1 Design

1.1 System Architecture

In this section, I will lay out the overall logic, and an overview of the steps involved in running my program. By decomposing the program into individual abstracted stages, I can focus on the workings and functionality of each section individually, which makes documenting and coding each section easier.I have also included a flowchart to illustrate the logic of each screen of the program.

I will also create an abstracted GUI prototype in order to showcase the general functionality of the user experience, while acting as a reference for further stages of graphical development. It will consist of individually drawn screens for each stage of the program, as shown in the top-level overview. The elements and layout of each screen are also documented below.

The following is a top-level overview of the logic of the program:

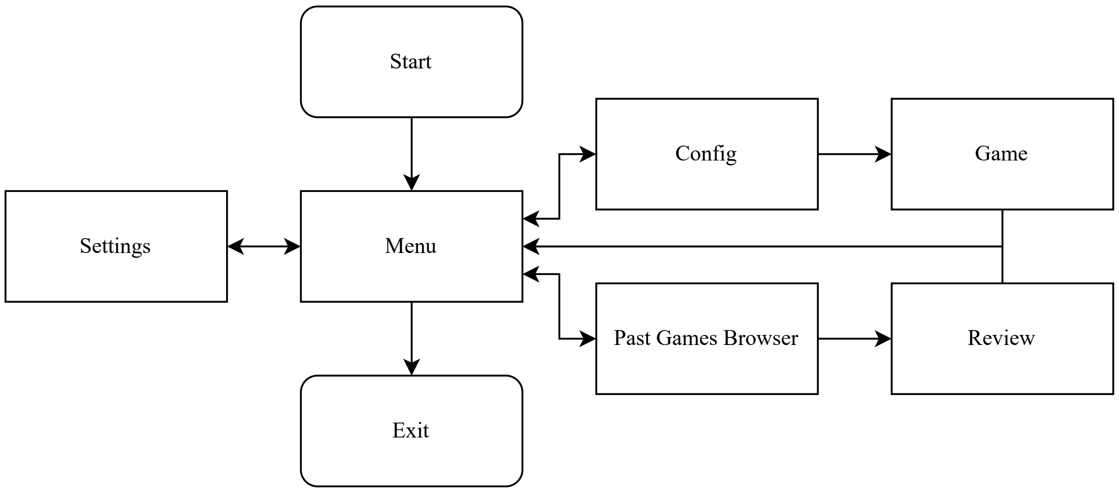


Figure 1: Flowchart for Program Overview

1.1.1 Main Menu

The main menu will be the first screen to be displayed, providing access to different stages of the game. The GUI should be simple yet effective, containing clearly-labelled buttons for the user to navigate to different parts of the game.

1.1.2 Settings

The settings menu allows for the user to customise settings related to the program as a whole. The settings will be changed via GUI elements such as buttons and sliders, offering the ability to customize display mode, volume, board colour etc. Changes to settings will be stored in an intermediate code class, then stored externally into a JSON file. Game settings will instead be changed in the Config screen.

The setting screen should provide a user-friendly interface for changing the program settings intuitively; I have therefore selected appropriate GUI widgets for each setting:

- Windowed or Fullscreen - Drop-down list for selecting between pre-defined options

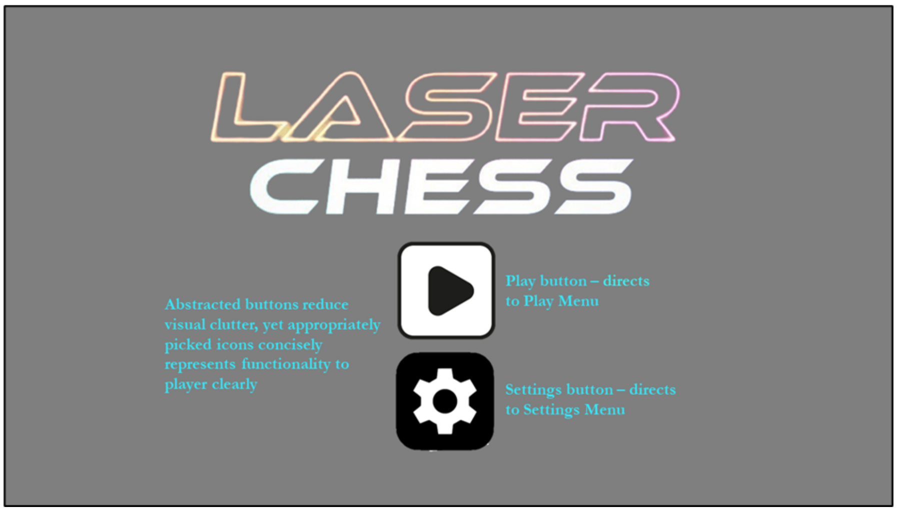
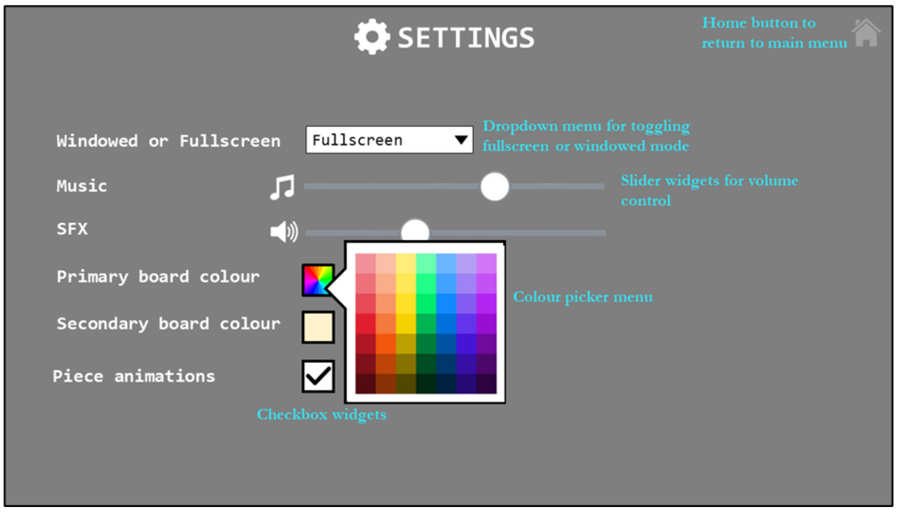

LASER
CHESS
Abstracted buttons reduce visual clutter, yet appropriately picked icons concisely represents functionality to player clearly
Play button – directs to Play Menu
Settings button – directs to Settings Menu

Figure 2: Main Menu screen prototype


SETTINGS
Home button to return to main menu
Dropdown menu for toggling fullscreen or windowed mode
Slider widgets for volume control
Colour picker menu
Checkbox widgets

Windowed or Fullscreen   Fullscreen
Music
SFX
Primary board colour
Secondary board colour
Piece animations   ✓

Figure 3: Settings screen prototype

- Music and SFX - Slider for selecting audio volume, a continuous value
- Board colour - Colour grid for the provision of multiple pre-selected colours
- Piece animation - Checkbox for toggling between on or off

Additionally, each screen is provided with a home button icon on the top right (except the main menu), as a shortcut to return to the main menu.

1.1.3 Past Games Browser

The Past Games Browser menu displays a list of previously played games to be replayed. When selecting a game, the replay will render out the saved FEN string into a board position identical

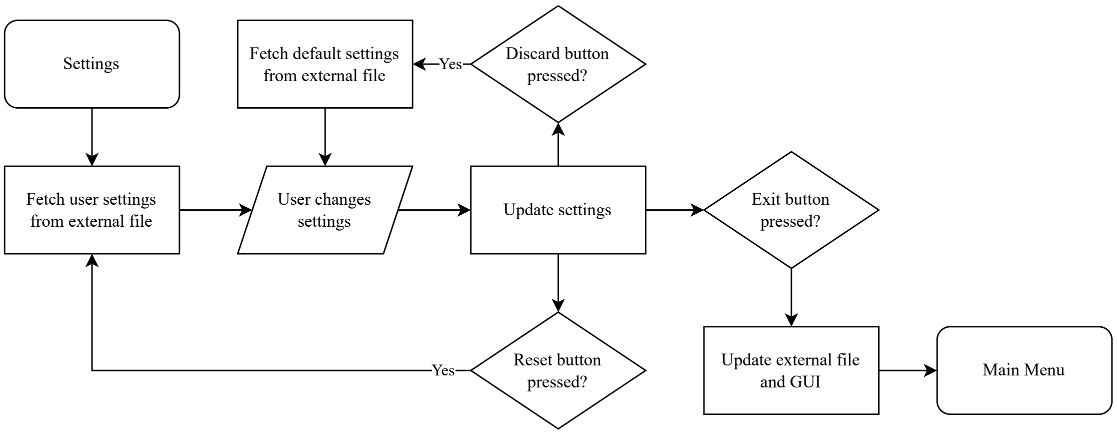


Figure 4: Flowchart for Settings

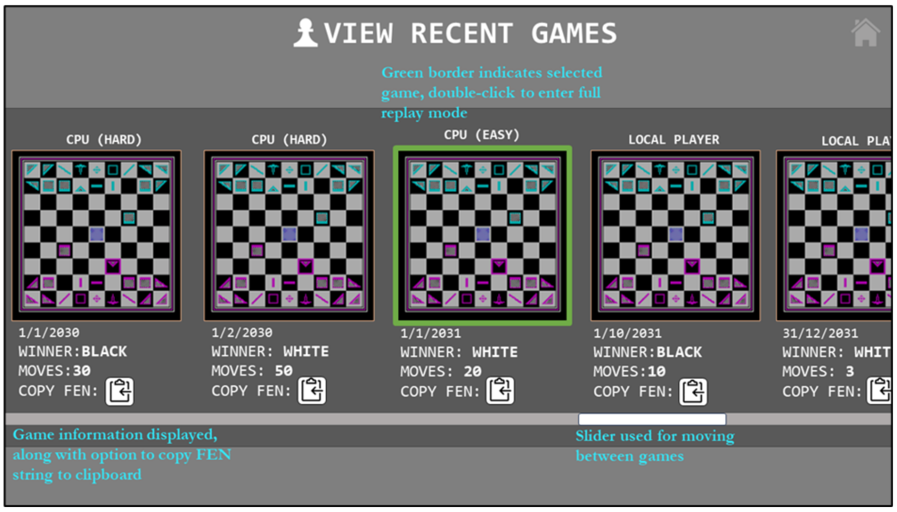


Figure 5: Browser screen prototype

to the one played previously, except the user is limited to replaying back and forth between recorded moves. The menu also offers the functionality of sorting games in terms of time, game length etc.

For the GUI, previous games will be displayed on a strip, scrolled through by a horizontal slider. Information about the game will be displayed for each instance, along with the option to copy the FEN string to be stored locally or to be entered into the Review screen. When choosing a past game, a green border will appear to show the current selection, and double clicking enters the user into the full replay mode. While replaying the game, the GUI will appear identical to an actual game. However, the user will be limited to scrolling throughout the moves via the left and right arrow keys.

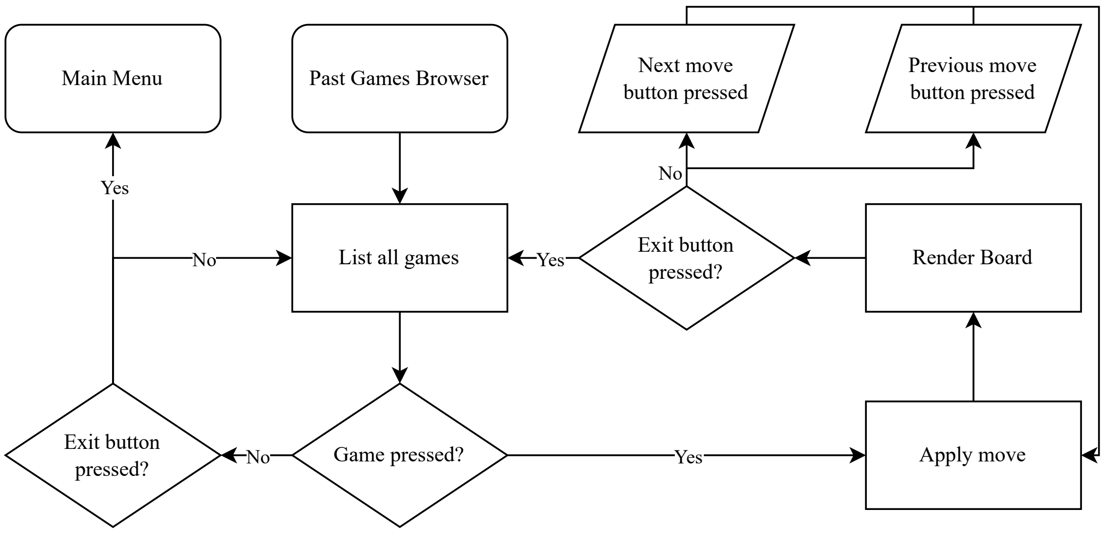
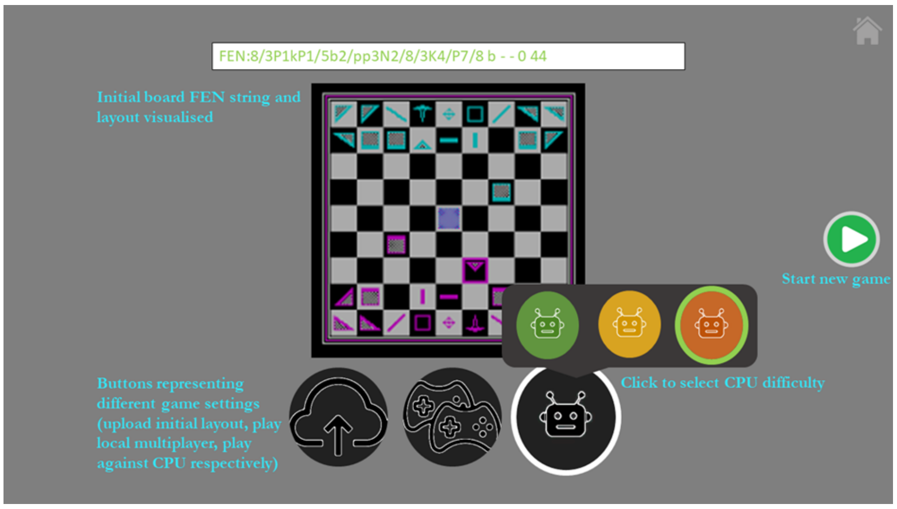


Figure 6: Flowchart for Browser


Initial board FEN string and layout visualised
Start new game
Click to select CPU difficulty
Buttons representing different game settings (upload initial layout, play local multiplayer, play against CPU respectively)

FEN:8/3P1kP1/5b2/pp3N2/8/3K4/P7/8 b - - 0 44

Figure 7: Config screen prototype

1.1.4 Config

The config screen comes prior to the actual gameplay screen. Here, the player will be able to change game settings such as toggling the CPU player, time duration, playing as white or black etc.

The config menu is loaded with the default starting position. However, players may enter their own FEN string as an initial position, with the central board updating responsively to give a visual representation of the layout. Players are presented with the additional options to play against a friend, or against a CPU, which displays a drop-down list when pressed to select the CPU difficulty.

$$\begin{tikzcd}
 & \text{Game} & \text{Menu} & \text{Board Editor} & & \\
\text{Config} \arrow[r] & \text{Start button pressed?} \arrow[u, "Yes"] \arrow[r, "No"] & \text{Exit button pressed?} \arrow[u, "Yes"] \arrow[r, "No"] & \text{Editor button pressed?} \arrow[u, "Yes"] \arrow[r, "No"] & \text{Game config changed?} \arrow[r, "Yes"] \arrow[lll, bend left, "No"] & \text{Update game settings} \arrow[llll, bend left]
\end{tikzcd}$$

Figure 8: Flowchart for Config

1.1.5 Game

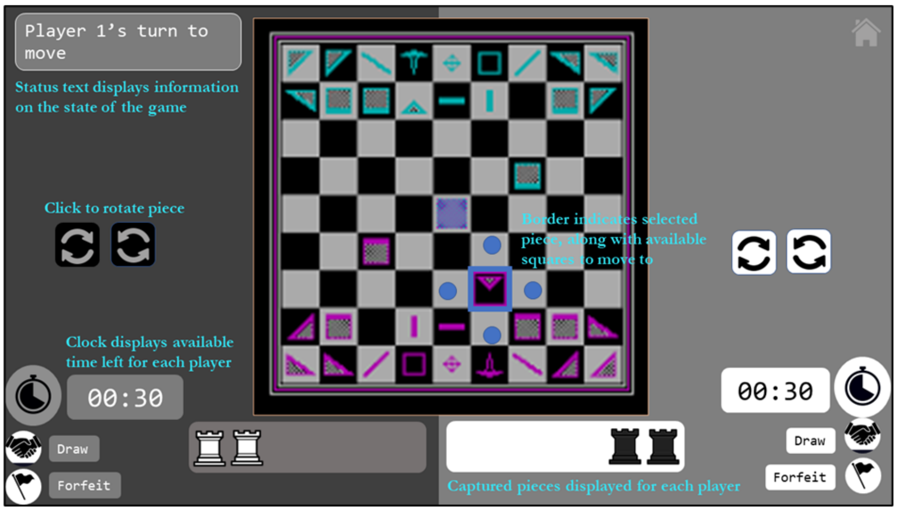

Player 1's turn to move
Status text displays information on the state of the game
Click to rotate piece
Border indicates selected piece, along with available squares to move to
Clock displays available time left for each player
00:30
00:30
Draw
Forfeit
Draw
Forfeit
Captured pieces displayed for each player

Figure 9: Game screen prototype

During the game, handling of the game logic, such as calculating player turn, calculating CPU moves or laser trajectory, will be computed by the program internally, rendering the updated GUI accordingly in a responsive manner to provide a seamless user experience.

In the game screen, the board is positioned centrally on the screen, surrounding by accompanying widgets displaying information on the current state of the game. The main elements include:

- Status text - displays information on the game state and prompts for each player move
- Rotation buttons - allows each player to rotate the selected piece by 90° for their move
- Timer - displays available time left for each player
- Draw and forfeit buttons - for the named functionalities, confirmed by pressing twice
- Piece display - displays material captured from the opponent for each player

Additionally, the current selected piece will be highlighted, and the available squares to move to will also contain a circular visual cue. Pieces will either be moved by clicking the target square, or via a drag-and-drop mechanism, accompanied by responsive audio cues. These implementations aim to improve user-friendliness and intuitiveness of the program.

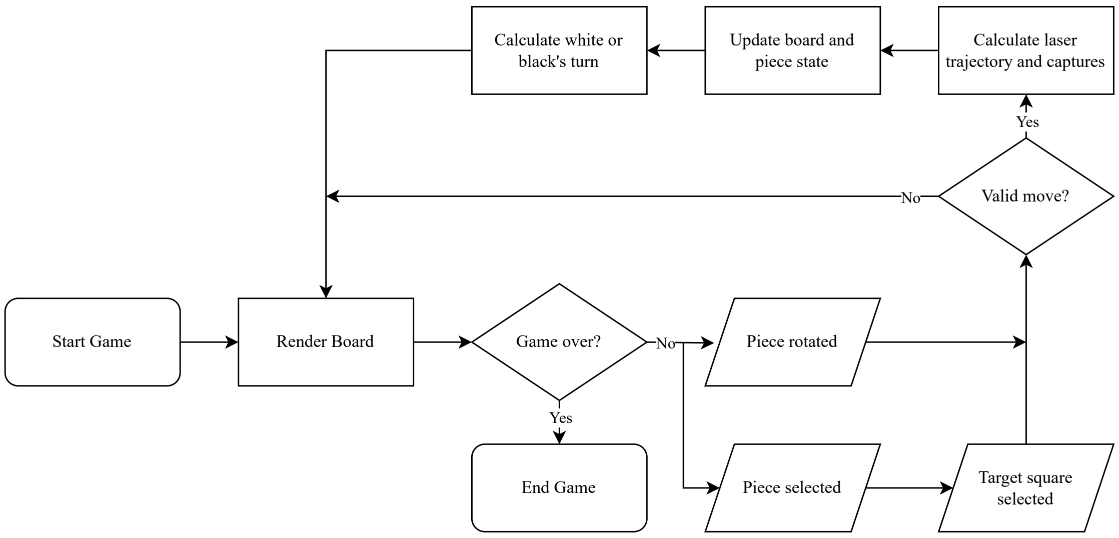


Figure 10: Flowchart for Game

1.1.6 Board Editor

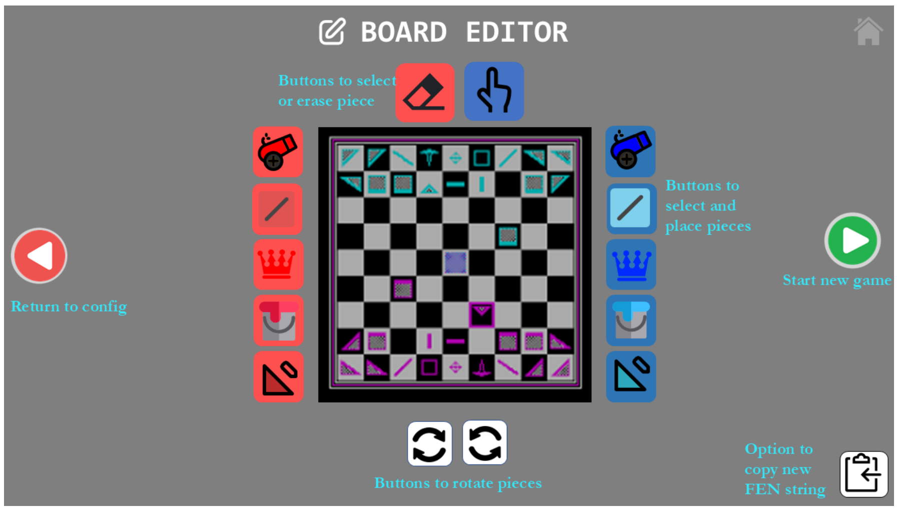

BOARD EDITOR
Buttons to select or erase piece
Buttons to select and place pieces
Return to config
Start new game
Option to copy new FEN string
Buttons to rotate pieces

Figure 11: Editor screen prototype

The editor screen is used to configure the starting position of the board. Controls should include the ability to place all piece types of either colour, to erase pieces, and easy board manipulation shortcuts such as dragging pieces or emptying the board.

For the GUI, the buttons should clearly represent their functionality, through the use of icons and appropiate colouring (e.g. red for delete).

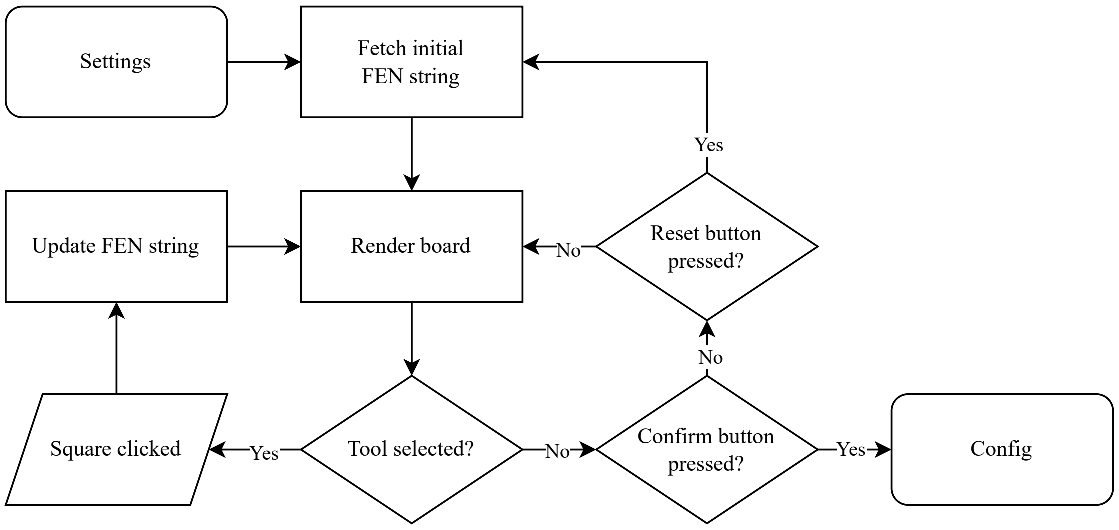


Figure 12: Flowchart for board editor

1.2 Algorithms and Techniques

1.2.1 Minimax

Minimax is a backtracking algorithm commonly used in zero-sum games used to determine the score according to an evaluation function, after a certain number of perfect moves. Minimax aims to minimize the maximum advantage possible for the opponent, thereby minimizing a player's possible loss in a worst-case scenario. It is implemented using recursive depth-first search, alternating between minimizing and maximizing the player's advantage in each recursive call.

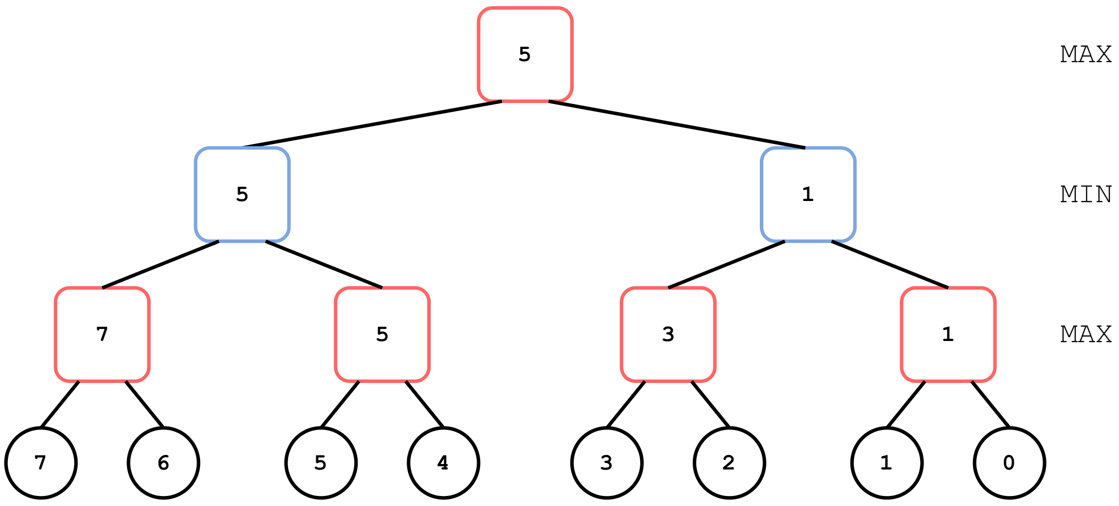


Figure 13: Example minimax tree

For the example minimax tree show in Figure 13, starting from the bottom leaf node evalua-

tions, the maximizing player would choose the highest values (7, 5, 3, 1). From those values, the minimizing player would choose the lowest values (5, 1). The final value chosen by the maximum player would therefore be the highest of the two, 5.
Implementation in the form of pseudocode is shown below:

Algorithm 1 Minimax pseudocode

```
function MINIMAX(node, depth, maximizingPlayer)
    if $depth = 0$ OR node equals game over then
        return EVALUATE
    end if

    if maximisingPlayer then
        $value \leftarrow -\infty$
        for child of node do
            $value \leftarrow \text{MAX}(value, \text{MINIMAX}(child, depth - 1, false))$
        end for
        return $value$
    else
        $value \leftarrow +\infty$
        for child of node do
            $value \leftarrow \text{MIN}(value, \text{MINIMAX}(child, depth - 1, true))$
        end for
        return $value$
    end if
end function
```

1.2.2 Minimax improvements

Alpha-beta pruning

Alpha-beta pruning is a search algorithm that aims to decrease the number of nodes evaluated by the minimax algorithm. Alpha-beta pruning stops evaluating a move in the game tree when one refutation is found in its child nodes, proving the node to be worse than previously-examined alternatives. It does this without any potential of pruning away a better move. The algorithm maintains two values: alpha and beta. Alpha (α), the upper bound, is the highest value that the maximizing player is guaranteed of; Beta (β), the lower bound, is the lowest value that the minimizing player is guaranteed of. If the condition $\alpha \geq \beta$ for a node being evaluated, the evaluation process halts and its remaining children nodes are 'pruned'.
This is shown in the following maximizing example:

Since minimax is a depth-first search algorithm, nodes C and G and their α and β have already been searched. Next, at node F, the current α and β are $-\infty$ and 1 respectively, since the β is passed down from node C. Searching the first leaf node, the α subsequently becomes $\alpha = max(-\infty, 2)$. This means that the maximising player at this depth is already guaranteed an evaluation of 2 or greater. Since we know that the minimising player at the depth above is guaranteed a value of 1, there is no point in continuing to search node F, a node that is returns a value of 2 or greater. Hence at node F, where $\alpha \geq \beta$, the branches are pruned.

Alpha-beta pruning therefore prunes insignificant nodes by maintain an upper bound α and lower bound β. This is an essential optimization as a simple minimax tree increases exponentially

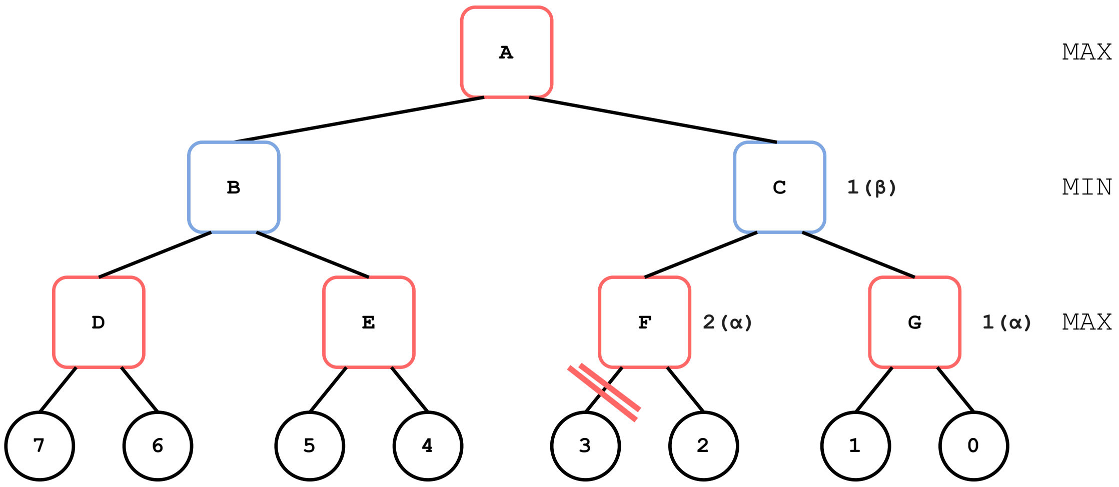


Figure 14: Example minimax tree with alpha-beta pruning

in size with each depth ($O(b^d)$, with branching factor b and d ply depth), and alpha-beta reduces this and the associated computational time considerably.
The pseudocode implementation is shown below:

Transposition Tables & Zobrist Hashing

Transition tables, a memoisation technique, again greatly reduces the number of moves searched. During a brute-force minimax search with a depth greater than 1, the same positions may be searched multiple times, as the same position can be reached from different sequences of moves. A transposition table caches these same positions (transpositions), along with the its associated evaluations, meaning commonly reached positions are not unnecessarily re-searched.

Flags and depth are also stored alongside the evaluation. Depth is required as if the current search comes across a cached position with an evaluation calculated at a lower depth than the current search, the evaluation may be inaccurate. Flags are required for dealing with the uncertainty involved with alpha-beta pruning, and can be any of the following three.

Exact flag is used when a node is fully searched without pruning, and the stored and fetched evaluation is accurate.

Lower flag is stored when a node receives an evaluation greater than the β, and is subsequently pruned, meaning that the true evaluation could be higher than the value stored. We are thus storing the α and not an exact value. Thus, when we fetch the cached value, we have to recheck if this value is greater than β. If so, we return the value and this branch is pruned (fail high); If not, nothing is returned, and the exact evaluation is calculated.

Upper flag is stored when a node receives an evaluation smaller than the α, and is subsequently pruned, meaning that the true evaluation could be lower than the value stored. Similarly, when we fetch the cached value, we have to recheck if this value is lower than α. Again, the current branch is pruned if so (fail low), and an exact evaluation is calculated if not.
The pseudocode implementation for transposition tables is shown below:

Algorithm 2 Minimax with alpha-beta pruning pseudocode

```
function MINIMAX(node, depth, $\alpha$, $\beta$, maximizingPlayer)
    if $depth = 0$ OR node equals game over then
        return EVALUATE
    end if

    if maximisingPlayer then
        $value \leftarrow -\infty$
        for child of node do
            $value \leftarrow \text{MAX}(value, \text{MINIMAX}(child, depth - 1, \alpha, \beta, false))$
            if $value > \beta$ then break
            end if
            $\alpha \leftarrow \text{MAX}(\alpha, value)$
        end for
        return $value$
    else
        $value \leftarrow +\infty$
        for child of node do
            $value \leftarrow \text{MIN}(value, \text{MINIMAX}(child, depth - 1, \alpha, \beta, true))$
            if $value < \alpha$ then break
            end if
            $\beta \leftarrow \text{MIN}(\beta, value)$
        end for
        return $value$
    end if
end function
```

Algorithm 3 Minimax with transposition table pseudocode

```
function MINIMAX(node, depth, $\alpha$, $\beta$, maximizingPlayer)
    $hash\_key \leftarrow$ HASH($node$)
    $entry \leftarrow$ GETENTRY($hash\_key$)

    if $entry.hash\_key = hash\_key$ AND $entry.hash\_key \geq depth$ then
        if $entry.hash\_key =$ EXACT then
            return $entry.value$
        else if $entry.hash\_key =$ LOWER then
            $\alpha \leftarrow$ MAX($\alpha, entry.value$)
        else if $entry.hash\_key =$ UPPER then
            $\beta \leftarrow$ MIN($\beta, entry.value$)
        end if
        if $\alpha \geq \beta$ then
            return $entry.value$
        end if
    end if

     ...normal minimax...

    $entry.value \leftarrow value$
    $entry.depth \leftarrow depth$
    if $value \leq \alpha$ then
        $entry.flag \leftarrow$ UPPER
    else if $value \geq \beta$ then
        $entry.flag \leftarrow$ LOWER
    else
        $entry.flag \leftarrow$ EXACT
    end if

     return $value$
end function
```

The current board position will be used as the index for a transposition table entry. To convert our board state and bitboards into a valid index, Zobrist hashing may be used. For every square on the chessboard, a random integer is assigned to every piece type (12 in our case, 6 piece type, times 2 for both colours). To initialise a hash, the random integer associated with the piece on a specific square undergoes a XOR operation with the existing hash. The hash is incrementally update with XOR operations every move, instead of being recalculated from scratch improving computational efficiency. Using XOR operations also allows moves to be reversed, proving useful for the functionality to scroll throughh previous moves. A Zobrist hash is also a better candidate than FEN strings in checking for threefold-repetition, as they are less intensive to calculate for every move.
The pseudocode implementation for Zobrist hashing is shown below:

Algorithm 4 Zobrist hashing pseudocode

```
*RANDOMINTS represents a pre-initialised array of random integers for each piece type for each square*

function HASHBOARD(board)
    $hash \leftarrow 0$
    for each square on board do
        if square is not empty then
            $hash \oplus RANDOMINTS[square][piece\ on\ square]$
        end if
    end for
    return hash
end function

function UPDATEHASH(hash, move)
    $hash \oplus RANDOMINTS[source\ square][piece]$
    $hash \oplus RANDOMINTS[destination\ square][piece]$
    if red to move then
        $hash \oplus hash\ for\ red\ to\ move$ ▷ Hash needed for move colour, as two identical positions are different if the colour to move is different
    end if
    return hash
end function
```

1.3 Classes