 is to the right of cell2

maze[cell2.row, (cell1.column + cell2.column) / 2] = PATH

ENDIF

ELSIF cell1.column == cell2.column: // Cells are in the same column

IF cell1.row < cell2.row:

// cell2 is below cell1

maze[(cell1.row + cell2.row) / 2, cell1.column] = PATH

ELSE

// cell1 is below cell2

maze[(cell1.row + cell2.row) / 2, cell2.column] = PATH

ENDIF

ENDIF

ENDFUNCTION

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Process | Storage | Output |
| Maze Gen Algorithm (DFS, Prim's, Kruskal's) | - DFS: Recursive cell traversal, marking visited, random neighbour selection, backtracking. - Prim's: Wall-based, random start, neighbour wall queue, lowest-cost path creation. - Kruskal's: Node-based, random wall removal, disjoint set to prevent loops. | - Maze Grid: 2D array of cells with wall data. - Visited Cells: Tracked cells during generation. | - Generated Maze: Visual grid with walls/paths. - Maze Structure: In-memory grid representation. |
| Maze Dimensions (Width, Height) | - Grid Init: Create 2D grid, set default cell state. - Cellular Rep: Allocate memory for cell properties. | - Cell Data: Wall status, visited, coordinates. - Grid Size: Width, height, cell count. | - Grid with dimensions: Initialized grid matching input. - Initialized Cells: Cells with default properties. |
| Path find Algorithm (A\*, Dijkstra's, BFS) | - A: \* Heuristic cost estimate, open/closed sets, lowest total cost path. - Dijkstra's: Path exploration from start, cost calculation, lowest-cost expansion. - BFS: Level-by-level exploration for shortest path. | - Open/Closed Sets: Nodes for evaluation (A\*, Dijkstra's). - Path Cost: Distance from start to each node. - Parent Node: For path reconstruction. | - Shortest/Valid Path: Cell sequence from start to end. - Path Cost: Total path distance. |
| Start Point (Coordinates) | - Start Definition: Specify pathfinding start cell. - Start Init: Set start cell as current, init properties. | - Start Coords: X, Y coordinates of start. | - Highlighted Start: Visual marker for start cell. |
| End Point (Coordinates) | - End Definition: Specify pathfinding goal cell. - Goal Test: Check if current cell is the goal. | - End Coords: X, Y coordinates of goal. | - Highlighted End: Visual marker for goal cell. |
| Maze Data Struct (2D Array, Grid) | - Structure manipulation: Modify cell properties, get maze state. - Grid Traversal: Access/update cells during algorithms. | - Grid Updates: Store cell changes (walls, visited). | - Updated Maze: Data structure after changes. |
| UI Input (Clicks, Params) | - Event Handle: Capture user interactions. - Input Process: Extract algorithm, params, actions. - Data Valid: Check input validity. | - User Preferences: Algorithm, dims, animation settings. - UI State: Open panels, selected options. | - Updated UI: Reflect input, display changes. |
| Animation Speed (User-Defined) | - Animation Timing: Control delay between visualization steps. - Render Updates: Update visual maze at set intervals. | - Delay Values: Animation speeds to delay mappings. | - Animated Maze: Step-by-step visual of generation/solving. |
| Algorithm Visuals (Graphical Steps) | Visualisation Data Generation: Create data for visualization (cell colours, etc.). - Graph Render: Use data to show algorithm steps visually. | - Visualisation Data: Cell colours, path highlights, node order. | - Algorithm Visual: Graphical display of algorithm execution. |
| Intermediate Maze State (Partial Maze) | - Structure Modifications: Algorithms change the maze grid. - Grid Updates: Store the maze's partial state. | - Grid Updates: Store intermediate maze states. | - Partial Maze: Maze state during generation. |
| Intermediate Path State (Partial Path) | - Path Track: Track pathfinding progress. - Path Updates: Update path data during search. | - Path Updates: Store intermediate path states. | - Partial Path: Path state during pathfinding. |

A diagram of a software algorithm

AI-generated content may be incorrect.

A diagram of a computer

AI-generated content may be incorrect.

A diagram of cell

AI-generated content may be incorrect.

**Class Diagram Overview**

The class diagram outlines the **structural organization** of a maze generation and pathfinding system. It declares key classes:

* MazeGenerator: An abstract class with concrete implementations like RecursiveBacktracker, Prim, and HuntAndKill.
* Pathfinder: Another abstract class containing implementations like AStar, Dijkstra, and BFS.
* Statistics: Included for monitoring **performance indicators**.
* Grid: Used to map maze setup.
* Cell: The basic building block of the grid, with fields for wall setup, visitation flag, and start and end points.

**Cell Details**

Cell has methods for:

* Representation
* Neighbour checking

**System Architecture**

Both pathfinders and generators utilize Statistics to keep track of **execution stats**. The diagram effectively shows interrelations and inheritance relationships between these classes, presenting a clear representation of the system architecture.

**Pathfinding**

**A\* Pathfinding algorithm**

A cost function is a mathematical formula that calculates the total cost to reach a specific part of a maze from the starting position. This is the cost function for the A\* pathfinding algorithm

A diagram of a function

Description automatically generated

A\* finds the shortest path exactly like Dijkstra while also having an educated guess. The function f(n) = g(n) is Dijkstra’s algorithm with the h(n) being the guess.

f(n) = estimate of the total cost from start node to the target node through node n

g(n) = actual cost from start node to node n

h(n) = estimated cost from node n to target node

The function h(n) is the heuristic

If h(n) = 0, A\* becomes Dijkstra’s which will get the shortest path.

**Path node (g(n))**

We can express g(n) mathematically for a path through start to current node as:

Where is the weight of the edge connecting

**Heuristic Function (h(n))**

**Manhattan**

**Euclidean**

**A graph of a line with a cross

Description automatically generated with medium confidence**

**Pseudocode for A\* Pathfinding:**

function A\_Star(start, goal):

// Initialize open and closed lists

openList = [start] // Nodes to be evaluated

closedList = [] // Nodes already evaluated

// Initialize node properties

start.g = 0 // Cost from start to start is 0

start.h = heuristic(start, goal) // Estimate to goal

start.f = start.g + start.h // Total estimated cost

start.parent = null // For path reconstruction

while openList is not empty:

// Get node with lowest f value - implement using a priority queue

// for faster retrieval of the best node

current = node in openList with lowest f value

// Check if we've reached the goal

if current = goal:

return reconstruct\_path(current)

// Move current node from open to closed list

remove current from openList

add current to closedList

// Check all neighbouring nodes

for each neighbour of current:

if neighbour in closedList:

continue // Skip already evaluated nodes

// Calculate tentative g score

tentative\_g = current.g + distance(current, neighbour)

if neighbour not in openList:

add neighbour to openList

else if tentative\_g >= neighbour.g:

continue // This path is not better

// This path is the best so far

neighbour.parent = current

neighbour.g = tentative\_g

neighbour.h = heuristic(neighbour, goal)

neighbour.f = neighbour.g + neighbour.h

return failure // No path exists

function reconstruct\_path(current):

path = []

while current is not null:

add current to beginning of path

current = current.parent

return path

**Testing**

## **Technical implementation**

Index.html:

<!DOCTYPE html>

<html>

<head>

<title>Maze Generator & Solver</title>

<link rel="stylesheet" href="styles.css">

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/intro.js/7.0.1/introjs.min.css">

<script src="https://cdnjs.cloudflare.com/ajax/libs/intro.js/7.0.1/intro.min.js"></script>

</head>

<body>

<div class="header">

<h1>Maze Generator & Solver</h1>

<div class="header-buttons">

<button onclick="startTour()" class="help-button" title="Start guided tour">

<i class="fas fa-route"></i> Tour

</button>

<button onclick="toggleHelpMenu()" class="help-button" title="Open help menu">

<i class="fas fa-question-circle"></i> Help

</button>

</div>

</div>

<div class="controls">

<div class="main-controls">

<div class="control-group" data-intro="Select your maze generation algorithm here" data-step="1">

<select id="genAlgorithm" class="algorithm-select" title="Choose the algorithm to generate your maze">

<option value="recursive\_backtracker">Recursive Backtracker</option>

<option value="prims">Prim's Algorithm</option>

<option value="hunt\_and\_kill">Hunt and Kill</option>

</select>

<button onclick="generateMaze()" title="Generate a new maze using selected algorithm">Generate Maze</button>

</div>

<div class="control-group" data-intro="Select your pathfinding algorithm here" data-step="2">

<select id="pathAlgorithm" class="algorithm-select" title="Choose the algorithm to find the path from start to end">

<option value="astar">A\* Pathfinding</option>

<option value="dijkstra">Dijkstra's Algorithm</option>

<option value="dfs">Depth-First Search</option>

</select>

<button onclick="solveMaze()" title="Solve the maze using selected algorithm">Solve Maze</button>

</div>

</div>

<div class="size-controls">

<div class="size-group">

<label for="mazeSize">Maze Size:</label>

<select id="mazeSize" class="size-select" title="Select the dimensions of your maze">

<option value="10">10x10 (Small)</option>

<option value="25" selected>25x25 (Medium)</option>

<option value="40">40x40 (Large)</option>

<option value="60">60x60 (Extra Large)</option>

<option value="custom">Custom Size</option>

</select>

</div>

<div class="custom-size" id="customSizeInputs" style="display: none;">

<div class="size-input-group">

<label for="customWidth">Width:</label>

<input type="number" id="customWidth" min="5" max="100" value="25"

title="Enter width (5-100 cells)">

</div>

<div class="size-input-group">

<label for="customHeight">Height:</label>

<input type="number" id="customHeight" min="5" max="100" value="25"

title="Enter height (5-100 cells)">

</div>

<button id="applyCustomSize" class="size-button" title="Apply custom dimensions to maze">Apply Size</button>

</div>

</div>

<div class="custom-points-controls">

<div class="custom-points-controls">

<div class="control-group">

<label for="startRow">Start Row (0-<span id="maxRowStart"></span>):</label>

<input type="number" id="startRow" min="0" value="0">

<label for="startCol">Start Col (0-<span id="maxColStart"></span>):</label>

<input type="number" id="startCol" min="0" value="0">

<button class="custom-point-button"

onclick="setStart(parseInt(document.getElementById('startCol').value), parseInt(document.getElementById('startRow').value))">Set

Start</button>

</div>

<div class="control-group">

<label for="endRow">End Row (0-<span id="maxRowEnd"></span>):</label>

<input type="number" id="endRow" min="0" value="">

<label for="endCol">End Col (0-<span id="maxColEnd"></span>):</label>

<input type="number" id="endCol" min="0" value="">

<button class="custom-point-button"

onclick="setEnd(parseInt(document.getElementById('endCol').value), parseInt(document.getElementById('endRow').value))">Set

End</button>

</div>

</div>

</div>

<div class="playback-controls" data-intro="Control the animation process" data-step="3">

<button id="instantButton" class="control-button instant-button"

title="Complete the current operation immediately">

<i class="fas fa-bolt"></i> Instant

</button>

<button id="resetButton" class="control-button" title="Clear the maze and start over">

<i class="fas fa-undo"></i> Reset

</button>

<button onclick="exportMazeAsPNG()" class="control-button" title="Export maze as PNG">

<i class="fas fa-download"></i> Export PNG

</button>

</div>

<div class="statistics-panel">

<div class="stat-group">

<div class="stat-item">

<i class="fas fa-stopwatch"></i>

<span>Time: </span>

<span id="timeElapsed">0.0s</span>

</div>

<div class="stat-item">

<i class="fas fa-shoe-prints"></i>

<span>Steps: </span>

<span id="stepCount">0</span>

</div>

</div>

<div class="stat-group">

<div class="stat-item">

<i class="fas fa-percentage"></i>

<span>Completion: </span>

<span id="completionRate">0%</span>

</div>

<div class="stat-item">

<i class="fas fa-chart-line"></i>

<span>Status: </span>

<span id="currentStatus">Ready</span>

</div>

</div>

</div>

<div class="speed-control" data-intro="Adjust the animation speed" data-step="3">

<label>Speed:</label>

<input type="range" min="1" max="100" value="50" id="speedControl" min="1" max="100" value="50"

title="Adjust the speed of generation and solving">

</div>

</div>

<div class="debug-panel">

<h3>Debug Log</h3>

<div id="debugLogDisplay" class="debug-log-display">

<!-- Log messages will be displayed here -->

</div>

<button onclick="clearDebugLog()" class="clear-log-button">Clear Log</button>

</div>

<canvas id="mazeCanvas" data-intro="The maze will be displayed here" data-step="4"></canvas>

<div id="helpModal" class="modal">

<div class="modal-content">

<span class="close">&times;</span>

<h2>Maze Generator & Solver Help</h2>

<div class="help-section">

<h3>Maze Size</h3>

<ul>

<li><strong>Preset Sizes:</strong> Choose from small (10x10) to extra large (60x60)</li>

<li><strong>Custom Size:</strong> Enter your own dimensions (5-100 cells)</li>

</ul>

</div>

<div class="help-section">

<h3>Generation Algorithms</h3>

<ul>

<li><strong>Recursive Backtracker:</strong> Creates long corridors and fewer dead ends</li>

<li><strong>Prim's Algorithm:</strong> Creates a more balanced maze with shorter corridors</li>

<li><strong>Hunt and Kill:</strong> Similar to recursive backtracker but with different patterns</li>

</ul>

</div>

<div class="help-section">

<h3>Pathfinding Algorithms</h3>

<ul>

<li><strong>A\* Pathfinding:</strong> Finds the shortest path using heuristics (fastest)</li>

<li><strong>Dijkstra's Algorithm:</strong> Guarantees the shortest path by exploring all possibilities</li>

<li><strong>Depth-First Search:</strong> Quickly finds a path but not necessarily the shortest</li>

</ul>

</div>

<div class="help-section">

<h3>Playback Controls</h3>

<ul>

<li><strong>Instant:</strong> Complete the current operation immediately</li>

<li><strong>Reset:</strong> Clear the maze and start over</li>

<li><strong>Speed Slider:</strong> Adjust the animation speed</li>

</ul>

</div>

<div class="help-section">

<h3>Colour Guide</h3>

<ul>

<li><strong>Green Cell:</strong> Start point (top-left)</li>

<li><strong>Red Cell:</strong> End point (bottom-right)</li>

<li><strong>Blue Path:</strong> Solution path from start to end</li>

<li><strong>Dark Grey:</strong> Visited cells during generation/solving</li>

</ul>

</div>

<div class="help-section">

<h3>Statistics Panel</h3>

<ul>

<li><strong>Time:</strong> Duration of the current operation</li>

<li><strong>Steps:</strong> Number of algorithm steps taken</li>

<li><strong>Completion:</strong> Percentage of maze generated/solved</li>

<li><strong>Status:</strong> Current state of operation</li>

</ul>

</div>

</div>

</div>

<script src="maze.js"></script>

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.15.4/css/all.min.css"></body>

</html>

Maze.js

// get the canvas element and its drawing context

const mazeCanvas = document.getElementById('mazeCanvas');

const context = mazeCanvas.getContext('2d');

// grid size settings

let gridColumns = 25;

let gridRows = 25;

let cellSize = 20;

// control and state variables

let isPlaying = false;

let isPaused = false;

let currentStep = 0;

let animationFrame;

let currentOperation = null;

let startTime = null;

let stepCount = 0;

let totalCells = gridColumns \* gridRows;

let visitedCells = 0;

// pathfinding-related sets

let openNodes = [];

let closedNodes = [];

let path = [];

// set canvas size dynamically

mazeCanvas.width = gridColumns \* cellSize;

mazeCanvas.height = gridRows \* cellSize;

// grid structure

let grid = [];

let stack = [];

let startCell, endCell;

let currentCell;

// animation control settings

let animationSpeed = 50;

let isGenerating = false;

let isSolving = false;

// customisable colours for different elements in the maze

const colors = {

background: '#1a1a1a', // almost black

wall: '#4a4a4a', // mid grey

visited: '#2a2a2a', // dark grey

current: '#000033', // very dark blue

start: '#4CAF50', // green

end: '#f44336', // red

path: '#2196F3' // blue

};

// stores the start and end positions in the grid

let startPos = { i: 0, j: 0 };

let endPos = { i: gridColumns - 1, j: gridRows - 1 };

function setStart(i, j) {

log(`Setting start to ${i}, ${j}`, 'info');

if (i >= 0 && i < gridColumns && j >= 0 && j < gridRows) {

startPos = { i: i, j: j };

defineBoundaries();

redrawMaze();

} else {

console.warn("Invalid start position");

}

}

function setEnd(i, j) {

log(`Setting end to ${i}, ${j}`, 'info');

if (i >= 0 && i < gridColumns && j >= 0 && j < gridRows) {

endPos = { i: i, j: j };

defineBoundaries();

redrawMaze();

} else {

console.warn("Invalid end position");

}

}

// initialize start and end points in the grid.

function defineBoundaries() {

if (!grid.length) {

console.warn("Grid not initialized yet! Ensure grid is generated before calling this.");

return;

}

grid.forEach(cell => {

cell.isStart = false;

cell.isEnd = false;

});

startCell = grid[index(startPos.i, startPos.j)];

endCell = grid[index(endPos.i, endPos.j)];

if (startCell) startCell.isStart = true;

if (endCell) endCell.isEnd = true;

}

// represents a single cell in the maze grid

class Cell {

constructor(i, j) {

this.i = i;

this.j = j;

this.walls = [true, true, true, true];

this.visited = false;

this.isStart = false;

this.isEnd = false;

this.inMaze = false;

}

show(highlight = false) {

let x = this.i \* cellSize;

let y = this.j \* cellSize;

context.strokeStyle = colors.wall;

context.fillStyle = this.determineFillColor(highlight);

context.fillRect(x, y, cellSize, cellSize);

context.beginPath();

if (this.walls[0]) { context.moveTo(x, y); context.lineTo(x + cellSize, y); } // Top wall

if (this.walls[1]) { context.moveTo(x + cellSize, y); context.lineTo(x + cellSize, y + cellSize); } // Right wall

if (this.walls[2]) { context.moveTo(x + cellSize, y + cellSize); context.lineTo(x, y + cellSize); } // Bottom wall

if (this.walls[3]) { context.moveTo(x, y + cellSize); context.lineTo(x, y); } // Left wall

context.stroke();

}

determineFillColor(highlight) {

if (this.isStart) return colors.start;

if (this.isEnd) return colors.end;

if (highlight) return colors.current;

if (this.visited) return colors.visited;

return colors.background;

}

checkNeighbours() {

let neighbours = [];

let directions = [[0, -1], [1, 0], [0, 1], [-1, 0]]; // Top, right, bottom, left

randomiseArray(directions);

for (let [dx, dy] of directions) {

let newI = this.i + dx;

let newJ = this.j + dy;

let neighbour = grid[index(newI, newJ)];

if (neighbour && !neighbour.visited) {

neighbours.push(neighbour);

}

}

return neighbours.length > 0 ? neighbours[0] : undefined;

}

}

// converts 2D grid coordinates to 1D array index

function index(i, j) {

if (i < 0 || j < 0 || i > gridColumns - 1 || j > gridRows - 1) return -1;

return i + j \* gridColumns;

}

// removes walls between two adjacent cells

function deleteWalls(a, b) {

let x = a.i - b.i;

let y = a.j - b.j;

if (x === 1) { // B is left of A

a.walls[3] = false;

b.walls[1] = false;

} else if (x === -1) { // B is right of A

a.walls[1] = false;

b.walls[3] = false;

}

if (y === 1) { // B is above A

a.walls[0] = false;

b.walls[2] = false;

} else if (y === -1) { // B is below A

a.walls[2] = false;

b.walls[0] = false;

}

}

// randomises array elements using Fisher-Yates shuffle

function randomiseArray(array) {

for (let i = array.length - 1; i > 0; i--) {

const j = Math.floor(Math.random() \* (i + 1));

[array[i], array[j]] = [array[j], array[i]];

}

return array;

}

// recursive backtracker maze generation algorithm

async function generateMazeRecursiveBacktracker() {

isGenerating = true;

current = start;

stack = [];

let visitedCount = 0;

const totalCells = gridColumns \* gridRows;

while (isGenerating) {

current.visited = true;

current.inMaze = true;

visitedCount++;

let next = current.checkNeighbours();

if (next) {

next.visited = true;

stack.push(current);

deleteWalls(current, next);

current = next;

} else if (stack.length > 0) {

current = stack.pop();

} else {

if (visitedCount === totalCells) break;

let unvisited = grid.find(cell => !cell.visited);

if (unvisited) {

let nearestVisited = searchNearestVisitedCell(unvisited);

if (nearestVisited) {

deleteWalls(unvisited, nearestVisited);

current = unvisited;

continue;

}

}

break;

}

statistics.incrementStep();

statistics.updateVisitedCells(visitedCount);

redrawMaze();

if (!isPaused) {

await new Promise(resolve => setTimeout(resolve, 101 - animationSpeed));

}

}

isGenerating = false;

statistics.stop();

}

// prim's Algorithm

async function primsMazeGeneration() {

isGenerating = true;

let frontierCells = new Set();

let inMaze = new Set();

let visitedCount = 0;

start.inMaze = true;

inMaze.add(start);

visitedCount++;

insertFrontierCells(start, frontierCells, inMaze);

while (frontierCells.size > 0 && isGenerating) {

let frontierArray = Array.from(frontierCells);

let current = frontierArray[Math.floor(Math.random() \* frontierArray.length)];

frontierCells.delete(current);

let mazeNeighbours = [];

let possibleDirs = [[0, -1], [1, 0], [0, 1], [-1, 0]];

randomiseArray(possibleDirs);

for (let [dx, dy] of possibleDirs) {

let newI = current.i + dx;

let newJ = current.j + dy;

let neighbour = grid[index(newI, newJ)];

if (neighbour && neighbour.inMaze) {

mazeNeighbours.push(neighbour);

}

}

if (mazeNeighbours.length > 0) {

let selectedNeighbour = mazeNeighbours[Math.floor(Math.random() \* mazeNeighbours.length)];

deleteWalls(current, selectedNeighbour);

current.inMaze = true;

current.visited = true;

inMaze.add(current);

visitedCount++;

insertFrontierCells(current, frontierCells, inMaze);

}

statistics.incrementStep();

statistics.updateVisitedCells(visitedCount);

redrawMaze();

if (!isPaused) {

await new Promise(resolve => setTimeout(resolve, 101 - animationSpeed));

}

}

isGenerating = false;

statistics.stop();

}

// hunt and Kill Algorithm

async function huntAndKillMazeGeneration() {

isGenerating = true;

let current = start;

let visitedCount = 0;

const totalCells = gridColumns \* gridRows;

while (current && isGenerating) {

current.visited = true;

current.inMaze = true;

visitedCount++;

let neighbours = current.checkNeighbours();

if (neighbours) {

deleteWalls(current, neighbours);

current = neighbours;

} else {

current = null;

huntLoop: for (let j = 0; j < gridRows; j++) {

for (let i = 0; i < gridColumns; i++) {

let cell = grid[index(i, j)];

if (!cell.visited) {

let dirs = [[0, -1], [1, 0], [0, 1], [-1, 0]];

randomiseArray(dirs);

for (let [dx, dy] of dirs) {

let newI = i + dx;

let newJ = j + dy;

let neighbour = grid[index(newI, newJ)];

if (neighbour && neighbour.visited) {

deleteWalls(cell, neighbour);

current = cell;

break huntLoop;

}

}

}

}

}

}

statistics.incrementStep();

statistics.updateVisitedCells(visitedCount);

redrawMaze();

if (!isPaused) {

await new Promise(resolve => setTimeout(resolve, 101 - animationSpeed));

}

if (visitedCount === totalCells) {

break;

}

}

isGenerating = false;

statistics.stop();

}

// adds cells to frontier

function insertFrontierCells(cell, frontierCells, inMaze) {

let dirs = [[0, -1], [1, 0], [0, 1], [-1, 0]];

for (let [dx, dy] of dirs) {

let newI = cell.i + dx;

let newJ = cell.j + dy;

let neighbour = grid[index(newI, newJ)];

if (neighbour && !neighbour.inMaze && !frontierCells.has(neighbour)) {

frontierCells.add(neighbour);

}

}

}

// finds nearest visited cell to connect isolated cells

function searchNearestVisitedCell(cell) {

let visited = grid.filter(c => c.visited);

if (visited.length === 0) return null;

let nearest = visited[0];

let minDist = Math.abs(cell.i - nearest.i) + Math.abs(cell.j - nearest.j);

for (let v of visited) {

let dist = Math.abs(cell.i - v.i) + Math.abs(cell.j - v.j);

if (dist < minDist) {

minDist = dist;

nearest = v;

}

}

return nearest;

}

// manage stats and status

class Statistics {

constructor() {

this.timeElement = document.getElementById('timeElapsed');

this.stepElement = document.getElementById('stepCount');

this.completionElement = document.getElementById('completionRate');

this.statusElement = document.getElementById('currentStatus');

this.updateInterval = null;

}

start(operationType) {

startTime = Date.now();

stepCount = 0;

visitedCells = 0;

totalCells = gridColumns \* gridRows;

this.updateStatus(`${operationType} in progress...`);

this.startTimeUpdate();

}

stop() {

if (this.updateInterval) {

clearInterval(this.updateInterval);

this.updateInterval = null;

}

if (startTime) {

this.updateTimeElapsed();

}

this.updateStatus('Complete');

this.updateVisitedCells(totalCells);

}

reset() {

startTime = null;

stepCount = 0;

visitedCells = 0;

clearInterval(this.updateInterval);

this.timeElement.textContent = '0.0s';

this.stepElement.textContent = '0';

this.completionElement.textContent = '0%';

this.updateStatus('Ready');

}

incrementStep() {

stepCount++;

this.stepElement.textContent = stepCount;

}

updateVisitedCells(count) {

visitedCells = count;

if (this.completionElement) {

const completion = Math.min(100, Math.round((visitedCells / totalCells) \* 100));

this.completionElement.textContent = `${completion}%`;

console.log('Completion:', completion + '%');

}

}

updateStatus(status) {

this.statusElement.textContent = status;

}

startTimeUpdate() {

this.updateTimeElapsed();

this.updateInterval = setInterval(() => this.updateTimeElapsed(), 100);

}

updateTimeElapsed() {

if (!startTime) return;

const elapsed = (Date.now() - startTime) / 1000;

this.timeElement.textContent = `${elapsed.toFixed(1)}s`;

}

setSteps(steps) {

stepCount = steps;

if (this.stepElement) {

this.stepElement.textContent = steps;

console.log('Steps set to:', steps, 'info');

}

}

setInstantComplete(steps) {

if (this.updateInterval) {

clearInterval(this.updateInterval);

this.updateInterval = null;

}

this.setSteps(steps);

const theoreticalTime = calculateTheoreticalTime(steps);

if (this.timeElement) {

this.timeElement.textContent = `${theoreticalTime.toFixed(1)}s (instant)`;

}

this.updateVisitedCells(totalCells);

this.updateStatus('Completed Instantly');

startTime = null;

console.log('Instant completion:', {

steps: steps,

theoreticalTime: theoreticalTime,

completion: '100%'

});

}

}

const statistics = new Statistics();

async function generateMaze() {

log("Generating maze...", "info");

if (isGenerating || isSolving) return;

resetMaze();

currentOperation = 'generating';

isGenerating = true;

statistics.start('Generation');

const selectedAlgorithm = document.getElementById('genAlgorithm').value;

current = start;

stack = [];

switch(selectedAlgorithm) {

case 'prims':

await primsMazeGeneration();

break;

case 'hunt\_and\_kill':

await huntAndKillMazeGeneration();

break;

case 'recursive\_backtracker':

default:

await generateMazeRecursiveBacktracker();

break;

}

log("Maze generation complete.", "info");

}

// solves maze using selected pathfinding algorithm

async function solveMaze() {

if (isGenerating || isSolving) return;

currentOperation = 'solving';

isSolving = true;

isPaused = false;

path = [];

openNodes = [];

closedNodes = [];

grid.forEach(cell => {

cell.visited = false;

cell.previous = null;

cell.f = 0;

cell.g = 0;

cell.distance = Infinity;

});

const algorithm = document.getElementById('pathAlgorithm').value;

switch(algorithm) {

case 'dijkstra':

start.distance = 0;

openNodes = [start];

break;

case 'dfs':

openNodes = [start];

break;

case 'astar':

default:

start.g = 0;

start.f = heuristic(start, end);

openNodes = [start];

break;

}

statistics.start('Pathfinding');

if (!isPaused) {

await startSolvingAnimation();

}

}

async function startSolvingAnimation() {

const algorithm = document.getElementById('pathAlgorithm').value;

let done = false;

let pathFound = false;

while (!done && isSolving && !isPaused) {

switch(algorithm) {

case 'dijkstra':

done = await solvingStepDijkstra();

break;

case 'dfs':

done = await solvingStepDFS();

break;

case 'astar':

default:

done = await solvingStepAStar();

break;

}

statistics.incrementStep();

const visitedCount = closedNodes.length;

statistics.updateVisitedCells(visitedCount);

redrawMaze();

if (!isPaused && !done) {

await new Promise(resolve => setTimeout(resolve, 101 - animationSpeed));

}

}

if (done) {

isSolving = false;

statistics.stop();

}

return pathFound;

}

// solves maze using A\* algorithm

async function solveMazeAStar() {

openNodes = [start];

start.g = 0;

start.f = heuristic(start, end);

while (openNodes.length > 0 && isSolving) {

statistics.incrementStep();

let current = findMinFScore();

if (current === end) {

path = reconstructPath(end);

await renderPath(path);

break;

}

openNodes = openNodes.filter(cell => cell !== current);

closedNodes.push(current);

current.visited = true;

let neighbours = obtainValidNeighbours(current);

for (let neighbour of neighbours) {

if (closedNodes.includes(neighbour)) continue;

let tempG = current.g + 1;

if (!openNodes.includes(neighbour)) {

openNodes.push(neighbour);

} else if (tempG >= neighbour.g) {

continue;

}

neighbour.previous = current;

neighbour.g = tempG;

neighbour.f = neighbour.g + heuristic(neighbour, end);

}

redrawMaze();

await new Promise(resolve => setTimeout(resolve, 101 - animationSpeed));

}

isSolving = false;

statistics.stop();

}

// gets valid neighbouring cells for pathfinding

function obtainValidNeighbours(cell) {

let neighbours = [];

let i = cell.i;

let j = cell.j;

if (!cell.walls[0] && j > 0) neighbours.push(grid[index(i, j-1)]); // top

if (!cell.walls[1] && i < gridColumns-1) neighbours.push(grid[index(i+1, j)]); // right

if (!cell.walls[2] && j < gridRows-1) neighbours.push(grid[index(i, j+1)]); // bottom

if (!cell.walls[3] && i > 0) neighbours.push(grid[index(i-1, j)]); // left

return neighbours;

}

function findMinFScore() {

let lowest = openNodes[0];

for (let i = 1; i < openNodes.length; i++) {

if (openNodes[i].f < lowest.f) {

lowest = openNodes[i];

}

}

return lowest;

}

// calculates heuristic distance for A\* algorithm im using manhattan distance

function heuristic(a, b) {

return Math.abs(a.i - b.i) + Math.abs(a.j - b.j);

}

// solves maze using Dijkstra's algorithm

async function solveMazeDijkstra() {

if (isGenerating || isSolving) return;

isSolving = true;

grid.forEach(cell => {

cell.distance = Infinity;

cell.previous = null;

cell.visited = false;

});

start.distance = 0;

let unvisited = [...grid];

while (unvisited.length > 0) {

let current = unvisited.reduce((min, cell) =>

cell.distance < min.distance ? cell : min, unvisited[0]);

if (current === end || current.distance === Infinity) break;

unvisited.splice(unvisited.indexOf(current), 1);

let neighbours = [];

if (!current.walls[0]) neighbours.push(grid[index(current.i, current.j-1)]);

if (!current.walls[1]) neighbours.push(grid[index(current.i+1, current.j)]);

if (!current.walls[2]) neighbours.push(grid[index(current.i, current.j+1)]);

if (!current.walls[3]) neighbours.push(grid[index(current.i-1, current.j)]);

for (let neighbour of neighbours) {

if (!neighbour) continue;

let alt = current.distance + 1;

if (alt < neighbour.distance) {

neighbour.distance = alt;

neighbour.previous = current;

}

}

current.show(true);

await new Promise(resolve => setTimeout(resolve, 101 - animationSpeed));

}

let path = reconstructPath(end);

await renderPath(path);

isSolving = false;

}

// solves maze using Depth-First Search

async function solveMazeDFS() {

if (isGenerating || isSolving) return;

isSolving = true;

grid.forEach(cell => {

cell.visited = false;

cell.previous = null;

});

let stack = [start];

let found = false;

while (stack.length > 0 && isSolving && !found) {

statistics.incrementStep();

let current = stack.pop();

current.visited = true;

if (current === end) {

found = true;

break;

}

let neighbours = obtainValidNeighbours(current);

for (let neighbour of neighbours.reverse()) {

if (!neighbour.visited) {

neighbour.previous = current;

stack.push(neighbour);

}

}

const visitedCount = grid.filter(cell => cell.visited).length;

statistics.updateVisitedCells(visitedCount);

redrawMaze();

if (!isPaused) {

await new Promise(resolve => setTimeout(resolve, 101 - animationSpeed));

}

}

if (found) {

path = reconstructPath(end);

await renderPath(path);

}

isSolving = false;

statistics.stop();

}

// reconstructs path from end to start

function reconstructPath(endCell) {

let path = [];

let current = endCell;

while (current) {

path.unshift(current);

current = current.previous;

}

console.log("Reconstructed Path:", path);

return path;

}

async function renderPath(path) {

if (!path) return;

for (let i = 0; i < path.length - 1; i++) {

let current = path[i];

let next = path[i + 1];

context.beginPath();

context.strokeStyle = colors.path;

context.lineWidth = cellSize / 4;

context.moveTo(current.i \* cellSize + cellSize/2, current.j \* cellSize + cellSize/2);

context.lineTo(next.i \* cellSize + cellSize/2, next.j \* cellSize + cellSize/2);

context.stroke();

if (!isPaused) {

await new Promise(resolve => setTimeout(resolve, 50));

}

}

}

// speed control

document.getElementById('speedControl').addEventListener('input', function(e) {

animationSpeed = parseInt(e.target.value);

});

// generate initial maze

document.addEventListener('DOMContentLoaded', function() {

initialiseGrid();

displayEmptyGrid();

updateMaxValues();

document.getElementById('resetButton').addEventListener('click', resetMaze);

document.getElementById('instantButton').addEventListener('click', instantComplete);

const sizeSelect = document.getElementById('mazeSize');

sizeSelect.addEventListener('change', updateMazeSize);

document.getElementById('applyCustomSize').addEventListener('click', function() {

const width = parseInt(document.getElementById('customWidth').value);

const height = parseInt(document.getElementById('customHeight').value);

if (width < 5 || width > 100 || height < 5 || height > 100) {

alert('Please enter dimensions between 5 and 100');

return;

}

resizeMaze(width, height);

});

document.addEventListener('keydown', function(event) {

if (event.key === 'Escape') {

document.getElementById('helpModal').style.display = 'none';

}

});

});

function initialiseGrid() {

grid = [];

for (let j = 0; j < gridRows; j++) {

for (let i = 0; i < gridColumns; i++) {

grid.push(new Cell(i, j));

}

}

start = grid[0];

end = grid[grid.length - 1];

}

function displayEmptyGrid() {

context.fillStyle = colors.background;

context.fillRect(0, 0, mazeCanvas.width, mazeCanvas.height);

grid.forEach(cell => cell.show());

}

const algorithms = {

generation: {

RECURSIVE\_BACKTRACKER: 'recursive\_backtracker',

PRIMS: 'prims',

HUNT\_AND\_KILL: 'hunt\_and\_kill'

},

pathfinding: {

ASTAR: 'astar',

DIJKSTRA: 'dijkstra',

DFS: 'dfs'

}

};

let currentGenAlgorithm = algorithms.generation.RECURSIVE\_BACKTRACKER;

let currentPathAlgorithm = algorithms.pathfinding.ASTAR;

// helper functions

function addFrontierCells(cell, frontierCells) {

let neighbours = getUnvisitedNeighbours(cell);

for (let neighbour of neighbours) {

if (!frontierCells.includes(neighbour)) {

frontierCells.push(neighbour);

}

}

}

// returns array of visited neighbours in all four directions

function getVisitedNeighbours(cell) {

let neighbours = [];

let possibleNeighbours = [

grid[index(cell.i, cell.j - 1)], // Top

grid[index(cell.i + 1, cell.j)], // Right

grid[index(cell.i, cell.j + 1)], // Bottom

grid[index(cell.i - 1, cell.j)] // Left

];

for (let neighbour of possibleNeighbours) {

if (neighbour && neighbour.visited) {

neighbours.push(neighbour);

}

}

return neighbours;

}

function hasVisitedNeighbour(cell) {

return getVisitedNeighbours(cell).length > 0;

}

// returns random visited neighbour from available neighbours

function getRandomVisitedNeighbour(cell) {

let neighbours = getVisitedNeighbours(cell);

return neighbours[Math.floor(Math.random() \* neighbours.length)];

}

async function renderPath(path) {

context.beginPath();

context.strokeStyle = colors.path;

context.lineWidth = cellSize/3;

context.lineCap = 'round';

context.lineJoin = 'round';

context.shadowColor = colors.path;

context.shadowBlur = 5;

for (let i = 0; i < path.length - 1; i++) {

context.beginPath();

context.moveTo(

path[i].i \* cellSize + cellSize/2,

path[i].j \* cellSize + cellSize/2

);

context.lineTo(

path[i+1].i \* cellSize + cellSize/2,

path[i+1].j \* cellSize + cellSize/2

);

context.stroke();

await new Promise(resolve => setTimeout(resolve, 101 - animationSpeed));

}

context.shadowBlur = 0;

}

// tour

function startTour() {

introJs().setOptions({

steps: [

{

element: document.querySelector('.control-group:first-child'),

title: 'Generation Algorithm',

intro: 'Select an algorithm to generate your maze. Each creates different patterns.'

},

{

element: document.querySelector('.control-group:nth-child(2)'),

title: 'Solving Algorithm',

intro: 'Choose how to solve the maze. Each algorithm has different strategies.'

},

{

element: '.size-controls',

title: 'Maze Size',

intro: 'Choose the size of your maze or enter custom dimensions.'

},

{

element: '.custom-points-controls',

title: 'Start and End Points',

intro: 'Set custom start and end points for the maze.'

},

{

element: '.playback-controls',

title: 'Playback Controls',

intro: 'Control the animation: Complete Instantly, Reset or save Maze.'

},

{

element: '.speed-control',

title: 'Animation Speed',

intro: 'Adjust how fast the algorithms run.'

},

{

element: '.statistics-panel',

title: 'Statistics',

intro: 'View real-time information: time elapsed, steps taken, and completion progress.'

},

{

element: '#mazeCanvas',

title: 'Maze Display',

intro: 'The maze is displayed here. Green cell is start, red cell is end, and blue shows the solution path.'

},

{

title: 'Ready to Start',

intro: 'Generate a maze and try solving it with different algorithms!'

}

],

tooltipClass: 'customTooltip',

highlightClass: 'customHighlight',

exitOnOverlayClick: false,

showStepNumbers: true,

showBullets: true,

showProgress: true,

disableInteraction: false

}).start();

}

// help menu

function toggleHelpMenu() {

const modal = document.getElementById('helpModal');

modal.style.display = modal.style.display === 'block' ? 'none' : 'block';

}

// close when click x

document.querySelector('.close').onclick = function() {

document.getElementById('helpModal').style.display = 'none';

}

// close when click outside

window.onclick = function(event) {

const modal = document.getElementById('helpModal');

if (event.target === modal) {

modal.style.display = 'none';

}

}

function resetMaze() {

currentStep = 0;

currentOperation = null;

visitedCells = 0;

totalCells = gridColumns \* gridRows;

if (statistics) {

statistics.reset();

}

initialiseGrid();

defineBoundaries();

displayEmptyGrid();

}

function redrawMaze() {

context.fillStyle = colors.background;

context.fillRect(0, 0, mazeCanvas.width, mazeCanvas.height);

grid.forEach(cell => cell.show());

if (current) current.show(true);

}

// handle maze size changes

function updateMazeSize() {

const sizeSelect = document.getElementById('mazeSize');

const customSizeInputs = document.getElementById('customSizeInputs');

if (sizeSelect.value === 'custom') {

customSizeInputs.style.display = 'flex';

return;

}

customSizeInputs.style.display = 'none';

const size = parseInt(sizeSelect.value);

if (!isNaN(size)) {

resizeMaze(size, size);

}

}

function updateMaxValues() {

document.getElementById('maxRowStart').innerText = gridRows - 1;

document.getElementById('maxColStart').innerText = gridColumns - 1;

document.getElementById('maxRowEnd').innerText = gridRows - 1;

document.getElementById('maxColEnd').innerText = gridColumns - 1;

document.getElementById('endRow').max = gridRows - 1;

document.getElementById('endCol').max = gridColumns - 1;

document.getElementById('startRow').max = gridRows - 1;

document.getElementById('startCol').max = gridColumns - 1;

document.getElementById('endRow').value = gridRows - 1;

document.getElementById('endCol').value = gridColumns - 1;

}

function resizeMaze(newgridColumns, newgridRows) {

log(`Resizing maze to ${newgridColumns}x${newgridRows}`, 'info');

gridColumns = newgridColumns;

gridRows = newgridRows;

const maxWidth = Math.min(window.innerWidth \* 0.8, 800);

const maxHeight = Math.min(window.innerHeight \* 0.6, 600);

const cellWidth = Math.floor(maxWidth / gridColumns);

const cellHeight = Math.floor(maxHeight / gridRows);

cellSize = Math.min(cellWidth, cellHeight, 40);

mazeCanvas.width = gridColumns \* cellSize;

mazeCanvas.height = gridRows \* cellSize;

resetMaze();

updateMaxValues();

}

// update initialiseGrid function to use current dimensions

function initialiseGrid() {

log('Initialising grid', 'info');

grid = [];

for (let j = 0; j < gridRows; j++) {

for (let i = 0; i < gridColumns; i++) {

grid.push(new Cell(i, j));

}

}

start = grid[0];

end = grid[grid.length - 1];

}

// instant completion

async function instantComplete() {

if (!currentOperation) return;

const instantButton = document.getElementById('instantButton');

instantButton.disabled = true;

isPaused = true;

cancelAnimationFrame(animationFrame);

try {

if (currentOperation === 'generating') {

initializeGrid();

await completeGeneration();

} else if (currentOperation === 'solving') {

openNodes = [];

closedNodes = [];

path = [];

grid.forEach(cell => {

cell.visited = false;

cell.distance = Infinity;

cell.previous = null;

cell.f = 0;

cell.g = 0;

cell.h = 0;

});

const algorithm = document.getElementById('pathAlgorithm').value;

switch(algorithm) {

case 'dijkstra':

start.distance = 0;

openNodes = [start];

break;

case 'dfs':

openNodes = [start];

break;

case 'astar':

default:

start.g = 0;

start.f = heuristic(start, end);

openNodes = [start];

break;

}

await completeSolving();

}

} catch (error) {

console.error('Error during instant completion:', error);

} finally {

instantButton.disabled = false;

redrawMaze();

}

}

// instantly completes maze generation

async function completeGeneration() {

const algorithm = document.getElementById('genAlgorithm').value;

let steps = 0;

try {

switch(algorithm) {

case 'prims':

steps = await completePrimsGeneration();

break;

case 'hunt\_and\_kill':

steps = await completeHuntAndKillGeneration();

break;

case 'recursive\_backtracker':

default:

steps = await completeRecursiveBacktracker();

break;

}

statistics.setInstantComplete(steps);

redrawMaze();

isGenerating = false;

currentOperation = null;

return steps;

} catch (error) {

console.error('Error in maze generation:', error);

isGenerating = false;

currentOperation = null;

throw error;

}

}

// instantly completes maze solving using selected algorithm

async function completeSolving() {

const algorithm = document.getElementById('pathAlgorithm').value;

let steps = 0;

try {

grid.forEach(cell => {

cell.visited = false;

cell.distance = Infinity;

cell.previous = null;

cell.f = 0;

cell.g = 0;

cell.h = 0;

});

switch(algorithm) {

case 'dijkstra':

steps = await completeDijkstraSolve();

break;

case 'dfs':

steps = await completeDFSSolve();

break;

case 'astar':

steps = await completeAStarSolve();

break;

}

statistics.setInstantComplete(steps);

redrawMaze();

isSolving = false;

currentOperation = null;

return steps;

} catch (error) {

console.error('Error in maze solving:', error);

isSolving = false;

currentOperation = null;

throw error;

}

}

async function completeAStarSolve() {

let steps = 0;

while (openNodes.length > 0) {

steps++;

let current = findMinFScore();

if (current === end) {

await renderPath(reconstructPath(end));

break;

}

openNodes = openNodes.filter(cell => cell !== current);

closedNodes.push(current);

let neighbours = obtainValidNeighbours(current);

for (let neighbour of neighbours) {

if (closedNodes.includes(neighbour)) continue;

let tempG = current.g + 1;

if (!openNodes.includes(neighbour)) {

openNodes.push(neighbour);

} else if (tempG >= neighbour.g) {

continue;

}

neighbour.previous = current;

neighbour.g = tempG;

neighbour.f = neighbour.g + heuristic(neighbour, end);

}

}

return steps;

}

async function completeDijkstraSolve() {

let steps = 0;

let unvisited = [...grid];

start.distance = 0;

while (unvisited.length > 0) {

steps++;

let current = unvisited.reduce((min, cell) =>

cell.distance < min.distance ? cell : min, unvisited[0]);

if (current === end || current.distance === Infinity) break;

unvisited = unvisited.filter(cell => cell !== current);

current.visited = true;

let neighbours = obtainValidNeighbours(current);

for (let neighbour of neighbours) {

if (!neighbour.visited) {

let alt = current.distance + 1;

if (alt < neighbour.distance) {

neighbour.distance = alt;

neighbour.previous = current;

}

}

}

}

if (end.previous) {

await renderPath(reconstructPath(end));

}

return steps;

}

async function completeDFSSolve() {

let steps = 0;

let stack = [start];

start.visited = true;

while (stack.length > 0) {

steps++;

let current = stack.pop();

if (current === end) {

path = reconstructPath(end);

await renderPath(path);

break;

}

let neighbours = obtainValidNeighbours(current);

for (let neighbour of neighbours) {

if (!neighbour.visited) {

neighbour.visited = true;

neighbour.previous = current;

stack.push(neighbour);

}

}

}

return steps;

}

// calculate theoretical time

function calculateTheoreticalTime(steps) {

const speed = document.getElementById('speedControl').value;

const delayPerStep = 101 - speed;

const totalTime = steps \* delayPerStep;

return totalTime / 1000;

}

async function completePrimsGeneration() {

let steps = 0;

let frontierCells = new Set();

let inMaze = new Set();

start.inMaze = true;

inMaze.add(start);

addToFrontier(start, frontierCells, inMaze);

while (frontierCells.size > 0) {

steps++;

let frontierArray = Array.from(frontierCells);

let randomIndex = Math.floor(Math.random() \* frontierArray.length);

let current = frontierArray[randomIndex];

frontierCells.delete(current);

let mazeNeighbours = getMazeNeighbours(current, inMaze);

if (mazeNeighbours.length > 0) {

let randomNeighbour = mazeNeighbours[Math.floor(Math.random() \* mazeNeighbours.length)];

deleteWalls(current, randomNeighbour);

current.inMaze = true;

current.visited = true;

inMaze.add(current);

addToFrontier(current, frontierCells, inMaze);

}

}

return steps;

}

// adds cell to frontier

function addToFrontier(cell, frontierCells, inMaze) {

let i = cell.i;

let j = cell.j;

if (j > 0) checkAndAddFrontier(grid[index(i, j-1)], frontierCells, inMaze);

if (i < gridColumns-1) checkAndAddFrontier(grid[index(i+1, j)], frontierCells, inMaze);

if (j < gridRows-1) checkAndAddFrontier(grid[index(i, j+1)], frontierCells, inMaze);

if (i > 0) checkAndAddFrontier(grid[index(i-1, j)], frontierCells, inMaze);

}

// adds cell to frontier if it's not in maze and not already in frontier

function checkAndAddFrontier(cell, frontierCells, inMaze) {

if (cell && !inMaze.has(cell) && !frontierCells.has(cell)) {

frontierCells.add(cell);

}

}

// gets maze neighbours for Hunt and Kill algorithm

function getMazeNeighbours(cell, inMaze) {

let neighbours = [];

let i = cell.i;

let j = cell.j;

if (j > 0 && inMaze.has(grid[index(i, j-1)])) neighbours.push(grid[index(i, j-1)]);

if (i < gridColumns-1 && inMaze.has(grid[index(i+1, j)])) neighbours.push(grid[index(i+1, j)]);

if (j < gridRows-1 && inMaze.has(grid[index(i, j+1)])) neighbours.push(grid[index(i, j+1)]);

if (i > 0 && inMaze.has(grid[index(i-1, j)])) neighbours.push(grid[index(i-1, j)]);

return neighbours;

}

async function completeHuntAndKillGeneration() {

let steps = 0;

let currentCell = start;

let unvisited = new Set(grid);

while (unvisited.size > 0) {

steps++;

if (currentCell) {

currentCell.visited = true;

unvisited.delete(currentCell);

let next = getRandomUnvisitedNeighbour(currentCell, unvisited);

if (next) {

deleteWalls(currentCell, next);

currentCell = next;

} else {

currentCell = huntForNext(unvisited);

}

} else {

currentCell = huntForNext(unvisited);

if (!currentCell) break;

}

}

return steps;

}

// gets random unvisited neighbour

function getRandomUnvisitedNeighbour(cell, unvisited) {

let neighbours = [];

let i = cell.i;

let j = cell.j;

if (j > 0 && unvisited.has(grid[index(i, j-1)])) neighbours.push(grid[index(i, j-1)]);

if (i < gridColumns-1 && unvisited.has(grid[index(i+1, j)])) neighbours.push(grid[index(i+1, j)]);

if (j < gridRows-1 && unvisited.has(grid[index(i, j+1)])) neighbours.push(grid[index(i, j+1)]);

if (i > 0 && unvisited.has(grid[index(i-1, j)])) neighbours.push(grid[index(i-1, j)]);

return neighbours.length > 0 ? neighbours[Math.floor(Math.random() \* neighbours.length)] : null;

}

// hunts for next cell to connect

function huntForNext(unvisited) {

for (let cell of unvisited) {

let visitedNeighbours = getVisitedNeighbours(cell);

if (visitedNeighbours.length > 0) {

let randomNeighbour = visitedNeighbours[Math.floor(Math.random() \* visitedNeighbours.length)];

deleteWalls(cell, randomNeighbour);

return cell;

}

}

return null;

}

async function completeRecursiveBacktracker() {

let steps = 0;

let currentCell = start;

let stack = [];

let unvisited = new Set(grid);

while (unvisited.size > 0) {

steps++;

currentCell.visited = true;

unvisited.delete(currentCell);

let next = getRandomUnvisitedNeighbour(currentCell, unvisited);

if (next) {

stack.push(currentCell);

deleteWalls(currentCell, next);

currentCell = next;

} else if (stack.length > 0) {

currentCell = stack.pop();

} else {

if (unvisited.size > 0) {

let newCell = unvisited.values().next().value;

let visitedNeighbour = getVisitedNeighbours(newCell)[0];

deleteWalls(newCell, visitedNeighbour);

currentCell = newCell;

}

}

}

return steps;

}

// debugging and error Tracking

let debugLog = [];

function log(message, type = 'info') {

const timestamp = new Date().toLocaleTimeString();

const logEntry = { timestamp, type, message };

debugLog.push(logEntry);

console.log(`[${timestamp}] ${type.toUpperCase()}: ${message}`);

updateDebugLogDisplay();

}

function clearDebugLog() {

debugLog = [];

updateDebugLogDisplay();

}

function updateDebugLogDisplay() {

const logDisplay = document.getElementById('debugLogDisplay');

if (!logDisplay) return;

logDisplay.innerHTML = '';

debugLog.forEach(logEntry => {

const logItem = document.createElement('div');

logItem.classList.add('log-item', logEntry.type);

logItem.innerHTML = `

<span class="log-timestamp">${logEntry.timestamp}</span>

<span class="log-type">${logEntry.type.toUpperCase()}</span>

<span class="log-message">${logEntry.message}</span>

`;

logDisplay.appendChild(logItem);

});

logDisplay.scrollTop = logDisplay.scrollHeight;

}

function exportMazeAsPNG() {

const dataURL = mazeCanvas.toDataURL("image/png");

const link = document.createElement('a');

link.href = dataURL;

link.download = 'maze.png';

document.body.appendChild(link);

link.click();

document.body.removeChild(link);

}

async function solvingStepDijkstra() {

if (openNodes.length > 0) {

let current = openNodes.reduce((min, cell) =>

cell.distance < min.distance ? cell : min, openNodes[0]);

if (current === end) {

path = reconstructPath(end);

await renderPath(path);

return true;

}

openNodes = openNodes.filter(cell => cell !== current);

closedNodes.push(current);

current.visited = true;

let neighbours = obtainValidNeighbours(current);

for (let neighbour of neighbours) {

if (!closedNodes.includes(neighbour)) {

let tentativeDistance = current.distance + 1;

if (!openNodes.includes(neighbour)) {

openNodes.push(neighbour);

} else if (tentativeDistance >= neighbour.distance) {

continue;

}

neighbour.previous = current;

neighbour.distance = tentativeDistance;

}

}

return false;

}

return true;

}

async function solvingStepDFS() {

if (openNodes.length > 0) {

let current = openNodes.pop();

current.visited = true;

if (current === end) {

path = reconstructPath(end);

await renderPath(path);

return true;

}

closedNodes.push(current);

let neighbours = obtainValidNeighbours(current);

for (let neighbour of neighbours) {

if (!closedNodes.includes(neighbour) && !openNodes.includes(neighbour)) {

neighbour.previous = current;

openNodes.push(neighbour);

}

}

return false;

}

return true;

}

async function solvingStepAStar() {

if (openNodes.length > 0) {

let current = findMinFScore();

if (current === end) {

path = reconstructPath(end);

await renderPath(path);

return true;

}

openNodes = openNodes.filter(cell => cell !== current);

closedNodes.push(current);

current.visited = true;

let neighbours = obtainValidNeighbours(current);

for (let neighbour of neighbours) {

if (closedNodes.includes(neighbour)) continue;

let tempG = current.g + 1;

if (!openNodes.includes(neighbour)) {

openNodes.push(neighbour);

} else if (tempG >= neighbour.g) {

continue;

}

neighbour.previous = current;

neighbour.g = tempG;

neighbour.f = neighbour.g + heuristic(neighbour, end);

}

return false;

}

return true;

}

Style.css

:root {

--primary-blue: #2196F3;

--primary-blue-dark: #1976D2;

--primary-black: #1a1a1a;

--secondary-black: #2a2a2a;

}

body {

background: var(--primary-black);

display: flex;

flex-direction: column;

align-items: center;

justify-content: center;

min-height: 100vh;

margin: 0;

font-family: 'Arial', sans-serif;

}

canvas {

border: 2px solid #4a4a4a;

border-radius: 8px;

box-shadow: 0 0 20px rgba(0, 150, 255, 0.3);

background: #000;

}

.controls {

margin: 20px;

display: flex;

flex-direction: column;

gap: 20px;

align-items: center;

padding-top: 20px;

}

.main-controls {

display: flex;

gap: 20px;

}

.control-group {

display: flex;

flex-direction: column;

gap: 10px;

align-items: center;

}

.algorithm-select {

padding: 10px 15px;

border-radius: 25px;

border: 2px solid #2196F3;

background: #2a2a2a;

color: white;

font-size: 14px;

cursor: pointer;

width: 200px;

text-align: center;

outline: none;

transition: all 0.3s ease;

}

.algorithm-select:hover {

border-color: #1976D2;

box-shadow: 0 0 10px rgba(33, 150, 243, 0.3);

}

.algorithm-select option {

background: #2a2a2a;

color: white;

padding: 10px;

}

button {

padding: 12px 24px;

font-size: 16px;

border: none;

border-radius: 25px;

cursor: pointer;

transition: all 0.3s ease;

background: #2196F3;

color: white;

text-transform: uppercase;

letter-spacing: 1px;

box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);

width: 200px;

}

button:hover {

transform: translateY(-2px);

box-shadow: 0 6px 8px rgba(0, 0, 0, 0.2);

background: #1976D2;

}

button:active {

transform: translateY(0);

}

.speed-control {

display: flex;

align-items: center;

color: white;

gap: 10px;

background: #2a2a2a;

padding: 10px 20px;

border-radius: 25px;

border: 2px solid #2196F3;

margin-top: 10px;

}

input[type="range"] {

width: 150px;

accent-color: #2196F3;

}

/\* Media Queries \*/

@media (max-width: 768px) {

.main-controls {

flex-direction: column;

}

}

@media (max-width: 480px) {

.algorithm-select,

button {

width: 150px;

}

}

.header {

display: flex;

justify-content: space-between;

align-items: center;

width: 100%;

max-width: 800px;

margin-bottom: 20px;

padding: 0 20px;

box-sizing: border-box;

}

.header h1 {

color: white;

margin: 0;

font-size: 24px;

}

.header-buttons {

display: flex;

gap: 10px;

}

.help-button {

padding: 8px 16px;

width: auto;

font-size: 14px;

background: #3f51b5;

}

.help-button:hover {

background: #303f9f;

}

.modal {

display: none;

position: fixed;

z-index: 1000;

left: 0;

top: 0;

width: 100%;

height: 100%;

background-color: rgba(0,0,0,0);

}

.modal-content {

background-color: #2a2a2a;

margin: 5% auto;

padding: 20px;

border: 2px solid #4a4a4a;

border-radius: 8px;

width: 80%;

max-width: 600px;

max-height: 80vh;

overflow-y: auto;

color: white;

box-shadow: 0 0 20px rgba(0, 150, 255, 0.3);

}

.close {

color: #aaa;

float: right;

font-size: 28px;

font-weight: bold;

cursor: pointer;

}

.close:hover {

color: white;

}

.help-section {

margin-bottom: 20px;

padding: 15px;

background: #1a1a1a;

border-radius: 8px;

}

.help-section h3 {

color: #2196F3;

margin-top: 0;

}

.help-section ul {

list-style-type: none;

padding-left: 0;

}

.help-section li {

margin-bottom: 10px;

line-height: 1.4;

}

.help-section strong {

color: #64b5f6;

}

/\* Tooltip customisation \*/

[title] {

position: relative;

}

/\* Intro.js customisation \*/

.introjs-tooltip {

background: #ffffff;

color: #000;

border-radius: 8px;

box-shadow: 0 2px 10px rgba(0, 0, 0, 0.2);

}

.introjs-tooltiptext {

color: #000;

}

.introjs-tooltipbuttons {

border-top: 1px solid #e0e0e0;

}

.introjs-button {

background: #2196F3;

color: white;

border: none;

text-shadow: none;

border-radius: 25px;

padding: 8px 16px;

}

.introjs-button:hover {

background: #1976D2;

color: white;

}

.introjs-helperLayer {

background-color: rgba(255, 255, 255, 0.1);

}

.introjs-tooltipReferenceLayer \* {

color: #000;

}

.introjs-arrow {

border-color: #ffffff;

}

.introjs-tooltip-title {

color: #000;

}

.introjs-tooltip-header {

color: #000;

}

@media (max-width: 768px) {

.header {

flex-direction: column;

text-align: center;

gap: 15px;

}

.modal-content {

width: 95%;

margin: 10% auto;

}

}

.playback-controls {

display: flex;

gap: 10px;

margin: 15px 0;

}

.control-button {

padding: 8px 16px;

width: auto;

min-width: 100px;

font-size: 14px;

background: #2196F3;

display: flex;

align-items: center;

justify-content: center;

gap: 5px;

}

.control-button:disabled {

background: #666;

cursor: not-allowed;

transform: none;

}

.control-button i {

font-size: 16px;

}

/\* Statistics Panel \*/

.statistics-panel {

background: var(--secondary-black);

border-radius: 8px;

padding: 15px;

margin: 15px 0;

box-shadow: 0 2px 10px rgba(0, 0, 0, 0.1);

width: 100%;

max-width: 600px;

display: flex;

flex-wrap: wrap;

gap: 15px;

color: white;

}

.stat-group {

flex: 1;

min-width: 250px;

display: flex;

flex-direction: column;

gap: 10px;

}

.stat-item {

display: flex;

align-items: center;

gap: 10px;

padding: 8px;

background: var(--primary-black);

border-radius: 6px;

font-size: 14px;

color: white;

}

.stat-item i {

color: var(--primary-blue);

width: 20px;

text-align: center;

}

.stat-item span:last-child {

margin-left: auto;

font-weight: bold;

color: var(--primary-blue);

}

@media (max-width: 768px) {

.statistics-panel {

flex-direction: column;

}

.stat-group {

min-width: unset;

}

}

/\* size controls \*/

.size-controls {

display: flex;

flex-direction: column;

gap: 10px;

margin: 15px 0;

background: var(--secondary-black);

padding: 15px;

border-radius: 8px;

box-shadow: 0 2px 10px rgba(0, 0, 0, 0.1);

color: white;

}

.size-group {

display: flex;

align-items: center;

gap: 10px;

}

.size-select {

padding: 8px 15px;

border-radius: 25px;

border: 2px solid var(--primary-blue);

background: var(--secondary-black);

color: white;

font-size: 14px;

cursor: pointer;

outline: none;

transition: all 0.3s ease;

}

.size-select:hover {

border-color: #1976D2;

box-shadow: 0 0 10px rgba(33, 150, 243, 0.3);

}

.custom-size {

display: flex;

gap: 15px;

align-items: center;

flex-wrap: wrap;

}

.size-input-group {

display: flex;

align-items: center;

gap: 8px;

}

.size-input-group input {

width: 60px;

padding: 6px 10px;

border: 2px solid var(--primary-blue);

border-radius: 15px;

font-size: 14px;

outline: none;

background: var(--secondary-black);

color: white;

}

.size-button {

padding: 6px 15px;

font-size: 14px;

background: #2196F3;

color: white;

border: none;

border-radius: 15px;

cursor: pointer;

transition: all 0.3s ease;

}

.size-button:hover {

background: #1976D2;

transform: translateY(-2px);

}

@media (max-width: 768px) {

.custom-size {

flex-direction: column;

align-items: flex-start;

}

}

/\* instant button \*/

.instant-button {

background: #ff9800 !important;

}

.instant-button:hover {

background: #f57c00 !important;

}

.instant-button:disabled {

background: #666 !important;

}

/\* help section \*/

.help-section {

background: var(--secondary-black);

border-radius: 8px;

padding: 15px;

margin-bottom: 15px;

color: white;

}

.help-section h3 {

color: var(--primary-blue);

margin-top: 0;

margin-bottom: 10px;

font-size: 18px;

}

.help-section ul {

margin: 0;

padding-left: 20px;

}

.help-section li {

margin-bottom: 8px;

line-height: 1.4;

}

.help-section strong {

color: var(--primary-blue);

}

/\* tooltip styles \*/

[title] {

position: relative;

}

[title]:hover::after {

content: attr(title);

position: absolute;

bottom: 100%;

left: 50%;

transform: translateX(-50%);

padding: 5px 10px;

background: rgba(0, 0, 0, 0.8);

color: white;

border-radius: 4px;

font-size: 12px;

white-space: nowrap;

z-index: 1000;

pointer-events: none;

}

/\* Intro.js customisation \*/

.introjs-tooltip {

background: #ffffff;

color: #000;

border-radius: 8px;

box-shadow: 0 2px 10px rgba(0, 0, 0, 0.2);

}

.introjs-tooltiptext {

color: #000;

}

.introjs-tooltip-title,

.introjs-tooltip-header {

color: #000;

}

.introjs-arrow {

border-color: #ffffff;

}

.custom-points-controls {

display: flex;

gap: 20px;

margin-top: 20px;

}

.custom-points-controls .control-group {

display: flex;

flex-direction: column;

align-items: center;

}

.custom-points-controls label {

color: white;

margin-bottom: 5px;

}

.custom-points-controls input[type="number"] {

width: 60px;

padding: 8px;

border-radius: 5px;

border: 1px solid #4a4a4a;

background-color: #333;

color: white;

text-align: center;

}

.custom-point-button {

padding: 8px 16px;

font-size: 14px;

border: none;

border-radius: 5px;

cursor: pointer;

transition: all 0.3s ease;

background: #673ab7;

color: white;

text-transform: uppercase;

letter-spacing: 0.5px;

}

.custom-point-button:hover {

background: #512da8;

}

/\* Debug Panel \*/

.debug-panel {

background: #333;

border-radius: 8px;

padding: 15px;

margin: 15px 0;

box-shadow: 0 2px 10px rgba(0, 0, 0, 0.2);

width: 100%;

max-width: 800px;

color: #eee;

font-size: 14px;

font-family: monospace;

box-sizing: border-box;

}

.debug-panel h3 {

color: #f44336;

margin-top: 0;

margin-bottom: 10px;

font-size: 16px;

}

.debug-log-display {

height: 200px;

overflow-y: scroll;

padding: 10px;

border: 1px solid #555;

border-radius: 6px;

background: #222;

white-space: pre-wrap;

}

.log-item {

margin-bottom: 5px;

padding: 5px;

border-radius: 4px;

background: #444;

display: flex;

gap: 10px;

align-items: center;

}

.log-timestamp {

color: #999;

font-size: 12px;

}

.log-type {

font-weight: bold;

text-transform: uppercase;

}

.log-message {

flex: 1;

}

.log-item.info {

background: #444;

}

.log-item.warn {

background: #ff980020;

}

.log-item.error {

background: #f4433620;

}

.clear-log-button {

padding: 8px 16px;

font-size: 14px;

border: none;

border-radius: 5px;

cursor: pointer;

transition: all 0.3s ease;

background: #607d8b;

color: white;

text-transform: uppercase;

letter-spacing: 0.5px;

margin-top: 10px;

}

.clear-log-button:hover {

background: #455a64;

}