Analysis

# Background

Currently, there is no lightweight app which allows one to create random mazes of distinctive styles with ease, whilst also providing access to past creations. My client, Greg, develops a puzzle app which allows people to request puzzles from a local network and solve them on their device, or request a solution from the network. He has asked me to create an app for users which allows them to request and solve mazes, and a server which can run the generation and solving algorithms, to extend his own puzzle app.

Intrinsically, the apps need to run on computers with access to a network connection with an open port (a public point over which data relevant to the app can be transferred). The hardware requirements of the computers vary depending on their job. The user’s computer should need little processing power as it is just sending and receiving data and allowing basic interaction with the maze. The computer hosting the program which generates and solves mazes may need to have higher-end hardware to facilitate the algorithms and networking.

Henceforth, the app which the user runs will be referred to as the client, and the app generating and solving mazes the server.

# Problem

## Recognition

There are three main problems to solve in the solution:

* Finding a way for a client on one system to interact with the server on another.
* Generating mazes via different algorithms.
* Solving the mazes by the shortest path.

Of these problems, by far the hardest is the first. Once the issue of interaction across a network is solved, adding new generation algorithms, and solving algorithms is simply a case of adding to the pre-existing network structure.

## Decomposition

The networking issue can be split into smaller steps. A first approach at this would be:

1. The server is opened across a port on the network.
2. The client is opened, then looks for and connects to the server.
3. The server assigns the client an ID, or uses an account ID, dependant on my client’s requirements.
4. The client chooses maze settings and creates a JSON/XML file out of them and sends it to the server.
5. The server adds this to a queue.
6. The server generates and perhaps solves the first item in the queue and sends it to the respective client where they can view and interact with it.

Steps 5 and 6 would be repeated while there are requests in the queue.

I will cover maze solving and generation in the Research section of the analysis, as I plan to implement several different algorithms for both.

# Client Interview

## Interview Questions

I will outline the questions I will ask Greg and summarise his answers to each. The aim of the questions is to get to know more about the system and the key requirements he would be looking for in my solution.

1. **How big is your userbase?**  
   This question will inform how robust the network needs to be. I will have to stress test the network much harder if the userbase is large.
2. **Does your system currently accommodate touch screen users?**  
   This question informs whether I need on-screen buttons or whether I can simply use keyboard controls for solving the mazes manually.
3. **How many maze generation algorithms are you looking for in the initial solution at a minimum?**  
   This question establishes a minimum number of algorithms, so I will have to prioritise adding a certain number of algorithms dependant on Greg’s answer.
4. **Is a login system necessary?**  
   This question informs whether a system which attributes a record in a database to a user via a username and password system should be implemented.
5. **Should there be a system which tracks statistics?**  
   This question determines the need for a system which could track some statistics on the server (mazes solved automatically, manually, types of mazes generated, etc.) is necessary to implement or can be left as an additional feature.
6. **What should happen if the user is unable to solve the maze?**  
   This question determines whether the solve option should only be available on a generation request, or whether it should be available post-generation. Either option will make a significant impact on the program which handles solve requests.

## Answers

1. Roughly 300 people.
2. Yes, around 40% of the users are on touch screen devices.
3. 3-4.
4. The current app stores puzzles locally, so a login system would be appreciated but not absolutely needed.
5. Again, there isn’t something like that in place already, but it would be nice to have.
6. They should have the option to allow the software to solve it.

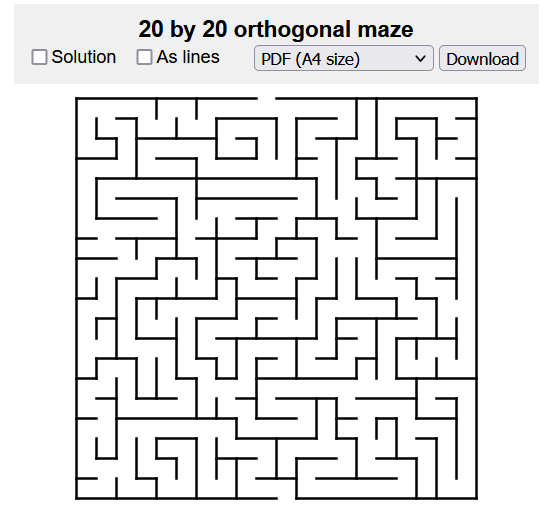
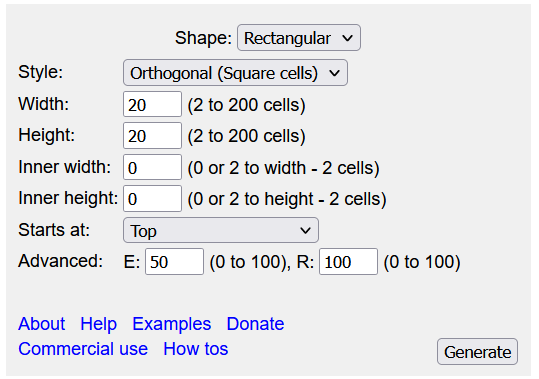
## Analysis

300 people at most would be connected to the one server. This means I will need to stress test the server to confirm it can handle consecutive or simultaneous requests from a proportion of the userbase. There is a considerable proportion of users who may not use a physical keyboard with the app: I will need to add onscreen buttons to accommodate them. If I also implement keyboard controls, I may need to add an onscreen indicator that the keyboard controls are also available. I may be able to put this into a small help menu along with other information on using the app, but this is not a key function of the app so the implementation should stay simple. Greg says that a login system is not a necessity for the solution, so I should only implement this after adding all other key features. A login system, if it is added, would be useful for tracking user-specific stats and server-side saving. Similarly, a stats system should be implemented last alongside the login system. Windows forms comes with a graph control I could use to pull data from a database into a charted format. Finally, the need for the user to always have the option to solve the maze means the server always needs to be listening for solve requests as well as generation requests, and the code for handling a solve will likely look different.

# Research

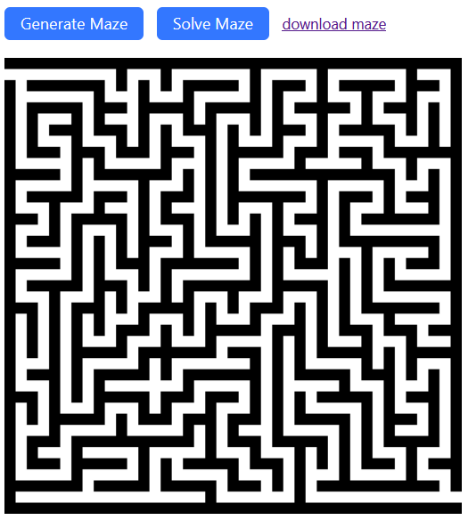
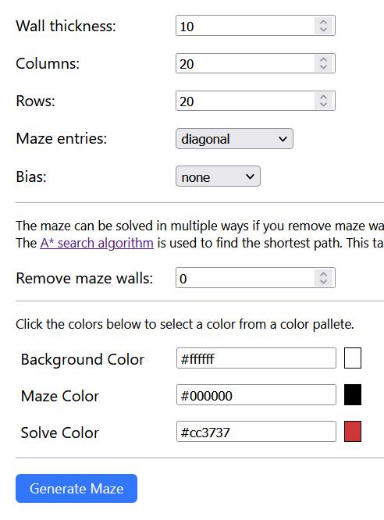
## Existing solutions

### https://www.mazegenerator.net/

  
This website is a free online maze generation platform, available on any web browser. The program can generate mazes in different shapes and sizes, as well as using a structural grid rather than an array-based grid to allow mazes where the cells are not squares. The user interface is simple and uses drop-down menus and text boxes to allow user input for maze parameters. However, there is no clear explanation of the advanced parameters E and R, nor can the user solve the maze on their browser. There is also no way to change the algorithm used.

### Parts I can use

I and my client like the simplicity of the interface and think the use of drop-down boxes to specify qualitative parameters is good. We like the way the maze is displayed as the walls are lines rather than thick cells. The generator also measures the maze height and width in cell number rather than array length, which allows for any size of maze instead of only odd numbers. This is something I will implement in my solution.

https://keesiemeijer.github.io/maze-generator/  


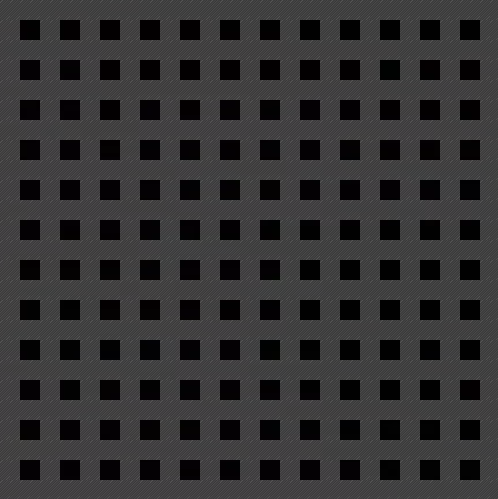
This website is another free online maze generation and solving platform. The parameter selection gives more meaningful names, so you can tell what each mean without a help page. The option for colour is given, but we think that for the initial solution, nonessential graphics are optional. We agreed that the option to remove walls is a straightforward way to generate a labyrinth style maze without a dedicated algorithm. Comparing this with the previous, the maze walls being thick here does not look bad, so we agreed it would be better to not draw walls as lines if it made the program better.

### Parts I can use

I like the option to download mazes, but based on my interview with Greg, I think that this option would be better as an upload to server option, tying into accounts. If I am unable to implement a login in time, then downloading the maze would be a viable alternative. We also think that the use of A\* Pathfinding for solving the mazes is good, so it would be good to include it, or some other shortest path algorithm.

## Maze Generation

A maze on a computer has two ways of being represented. One is to have a grid of cells, which can either be walls or passages. Another is to have a grid of cells which each store information about the state of each of their four walls. In this system I have decided to use the former method, as the algorithms are already complex, so I would like to keep the other aspects simple. A maze-generation algorithm involves first creating a rectangle of walls, then breaking those walls to create a grid of open passages called “cells,” in a shape resembling a waffle.



A typical generation algorithm then has a “constructor” roam the cells, breaking walls between them until a maze is formed. The way the constructor moves around the cells differentiates the algorithms from each other and determines the shape of the maze. For example, certain algorithms will produce mazes with longer/shorter hallways, or more hallways may be horizontal/vertical. I have researched several generation algorithms which I may include in the solution. Note that all these algorithms produce “perfect” mazes: mazes where there is only 1 path between 2 arbitrarily selected points. I should also implement a generator for a “labyrinth” style maze, where 2 points may be connected by multiple paths of varying length.

* Randomized Depth-First Search Algorithm  
  Aside from the binary tree algorithm, this is perhaps the simplest way to generate a maze computationally. Starting from the entrance, the computer selects a random unvisited cell, moves the constructor into it, and destroys the wall between the two cells. If the passage is a dead end – there are no unvisited cells surrounding the constructor’s cell, the constructor backtracks until it finds an unvisited cell. This process is repeated until there are no unvisited cells left in the maze. This algorithm, while simple, tends to have a bias for long passages and little branches.  
  This algorithm can be implemented in two ways:
  + Recursive  
    A new depth-first search is started on each unvisited cell. When a dead end is reached, the depth-first search ceases, and control returns to the previous search.
  + Stack  
    The recursive algorithm, whilst efficient, can cause stack overflow issues on older computers. A stack can instead be used to create larger mazes. The route the constructor takes is stored as cell coordinates and is appended to a stack. When a dead end is reached, the constructor backtracks down the stack, removing the top coordinate each time.
* Kruskal’s Algorithm  
  This algorithm uses a list of all walls and sets each containing one cell to produce a maze with a bias for many branches and short passages. It randomly selects a wall, and if the cells divided by it are in distinct sets, it destroys the wall and joins the two cells, creating one larger set. The process is repeated until only one set remains. The mazes produced by this algorithm tend to be easier to solve.
* Prim’s Algorithm  
  This algorithm would produce mazes identical to those of Kruskal’s algorithm, but Prim’s algorithm effectively introduces weighting to walls, affecting the randomization and creating a stylistic difference. This is done by not using a list of all walls. Instead, only the walls visible to the main chuck of the maze are added to the wall list.
* Wilson’s Algorithm  
  This algorithm creates an unbiased, uniform maze, which tends to be harder to solve. This is done by using a method called a loop-erased random walk. The algorithm starts at the starting cell and starts walking randomly until it reaches the end cell. However, if at any point it runs into its own path, it will backtrack, removing any destruction of walls along the way. Once this initial path has been formed, it picks another random start cell and performs another random walk until it reaches the maze. This is repeated until all cells are visited.

## Maze Solving

A typical maze-solving algorithm works similarly to the generation algorithm, in that it has a “solver” roam the maze and take a path at each crossroad, depending on a set of rules. If the path is a dead end, it will backtrack and take a different route. I have researched some solving/pathfinding algorithms which I could implement.

* Random Mouse Algorithm  
  The most primitive maze-solving algorithm, by far. This algorithm simply proceeds down a passage until it reaches a crossroad, before making a random decision about which passage it should go down. This algorithm is simple to implement and will always find a solution, but can be extremely slow, and this is noticeable on large mazes.
* Trémaux’s Algorithm  
  This is an efficient algorithm, which, while it is guaranteed to solve a maze, may not always find the shortest route. The algorithm works by marking the entrances to the passages in the maze. When a passage is entered, the entrance is marked. The direction to take at a crossroad is decided by the marks of the passage entrances. If none of the new passage entrances are marked, a random one is picked to go down. If any of the new entrances are marked, backtrack down the entrance that was just passed through, unless it is marked twice. Going back through the entrance in this way will mark it again. This rule will always apply at a dead end. Otherwise, pick the entrance with the fewest marks. When the exit is located, the marked passages will form a path back to the entrance, but the random nature of the algorithm means it may not always be the shortest. This algorithm is fast, not too complex, and can solve labyrinth mazes.

* Maze-Routing Algorithm  
  This heuristic algorithm uses Manhattan Distance (The absolute difference between two point’s Cartesian coordinates, henceforth MD), and the idea that on a grid, this distance measurement will change by exactly 1 when moving to a neighbouring cell. This algorithm can find paths between not only the entrance and exit of the maze, but even just any two points, with little information. It can also detect whether the maze is unsolvable. However, it may not find the shortest path. The algorithm works by first deciding if there is a productive path, based on comparing the current MD with the MD of a path. If the path is productive, take it. Otherwise, the algorithm will turn either left or right at a given crossroad, by determining which direction would best fit onto an imaginary line drawn between the start and exit cells. This process is repeated until the current and exit coordinates are the same.

* A\* Pathfinding Algorithm  
  A\* is a complex algorithm for finding the shortest path between the entrance and exit. Put simply, it creates a tree, with branches representing paths through the maze, and extends each branch by 1 cell until the exit is reached. The algorithm uses complex maths to determine which path to extend on each loop, using MD and a cost system to select nodes in a priority queue. This algorithm is complicated in design but will always find the shortest path, even in a labyrinth maze.

## Networking

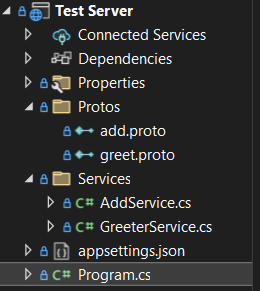
There are three ways of creating a server in a way that works in my solution. One is using a server object, and a console app which references this object, listening for requests to use its functions across an open network port. Another is to use sockets to handle requests rather than a server object, which compresses the server into a single project but may be more complex. I could also use gRPC’s C# implementation, which operates across 2 separate projects (a client and server) using shared protocols defined in proto3 files. This would retain the complexity inherent to a network environment whilst not introducing deprecated features into the project.

# Requirements

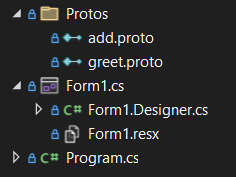
1. The Windows Forms client app which the user will interact with.
   1. The ability for the user to customize their request for a maze with several parameters.
      1. The ability to change the maze generation algorithm used to make the maze.
         1. An implementation of an algorithm to make perfect mazes with a bias for long corridors and low branching.
         2. An implementation of an algorithm to make perfect mazes with a bias for short corridors and high branching.
         3. An implementation of Wilson’s algorithm to make unbiased perfect mazes with a uniform distribution of branches and corridor lengths.
         4. (Optional) Add more algorithms by which the user can select and generate mazes.
      2. The ability for the user to select the width and height of the maze they generate, measured in cells.
         1. A width parameter which changes the horizontal cell size of the maze.
         2. A height parameter which changes the vertical cell size of the maze.
      3. The option to remove a user-specified number of walls in the maze, allowing for a performant solution of generating uniquely styled labyrinth mazes.
   2. The ability for the user to allow the server to solve the maze after it has been generated.
      1. An implementation of a simple algorithm for performant solves.
      2. An implementation of the Maze-Routing algorithm allowing for shorter-path solves.
      3. An implementation of any shortest path algorithm, such as A\* or the First-Breadth Search algorithm allowing the user to find the shortest possible path in a labyrinth maze.
   3. The ability for the user to request a maze to be generated by the server with their selected parameters, and have it be returned to their client on completion.
   4. A button on the client available post-generation allowing the user to request a solve of their maze.
   5. A screen on the client which displays the current maze after receiving it from the server.
   6. The ability for the user to navigate the displayed maze pre algorithmic solve and attempt to solve it themselves.
      1. Navigation controls mapped to buttons on the keyboard.
      2. Buttons on the client interface which allow navigation.
   7. The ability for a user to save their generated maze.
      1. If the login system is completed, this should save the maze to the server database. Otherwise, this feature will be a local download.
   8. (Optional) A section of the client which uses the Windows Forms Graphs library to display a user-specified graph of information stored in the server database.
      1. This aspect should allow the user to see personal stats, as well as global stats if the login system is completed.
2. The Console App server which will handle the backend operations of the system, such as maze generation, solving, and database handling.
   1. A database structure storing statistics and login information, as well as storing saved mazes.
   2. A console interface which logs requests.
   3. Services to facilitate logging in.
   4. Services to facilitate registering a new user.
   5. Services to facilitate database management.
      1. Saving/loading mazes on request.
      2. Organize global and user times.
   6. Services to facilitate getting data from the server database.
      1. Fetching user stats
      2. Fetching global stats
      3. Fetching saved mazes

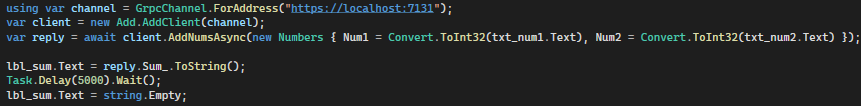
# Modelling

As a prototype of a potential solution, I have created a small client-server network via gRPC to see how the project may be structured.

The structure of the overall solution is split into 2 projects when using gRPC. The server contains a Protos file, containing files which document the format in which data will be transferred for a certain service. These are referenced in an item group in the project file so the objects defined can be generated for use in services. It also contains a Services folder, which contains programs defining the logic of how that request data will be handled and sent back to the respective client. These services are mounted to the server in the main Program.cs file.

I have created a simple Windows Forms client to test if the server will be able to contact a graphical client.

The client also contains a Protos file with copies of the server protocols. These are also referenced in an item group like in the server. The client contains no definition of the services however, since it will only send data to the server so the server app can perform the service. This keeps the client lightweight.



This is a call to the server, requesting its “AddNums” service. We establish a connection to the server across a predefined port (here, 7131), and create a clientside request. We pass the server the “Numbers” object defined in the protocol and await its reply. We then display the data in the reply to the client’s form.

The solution could use a similar principle, albeit much more complex with many more services and calls.