(Vecteezy, n.d.)

Herm0ni Chess Bot

Design Document 2024-2025

Student Name: Seán Rourke

Student Number: C00251168

Supervisor: Joseph Kehoe

Contents

1. Introduction

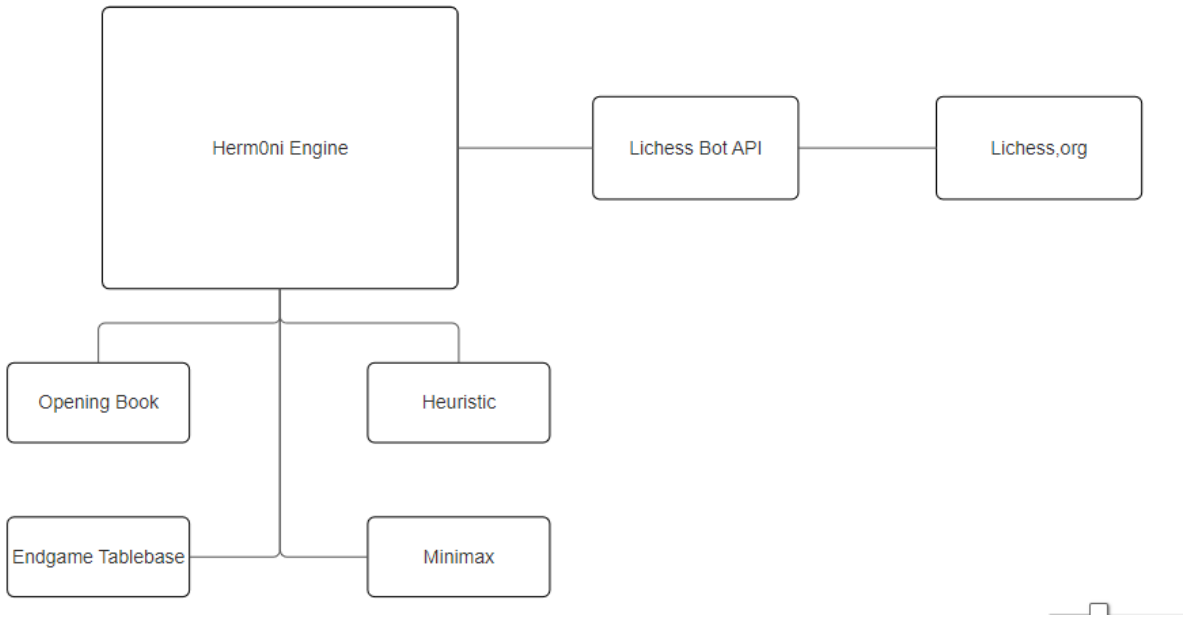
Herm0ni is an artificial intelligence (AI) chess bot aimed at being played by players for practice or fun, and other chess bots to determine comparative strength. The bot is designed to evaluate chess positions and determine the next move to make without the need for human input.

1.1. Document Purpose

The purpose of this document is to detail the design and structure of the Herm0ni bot and its implementation on lichess.org. Interactions between different parts of the system will be illustrated using sequence diagrams. This document will contain details of the various technologies used to create the bot and the reasoning behind using those technologies. The data structures used by the bot will be explained and visualised. This document will also contain details of the various complex algorithms that the bot uses to play chess.

2. System Architecture

2.1. Component Architecture



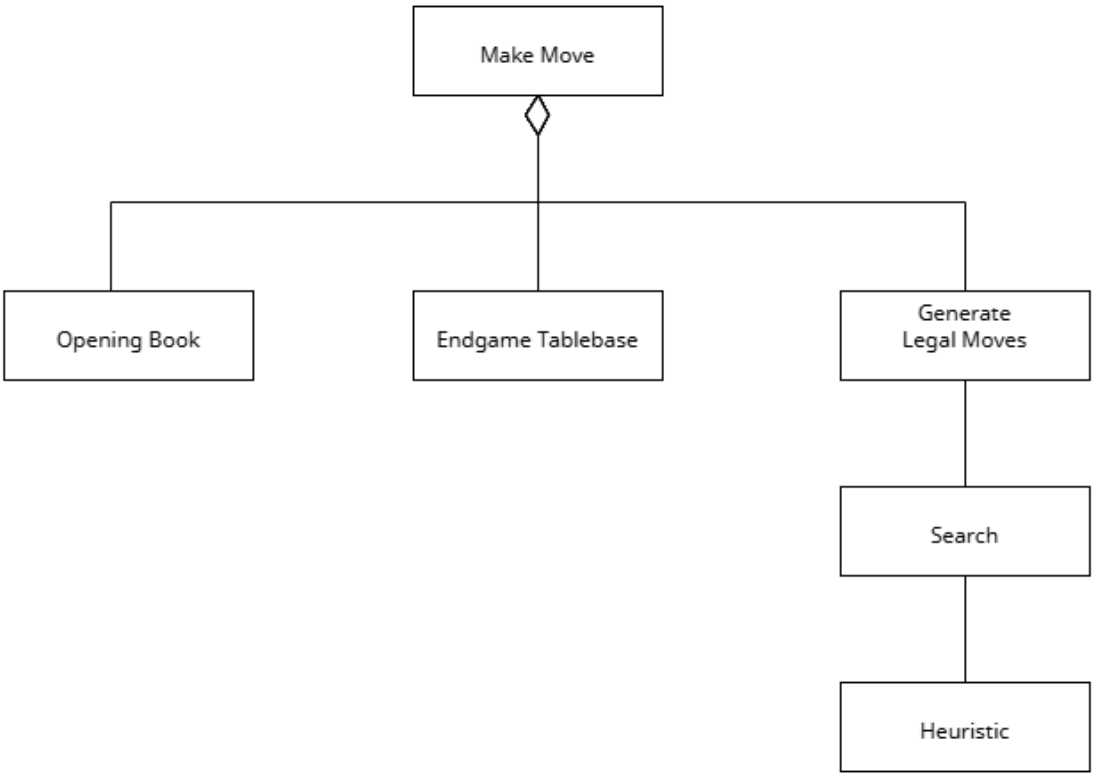
2.2. Chess Engine

The engine decides on its next move through one of three methods.

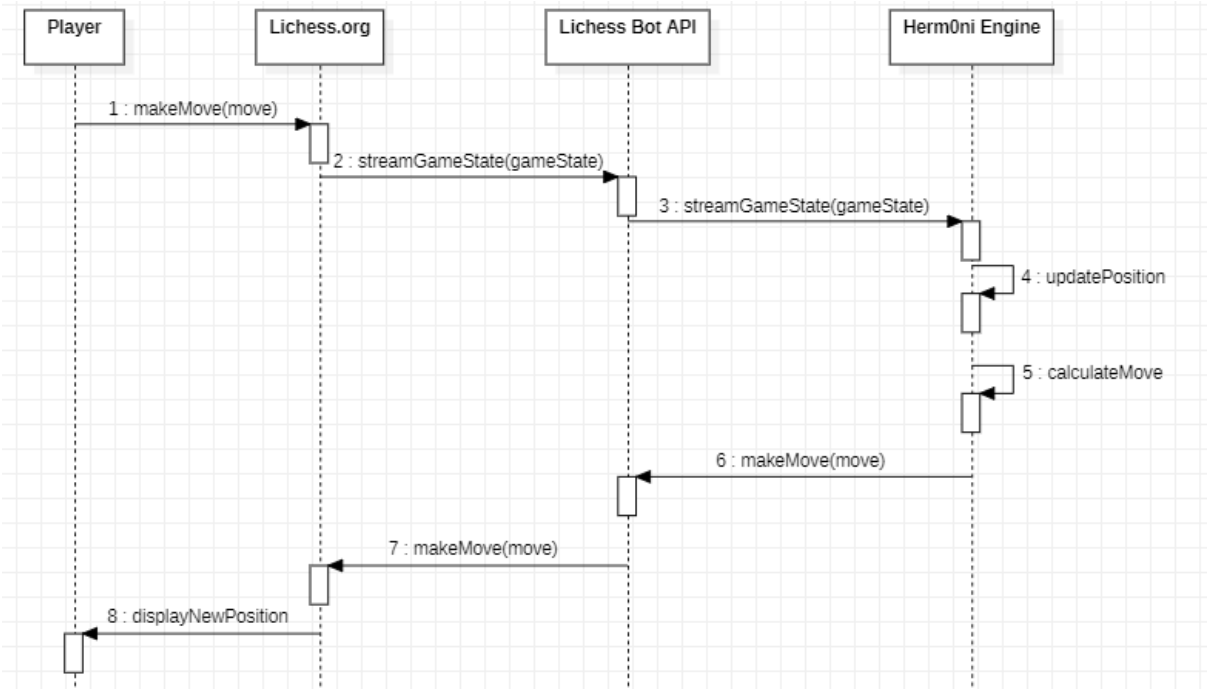
It can utilise an opening book that contains a sequence of moves it must follow for the initial moves of the game.

It can use an endgame tablebase once there are less than eight pieces left on the board. The tablebase contains the best move in every scenario for any combination of seven pieces or less.

If neither the opening book nor endgame tablebase can be used, the engine will calculate its next move using its own algorithms.



3. Sequence Diagram



4. Technologies

4.1. C++

The code for the bot itself will be written in C++. The high speed of C++ is beneficial for efficient computation of complex algorithms. This is especially crucial in short chess games, such as blitz or bullet games. Being able to perform these algorithms quickly ensures that the skill level of the bot doesn't take a hit when the time it it given to make each move is reduced.

4.2. Lichess Bot API

The lichess bot API allows the bot to interact with the lichess.org website. The API allows a bot account on lichess.org to be controlled by the engine. It can send commands to the engine and process responses. The API send messages to the engine in JSON. Information about moves made by the player is presented in universal chess interface (UCI).

4.3. Lichess Bot Client

The lichess bot client is a bridge between the lichess bot API and the chess engine. The client allows the engine to interact with the website, granting the ability for the bot to play games against other bots or real players on lichess.org.

Having the bot implemented on lichess.org means that time and effort does not need to be spent developing a graphical user interface (GUI) for the games to be played. This means more time can be spent optimising the engine's performance for improved chess ability.

4.4. Hosting Platform

The hosting platform for the bot is currently undecided. Google Cloud and Hetzner are currently being considered as options

5. Data Structures

5.1. Bitboards

The position of a game is stores using bitboards. These are implemented using 64-bit integers, with each piece for each colour having their own bitboard. A 64-bit integer is used as each bit can represent a square of the chess board. A 1 bit in the integer represents the presence of a piece on that square. For example, at the beginning of the game, the white pawns take up eight squares along the second rank. For example, using a 64-bit integer, the white pawns are initialised as seen below.

Bitboard whitePawns = 0x000000000000FF00;

Separate bitboards are stores for each piece of each colour, along with a bitboard for all white pieces, all black pieces, and all pieces. This allows for easy checking of is a square occupied and is a piece able to be captured.

Operations can be performed on these bitboards to easily update positions. For example, the bitboard containing all pieces is created by combining the bitboards for the white and black pieces.

allPieces = whitePieces | blackPieces;

5.2. Minimax Tree

When performing through the minimax algorithm, each position after a potential move represents a node on the tree. The root node represents the current game state and leaf nodes are made up of positions reached after looking ahead three moves, or when a worse position is discovered, and the branch is pruned.

6. Algorithms

6.1. Heuristic Algorithm

The heuristic algorithm is used to evaluate a position, determining who is in a more advantageous position, white or black, or whether the position is equal. An advantage for white will be represented with a positive number, an advantage for black will be represented with a negative number, an evaluation of 0 means that the position is equal. This algorithm is based on many factors. All evaluation functions will be run once, with values being added to the evaluation for the white pieces and subtracted from the evaluation for the black pieces. This is more efficient than running each function separately for the white and black pieces and then subtracting the black evaluation from the white.

6.1.1. Heuristic Function:

```
function EvaluatePosition()
    score = 0
    for each piece in board
        if white.owner == "white"
            score += EvaluatePiece(piece)
        else if piece.owner == "black"
            score -= EvaluatePiece(piece)
        end if
    end for
    return score
end function


function EvaluatePiece(piece)
```


```
    score = 0
    score += MaterialValue(piece)
    score += PieceActivity(piece)
    if piece.type == "king"
        score += KingSafety(piece)
    else if piece.type == "pawn"
        score += PawnStructure(piece)
    end if
    score += ControlOfCenter(piece)
    score += Development(piece)
    score += SpaceContribution(piece)
    score += Coordination(piece)
    score += Mobility(piece)
    return score
end function
```

The most basic evaluation factor material count. Each piece is given a value:

- Pawn: 1
- Knight: 3
- Bishop: 3
- Rook: 5
- Queen: 9
- King: 0 (not directly scored)

Material Count Funtion

```
function MaterialBalance()
    score = 0
    for each piece in board
        if piece.type == "pawn"
            value = 1
        else if piece.type == "knight" or piece.type == "bishop"
            value = 3
        else if piece.type == "rook"
```


```
            value = 5
        else if piece.type == "queen"
            value = 9
        else
            value = 0

        if piece.owner == "white"
            score += value
        else if piece.owner == "black"
            score -= value
        end if
    end for
    return score
end function
```

Piece activity is another factor to be considered when evaluating a position. For example, a bishop in the middle of the board can see more squares than at the edge of the board and is considered more active.

Piece Activity Function

```
function PieceActivity()
    score = 0
    for each piece in board
        legalMoves = generateLegalMoves(piece)
        activity = length(legalMoves) * activityWeight(piece)

        if piece.owner = "white"
            score += activity
        else if piece.owner = "black"
            score -= activity
        end if
    end for
    return score
```


```
end function
```

A major factor in the evaluation of a position is king safety.

King Safety Function

```
funtion KingSafety()
    score = 0
    for each king in board
        safety = 0
        if isFileOpen(king.position) or isDiagonalOpen(king.position)
            score -= 2
        end if
        for each square in AdjacentSquares(king.position)
            if board[square] != "pawn" or board[square]. owner != king.owner
                safety -= 1
            end if
        end for

        if king.owner = "white"
            score += safety
        else if king.owner = "black"
            score -= safety
        end if
    end for
    return score
end function
```

Pawn structure is also an important factor in evaluating a position. Isolated and doubled pawns are considered a negative, whereas passed pawns are considered a positive.

Pawn Structure Function

```
function PawnStructure
```


```
    score = 0
    for each pawn in board
        structure = 0
        if isIsolated(pawn)
            structure -= 1
        if isDoubled(pawn)
            structure -= 1
        if isPassed(pawn)
            structure += 2
        end if
        if pawn.owner = "white"
            score += structure
        else if pawn.owner = "black"
            score -= structure
        end if
    end for
    return score
end function
```

6.2. Minimax with Alpha-Beta Pruning

The minimax algorithm is used for the calculation of the next move. From a position, all possible moves are considered, then the possible responses to this move are considered. The position after these moves is evaluated to determine the best move to make.

Minimax Function

```
function Minimax(position, depth, alpha, beta, maximisingPlayer)
    if depth == 0 or gameOver(position)
        return evaluate(position)
    end if

    if maximisingPlayer
```


```
        maxEval = -infinity
        for move in generateLegalMoves(position)
            makeMove(position, move)
            eval = minimax(position, depth -1, alpha, beta, false)
            undoMove(position, move)
            maxEval = max(alpha, eval)
            alpha = max(alpha, eval)
            if beta <= alpha
                break
            end if
        end for
        return maxEval
    else
        minEval = infinity
        for move in generateLegalMoves(position)
            makeMove(position, move)
            eval = minimax(position, depth, alpha, beta, true)
            undoMove(position, move)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha
                break
            end if
        end for
        return minEval
    end if
end function

function BestMove
    bestEval = -infinity
    bestMove = null
    alpha = -infinity
```


```
    beta = infinity

    for move in generateLegalMoves(position)
        moveMove(position, move)
        eval = minimax(position, depth -1, alpha, beta, false)
        undo_move(position, move)
        if eval > bestEval
            bestEval = eval
            bestMove = move
        end is
    end for
    return bestMove
end function
```

References

Vecteezy, n.d. *Vecteezy.* [Online]
Available at: https://www.vecteezy.com/free-vector/chess-silhouette
[Accessed 26 November 2024].