Chess AI Project

Table of Contents

Introduction

Game Theory and AI in Chess

What is Game Theory?

Game theory is the mathematical study of decision-making in competitive situations where the outcome depends on the actions of multiple agents, often referred to as players. It provides frameworks and strategies for making optimal decisions in games and other interactive systems.

Key Concepts in Game Theory

1. Players: The agents participating in the game (e.g., White and Black in chess).
2. Strategies: The plans or sequences of actions a player can take.
3. Payoff: The reward or outcome of a game for each player, often represented as points or utility.
4. Game Types:
    - Zero-Sum Games: The gain of one player is equal to the loss of another (e.g., chess, where one player's win is another's loss).
    - Non-Zero-Sum Games: Both players can win or lose together (e.g., some cooperative games).
5. Perfect Information: Games where all players know the state of the game at all times (e.g., chess and checkers).
6. Nash Equilibrium: A state where no player can benefit by changing their strategy while the others keep theirs constant.

Chess and Game Theory

Chess is a two-player, zero-sum, perfect-information game, making it an ideal candidate for analysis using game theory.

1. Zero-Sum: If one player wins (+1), the other loses (-1). A draw results in a payoff of 0 for both.
2. Perfect Information: Both players see the board and know all possible moves at any time.
3. Deterministic: There are no random elements; outcomes depend purely on the players' decisions.

Overview

This project implements a Chess game with a graphical user interface (GUI) and an Artificial Intelligence (AI) component. It includes three Python scripts:

- **ChessMain.py**: Handles user interaction and displays the game state.
- **ChessEngine.py**: Manages game rules, board state, and move validation.
- **AiChess.py**: Implements the AI algorithms for move evaluation and selection.

This documentation explains the project, its components, functionality, code, and AI strategies. It is designed for readers without prior Python or chess knowledge.

Functionalities

Chess Game Features

- Playable chess game with a graphical interface using Pygame.
- Human vs. AI .
- Thematic customization with five board themes.
- Valid move highlighting and real-time updates.
- Support for special moves: castling, en passant, and pawn promotion.

- Move log for reviewing game history.
- Checkmate and stalemate detection.

Artificial Intelligence (AI)

- Minimax algorithm with alpha-beta pruning for efficient decision-making.
- Position scoring is based on piece value and positional heuristics.
- Adjustable search depth.

Project Breakdown

1. ChessMain.py

Handles user input, game rendering, and interaction between the AI and the game engine.

Key Components

- **Initialization**:
    - Sets up the game board, themes, and Pygame display.
- **Game Loop**:
    - Listens for user input and updates the board and move log.
    - Manages AI turns using multiprocessing.
- **Graphics**:
    - Renders the board, pieces, valid move highlights.

Special Functionalities

- **Thematic Board Customization**: Press keys 1-5 to switch themes dynamically.
- **Undo Button**
    - U key: Undo the last move.
- **Move Animation**: Smoothly animates piece movement for better user experience.

---

2. ChessEngine.py

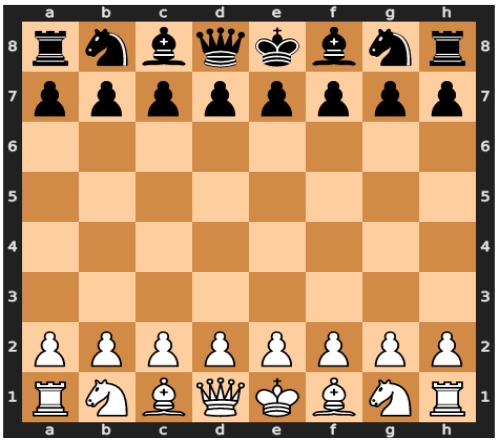
Implements the chess game logic, including board state management, move validation, and special rules.

Key Components

- **GameState Class**: Represents the current state of the board and tracks:
    - Board configuration.
    - Current player's turn.
    - Castling rights.
    - En passant opportunities.
- **Move Generation**:
    - Validates legal moves for each piece.
    - Detects special moves (e.g., castling, en passant and Pawn Promotion automatically to Queen).
    - Evaluates checkmate and stalemate conditions.
- **Move Class**: Converts board coordinates to standard chess notation and encapsulates move properties.

Explaining Rows, Columns, and Move ID

- **Board Representation**: The chess board is an 8x8 grid where rows are indexed 0-7 (from top to bottom) and columns are indexed 0-7 (from left to right).



- **Move Class IDs**: The Move class generates a unique ID for each move using the formula:

```
self.moveID = self.startRow * 1000 + self.startCol * 100 + self.endRow * 10 + self.endCol
```

This ensures every move is uniquely identifiable based on its start and end positions on the board.

Special Functionalities

- **Castling Rights**:
    - Tracks separately for kingside .
- **En Passant**:
    - Allows pawns to capture enemy pawns on special conditions.
- **Promotion**:
    - Automatically promotes pawns to queens.

---

3. AiChess.py

Implements the AI logic for move evaluation and selection.

Key Components

- **Scoring System**: Assigns values to pieces:
    - King: 0 (game-ending piece).
    - Queen: 10.
    - Rook: 5.
    - Bishop/Knight: 3.
    - Pawn: 1.
- **Positional Heuristics**: Adjusts scores based on positional tables for each piece type.
- **Minimax with Alpha-Beta Pruning**:

    - Searches possible moves to maximize/minimize scores.
    - Reduces unnecessary computations for improved efficiency.
    - Dynamically adjusts alpha (lower bound) and beta (upper bound) to prune suboptimal branches.
- **Random Move Selection**: Used as a fallback if no better move is found.

```
def findRandomMove(validMoves):   1 usage
    return random.choice(validMoves)
```

Minimax Algorithm with Alpha-Beta Pruning

```
def findMoveMinimaxAlphaBeta(gs, validMoves, depth, alpha, beta, maximizingPlayer):  3 usages
    global nextMove
    if depth == 0:
        return scoreBoard(gs)
    if maximizingPlayer:
        maxScore = -CHECKMATE
        for move in validMoves:
            gs.makeMove(move)
            nextMoves = gs.getValidMoves()
            score = findMoveMinimaxAlphaBeta(gs, nextMoves, depth - 1, alpha, beta, maximizingPlayer: False)
            if score > maxScore:
                maxScore = score
                if depth == DEPTH:
                    nextMove = move
            gs.undoMove()
            alpha = max(alpha, score)
            if alpha >= beta:
                break
        return maxScore
    else:
        minScore = CHECKMATE
        for move in validMoves:
            gs.makeMove(move)
            nextMoves = gs.getValidMoves()
            score = findMoveMinimaxAlphaBeta(gs, nextMoves, depth - 1, alpha, beta, maximizingPlayer: True)
            if score < minScore:
                minScore = score
                if depth == DEPTH:
                    nextMove = move
            gs.undoMove()
            beta = min(beta, score)
            if alpha >= beta:
```

This function efficiently evaluates the best possible move by pruning branches that cannot yield better outcomes, reducing the search space.

Optimality

Minmax algorithm is optimal if both opponents are playing optimally.

Alpha-Beta Pruning

The minmax algorithm can be optimized by pruning few branches. Pruned branches are the ones that are not going to affect result. It will improve time-complexity. This version minmax is knows as minmax with alpha-beta pruning. It is also called as alpha-beta algorithm.

In Alpha-Beta Pruning, there are two values, Alpha and Beta. Below are the few points to consider about alpha and beta:

Some point about Alpha(α)

• Alpha is the highest value, that is found along the MAX path.

• Initial value of alpha is negative infinity because alpha value will keep on increasing or remain same with every move. If we choose some value other than negative infinity, then the scenario may occur in which all values of alpha may be less than chosen value. So, we have to choose lowest possible value, and that is negative infinity.

• Alpha is only updated by MAX player.

Some points about Beta(β)

• Beta is the lowest value, that is found along the MIN path.

• Initial value of beta is positive infinity because beta value will keep on decreasing or remain same with every move. If we choose some value other than positive infinity, then scenario may occur in which all values of beta may be more than chosen value. So, we have to choose maximum possible value, and that is positive infinity.

• Beta value is only updated by MIN player.

While backtracking only node value is passed to parent, not the alpha and beta value.

The Condition for alpha-beta pruning:

α >= β

Move Ordering

The effectiveness of alpha-beta algorithm is highly depending on order of traversal. It plays crucial role in Time and Space Complexity.

Worst Ordering

In some cases, no node or sub-tree is pruned out of Game Tree. In this case, best move occurs in right sub-tree of Game Tree. This will result in increased Time Complexity.

Ideal Ordering

In this case, maximum number of node and sub-tree is pruned. Best moves occur in left subtree. It will reduce the Time and Space Complexity.

Algorithm Analysis

Minimax with Alpha-Beta Pruning

- **Time Complexity**: O(bd)O(b^d), where:
    - bb: Average branching factor (approximately 35 in chess).
    - dd: Depth of the search tree.
- **Optimized Complexity**: Reduced by pruning branches, effectively lowering the number of nodes evaluated.

Why Minimax is Slow and Limited

- **Branching Factor**: Chess has a high branching factor (~35 moves per turn). Even with pruning, the number of states to evaluate grows exponentially with depth.

- **Fixed Depth**: Setting a fixed depth (e.g., 2 or 3 moves ahead) limits strategic foresight, especially in complex positions.
- **Limited Heuristics**: While material value and basic position scoring are included, the algorithm does not understand advanced strategies like long-term planning or sacrifices.
- **Inefficiency in Dynamic Scenarios**: Positions requiring precise calculations (e.g., tactics or endgames) expose weaknesses in static evaluation and fixed-depth search.

Suggestions for Smarter AI

- **Dynamic Depth Adjustment**: Increase depth adaptively based on the game phase or critical positions.
- **Advanced Heuristics**: Incorporate factors like pawn structure, king safety, and control of open files.
- **Learning Mechanisms**: Use reinforcement learning or neural networks to recognize patterns and strategies.
- **Endgame Databases**: Integrate precomputed endgame tablebases for precise late-game moves.
- **Adding NegaMax Alpha Beta Pruning**.

Bugs and Improvements

Identified Bugs

- **Move Class Equality**:
    - Overriding __eq__ based on moveID can cause issues if multiple moves share the same ID.
- **Pawn Promotion**:
    - Automatically promotes a queen without player choice.
- **Castling happens from the queen's side:**
    - Which is not a valid move in chess.

Future Improvements

- Allow user-selectable pawn promotions (e.g., to knights).
- Implement transposition tables to enhance AI efficiency.
- Add difficulty levels by adjusting search depth dynamically.
- Fix Bugs.

Conclusion

This documentation provides a thorough explanation of the Chess AI project. By understanding the game's rules, code structure, and AI logic. Special features like Minimax with Alpha-Beta Pruning and customizable board themes enhance the user experience. While the AI performs well for beginner gameplay, improvements in depth, heuristics, and learning could significantly enhance its capabilities.