**Chess Program Analysis Report**

**By**

**Abdul Moiz**

**Introduction**

This report analyzes a Python chess program implementation that allows a human player to compete against a computer opponent. The program implements all standard chess rules, including special moves like castling and en passant, and uses a minimax algorithm with alpha-beta pruning for the computer's decision-making process.

**Program Architecture**

The chess program is structured around several key classes:

1. Piece Classes: A hierarchy of chess piece classes
2. ChessGame Class: The main game controller
3. Game Logic Functions: Methods for move validation, board evaluation, and AI

**Key Components Analysis**

**1. Piece Representation System**

The program uses a class hierarchy for chess pieces:

* Piece: The base class containing common attributes like color, symbol, and value
* Specialized pieces (King, Queen, Rook, Bishop, Knight, Pawn): Each with unique movement logic

Each piece implements a get\_moves() method that returns all possible moves for that piece from a given position, taking into account the current board state.

class Piece:

def \_\_init\_\_(self, color, symbol, value):

self.color = color

self.symbol = symbol

self.value = value

self.has\_moved = False

def get\_moves(self, board, pos):

"""Return a list of possible moves for this piece"""

pass

Unicode chess symbols (♔, ♕, ♖, etc.) are used for visualizing pieces on the board, creating an intuitive user interface in the terminal.

**2. Board Representation and Display**

The chess board is represented as an 8×8 matrix where:

* Each cell contains either a piece object or None for empty squares
* The board is displayed with coordinate labels (a-h, 1-8) following standard chess notation
* Empty squares use a checkerboard pattern with dots and spaces for visual clarity

The display function creates a clear visualization:

def display\_board(self):

print(" a b c d e f g h")

print(" +-----------------+")

# Board printing logic

print(" +-----------------+")

print(" a b c d e f g h")

**3. Move Validation and Execution**

Move validation follows a multi-step process:

1. Basic validation: Checks if the starting position contains a piece of the current player's color
2. Move pattern validation: Validates the move against the piece's allowed movement patterns
3. Special move handling: Implements special rules for:
   * Castling (king moves two squares and rook jumps over)
   * En passant captures (pawn captures diagonally an opponent's pawn that just moved two squares)
   * Pawn promotion (automatically promotes to queen when reaching the opposite rank)
4. Check validation: Ensures moves don't leave the player's king in check

The program converts between chess notation (e.g., "e2e4") and internal board coordinates (row, column indices).

**4. AI Implementation with Minimax**

The computer opponent uses a minimax algorithm with alpha-beta pruning to determine optimal moves:

def minimax(self, depth, alpha, beta, maximizing\_player):

"""Minimax algorithm with alpha-beta pruning"""

if depth == 0:

return self.evaluate\_board(), None

# Algorithm implementation for both maximizing and minimizing players

Key features of the AI:

* Board evaluation function: Assigns a numerical score to board positions based on:
  + Material advantage (piece values)
  + Check/checkmate status
  + Mobility (number of legal moves)
  + Center control (bonus for controlling center squares)
* Depth-limited search: Uses a depth of 2 to balance performance and calculation time
* Alpha-beta pruning: Optimizes the search by eliminating branches that won't affect the final decision
* Fallback mechanisms: Includes error handling to select random legal moves if the primary algorithm fails

**5. Game Control Flow**

The main game loop manages the alternating turns between human and computer:

1. Display the current board state
2. Check for game-ending conditions:
   * Checkmate (win for either side)
   * Stalemate (draw)
   * Move limit (draw after 100 moves)
3. Handle human turns:
   * Parse input in the format "e2e4"
   * Validate the move
   * Execute the move if valid
4. Handle computer turns:
   * Calculate the best move using minimax
   * Execute the move
   * Display the move to the player

**Special Chess Rules Implementation**

**Castling**

The program correctly implements castling rules:

* King and rook must not have moved previously
* Spaces between king and rook must be empty
* King cannot castle out of, through, or into check
* Both king and rook move in a single turn

**En Passant**

En passant capture is implemented with:

* Detection of opponent pawns that just moved two squares
* Special flag to mark these pawns as vulnerable to en passant
* Proper handling of the capture (removing the opponent's pawn from its original row)

**Pawn Promotion**

Pawns are automatically promoted to queens when reaching the opposite rank, simplifying the implementation while maintaining core functionality.

**Results and Game Play**

When executed, the program:

1. Prompts the player to choose a color (white or black)
2. Initializes the game board with all pieces in standard positions
3. Alternates between human and computer turns
4. Provides a clear interface for move input using chess notation
5. Displays the current board state after each move
6. Enforces all chess rules including special moves
7. Determines the game outcome (checkmate, stalemate, or draw)

The minimax algorithm provides a challenging opponent by evaluating various factors in the board position and planning moves ahead.

**Limitations and Potential Improvements**

While the program implements a complete chess game, some potential improvements include:

1. Adjustable AI difficulty: Currently fixed at depth 2, could be made variable
2. Choice of promotion piece: Currently always promotes to queen
3. Better heuristics: The evaluation function could be enhanced with more sophisticated chess principles
4. Opening book: Adding predefined openings would improve early-game play
5. Game history and save/load: Ability to record games and continue later
6. Graphical user interface: A GUI would improve usability over the current terminal interface

**Conclusion**

This chess program successfully implements all the fundamental rules of chess and provides a functional AI opponent. The object-oriented design makes the code modular and extensible, while the minimax algorithm with alpha-beta pruning provides a computational foundation for the computer's decision-making.

The program serves as an excellent example of how classic board games can be modeled in code, combining both the logical rules of chess with artificial intelligence concepts for computer opponents.