Artificial Intelligence 12th project
Chess game

Dr: Aliaa Youssef
TA: Amr Daba
By: Aya Khattab,
Mayar Elminesy,
Yousra Okasha,
Yassin zakaria

Chess game

Introduction

The code of the chess game is Python application that implements a graphical user interface (GUI) for playing chess. It allows for both Player vs. Player (1v1) and Player vs. AI (1vAI) modes, complete with various functionalities to make gameplay interactive and engaging. The application uses the `tkinter` library for the GUI and the `python-chess` library to handle chess mechanics and game rules.

Here's an introduction to the application's features:

Chess GUI Application Overview

This chess application is designed for interactive gameplay, offering a variety of features:

1. **Game Modes**:
    - **1v1 Mode**: Two players can compete locally on the same device.
    - **1vAI Mode**: Play against the computer with adjustable difficulty levels:
        - Easy
        - Medium
        - Hard
2. **Core Features**:
    - **Chessboard Rendering**: Displays an 8x8 chessboard using a `tk.Canvas` widget.
    - **Piece Movement**: Supports legal chess moves with visual indicators for available moves.
    - **Game Rules**: Enforces chess rules, including check, checkmate, stalemate, and pawn promotion.
    - **Captured Pieces**: Tracks and displays captured pieces for both players.
    - **Undo Moves**: Not directly implemented in the provided code, but could be extended.
    - **Pawn Promotion**: Allows the player to choose a promotion piece when a pawn reaches the opponent's end of the board.
3. **Interactive Elements**:
    - **Dynamic Popups**: Includes alerts for check conditions and game-over scenarios.
    - **Pawn Promotion Popup**: Prompts the player to select a piece when a pawn is promoted.
    - **Captured Pieces Viewer**: Opens a new window to display captured pieces with a scrollable interface.
4. **AI Integration**:
    - The AI makes moves based on difficulty:
        - **Easy**: Uses a simple heuristic evaluation of material advantage.
        - **Medium/Hard**: Implements a minimax algorithm with adjustable depth for better decision-making.

.

5. **Customizable UI**:
   - Widgets like labels, buttons, and frames are styled for better visual appeal.
   - The application dynamically adjusts to screen size for a centered display.
6. **Game Reset and Navigation**:
   - Restart the game with a single click.
   - Navigate back to the main menu to choose another game mode.

How It Works

1. **Initialization**:
   - The `ChessApp` class manages the entire application, initializing the GUI, game logic, and event handling.
   - The chessboard is rendered using a grid of colored squares on a canvas.
2. **Player Interactions**:
   - Players click on pieces to select them, and valid moves are highlighted.
   - After a move, the application updates the board and handles checks for game-over conditions.
3. **AI Functionality**:
   - For AI moves, the application evaluates the board state and calculates optimal moves based on the difficulty level.
4. **End-of-Game Scenarios**:
   - The application detects checkmate, stalemate, insufficient material, and draws, displaying the result in a popup.

AI logic

The AI logic in the chess application uses the **Minimax Algorithm**, a decision-making algorithm commonly used in two-player games like chess.

The **Minimax Algorithm** is a recursive method used for decision-making in games. The goal is to minimize the possible loss for a worst-case scenario. In chess:

- The **maximizing player** (AI in this case) tries to achieve the highest score.
- The **minimizing player** (the human opponent) tries to minimize the AI's score.

Minimax in the Chess Application

Here’s how Minimax is implemented :

1. Minimax Move Selection

The AI selects the best move using the minimax_move function

```
def minimax_move(self, depth):

best_move = None

   best_score = -float('inf')  # AI maximizes the score

for move in self.board.legal_moves:

      self.board.push(move)  # Make the move

      score = self.minimax(depth - 1, False)  # Recursively evaluate

      self.board.pop()  # Undo the move

         if score > best_score:

         best_score = score

         best_move = move

           return best_move
```

- The AI iterates through all legal moves, simulates each move on the board, and evaluates the resulting state using the `minimax` function.
- The move with the highest score is selected as the best move.

2. Minimax Function

The recursive logic is implemented in the `minimax` function:

```
def minimax(self, depth, is_maximizing):

   if depth == 0 or self.board.is_game_over():

      return self.evaluate_board()  # Evaluate the board state

    if is_maximizing:

      best_score = -float('inf')

      for move in self.board.legal_moves:

         self.board.push(move)
```

```
            score = self.minimax(depth - 1, False)

            self.board.pop()

            best_score = max(best_score, score)

        return best_score

    else:

        best_score = float('inf')

        for move in self.board.legal_moves:

            self.board.push(move)

            score = self.minimax(depth - 1, True)

            self.board.pop()

            best_score = min(best_score, score)

        return best_score
```

- Base Case: If the depth reaches 0 or the game is over, the function evaluates the board using `evaluate_board`.
- Recursive Case:
  For the maximizing player, it selects the move with the highest score.
  For the minimizing player, it selects the move with the lowest score

3. Board Evaluation

The board is evaluated using `evaluate_board`:

```
def evaluate_board(self):

    if self.board.is_checkmate():

        return -1000 if self.board.turn == chess.WHITE else 1000

    elif self.board.is_stalemate() or self.board.is_insufficient_material() or
self.board.is_seventyfive_moves():

        return 0

     material_score = 0
```

```
    for square in chess.SQUARES:
        piece = self.board.piece_at(square)
if piece:
            material_score += self.get_piece_value(piece)
return material_score
```

The function considers:

- Checkmate: Assigns a high positive or negative score depending on the winner.
- Stalemate or Draw: Returns a neutral score of 0.
- Material Value: Calculates the score based on the value of pieces on the board using `get_piece_value`.

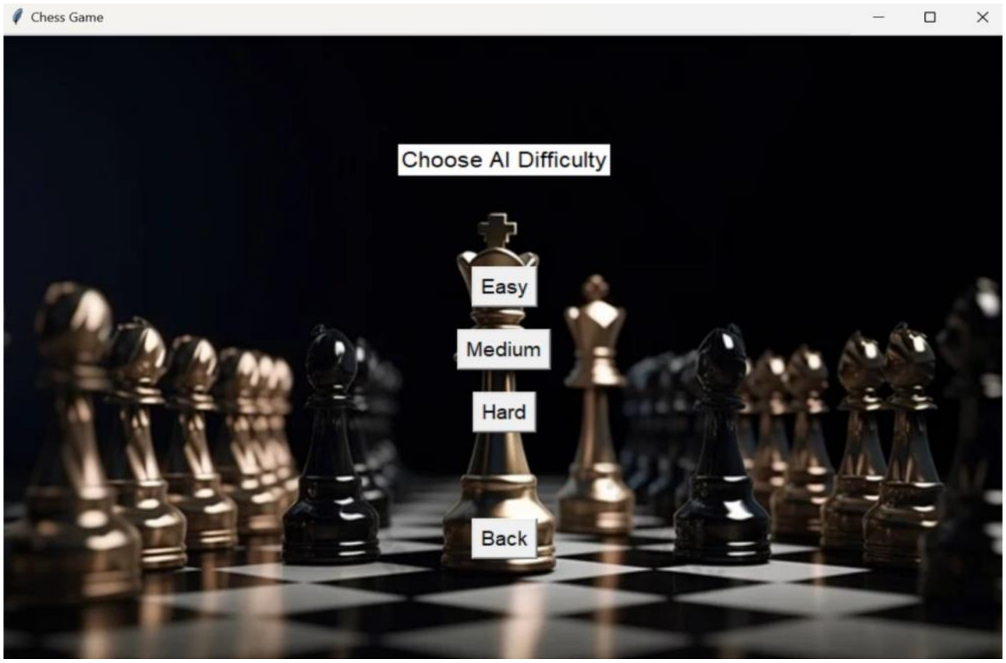
Advantages:

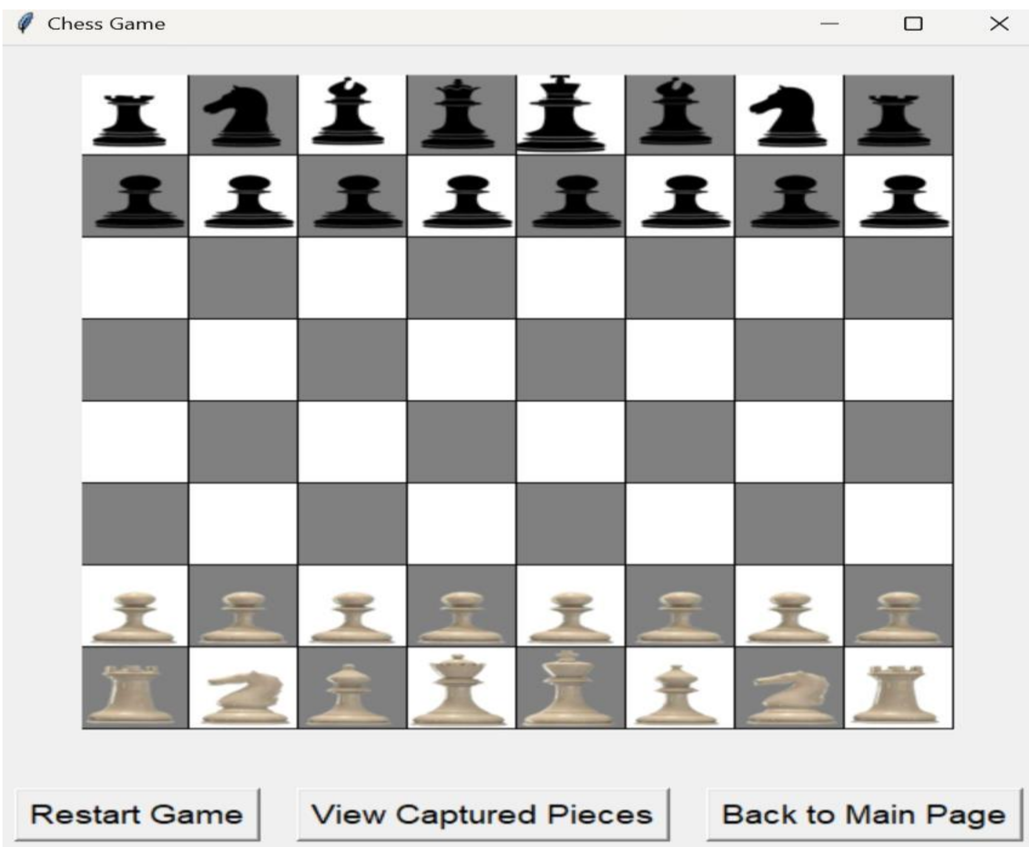
1. **Strategic Thinking**: Evaluates multiple moves ahead, making it challenging for the opponent.
2. **Customizable Depth**: Allows adjusting the difficulty level by varying the search depth.
3. **Sound Decision-Making**: Considers all possible outcomes, ensuring optimal play.

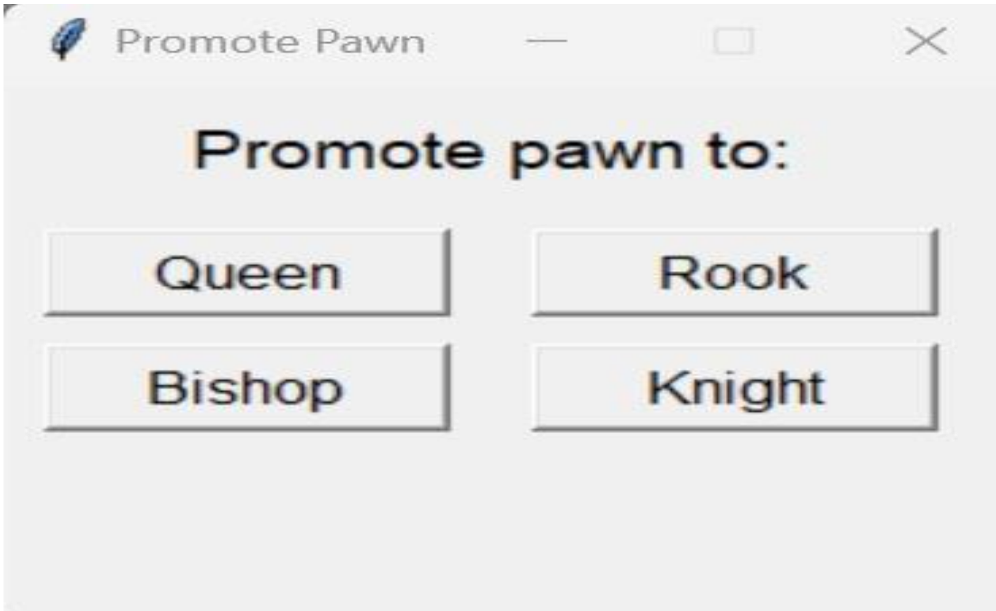
Limitations:

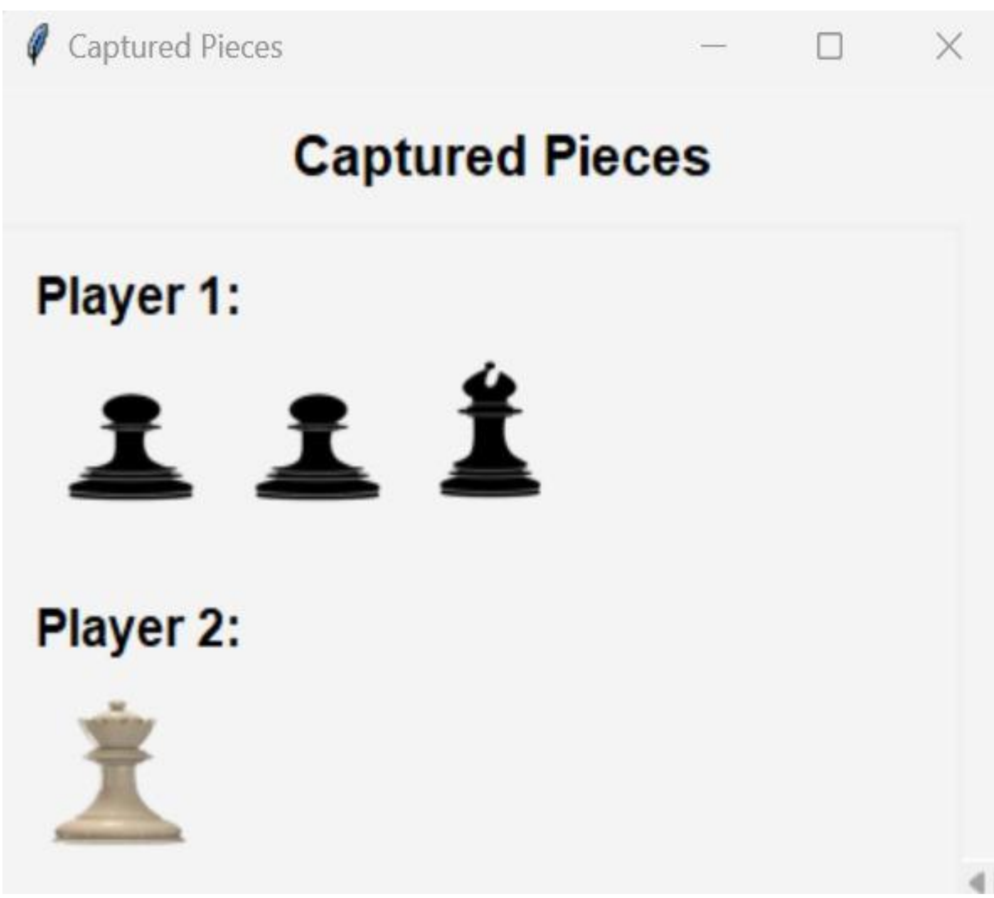
1. **Computational Cost**: The number of board states grows exponentially with depth, making it computationally expensive.
    - Example: If each position has 30 possible moves and depth = 3, the algorithm evaluates 303=27,00030^3 = 27,000303=27,000 positions.
2. **Limited Lookahead**: The depth limit means it may miss long-term strategies or traps.
3. **Heuristic Evaluation**: Relies on a simple evaluation function, which may not capture the full complexity of the board state.

Shots from the game:






Promote Pawn
Promote pawn to:
Queen
Rook
Bishop
Knight


Captured Pieces
Captured Pieces
Player 1:
Player 2: