Progress Report for AI for playing classical strategy games - Quoridor

Ace Shyjan

registration: 100390438

1. Project Introduction

1.1. Motivation and Aim

This project is about the classical board game Quoridor and integrating AI into it. My motivation for this project comes from my love for board games and my interest in creating computer AI and this project allows me to combine these interests by building a game to explore how AI makes decisions. Creating an efficient algorithm will be challenging due to Quoridor's dual objective nature of moving pawns and placing down fences; however, by using algorithms like Minimax or MCTS, I can explore strategic decision-making that balances both actions while considering the long-term consequences of each move. The project will be built in Godot which is a game engine I have had many years of experience in.

1.2. Objectives

My aim for this project is to create a fully functional Quoridor game that allows a player to compete against the computer. In addition to developing the core gameplay mechanics, I plan to explore and implement various AI algorithms, such as Minimax, Monte Carlo Tree Search, and possibly Neural Networks, to assess their effectiveness in making strategic decisions.

- Create a working game of Quoridor against 2 players

- Create a computer AI using Minimax Algorithm

- Explore other potential algorithms such as MCTS, Neural Network, etc.

- Add multiplayer functionality between 2 devices (Extra)

2. Progress Report

Currently, based on the Gantt Chart [7], I have started the coding stage and recently completed the design phase. The coding stage will be the longest, with the implementation of the algorithms being the most time-consuming part.

2.1. Planning

2.1.1. MoSCoW Table

Must Have	Should Have
Fence Placement System, Pawn Movement, Win Check, Illegal Fence Check, UI, AI Opponent	Customisability - Board Size; Pawn Colour; etc, Different AI Types
Could Have	**Won't Have**
Multiplayer Support, Save Game State, Move History	Leaderboard, AntiCheat, VR

Table 1: MoSCoW Table

I created a MoSCoW table to prioritise sections of the project. Since this project is a local game, the main focus will be on the game's mechanics, such as fence placement, pawn movement, and AI. Once the main mechanics have been implemented, I can then focus on additional features, such as comparing different types of AI algorithms.

2.2. Design

The key planning areas focused on the pre-game user interface, the board, fences, movement, and game loop. For the main menu, I created a list of possible options that the user would be able to navigate. Designing the board was straightforward, as it was simply an $X \times X$ grid of tiles, whereas designing the fences proved more challenging. I decided to use buttons to represent the fences, which were positioned at the center of every 2×2 grid of tiles. This design was much simpler to implement than other potential ideas, such as a drag-and-drop mechanism.

For movement, I took inspiration from Chess, where available tiles are highlighted upon selection, and I planned to implement a similar feature. Regarding the game loop, I chose to have players click a "Confirm" button to lock in their move. This approach allows players to explore possible moves and strategise — for example, they can click on different fences to consider how their opponent might respond.

2.3. Board Options

I began by focusing on the main menu and game options. The options implemented so far include player name, colour, and board size. These settings have been integrated into the game itself, as shown in Figure 1.

2.4. TileButton

The next step was to work on the actual board. I created a `TileButton` scene that inherits from Godot's Button class. I instantiated this button b × b times, where b represents the size of the board. The resulting board is shown in Figure 2.

2.5. FenceButton

The `FenceButton` class also inherits from the Button class. This class contains two panels to display horizontal and vertical fences. I instanced $(b-1)^2$ `FenceButtons` and offset them so that they appeared in the middle of a 2×2 grid of `TileButtons`, as shown in Figure 2.

I configured a toggle button to switch between horizontal and vertical modes. When a `FenceButton` is pressed, it displays the corresponding fence panel, and upon clicking confirm, the fence is placed. After the fence is placed, the adjacent `FenceButtons` are disabled to prevent overlapping. This behaviour is shown in Figure 5, where the red buttons represent the disabled `FenceButtons`.

2.6. Tile and Fence Connections

To separate the user interface from the game logic for future use, I created the `Fence` and `Tile` classes to store their connections and associated buttons. Each tile has a connection in each cardinal direction, and these are stored in a 1D array. Calculating the connections between tiles and fences uses the length of the board/fence grids.

Another connection that needed to be handled is the one that is removed when a fence is placed. The `Fence` class stores this information in a 3D array, where the first and second elements correspond to the horizontal and vertical fences, respectively. The second depth of the array stores the two possible sets of connections that can be

removed for that direction, and the third depth stores the actual two tiles that would lose their connection.

2.7. Board State

I created a `BoardState` class, which can be used later for the DFS and AI algorithms, as it stores the current state of the board, including tile connections, current pawn positions, and fences.

2.8. Pawn Spawning

Once the tiles and fences were in place, I started to handle the pawn spawning and movement. I created a `Pawn` class, which inherits from Godot's Panel class. I made the pawn a white circle so that I could change the colour by updating the panel's modulate. During my first attempt at spawning the pawns, I placed two pawns: one at the top centre for Player 2 and one at the bottom centre for Player 1. However, when I reached the pawn movement stage, I changed how the pawns were spawned. Instead, I spawned a pawn on each tile for each player. This way, for pawn movement, I could simply hide and show the pawn as needed.

2.9. Advanced Pawn Movement

Now that every tile has a connection, proper pawn movement can be set up. I first disable all tiles on the board, and then enable all tile connections for the tile that the player's pawn is currently on. When a fence is placed, the tile connection is severed, as shown in Figure 3.

2.9.1. Pawn Leaping

One of the rules states that if two pawns are adjacent, a pawn can leap over another pawn, provided that there isn't a fence behind the pawn to be leaped, as shown in Figure 6. I achieved this by checking if one of the connections contains the enemy pawn. I then determine the cardinal direction of that tile and multiply it by 2 to get the leaped tile position, if there is a connection between the enemy pawn and the leaped tile. Another rule is that to prevent a pawn from being entrapped, if the pawn cannot leap, it can move

to the enemy pawn's tile connections, as shown in Figure 5. To achieve this, if a fence is blocking the leaped tile, we then get the adjacent tiles of the enemy pawn's tile.

2.10. Game Loop

Once the movement was set, the last step was the game loop. I used a bit to store the current player, which allowed me to store both players' actions in arrays. After the player clicks confirm, the `current_player` bit is flipped. The bit has an attached setter, which then resets the board and stores the moveable locations. Lastly, I implemented a win condition that runs a check every time a pawn is moved, to determine if the pawns are at their respective goal lines (top and bottom of the board).

3. Progress Evaluation

3.1. Current Achievements

So far, I have been working on the first objective: "Create a working game of Quoridor against 2 players.". The base game is nearly completed, with the illegal fence check being the last step before having a fully working game of Quoridor.

3.2. Risk Analysis

The biggest problem so far was figuring out how to implement fences both graphically and logically. After a while, I realised that it was best to separate the fence logic and use it to update the tile connections. Another issue that arose were invalid tile connections. The problem stemmed from mapping the 2×2 `Tile Grid` with a single `Fence Button`, while also using the `board_size`. This was resolved later after examining the underlying logic.

A future concern will be optimising the AI and DFS, ensuring the efficient use of C# and threading in order to perform the calculations in a shorter period of time. Another oversight is that the `BoardState` class stores references to the actual class itself, rather than the index of the position in the grid. This might be a problem in the future, as duplicating board states to simulate moves may cause issues such as overwriting data on the original board, which is something I will need to be careful of.

3.3. Conclusion

In terms of a project plan, the next steps will be to implement the illegal fence check. This will require a depth-first search and will be written in C#. This is the final step in completing the local multiplayer. Once that is finished, I can work on implementing the Minimax and MCTS algorithms. I will be modifying the `BoardState` to use indexes, rather than references, as mentioned before, to make the DFS and AI algorithm implementations easier.
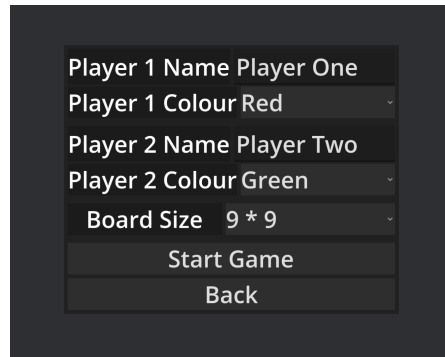
A. Appendix

Figure 1: Board Settings for local multiplayer.
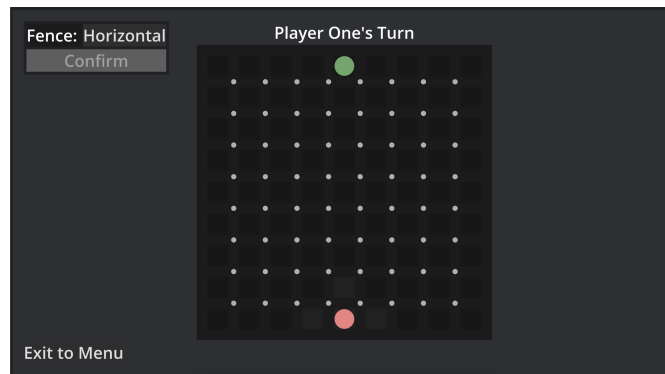
Figure 2: Current Board.
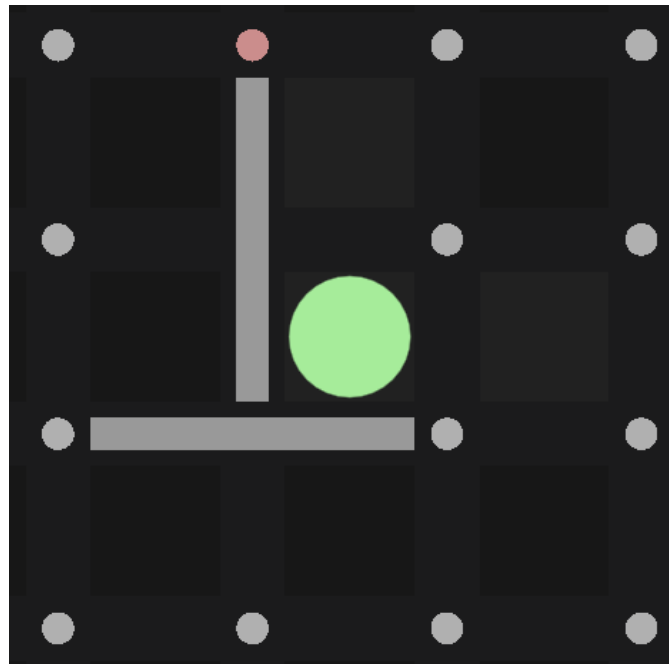
Figure 3: Fence Placement.
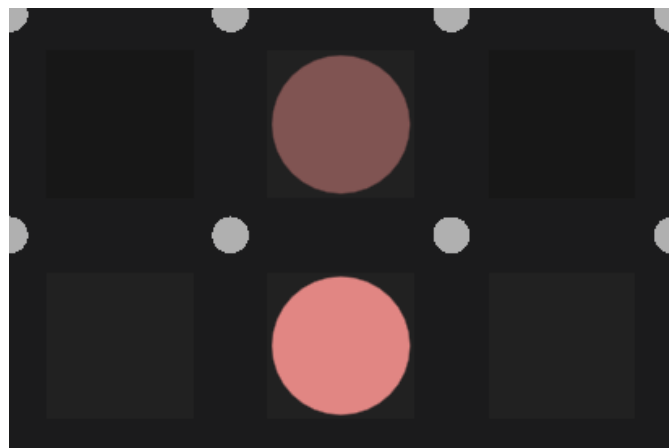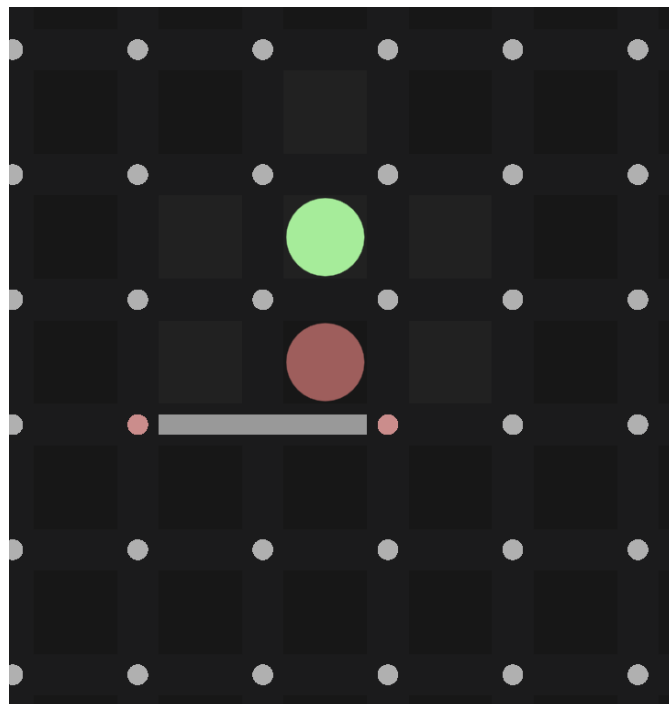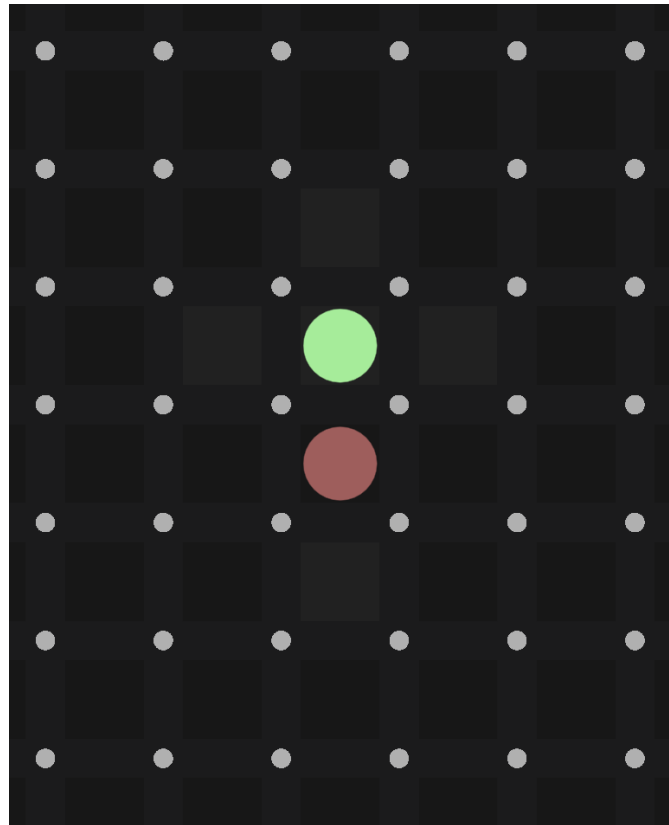

Figure 4: Pawn Movement.
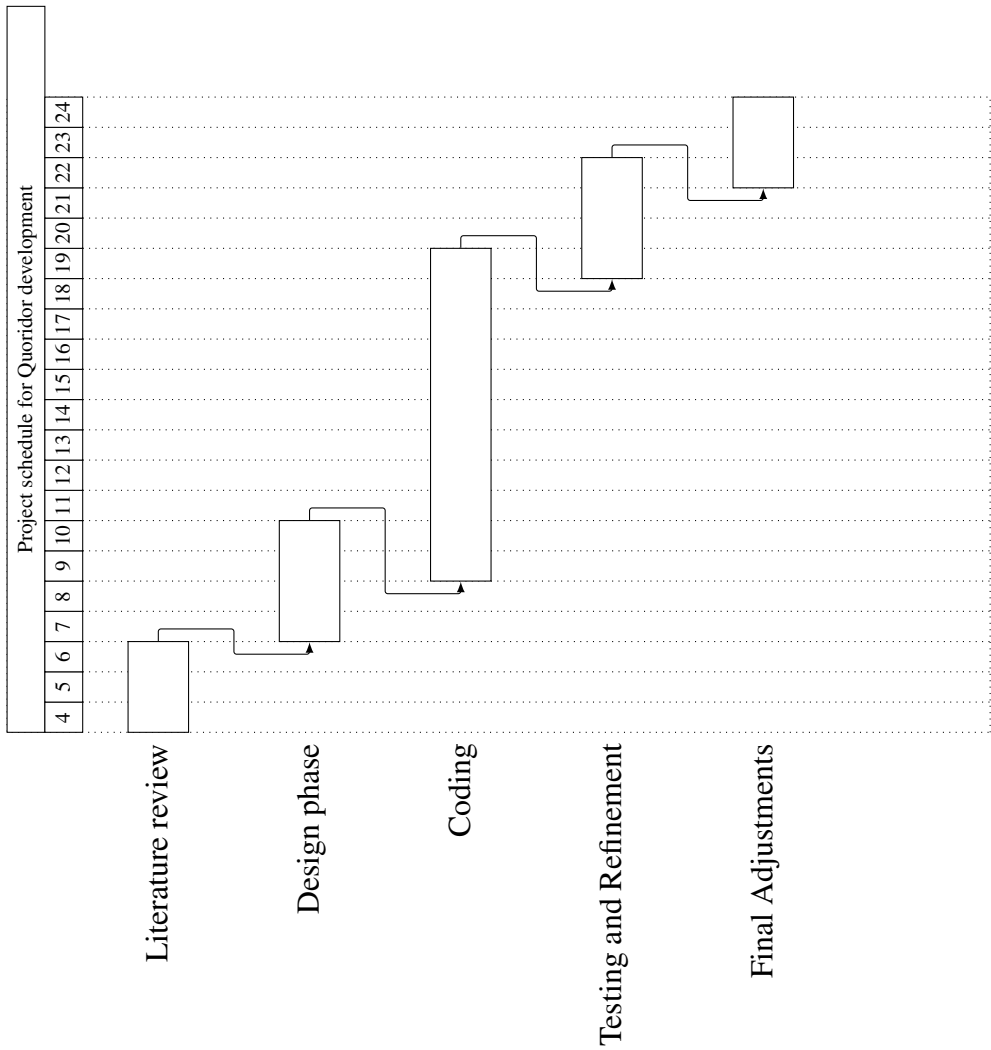
Figure 5: A fence blocking a leap.

Figure 6: Pawn Leaping.

Figure 7: Quoridor Development Gantt Chart