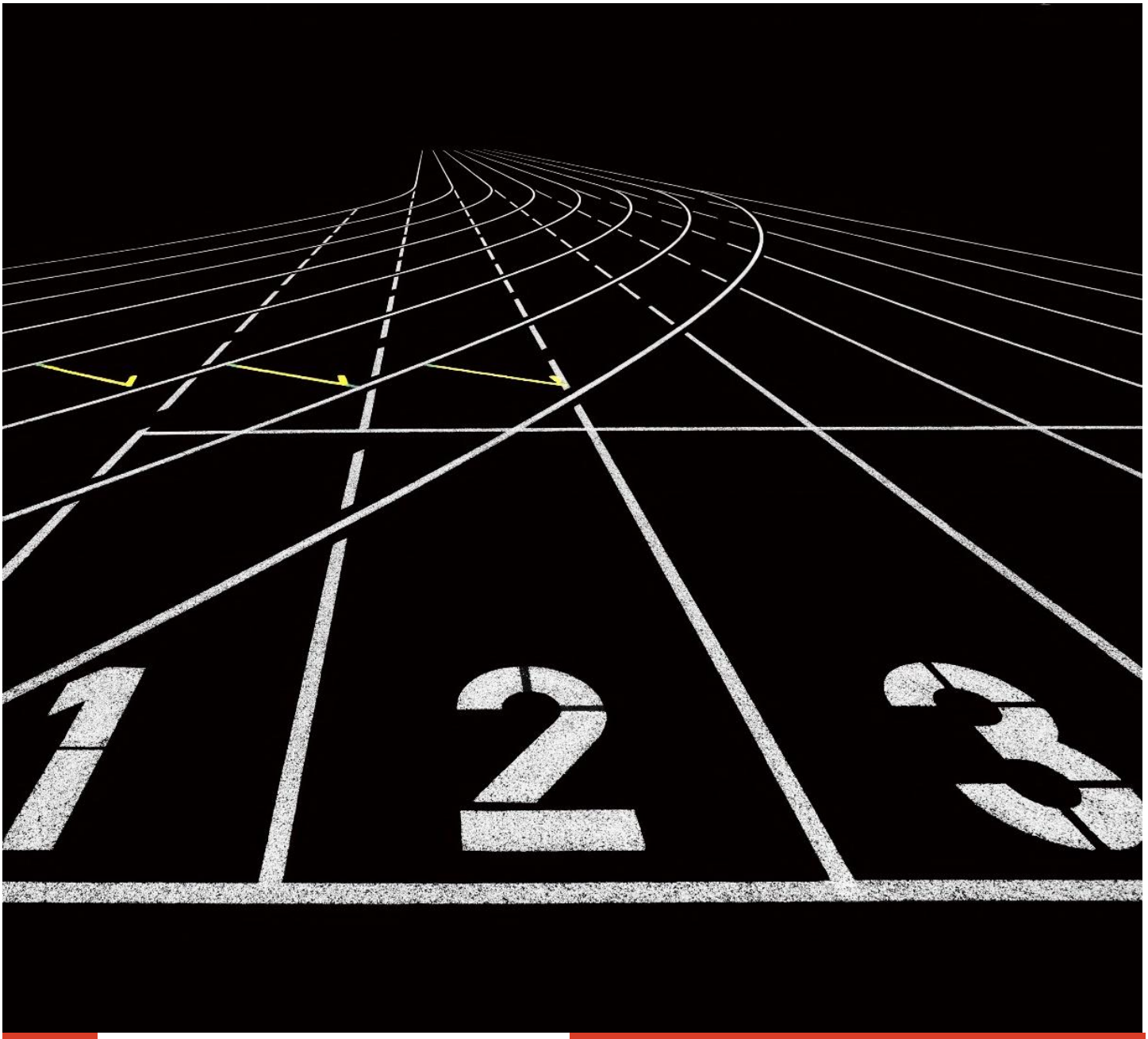
# CHESS DOCUMENTATION




# INTRO

This document provides detailed information about the chess game implemented using Python and the Pygame library. The game features a graphical user interface (GUI) for playing chess, an AI opponent, and support for chess rules like pawn promotion. Below, we describe the code structure, algorithms, and design details in depth.

# 1. OVERVIEW

This chess game allows users to:

- Play chess with a graphical interface.
- Compete against an AI opponent with adjustable difficulty.
- Use features like move highlighting and pawn promotion.

The game uses the Pygame library for rendering and the `python-chess` library for managing chess logic.

# 2. PACKAGES REQUIRED

The game depends on two main Python libraries:

## PYGAME

- **Purpose**: Provides the tools to create the graphical user interface, including the chessboard and piece rendering.
- **Features Used**:
  - Rendering 2D graphics (chessboard squares, highlighting, and text).
  - Handling user input via mouse and keyboard events.
- **Installation**: Run the following command in your terminal or command prompt to install Pygame:

  ```
  pip install pygame
  ```

- **Troubleshooting Installation**:
  - Ensure Python is installed and added to your PATH.
  - On some systems, you may need additional dependencies. Refer to the Pygame Installation Guide for details.
- **Documentation Reference**: Pygame Documentation

# PYTHON-CHESS

- **Purpose**: Handles chess game logic, including move validation, board state management, and game rules enforcement.
- **Features Used**:
  - Board representation.
  - Legal move generation and validation.
  - Chess-specific functionalities like pawn promotion and game result detection.
- **Installation**: Install python-chess with the following command:

  ```
  pip install python-chess
  ```

- **Troubleshooting Installation**:
  - Ensure Python version 3.6 or higher is installed.
  - If issues persist, use a virtual environment to isolate dependencies.
- **Documentation Reference**: python-chess Documentation

- **Assets**: The game requires chess piece images stored in an `assets` folder. Ensure the folder is structured as follows:

```
assets/
    white/
        wk.png (White King)
        wq.png (White Queen)
        wr.png (White Rook)
        wb.png (White Bishop)
        wn.png (White Knight)
        wp.png (White Pawn)
    black/
        bk.png (Black King)
        bq.png (Black Queen)
        br.png (Black Rook)
        bb.png (Black Bishop)
        bn.png (Black Knight)
        bp.png (Black Pawn)
```

- **Image Requirements**: Images should be in `.png` format and have dimensions matching the `SQUARE_SIZE` constant (e.g., 75x75 pixels for a 600x600 board).

# 3. GUI IMPLEMENTATION

The GUI is implemented using Pygame. Key elements include:

- **Board Drawing**: The chessboard consists of alternating light and dark squares, rendered dynamically in the draw_board() function.

- **Piece Display**: Chess piece images are loaded and displayed on their respective board squares using the draw_pieces() function.

- **Highlighting**: Possible moves and the selected piece are highlighted with colors (green for selection, blue for moves) via the highlight_moves() function.

**Dedication**

**Quality**

**Performance**

# 4. KEY FUNCTIONS

## -Draw board():

- **Purpose**: Draws the chessboard grid with alternating colors.
- **Implementation**: Iterates through rows and columns to fill squares with `WHITE` or `BLACK`)

## -Draw piaces board()

- **Purpose**: Displays chess pieces on the board.
- **Implementation**: Loops through all squares, checks for pieces, and renders corresponding images.

## -Handel user move()

- **Purpose**: Manages user interactions, such as selecting and moving pieces.
- **Algorithm**:
    1. Map the mouse position to a board square.
    2. Check for valid moves.
    3. Update the board state if the move is legal.)

## -AI move MCTS

- **Purpose**: Implements AI decision-making using Monte Carlo Tree Search (MCTS).
- **Algorithm**:
    1. Simulate random games from potential moves.
    2. Score moves based on outcomes.
    3. Select the move with the highest average score.)

## -Evaluate Board

- **Purpose**: Evaluates the board position to guide AI decisions.
- **Logic**: Assigns values to pieces (e.g., Queen = 9, Pawn = 1) and computes a score based on material balance.

## -Handel Pawn Promotion

- **Purpose**: Handles pawn promotion when a pawn reaches the last rank.
- **Implementation**: Prompts the user to select a promotion piece.

## -Menus

- **Purpose**: Display the initial menus for game start and difficulty selection.
- **Implementation**: Render text options and wait for user input.

# 5. AI ALGORITHMS

| Feature | Easy Level | Medium Level | Hard Level |
|---|---|---|---|
| **Algorithm** | Random Move Selection | Weighted Random Selection | Monte Carlo Tree Search (MCTS) |
| **Move Selection** | Purely random | Evaluates moves for immediate gains | Simulates games to evaluate outcomes |
| **Evaluation Depth** | None | Single move evaluation | Multiple moves ahead |
| **Strategy** | None | Basic (material gain, threats) | Advanced (heuristics, foresight) |
| **Computational Cost** | Low | Moderate | High |
| **Time Complexity** | O(1) | O(L) | O(L × D) |
| **Best Use Case** | Beginners | Intermediate learners | Experienced players |

## Easy Level: Random Move Selection

- **Description**: The AI selects a move entirely at random from the pool of legal moves. This method ensures simplicity and speed, suitable for beginners learning the game.
- **Example Scenario**:
  - Board state:
    - White pawns: d2, e2.
    - Black knight: f6.
  - AI (playing White) has the following legal moves:

    1. Move pawn from d2 to d3.
    2. Move pawn from e2 to e3.
  - Without evaluation, the AI randomly selects one of these moves, e.g., pawn from e2 to e3.
- **Key Characteristics**:
  - No evaluation of board state.
  - Suitable for testing basic game mechanics.

# Medium Level: Weighted Random Selection

- **Description**: The AI evaluates each move based on immediate gain and assigns weights to prioritize advantageous moves. It then randomly selects a move, biased by these weights.
- **Example Scenario**:
  - Board state:
    - AI (White) can:
      1. Capture a black pawn (score: +1).
      2. Move a knight to a safe square (score: 0).
  - AI assigns weights (e.g., 70% for capture, 30% for knight move) and selects a move based on these probabilities.
- **Key Characteristics**:
  - Introduces basic strategic thinking.
  - Balances challenge and computational efficiency.

# Hard Level: Monte Carlo Tree Search (MCTS)

- **Description**: This advanced method involves simulating multiple possible game outcomes for each legal move and choosing the move with the best statistical performance.
- **Example Scenario**:
  - AI considers three moves:
    1. Pawn advance (average outcome: -0.5).
    2. Knight attack (average outcome: +1.2).
    3. Defensive move (average outcome: +0.3).
  - Based on simulations, the AI selects the knight attack for the highest gain.
- **Key Characteristics**:
  - Depth and foresight for strategic gameplay.
  - Computationally intensive but adaptable by limiting simulation depth or time.

# DESCRIPTION LEVEL

### Easy Level: Random Move Selection

- **Description**: At this level, the AI chooses a move entirely at random from the list of all legal moves. This simplicity ensures that the AI is extremely fast and does not require any computation to evaluate the board state.
- **Advantages**:
  - Extremely fast as it requires no analysis or strategy.
  - Easy to implement, making it ideal for testing the basic framework of the game.
- **Performance**:
  - Provides minimal challenge, making it suitable for beginners learning the rules of chess.
- **Example Scenario**: Imagine the AI's turn in a game where it has three legal moves:

  1. Move a pawn forward.
  2. Capture an opponent's piece.
  3. Move a bishop to an open square. The AI randomly selects one of these moves without considering the consequences. For example, it might choose to move the bishop, even if capturing the opponent's piece would be more advantageous.

### Medium Level: Weighted Random Selection

- **Description**: Random selection with basic evaluation.
- **Advantages**:
  - Introduces strategy without significant computational cost.
  - Balances ease of play with moderate challenge.
- **Performance**:
  - Suitable for intermediate players practicing tactical thinking.

### Hard Level: Monte Carlo Tree Search (MCTS)

- **Description**:
  1. Generate all legal moves.
  2. Simulate random games (playouts) for each move.
  3. Track outcomes and evaluate move quality.
  4. Use backpropagation to update scores in the search tree.
  5. Select the move with the highest average score.
- **Advantages**:
  - Strategically robust; accounts for long-term consequences.
  - Adaptable through heuristic enhancements and time constraints.
- **Performance**:
  - Requires significant computational resources.
  - Provides a strong challenge for skilled players.

# 6. COMPLEXITY AND PERFORMANCE ANALYSIS COMPLEXITY

## Time complexity

- **GUI Rendering**: Constant per frame, dependent on board size.
- **AI MCTS**:
  - Move simulation: O(L × D), where L is the number of legal moves and D is the depth of random playouts.

## Space complexity

- **Board Representation**: O(1) for 64 squares.
- **AI**: Additional space for storing move scores and board copies during simulations.

## performance

- GUI updates are smooth at typical frame rates.
- AI performance is scalable based on time limits.

# 7. REFERENCES


1. **Pygame Documentation**: https://www.pygame.org/docs/

2. **python-chess Library**: https://python-chess.readthedocs.io/

3. **Monte Carlo Tree Search**: Browne, C. B., et al. "A survey of Monte Carlo tree search methods." *IEEE Transactions on Computational Intelligence and AI in Games*, 2012.

4. **Chess Rules and Mechanics**: FIDE Handbook