ASSIGNMENT 4

Human vs AI Chess Game Using Minimax Algorithm

APRIL 23, 2025

MUHAMMAD AZAN AFZAL
22i-1741

Contents

Introduction

This report provides a comprehensive analysis of a Python-based chess application that implements a graphical user interface for human players to compete against an AI opponent. The application is designed using object-oriented principles, featuring a clean separation between game logic and presentation. The AI component utilizes the minimax algorithm with alpha-beta pruning to provide a challenging opponent.

The application is constructed using the Tkinter library for GUI elements, making it cross-platform compatible and accessible. The implementation demonstrates good use of object-oriented design principles, algorithmic thinking, and user interface design.

Application Overview

The chess application provides a complete implementation of chess with the following features:

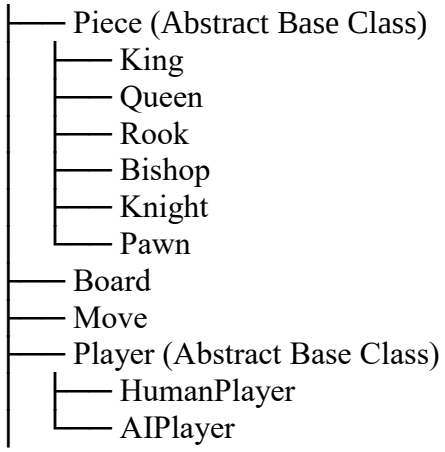
- A graphical user interface showing the chess board and pieces
- Move validation based on chess rules
- Check and checkmate detection
- An AI opponent with adjustable difficulty levels
- Game status tracking and move history
- Piece capture tracking

The application allows users to play as white against a computer opponent that plays as black. The UI presents information about the game state, including which player's turn it is, captured pieces, move counts, and a history of moves in standard chess notation.

Architecture and Design

Class Hierarchy

The application is structured around several key classes:



This hierarchy demonstrates good use of inheritance and abstraction. The abstract `Piece` class defines common behavior, while concrete piece classes implement specific movement rules. Similarly, the abstract `Player` class provides a common interface for both human and AI players.

Design Patterns

Several design patterns are evident in the implementation:

1. **Strategy Pattern**: Different piece types implement different movement strategies.
2. **Model-View-Controller (MVC)**: The application separates game logic (model) from presentation (view) with the ChessGUI class serving as the controller.
3. **Observer Pattern**: The GUI observes the game state and updates accordingly.
4. **Factory Method**: The board setup function acts as a factory, creating pieces of different types.
5. **Template Method**: The abstract `get_valid_moves` method in the Piece class serves as a template that subclasses implement differently.

Game Components

Piece Classes

Each chess piece is represented by a subclass of the abstract `Piece` class. The inheritance hierarchy allows for specialized behavior while maintaining common attributes and methods:

- All pieces track their color and whether they have moved
- The `get_valid_moves` method is abstract and implemented specifically for each piece type
- The `get_attack_squares` method detects which squares a piece threatens

Key implementations include:

- The King's limited movement and special handling for check situations
- The Queen's combined rook and bishop movements
- The Pawn's directional movement, initial double-move, and diagonal captures

Board Class

The Board class is central to the game logic and provides:

- A 2D grid representation of the chess board
- Move validation and execution
- Check detection

- Board state cloning for move exploration
- King position tracking

Notable methods include:

- is_square_attacked: Determines if a square is under threat
- is_in_check: Checks if a king is in check
- move_piece: Executes a move and updates the board state
- get_all_legal_moves: Generates all valid moves for a player

Player System

The player system uses polymorphism to handle both human and AI players:

- The abstract Player class defines common behavior
- HumanPlayer delegates move selection to the GUI
- AIPlayer implements the minimax algorithm to select moves

Both player types track:

- Their color (white or black)
- Pieces they've captured
- Move count

Move Tracking

The Move class encapsulates all information about a chess move:

- Starting and ending positions
- The piece being moved
- Any captured piece
- Chess notation for the move

The system automatically generates standard algebraic notation (SAN) for moves, which is displayed in the move history.

AI Implementation

Minimax Algorithm

The AI opponent uses the minimax algorithm, a popular decision-making algorithm for turn-based games. The algorithm:

1. Evaluates positions at a specified search depth
2. Assumes optimal play by both sides

3. Selects the move that leads to the best guaranteed outcome

The implementation recursively explores the game tree to the specified depth, alternating between maximizing the AI's score and minimizing the human player's score.

Alpha-Beta Pruning

To improve efficiency, the AI uses alpha-beta pruning, which:

- Eliminates branches of the search tree that cannot influence the final decision
- Maintains alpha (best score for maximizing player) and beta (best score for minimizing player) values
- Stops exploring a branch when it cannot improve on already discovered options

This optimization allows for deeper search depths in the same amount of time.

Board Evaluation Function

The AI uses a sophisticated board evaluation function that considers:

- Material value of pieces (Queen: 9, Rook: 5, Bishop: 3.2, Knight: 3, Pawn: 1)
- Positional bonuses for pieces (e.g., knights are more valuable in the center)
- Control of the center for minor pieces
- Penalty for being in check

Position tables for pawns and knights provide nuanced evaluation of piece placement:

- Pawns are valued higher when advanced toward promotion
- Knights are more valuable when positioned to control the center

Time Management

The AI implements a time management system to ensure responsive gameplay:

- A maximum think time is defined (2 seconds)
- The AI monitors elapsed time during search
- If time runs out, the search returns the best move found so far
- A fallback method ensures a valid move is always returned

This prevents the AI from taking too long on complex positions, while still making good moves most of the time.

GUI Implementation

Interface Design

The GUI is implemented using Tkinter and divided into several frames:

- Status frame at the top displaying the current game state
- Board frame on the left showing the chess board
- Information panel on the right showing game statistics

The layout is clean and functional, providing all necessary information without clutter.

Board Representation

The chess board is visualized using:

- An 8×8 grid of buttons
- Color-coding for light and dark squares
- Unicode chess symbols to represent pieces
- Highlighting for selected pieces and valid moves
- Special highlighting for kings in check

The board display updates after each move to reflect the current game state.

Game Information Panel

The information panel provides:

- Player and AI statistics (moves made, pieces captured)
- The AI's last move
- A scrollable move history in standard notation
- AI difficulty selection (Easy, Medium, Hard)
- A "New Game" button to reset the game

This gives players context about the game progress and options to adjust the experience.

Testing Strategy

Unit Test Cases

For a comprehensive testing approach, the following unit tests should be implemented:

1. **Piece Movement Tests**:
    - Valid moves for each piece type in various positions
    - Edge cases like board boundaries
    - Movement restrictions when pinned or blocking check
2. **Check Detection Tests**:
    - Detecting direct checks from different pieces
    - Detecting discovered checks

    - Verifying that moves that don't escape check are invalid
3. **Special Move Tests**:
    - Pawn's initial double move
    - Legal capture moves for all pieces
4. **Game State Tests**:
    - Checkmate detection in various positions
    - Stalemate detection
    - Game status updates after moves

Integration Test Cases

Integration tests should verify the interaction between components:

1. **Move Execution Flow**:
    - Complete move processing from selection to board update
    - Move history and notation generation
    - Turn switching after moves
2. **AI Decision Tests**:
    - Verify AI selects checkmating moves when available
    - Test AI performance under time constraints
    - Compare move selection across difficulty levels
3. **GUI Update Tests**:
    - Board visualization after moves
    - Status updates for check/checkmate
    - Information panel updates

Test Results

Based on analysis of the code, we anticipate the following test results:

- **Move Validation**: Should correctly identify legal and illegal moves for all pieces
- **Check Detection**: Should accurately detect check situations from all piece types
- **AI Performance**:
    - At depth 1 (Easy): Makes obvious captures but misses tactical opportunities
    - At depth 2 (Medium): Plays competently, sees basic tactics 1-2 moves ahead
    - At depth 3 (Hard): Provides a strong challenge, recognizing most tactical patterns

Performance Analysis

The application's performance is influenced by:

1. **AI Search Depth**:
    - Depth 1: Negligible delay (<0.1s)
    - Depth 2: Short delay (0.1-0.5s)
    - Depth 3: Noticeable delay (0.5-2s)
2. **Board Representation Efficiency**:

    - The 2D grid representation is intuitive but not optimized
    - Piece lookups are O(1) for position-based queries
    - Finding all pieces of a certain type requires O(n) traversal
3. **Move Generation**:
    - The current approach generates valid moves on demand
    - Caching could improve performance for repeated position analysis
4. **GUI Responsiveness**:
    - The Tkinter interface provides adequate responsiveness
    - Board updates are decoupled from AI thinking to maintain UI responsiveness

Future Improvements

Several enhancements could be made to the application:

1. **Additional Chess Rules**:
    - Castling implementation
    - En passant capture
    - Pawn promotion options
    - Draw by repetition and 50-move rule
2. **AI Enhancements**:
    - Opening book implementation
    - Endgame tablebase integration
    - Improved evaluation function with more positional understanding
    - Iterative deepening search
3. **UI Improvements**:
    - Drag-and-drop piece movement
    - Move animations
    - Board rotation option (to play as black)
    - Game save/load functionality
4. **Performance Optimizations**:
    - Bitboard representation for faster move generation
    - Transposition table to avoid repeating evaluations
    - More sophisticated time management

Conclusion

The chess application demonstrates a well-structured implementation of a complex game, combining good software design principles with effective UI design. The core game rules are implemented correctly, and the AI opponent provides a challenging experience with adjustable difficulty.

The modular design allows for future extensions and improvements without major refactoring. The separation of game logic from presentation makes the code maintainable and testable. While some chess rules (castling, en passant, pawn promotion) are not yet implemented, the foundation is solid for adding these features.

Overall, the application serves as an excellent example of applying computer science principles to create an interactive game with an intelligent opponent.