CHESS AI: Minimax Based Chess Engine

Kadir Bora Kara Author
TOBB ETU

Abstract-These - This paper outlines the creation of a Chess Engine featuring custom APIs for managing the chessboard and pieces, and a chess bot developed on top of these APIs. The bot uses the Minimax algorithm with Alpha-Beta pruning for improved efficiency in move selection. Key to this engine is the use of heuristic values based on the author's personal chess playing experience, which help the bot make smarter decisions.

I. Introduction

Chess, with its historical origins stretching back to the 6th century, began as "Chaturanga" in India, a game that mirrored the key components of an army: infantry, cavalry, elephants, and chariots. Over the centuries, it transformed into the strategic and intellectual game we know today, where the goal is to checkmate the opponent's king. Governed by the International Chess Federation (FIDE), chess not only challenges the mind but also offers a window into the strategic thinking of different cultures and eras.

In recent years, the intersection of chess and technology has opened new avenues for exploring artificial intelligence. The development of Chess Engines, which can simulate human-like strategic thinking, has become a field of keen interest. This paper delves into creating such an engine, employing the Minimax algorithm coupled with Alpha-Beta pruning.

II. Literature Review

A. Chess Programming Wiki

The development of this Chess Engine was informed by a thorough review of existing resources and strategies in the field. Key among these was the Chess Programming Wiki, a comprehensive platform detailing various coding algorithms and methods essential for building competitive chess bots [1]. This resource proved invaluable for understanding the core techniques used in chess programming.

B. Sebastian Lague's Chess AI Videos

In the Chess AI video series by Sebastian Lague, the construction of a chess bot is approached through the integration of several key methodologies [2]. Notably, the series covers the use of the Minimax algorithm and the enhancement of decision-making efficiency through Alpha-Beta pruning. The representation of the chessboard is addressed through the utilization of a bitboard format, providing a compact and efficient means of managing game states.

Lague's series further delves into the application of custom heuristic values tailored to the evaluation of chess positions, adding depth to the bot's strategic analysis. Additional techniques explored include move prioritization, which aids in the efficient exploration of the search space, and specialized endgame strategies aimed at maximizing positional advantage. The handling of specific chess elements, such as passed pawns and their implications for gameplay [3].

C. Tiny Chess Bot Tournament

The Tiny Chess Bot Tournament, hosted by Sebastian Lague, provided valuable insights into practical AI strategies for chess [4]. Observing the variety of approaches and analyzing how different bots performed in this tournament gave a clear view of effective tactics in the realm of chess AI [5]. This real-world application helped in choosing the right strategies for this project, ensuring a blend of innovation and proven effectiveness.

The winning bot in the Tiny Chess Bot Tournament, Boychesser (Bot_614), leverages a sophisticated combination of the Minimax algorithm with Alpha-Beta pruning and a bitboard for chessboard representation, ensuring efficient game state management. It stands out for its use of custom heuristic values for nuanced board evaluation and strategic move prioritization, allowing for swift yet deep analysis of potential moves. This bot's strategic prowess, particularly in endgame tactics and pawn management, played a pivotal role in its success in the tournament [6].

III. Methodology

MyBot, a chess engine, is designed with a deep understanding of chess strategies and employs a multifaceted approach to gameplay, combining classical chess principles with advanced computational techniques. At the foundation of MyBot's decision-making process is the Minimax algorithm, enhanced with alpha-beta pruning to efficiently explore the vast game tree of chess. This algorithm is dynamically adjusted in depth based on the game phase, allowing deeper searches in more critical positions where fewer pieces are on the board, thus optimizing computational resources while maintaining strategic depth.

During the Minimax search process, capture moves are prioritized for evaluation, enabling the desired moves to be identified more swiftly. This approach serves as an effective optimization, significantly enhancing the engine's performance.

```
Function MiniMax(node, depth, alpha, beta, maximizingPlayer)
    Check for terminal conditions:
        If depth is 0 OR game is in checkmate OR game is a draw
            Calculate and set the heuristic value of the node
            If game is in checkmate
                Adjust the heuristic value based on depth
            Return the node

    Generate all possible child moves for the current node

    If maximizingPlayer
        Initialize bestValue to negative infinity
        For each child move of the current node
            Make the move on the chessboard
            Recursively call MiniMax for the child node with decreased depth,
            Undo the move on the chessboard
            Update bestValue if the returned value is greater
            Update alpha with the maximum of alpha and bestValue
            Break if bestValue is greater or equal to beta (alpha-beta prune)
        Return the node with bestValue

    Else (minimizing player)
        Initialize bestValue to positive infinity
        For each child move of the current node
            Make the move on the chessboard
            Recursively call MiniMax for the child node with decreased depth,
            Undo the move on the chessboard
            Update bestValue if the returned value is less
            Update beta with the minimum of beta and bestValue
            Break if bestValue is less or equal to alpha (alpha-beta prune)
        Return the node with bestValue
End Function
```

Minimax algorithm
with alpha beta
pruning

Each chess piece is assigned a base value, reflecting its tactical and strategic importance: pawns are valued modestly at 10 points, reflecting their role as both defensive and offensive units, knights, and bishops slightly higher at 30 and 35 points due to their mobility and utility, rooks at 50 points for their power in both open and semi-open files, and queens at 90 points for their unparalleled range and versatility. The king, invaluable for the game's outcome, is assigned a prohibitively high score of 900 points, ensuring the engine prioritizes its safety above all.

TABLE I
Chess Pieces
Heuristic Values

White Pieces	Capture Heuristic
Pawn	10
Knight	30
Bishop	35
Rook	50
Queen	90
King	900

MyBot's evaluation function is refined further with positional values, encouraging control of the board's center and penalizing peripheral placements that limit a piece's effectiveness. This nuanced approach to piece valuation allows MyBot to not only capture material advantages but also to position its pieces optimally for both defense and offense.

TABLE II
Chess Board
Positional Values

-4	-4	-4	-3	-3	-4	-4	-4
-4	-4	-4	-3	-3	-4	-4	-4
-4	-2	-2	-3	-3	-2	-2	-4
-4	-2	-1	0	0	-1	-2	-4
-4	-2	-1	0	0	-1	-2	-4
-4	-2	-2	-3	-3	-2	-2	-4
-4	-4	-4	-3	-3	-4	-4	-4
-4	-4	-4	-3	-3	-4	-4	-4

In the endgame, MyBot shifts focus on king positioning, implementing a heuristic that increases in value as the opposing king is driven towards the corners, where checkmate probabilities are higher. This strategic push is balanced with a keen awareness of draw conditions, where the engine adjusts its heuristics to either avoid or seek a draw based on its material standing, demonstrating a deep strategic foresight.

In Addition, MyBot adjusts the values of certain pieces like bishops, knights, and especially pawns, recognizing their increased importance in a less crowded board. This adjustment helps in making strategic moves that capitalize on the endgame dynamics.

```
Function GetPushKingCornerHeuristic
    Get count of my pieces
    Get count of opponent's pieces

    If more than 5 pieces for both players
        Return 0

    Get opponent king's position

    Calculate distance from nearest corner:
    Corner distance = 8 - Min(7 - king's row, king's row)
                    + Min(7 - king's col, king's col)

    Calculate heuristic:
    Heuristic = Corner distance * 5 * (LateGameRate)^2

    Return heuristic
End Function
```

End Game Push
King Corner Logic

Strategic Heuristics in Different Game Phases

Pawn promotion is aggressively pursued in the endgame, with significant bonuses for advancing pawns, recognizing their potential to change the game's tide.

Castling is encouraged through positive heuristics, promoting king safety and rook mobilization, essential elements for mid-game security and flexibility.

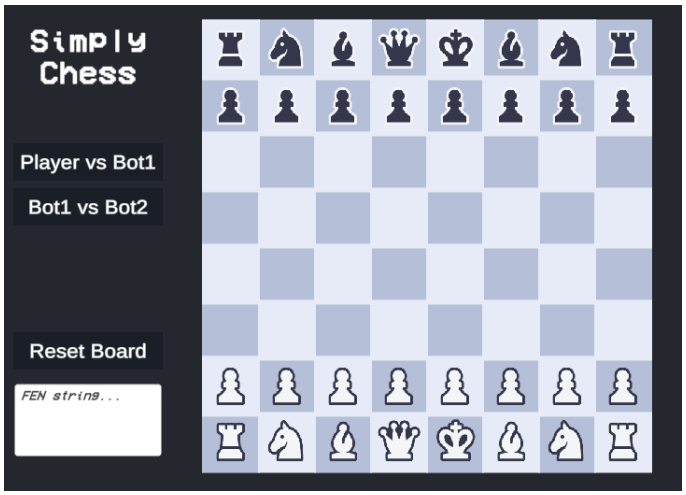
The engine discourages premature queen movements with a heuristic penalty, aligning with established chess principles that favor piece development and king safety in the opening.

Pawn structure and advancement are subtly incentivized, rewarding forward movement that supports control and potential breakthroughs.

MyBot's approach to each game is dynamic, with heuristics that adjust in real-time to the evolving board state, considering factors such as piece activity, king safety, and positional advantages. This adaptability, combined with a robust evaluation of material and positional factors, allows MyBot to execute strategies that are both tactically sound and strategically profound, making it a formidable chess engine capable of competing.

IV. Chess Game Menu Interface

In my chess game, players can experience two distinct modes. The first mode allows a player to challenge my custom-built bot, offering a personal test against its AI. The second mode features an exciting clash between two bots, letting players observe and analyze AI versus AI matches. Additionally, there's a handy feature in the bottom-left corner where users can input a FEN (Forsyth-Edwards Notation) string to set up specific board positions. This flexibility enables players to either face the bot from any desired game scenario or to watch how two bots navigate through predefined positions, adding a layer of customization and strategy exploration to the game experience [7].



Chess Game Menu

V. Chess Game API

A. Chess API

The Chess API serves as the backbone for interfacing between the game's logic layer and the user interface, providing a structured way to query and manipulate the game state. It abstracts complex chess logic into a set of accessible functions, enabling the chess engine or external entities to perform actions such as making moves, evaluating positions, and determining game status without delving into the intricacies of the underlying implementation.

Chess API include:

Determining which player's turn it is, thereby allowing the engine to consider moves for the correct set of pieces.

Retrieving lists of pieces for both players, which is essential for evaluating the current board state and planning subsequent moves.

Identifying specific pieces like kings and rooks, which is crucial for special moves such as castling.

Managing move history, which aids in implementing rules like the threefold repetition draw and assessing the game's progression.

Facilitating the execution and reversion of moves, thereby enabling the engine to explore future game states through move simulations.

Evaluating conditions such as check, checkmate, and draw, which are vital for determining the game's outcome and guiding the engine's decision-making process.

B. Chess Board API

The Chess Board API serves as a crucial interface between the game's logic and the chessboard's representation, streamlining interactions with the board and its pieces. It abstracts the complexities of board management, providing a suite of functions designed to query the board's state, manipulate piece positions, and evaluate potential moves and threats. Additionally, this API plays a pivotal role in enforcing the rules of chess, ensuring that each move adheres to the game's legal parameters and that special conditions like check, checkmate, and stalemate are accurately recognized and handled.

Chess Board API include:

Bounds Checking: Verifies if a given position falls within the board's boundaries, ensuring moves and piece placements are legal.

Square State: Determines whether a specific square is occupied or empty, which is fundamental for move generation and collision detection.

Threat Assessment: Evaluates whether a square is under threat by an opponent's piece, crucial for move validation and check/checkmate detection.

Distance Calculation: Computes the distance between two squares, aiding in movement logic and piece interaction.

Move Notation: Generates algebraic notations for moves, facilitating move recording and user interface display.

Castling Logic: Provides functions to check the feasibility of castling moves, adhering to chess rules regarding king and rook movements.

Move Execution and Reversion: Allows hypothetical moves to be made and undone on the board, supporting the engine's lookahead feature in move planning.

Special Move Checks: Includes functions to check for special moves like pawn promotion and en passant, ensuring all chess rules are covered.

Directional Move Checks: Offers methods to check for legal moves in horizontal, vertical, diagonal, and L-shaped patterns, accommodating the diverse move sets of chess pieces.

Encapsulating game logic within the Chess & Chess Board API ensures a clean division between the core functionalities of the chess engine and its interface, promoting a streamlined approach to development and maintenance. This method not only simplifies the complexities associated with managing the board state and piece movements but also empowers the engine to focus on strategic elements and decision-making. The design focuses on simplicity and clear structure, making it easier to update, fix issues, and work with other systems. This approach improves the chess engine's flexibility and overall performance.

VI. WAYS TO ENHANCING THE BOT

A. Opening Book Integration

Implementing an opening book can improve bot's performance in the early stages of the game. This would involve the bot having access to a database of established opening moves, allowing it to play the most theoretically sound moves in the opening phase.

B. Endgame Tablebases

Incorporating endgame tablebases can give bot a significant advantage in the endgame phase. These are databases that contain pre-calculated exhaustive analysis of endgame positions. Utilizing these can enable your bot to play perfect endgames for positions with a small number of pieces left on the board.

C. Positional Evaluation Enhancements

Beyond basic piece value assessments, enhancing bot's ability to evaluate positions more deeply by considering factors such as pawn structure, piece mobility, king safety, and control of key squares could significantly improve its strategic play.

D. Multi-threaded Search

Implementing a multi-threaded search algorithm can significantly speed up your bot's ability to analyze positions by allowing it to process multiple move sequences simultaneously, taking full advantage of modern multi-core processors.

E. Time Management Improvements

Enhancing the bot's ability to manage its time more effectively during games, ensuring it allocates its time resourcefully across different phases of the game, can lead to more thoughtful and profound analysis when it's most needed.

F. Dynamic Play Style

Developing the bot to adapt its play style based on the game situation (aggressive when ahead, defensive when behind, etc.) or the opponent's style could make it more unpredictable and challenging to face.

VII. RESULT

In conclusion, this chess engine, built upon the Minimax algorithm with Alpha-Beta pruning, showcases an estimated Elo rating of around 1200. The creation process involved utilizing custom-designed Chessboard and Chess API, which were effective shaping the bot's capabilities. With its current foundation, there's a clear pathway for further enhancements, including the addition of advanced features that could significantly elevate its competitive edge. The ambition moving forward is to evolve this engine into a stronger opponent, expanding its utility and complexity.

REFERENCES

[1] https://www.chessprogramming.org/Main_Page
[2] https://www.youtube.com/watch?v=U4ogK0MIzqk&t=1253s
[3] https://www.youtube.com/watch?v=_vqlIPDR2TU&t=1144s
[4] https://www.youtube.com/watch?v=Ne40a5LkK6A&t=1890s
[5] https://github.com/SebLague/Tiny-Chess-Bot-Challenge-Results
[6] https://sebastian.itch.io/tiny-chess-bots
[7] https://hinkirmunkur.itch.io/simply-chess