Design

# Implemented Packages

## gRPC Packages

gRPC is built across 4 packages:

* Grpc.AspNetCore is for the server.
* Google.Protobuf is for the client.
* Grpc.Net.Client is for the client.
* Grpc.Tools is for the client.

I have chosen to build the network using gRPC. This significantly changes the style of the solution. The system will be implemented via 2 separate projects: a client and server. A protos folder containing protocol files will be shared between these projects so the client and server know which services are available, and what the proper request/response objects for these calls are. The server will additionally have a folder defining classes for each of its services, with each protocol file requiring its own class. These classes are mapped to the server when it is run.

## Json Parsing

I will implement Newtonsoft.Json in my project to facilitate the transfer of objects across the network. In gRPC, data is wrapped in a collection defined by a protocol file to be sent across the network. However, these protocol files only support simple data structures, so to share Maze-derivative objects with the server I will have to parse them into a Json format first. I can use the JsonConvert class from this package to convert these. However, I will have to implement rigorous error handling, particularly for deserializing Json objects, as the object may have null components if inappropriate access modifiers are used for properties.

In addition, Json parsing in this way should allow me to make the client even more lightweight: both the client and server will use the Maze, Coordinate, and all child classes, but only the server will need to call their functions, which are purely to build the maze. Therefore, I should not need to provide the client with anything but the class’ properties to build the Json object.

## Database Management

I plan to use the System.Data.SQLite package to manage the databases used in the system. The database file itself can be stored on the server only; if the user needs data from a table, it can be accessed in a controlled way via a network request from their client. Additionally, this package allows the use of a tag system when creating an SQL command, helping prevent SQL injection attacks when using parameterized SQL with user input, such as storing login information.

## Data Visualization

Since the System.Windows.Forms.DataVisualization chart system is deprecated in the version of C# I am using, I shall implement this package[[1]](#footnote-1) to restore the feature so I can display global stats to the user.

# Entity-Relationship Diagram

|  |
| --- |
|  |

This is the structure I have chosen for my data management. When the user creates an account, a record in the User table will be created for them, with a unique UserID. See the security section for comments on password storage and the Salt field. The trigger CreateUserStats will be set up so that a record in the UserStats table will be initialized for that specific user, by referencing their UserID. When the user saves a maze to the server, a new record in the Mazes table is created, with a new MazeID to allow mazes with the same name. It will also reference the UserID of the creator of that maze, allowing only the user to see their own mazes. The GlobalStats table does not reference any other tables as it has no key: It serves as a permanent data store for statistics across the userbase.

# Data Structures

## Coordinate Class

|  |
| --- |
| Coordinate |
| -xPos : int -yPos : int  +XPos : int  +YPos : int  -visited : bool  +Visited : bool |
| +Coordinate(int : xPos, int : yPos) +GetCartesianCoordinates(Maze : maze) : Tuple<int, int>  +GetManhattanDistance(endPoint : Coordinate) : float |

The Coordinate class will be used whenever coordinates are necessary within the program. Most algorithms will utilise a stack of coordinates, adding new coordinates to the stack and popping the top value to backtrack.

The Coordinate class is useful since it allows more flexibility than just using tuples or similar data structures to store coordinates. A tuple can only store the two integers for example, but the coordinate class allows the code to get information from the object, which may simplify the more complex solving algorithms.

The private properties xPos and yPos will store the positions of the cell in a 2D array – this means that the Y coordinate will start at the top of the 2d grid and increase as you move down. I am therefore adding the GetCartesianCoordinates method to obtain traditional coordinates to a grid cell, which some solving algorithms will require. It needs to take an argument which contains the current maze in order to compute this. The public properties XPos and YPos will act as read-only accessors to their private counterparts using the get{} keyword in C#. This will allow coordinates to be initialised at creation, but not be accidentally changed afterward. The visited property is useful for some generation algorithms which need to remember if a certain cell has been visited. The method GetManhattanDistance will return a floating-point value equal to the Manhattan Distance between the coordinates of the Coordinate object and Coordinate argument, which will be useful in more complicated algorithms such as Maze-Routing or A\* pathfinding.

## Maze Class

|  |
| --- |
| *Maze* |
| -mazeActualWidth : int  +MazeActualWidth : int -mazeActualHeight : int +MazeActualHeight : int  -mazeCellWidth : int  +MazeCellWidth : int  -mazeCellHeight : int  +MazeCellHeight : int #mazeWalls : bool[,]  #mazeCoordinates : Coordinate[,] |
| {abstract} +initMaze() : void {abstract} +buildMaze() : void  {virtual} #cellVisited(cellPos : Coordinate) : bool |

The abstract class Maze will be the parent class of all maze types added and should streamline adding new maze types to the program. It provides essential properties for mazes and forces its subclasses to implement buildMaze.

The first 8 properties will hold the integers representing width and height, measured in array dimensions and number of cells, privately with read-only public access. The 2D array mazeWalls will hold the structure of the maze with booleans representing whether a cell contains a wall. I am using booleans since using char or string would make the maze files bigger and may affect network speed. Also, the maze will be displayed in a Windows Forms GUI so using letters would not add any extra utility to the program. I may also implement the 2D array mazeCoordinates as an array of Coordinate objects, as some algorithms may need the ability to always track the position of cells. The abstract method initMaze will be used to initialise the mazeWalls array with the pre-built maze structure, whilst buildMaze will implement the generation algorithm. The method cellVisited will query if a cell has been visited - with most algorithms this means if it still has 4 walls, but I am making it virtual in case any algorithms need their own definition.

## Hierarchy Charts

|  |
| --- |
|  |
| This is how the abstract class Maze would be implemented. Since the mazes I implement will inherit from Maze, they are forced to provide function bodies for the 4 abstract functions, and therefore all mazes must provide standardized output. Almost every algorithm in the entire solution implements Coordinate in some way, which I have shown here. Each class that defines an object which will be sent over the network must |
|  |
| Similarly, each SolvingAlgorithm is forced to implement the SolveMaze function by the abstract class SolvingAlgorithm. |

# Key Algorithms

## Generation Overview

The Maze abstract class means that, whilst the generation methods will be different, they will all have the same output: a 2D array of Booleans which represent the maze. Hence, each algorithm will be at its simplest level some way of manipulating the Coordinate system and the MazeWalls 2D array structure to create a maze. This standardised form is good since it should allow new algorithms to be easily implemented and added.

## Solving Overview

The abstract class Solving requires each solving algorithm to return 1 standardised output: a List of Coordinates which represent the solution to the maze. Therefore, in much the same way as generation, each solving algorithm will at its core be some way of interpreting and manipulating the MazeWalls structure and Visited Boolean of indices of MazeCoordinates respectively, in order to produce this List. The standard form should allow me to write less code to visualize the solution on the user interface.

## Initializing the maze

Whilst some algorithms may start with a grid full of walls, most maze generation algorithms will start with a waffle grid. This is the pseudocode for generating this waffle.

|  |
| --- |
|  |

First, the actual dimensions of the maze are calculated from the user input of cell dimensions. The mazeStructure array is then initialized with these sizes. Then, each cell is iterated through and is defined as either a wall or passage, with False representing passages and vice versa.

## Depth-First Generation

### Recursive Implementation

This algorithm takes the waffle and carves a perfect maze into the array. It uses recursion to backtrack through the passages it carved if a dead end is reached via the system call stack.

|  |
| --- |
|  |

A random cell in the waffle is passed to the subroutine as an argument. This cell is marked as visited. Then a list of all unvisited cells neighbouring the current cell is obtained. The pseudocode checks if the cell north, east, south, and west is visited. If it is not visited, it adds it to the list. I am adding 2 instead of 1 to the coordinates as I need to check the next cell, not the wall separating them. Next, a while loop is entered if there are unvisited neighbour cells. The algorithm picks a random cell from the list and destroys the wall between the current cell and the randomly selected one. The subroutine is then called recursively on the randomly selected cell. When a dead end is reached, the size of the list will equal 0, so the while loop will exit and control will return to the previous iteration of buildMaze, effectively using the call stack to backtrack until an unvisited cell is found.

### Stack implementation

The recursive algorithm is short and powerful but may exceed the maximum limit on recursion depth if the maze is too large. In the worst-case scenario, the algorithm will be called on every cell in the maze. If I run into issues with this, I may use an alternate implementation with an explicit stack of coordinates and a constructor which roams the maze, which has a less strict size limitation.

|  |
| --- |
|  |

A random cell is passed to the subroutine. The constructor is set to that cell and the constructor’s position is added to the stack. The number of unvisited cells in the maze is then calculated. Then, while there are still unvisited cells in the maze, the constructor checks if there are unvisited cells around it. If there are, then it picks a random one and travels to it, destroying the separating wall, adding its new position to the stack, and reducing the number of unvisited cells in the process. If, however, the constructor has reached a dead end, it backtracks down the stack until it finds an unvisited cell to enter. Eventually the constructor will have visited every cell and carved out a perfect maze in the process.

## Wilson’s Generation Algorithm

|  |
| --- |
|  |

This is an implementation of Wilson’s algorithm. Representing the logic as a flowchart is much neater than in pseudocode due to the large amount of data handling per step.

## Depth First Search Solving Algorithm

This is an easy to implement solving algorithm which find a solution, but it may not be the shortest one in a labyrinth-style maze. It is almost identical to the random mouse algorithm but uses the visited feature in the Coordinate class to make sure the solver never goes back down an already explored passage.

|  |
| --- |
|  |

Here, we start at the entrance. We get all the unvisited neighbour cells of the solver. If there are some, we go to the first one and mark it as visited, adding it to the solution in the process. By visiting the first one repeatedly, we essentially hug the left wall of the maze until an exit is found. However, this algorithm will not get stuck as we cannot loop due to marking cells as visited; if we find no unvisited neighbour cells, we backtrack down the solution stack. When the solver’s position is on the exit, we can return the solution stack, which will be a list of coordinates making up the final path the solver took to reach the exit.

## Maze Routing Solving Algorithm

With consideration of the Coordinate class’ implementation, I can implement this algorithm in a slightly different way to that described in the Analysis. I can have the algorithm never fail to find an improvement on its Manhattan Distance by implementing backtracking. The algorithm will have three states: Moving, Backtracking, and Trying Paths. Every update, the number of unvisited neighbour cells will be checked. If there is 0, backtrack: this behaviour replaces choosing based on which path best fits a line, as the algorithm would not normally have stack-based backtracking built in – it would just turn around. If there is exactly 1, it must be on a corridor, so continue moving down that corridor. If there are more than 1, it will try paths – in sequence, it moves down each path until it comes to a second crossroad, where it measures the Manhattan Distance between its current location and the maze exit. By doing this for each path, it can ascertain which path is likely the most productive, and so it will choose that one to proceed down. This will produce a good, but not perfect, solution.

# Security

Since I am implementing logins on a client-server network, I will need to implement some form of password obfuscation to prevent data theft; whilst no important data is stored on this database, the risk of a user choosing a password that is the same as their password on another platform is present and may turn the platform into a medium of attack for a malicious user. I plan to implement a password hashing algorithm, so the server never stores the passwords in plaintext.

## Data hiding

I plan to implement the PasswordChar feature into my user interface, so when the user is entering their password, it will be hidden by the standard dot character.

## Password Hashing

The PBKDF2 algorithm is built into C# and allows me to easily hash a password without the use of an external package. When the user has typed their password and requested a registration from the server, their password will be hashed and converted to hex/base64 so it can be sent as a string. In this way, the plaintext password will never be exposed from their local machine, and only ever the hash will be sent across the network and stored in the database, so it should be unintelligible to a malicious interceptor.

## Logging in

When logging in, the password the user entered on the login screen should be hashed on their local system and sent to the server. The hashes can then be compared using a fixed time comparison, which means that the state of the central system cannot be determined by the rejection time of a password (this could otherwise be used to guess how the password was hashed more easily). If the hashes are the same, the login should be correct, so the user is allowed access. Otherwise, access should be denied with a message that does not indicate what credentials were wrong.

## The Salt Field

The term “Salt” is used to describe a randomised string of characters appended to a password before it is hashed. This allows the same password to be hashed differently each time, increasing security somewhat. When logging in, if we hash the login password with the same salt as the original, we can get the same hash. I plan to store the hash in the login database alongside the salt to do this, but it is possible to extract the salt from the password if you know its length and encoding – I will not be doing this for the sake of more readable code.

# User Interface

|  |
| --- |
|  |

This is a first look at a potential user interface. I have added boxes, allowing the user to specify the parameters of the maze to request it. The box on the right holds a Table Layout Panel. I can use the CellPaint event of the panel to paint specific cells on the grid. However, I may have to have a dynamically sized form rather than a strictly sized one as pictured, since certain widths and heights cause small pixel errors which build up across cells, creating a large blank row and column across the bottom and right of the display. Additionally, high cell counts can make the cells too small. Both of these issues would be fixed by dynamic sizing.

1. https://github.com/kirsan31/winforms-datavisualization [↑](#footnote-ref-1)