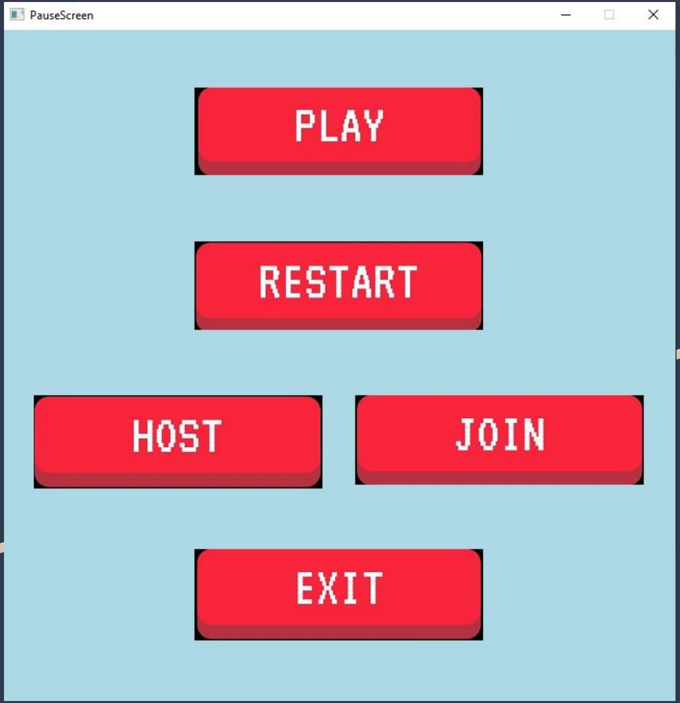
SDL Maze Game
And
Improved Maze Simulation(Merged)

Made By Arka Mandal

Starting Screen and Others

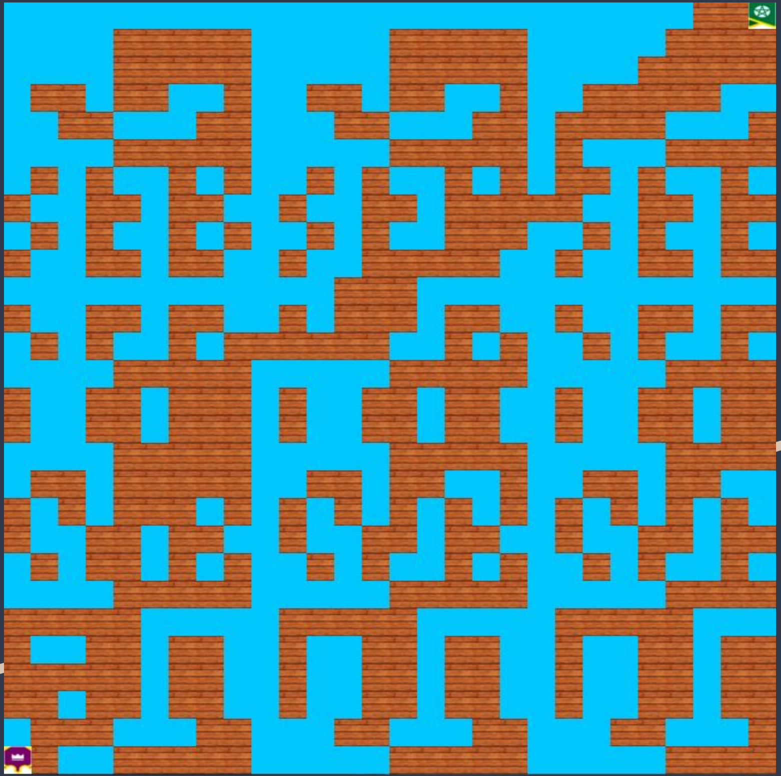


When the game starts, the player is greeted with the following screen. All buttons are self explanatory but we will go over them one by one.

Play Button starts the game. If there is no connection made, then game starts with 2 players in same game with different controls for each. Restart button refreshes the game, closing all connections and loading a new map. Host button makes the game act as a Server and allows another game using the Join button to join this game with the unique room id. Exit stops the game and closes all windows.

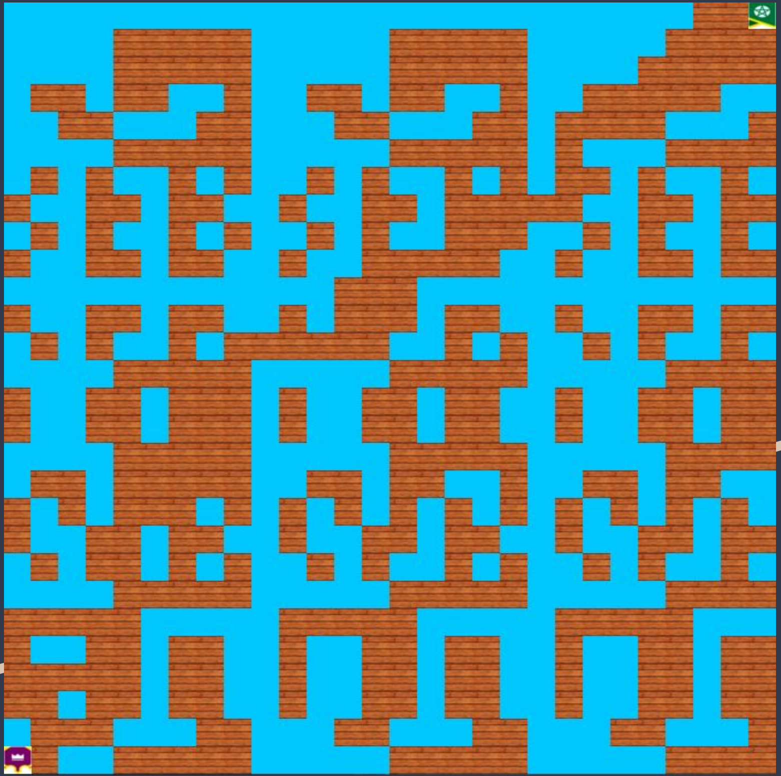
Also, the game will keep printing out different statistics throughout the game for the Player to see what's going on. Whose turn it is, if connection is happening or failing, everything is displayed on terminal.

Game Screen



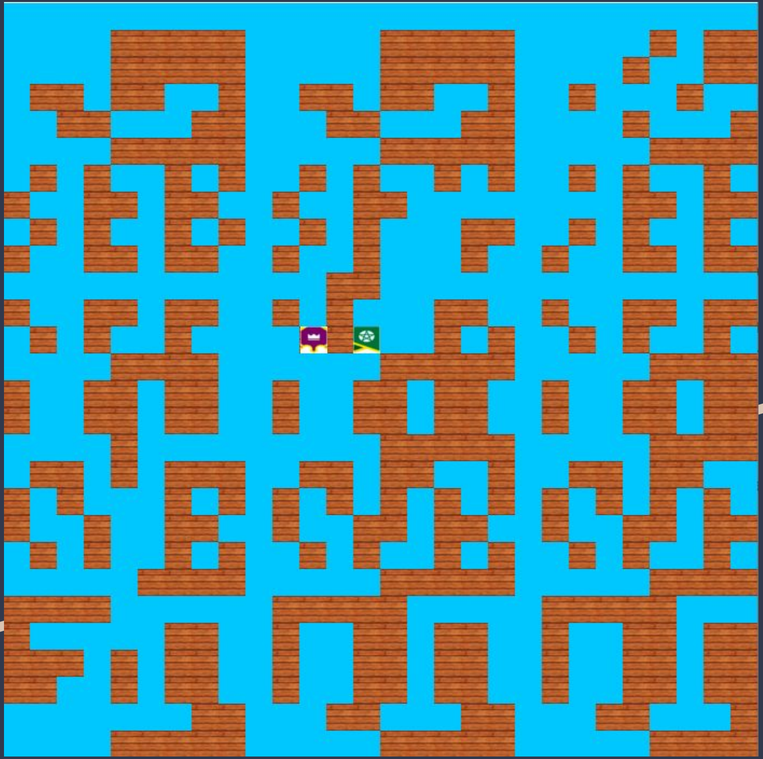
The game starts as shown. Both players at opposite corners. The bottom left purple flag refers to player 1 and top right green flag is player 2. They have to move towards each other and the player who overlaps the opponent position wins. This is a turn based game very similar to chess. The maze is randomly generated and will change every game.

Back-Story



A war has been going on between the Purple Kingdom and the Green Kingdom. Two brave soldiers from these kingdoms are trying to infiltrate their enemy kingdom stealthily using an old broken bridge not used by anyone. They see each other and try to move to their opponent's position in order to get rid of them. The problem is that the bridge they are using is old and weak. When they move from one wooden tile to another, the tile breaks and falls down.

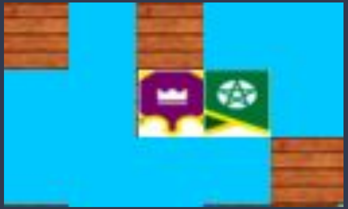
No Turning Back and Game Controls

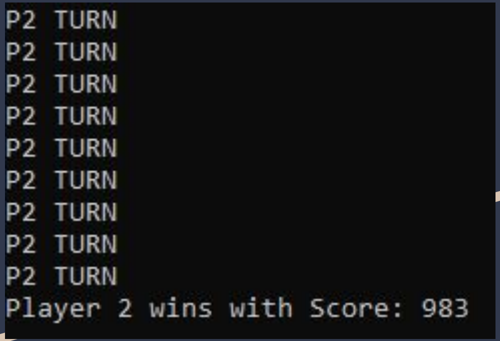


As players move from one tile to another, the tile disappears and the player cannot move back. This is done so that players don't move around in circles avoiding overlap by opponent. Players need to keep moving onto new tiles at each step.

Players are controlled by just 4 buttons. UP, DOWN, LEFT and RIGHT arrow keys moves the player 1. W,A,S,D moves player 2 in up, left, down, right respectively.

Checkmate, Winning and Scoring





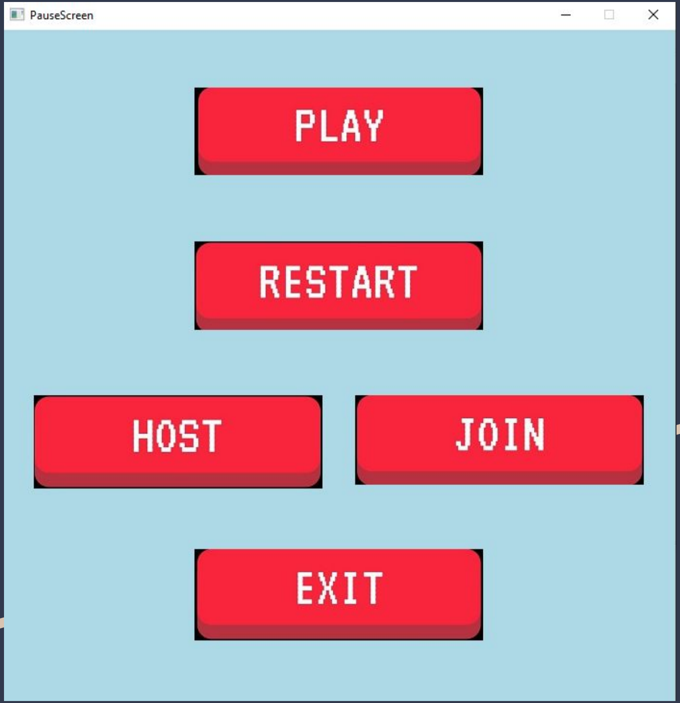
```
P2  TURN
P2  TURN
P2  TURN
P2  TURN
P2  TURN
P2  TURN
P2  TURN
P2  TURN
P2  TURN
Player 2 wins with Score: 983
```

Eventually, the game will come to a point like this. Whoever the next turn belongs to, they may come over to the opponent's position and win the game.

The winner will be declared along with their score in the terminal as shown.

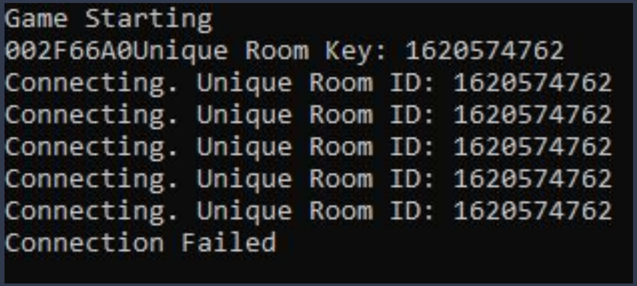
The scores are independent of who wins. It is only a measure of how well you played. Score reduces by 1 for every move you make. So, it is better to make lesser moves.

Pause Feature

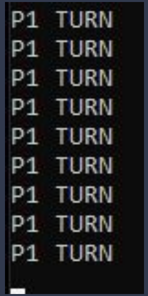


The game may be paused. While paused, any keyboard inputs will be ignored. However, the music will not stop. The background music will keep playing. Game Mode may be changed while paused or the game may be restarted as single player with a new map.

Hosting a Game



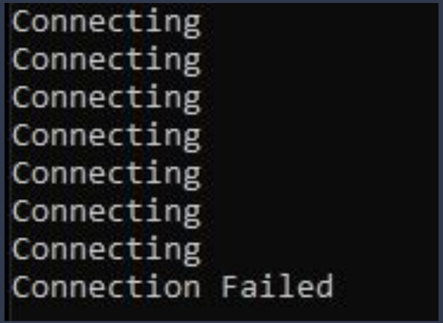
```
Game Starting
002F66A0Unique Room Key: 1620574762
Connecting. Unique Room ID: 1620574762
Connecting. Unique Room ID: 1620574762
Connecting. Unique Room ID: 1620574762
Connecting. Unique Room ID: 1620574762
Connecting. Unique Room ID: 1620574762
Connection Failed
```



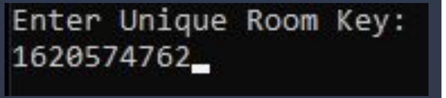
```
P1 TURN
P1 TURN
P1 TURN
P1 TURN
P1 TURN
P1 TURN
P1 TURN
P1 TURN
P1 TURN
```

Clicking the Host button starts the games a host. It gives a unique room key. The client must use this room key to join this game. For now, the host waits for 10 seconds before closing its connection. “Connection Successful” is printed if done else after 10 seconds it shows “Connection Failed”. After connection is successful, the game starts. The host controls Player 1. The turn indicator also starts informing the player that the game is waiting for an input from the host.

Joining a Game



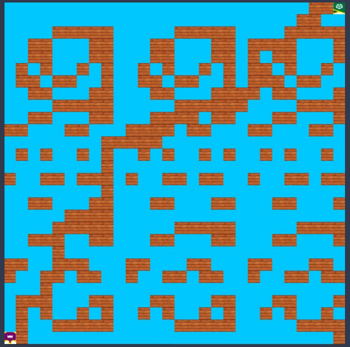
```
Connecting
Connecting
Connecting
Connecting
Connecting
Connecting
Connecting
Connection Failed
```

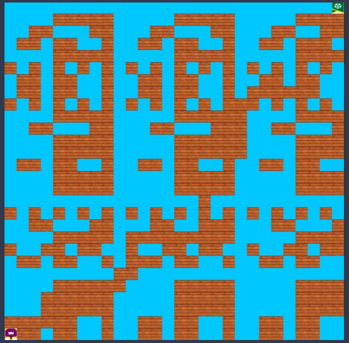


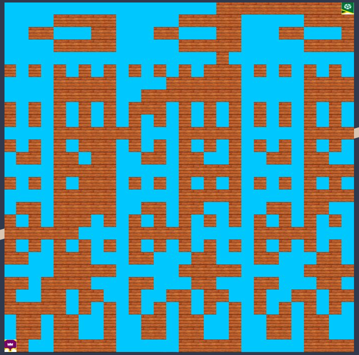
```
Enter Unique Room Key:
1620574762_
```

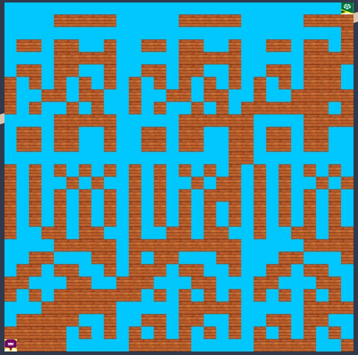
When Join button is pressed, it asks for unique room key as given by the server. Once the key is given, it tries to connect to the server. If connection is successful, it prints “Connection Successful” on both client and server side. Else it shows “Connection Failed” after 10 seconds of trial. After connection is successful, the game starts. The client controls the Player 2.

Maps/Mazes/Levels









There are no levels or stages as this game was made from start considering a multiplayer perspective. The maps/mazes are randomly generated at runtime.

For different games, the map has to be same. This is done using the unique room key. The unique room key ensures that same map is generated in both games.

The restart button also changes the map.

The wooden tiles are where the player can be. The blue tiles are traps in a maze. Trying to go into a blue tile will automatically result in the opponent's victory.

Maze Simulation and Solving

Arka Mandal

11 May 2021

1 Objective

We are supposed to generate a Maze building algorithm and then find a way to solve it by reaching 6 different cells. The data structure to be assumed for the Maze, algorithm to solve, data available at each step of iteration etc are flexible and open to interpretation.

2 Maze Structure

The maze will be defined by a simple 2D array data structure (A matrix). The matrix will consist of several nodes each representing a cell. Each node will have its position mentioned within it and the node will be connected to its neighbouring cells/nodes. These connections will be specified within the node using 4 Boolean variables each representing whether there is a connection towards up/down/left/right. If there is no connection that would signify a wall. We need an algorithm to procedurally generate a randomized maze.

3 Maze Formation

The maze forms by a DFS algorithm. We start at a cell. This can be any randomly selected cell but lets say in our implementation, we start from top left cell. We then check the which are the neighbouring cells that are not visited yet and we choose anyone from them and move there. We declare our original cell as visited and continue this way visiting cells one by one.

While doing this, we also define the paths. This is done by making connections in the graph joining different nodes. Algorithmically, this is achieved by declaring two sides of a cell as a wall. These two sides are specifically the ones where we are not moving and where we did not come from. For example, if we move from cell(2,2) to cell(2,3) to cell(3,3), then in cell(2,3), we keep declare a wall at top and right as we entered this cell from left and leaving towards down.

We keep continuing this way and eventually we come at a point where there is no way forward. All the cells are already visited. At a point like this, we start backtracking. We move back to a point where there was atleast one path leading to an unvisited cell. This is done using stacks. We use stacks to store the cells in the order we visit them and while backtracking, we pop from top. We continue all these steps and eventually we will complete the maze. We will know that our algorithm is complete when we return back to our starting node after pushing and popping into the stacks multiple times.

4 Maze Solving

Solving is very similar to making the maze. Before we start building an algorithm, we need to decide if all cells of maze is available to the solver at all times or is he only able to see the data in its current cell. For this simulation, we decided to see the maze from the perspective of someone inside it solving. So, only the data of the cell we are currently in will be available to us.

We need to find a way to reach 6 cells of the maze and then reach the original point. The best method is to see whatever paths are available to take from current cell and randomly choose one. Then we add this cell to a stack, mark it visited and move to the next chosen cell. We continue this and backtrack by popping when we reach a dead-end. This is very similar to maze-making algorithm as both these places we implement DFS. If the data of all the cells were available to us, then we could have opted for BFS as it would be faster and easier to visualize.

The fastest way to solve this maze is unknown to us as we can only see our neighbouring cells at each moment. So our best course of action is to remember our paths already taken and proceed forward randomly. Once all 6 items are acquired, we can backtrack without moving forward.

5 Time Complexity

- The time complexity associated with the maze-making algorithm can be understood as follows: We move forward to a random cell at each stage so in any case, we need to go to each cell and set up walls there. So, the the time complexity is proportional to number of cells in the maze. This is O(mn) where m and n are dimensions of the maze.
- The time complexity associated with the maze-solving algorithm can be understood as follows: We move to a random cell at each step. In the worst case, the last item may be present in the last cell. This would mean that we effectively traversed the entire maze. This time is proportional to total number of cells in the maze, mn. After this we have to backtrack and this is again ¡total number of cells. So, the worst case complexity of solving this is again mn. Therefore, net time complexity is O(mn) where m and n are dimensions of the maze. The worst case is described in a diagram below with an example maze.

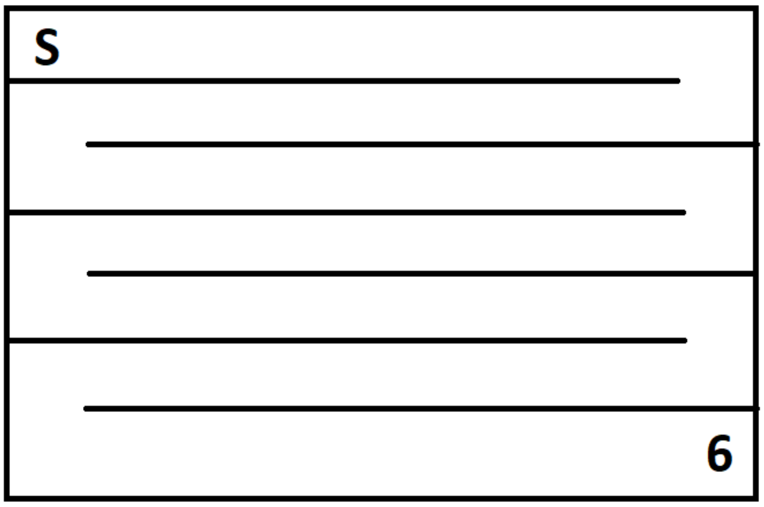


Figure 1: Worst Case Scenario Visualized

Here, S indicates starting position and 6 indicates the position of the last item to be collected. Here we can clearly see that to solve it, we need to traverse all the cells of the maze twice.

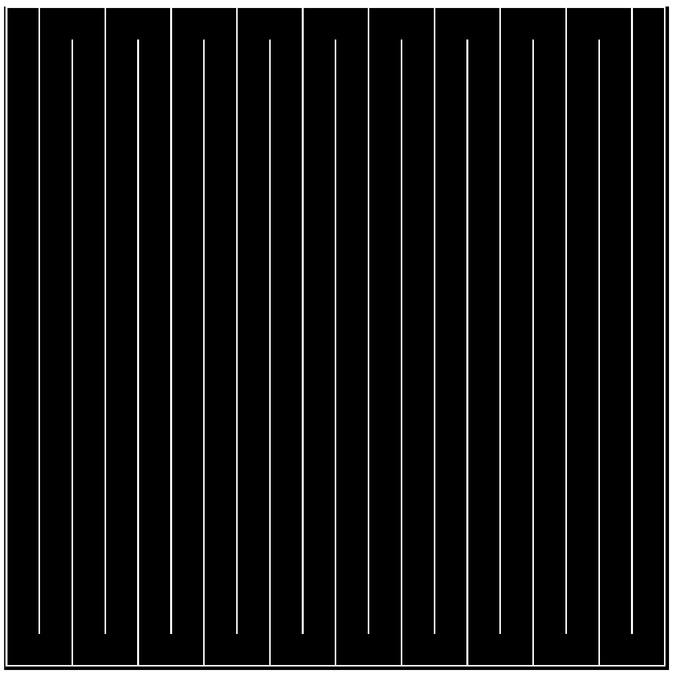


Figure 2: Worst Case Scenario Simulated

6 Correctness

We can prove correctness using PMI. Our goal is to ensure that all cells have atleast one connection to a nearby cell. We define our maze in this manner.

- Basis: At start we have only visited the first cell. So total cells visited and sides manipulated is 0 as we update visited cells after we leave the cell to go to the next cell. Our work will be complete when we reach the mn^{th} cell.
- Induction Hypothesis: At n^{th} iteration, let there be n visited cells.
- Induction Step: We are inducting on number of visited cells. At any step of the iteration, we move to a random unvisited cell and declared it visited and manipulate its sides by declaring two of them as walls. So at each step, number of visited cells increase by exactly one. At n^{th} step of iteration, number of visited cells is n from Induction step. From logic described above, we can say that while moving to $(n+1)^{th}$ iteration, we increase number of visited cells by one. Therefore, number of visited cells in n+1 steps is n+1.

Therefore, by strong induction, we can say that, our algorithm is correct and a maze as defined above will be formed.

7 Problems Associated and Solutions

The maze solving algorithm has no problems in it. However, there is a problem in the maze-making part. We only move to a random cell and block all other paths. This creates a very linear fashioned maze with only one way to move forward -not a very good maze in conventional sense.

One solution to this is by randomly selecting some non-corner cells and removing a random wall. Removal of these walls will mean more paths to choose from at any stage of movement. This will make the maze less linear and more challenging. Deciding the number of walls to remove is our next challenge. If we remove too less walls, the maze will still remain quite linear but if we remove too many walls, there will not be many defining paths and the maze will be empty. Through trial and error, we observe that removing one wall in each row is a good choice for small mazes.

8 Final Maze

A maze created after all these is shown below. Random Wall removal has not been done here as for a small maze like this it was not necessary.

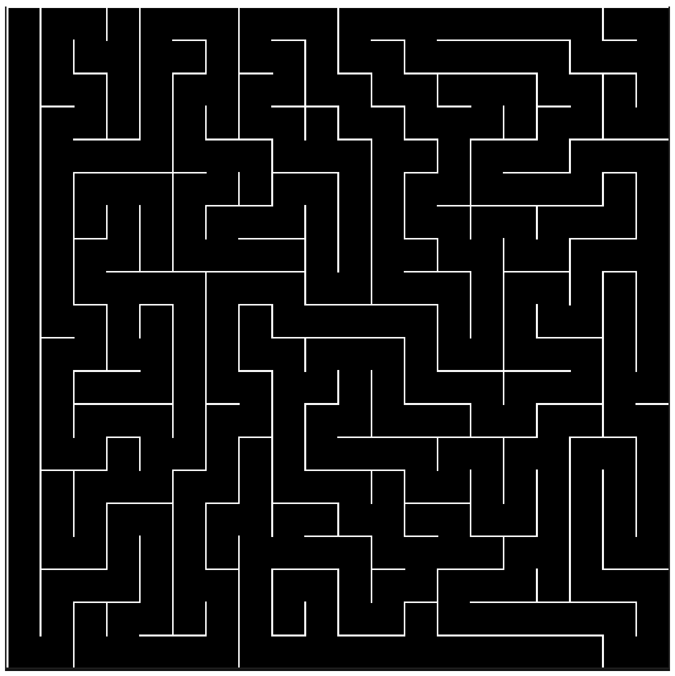


Figure 3: Sample 20x20 Maze Simulated Fig.1

Another sample is drawn below as a show of how random these mazes can be generated.

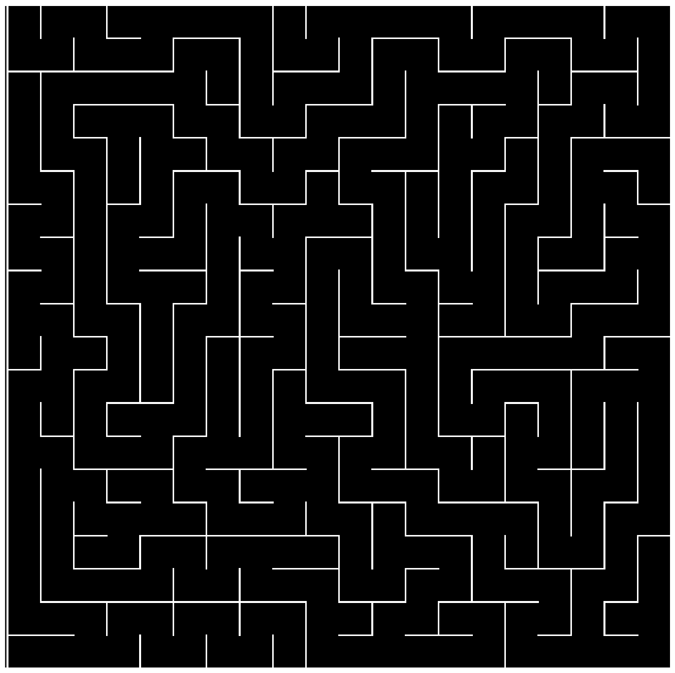


Figure 4: Sample 20x20 Maze Simulated Fig.2

9 Notes on Application

- The randomizing function used for deciding path has been designed by me for various reasons. The conventional rand() function is not used.
- We specify that our algorithm is inspired from DFS algorithm but no pointers are used as in conventional DFS algorithm on trees. We used DFS to traverse within a 2D array.
- Rendering the maze takes place directly with the help of 4 booleans specified in the "Maze Structure" section. We render only when wall is true.

- Positions of a node inside the matrix is used only for backtracking. While moving back we pop the last node and see its position to update our position.
- The entire application is done keeping speed and abstraction in mind. Functions are cleanly segmented and self-explanatory. Instead of classes, a lower level ”struct” implementation is chosen as our needs can be satisfied by that too. Finally, we tried to minimize the use of pointers and used only one stack for backtracking. The maze itself is made completely using a matrix instead of trees. The only downside of this is that the maze size cannot be dynamically changed. It needs change in array declaration within the code.
- Different Size mazes can also be made using the algorithm. The size input is just not taken during runtime and if changes need to be made, they have to be made by changing 2 macros. The same is applicable for window size too. Both window size and maze size is hardcoded within 4 macro. An example of a larger maze is shown below.

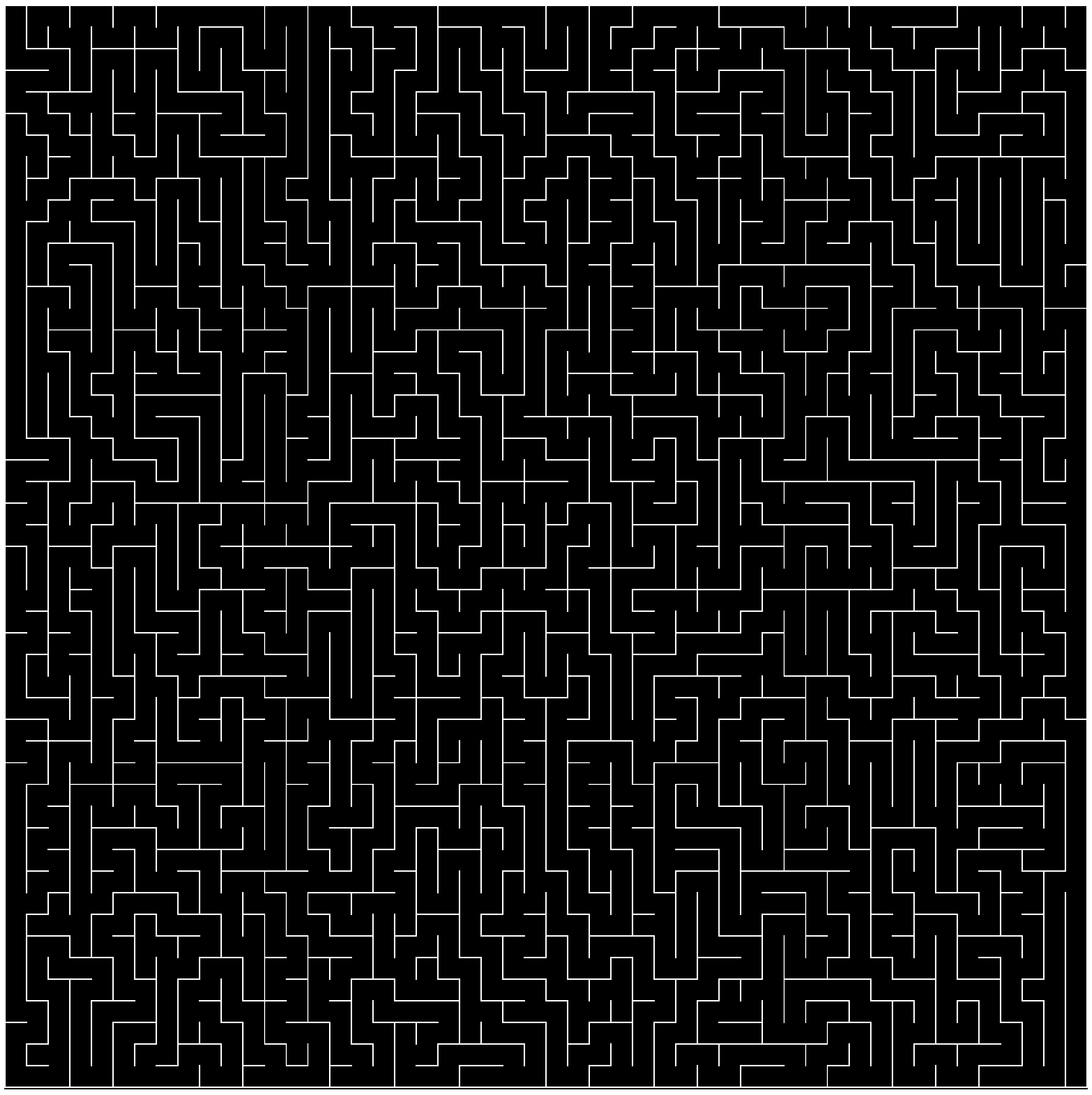


Figure 5: Big 50x50 Maze Simulated Fig.3

For reference, the previous mazes were drawn on 400x400 screen with 20x20 grids. This maze is of 50x50 grids and is drawn on 800x800 screen.