Chess Game Project Report

Objectives

The goal of this project was to develop a fully functional chess game where a player can compete against an AI opponent. The game adheres to standard chess rules, including special moves such as castling, en passant, and pawn promotion. It features a user-friendly graphical interface built with Tkinter and includes tactical analysis capabilities to identify patterns like pins, forks, and skewers, enhancing the player's learning experience.

Approach

The project was structured into several key components, each addressing a specific aspect of the chess game:

1. Board Representation

- **Implementation**: The chessboard is represented as an 8x8 2D grid within the Board class, where each cell can contain a piece object or be empty (None).
- **Details**: The grid is initialised with pieces in their standard starting positions. Methods are provided to move pieces, check if squares are under attack, and handle special conditions like en passant targets.

2. Piece Movement

- **Implementation**: Each chess piece type (Pawn, Rook, Knight, Bishop, Queen, King) is a subclass of the Piece base class, with a legal_moves method defining its possible moves.
- **Details**: Special rules are encoded, such as pawns moving forward two squares initially, castling for kings, and en passant captures. The move method updates the piece’s position and tracks whether it has moved.

3. Game Logic

- **Implementation**: The Game class manages the game state, including turn order, move history, and game-ending conditions (check, checkmate, stalemate).
- **Details**: It validates moves to ensure they don’t leave the player’s king in check and handles special moves by updating the board state accordingly. The undo_move method reverses moves, restoring captured pieces and special move states.

4. AI Opponent

- **Implementation**: The Chess AI class implements a minimax algorithm with alpha-beta pruning to choose the best move for the AI.
- **Optimizations**:
    - **Transposition Table**: Stores evaluated positions to avoid redundant calculations.
    - **Killer Moves**: Prioritises moves that previously caused beta cutoffs.
    - **History Heuristic**: Tracks move effectiveness for better move ordering.
    - **Quiescence Search**: Extends search depth for captures to mitigate the horizon effect.
    - **Adaptive Depth**: Adjusts search depth based on position complexity (e.g., fewer moves, checks, or endgame scenarios).
- **Evaluation**: The AI assesses positions based on material balance, piece-square tables, mobility, king safety, and pawn structure.

5. GUI

- **Implementation**: The Chessgui class uses Tkinter to create an interactive interface.
- **Features**: Displays the board with piece symbols, highlights selected squares and valid moves, and includes buttons for new games, undoing moves, showing tactics, and configuring AI settings. Multiple colour themes (Classic, Blue, Green) enhance visual appeal.
- **Interaction**: Players click squares to select and move pieces, with visual feedback for the last move and check conditions.

6. Tactical Analysis

- **Implementation**: The detect_tactics method in the Game class identifies pins, forks, and skewers.
- **Details**: Pins are detected by checking if a piece's movement exposes a more valuable piece to attack. Forks identify pieces attacking multiple opponent pieces. Skewers find aligned pieces where moving a valuable piece exposes a less valuable one. Results are highlighted on the GUI.

Results

- **AI Performance**: The AI performs competently at a depth of 3, handling most positions well, with deeper searches (up to 6) possible in complex scenarios due to adaptive depth. Optimisations reduce search time, making the AI responsive (10,000–50,000 nodes/second).
- **User Experience**: The GUI is intuitive, allowing easy piece selection and move execution. Features like move history and tactical highlights enhance engagement.
- **Tactical Analysis Accuracy**: The detection correctly identifies pins, forks, and skewers in tested positions, though it may miss subtle tactics in highly complex scenarios.

- **Performance Metrics**: The AI evaluates 10,000–50,000 nodes, depending on depth and position complexity, ensuring a smooth experience.

Challenges and Solutions

- **AI Efficiency**: Initial slow performance was improved with alpha-beta pruning, transposition tables, and move ordering, reducing computation time significantly.
- **Special Moves**: Correctly implementing castling and en passant required careful rule validation, achieved by tracking move history and board state.
- **GUI Responsiveness**: Tkinter's after method ensured the interface remained responsive during AI calculations.