National University of Computer and Emerging Sciences
Islamabad Campus

# Assignment 04
## Human vs AI
## Chess Game

**Submitted by:** Ayesha Areej
**Roll number:** i221711
**Date:** 23 April 2025

# Table of Contents

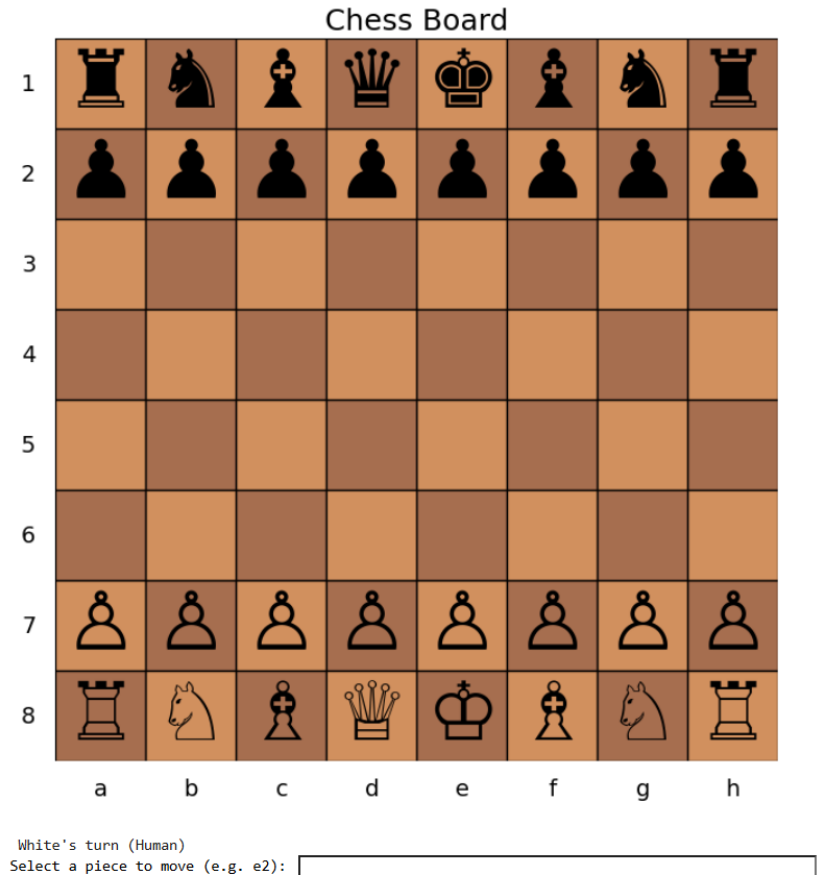
## Introduction

This assignment focuses on building a simplified Human vs AI Chess game using Python. The primary goal is to develop an interactive chess application where a human player competes against the AI opponent. The AI's decision-making is powered by Min-Max and Alpha beta Pruning to improve performance. To achieve this goal, the project adopts a modular, Object-Oriented programming approach. All core chess mechanics such as piece movement, special moves which include castling, promotion, en passant, check, checkmate, and stalemate detection must be accurately implemented. GUI is required to facilitate human interaction.

## Objective

The Objective of this assignment is to design and implement a simplified Chess game in Python that enables gameplay between a human and AI as an opponent. The AI should make strategic decisions using the minmax algorithm and alpha beta pruning. This should include the basic GUI implementation and the rules to be embedded.

## CHESS BOARD



## Classes Implementation

The code was designed for the Object Oriented Approach.

### ChessGame:

```
class ChessGame:
    def __init__(self):
        self.board = Board()
        self.players = {WHITE: HumanPlayer(WHITE), BLACK: AIPlayer(BLACK)}
        self.turn = WHITE

    def play(self):
        while True:
            player = self.players[self.turn]
```

The Functions it includes:

- def __init__(self):
- def play(self):
- def parse_move(self, move_input):
- def is_valid_move(self, move):
- def check_for_end_conditions(self):
- def is_in_check(color):
- def is_king_in_check(self, color):

### Board:

```
class Board:
    def __init__(self):
        self.grid = [[None]*8 for _ in range(8)]
        self.en_passant_target = None  # Square where en passant capture is possible
        self.castling_rights = {
            WHITE: {'kingside': True, 'queenside': True},
            BLACK: {'kingside': True, 'queenside': True}
        }
        self.setup_pieces()
```

The Functions it includes:

- def __init__(self):
- def setup_pieces(self):
- def move_piece(self, move):

- def get_piece(self, row, col):
- def display(self):
- def draw_chessboard(self):
- def is_square_under_attack(self, row, col, color):

## Move:

```
class Move:
    def __init__(self, start_pos, end_pos, piece, captured=None, move_type='normal'):
        self.start_pos = start_pos
        self.end_pos = end_pos
        self.piece = piece
        self.captured = captured
        self.move_type = move_type  # 'normal', 'castle', 'en_passant', 'promotion'

    def __str__(self):
        if self.move_type == 'castle':
            side = "kingside" if self.end_pos[1] > self.start_pos[1] else "queenside"
            return f"{side} castling"
```

The Functions it includes:

- def __init__(self, start_pos, end_pos, piece, captured=None, move_type='normal'):
- def __str__(self):

## Piece:

```
class Piece:
    def __init__(self, color):
        self.color = color
    @property
    def name(self):
        return PIECE_NAMES.get(self.symbol, 'Piece')

    def get_valid_moves(self, board, row, col):
        raise NotImplementedError
```

The Functions it includes:

- def __init__(self, color):
- def name(self):
- def get_valid_moves(self, board, row, col):

## King:

```
class King(Piece):
    def __init__(self, color):
        super().__init__(color)
        self.symbol = 'K'
        self.has_moved = False

    def get_valid_moves(self, board, row, col):
        moves = []
        # Regular king moves (don't check for check here)
        for dr in [-1, 0, 1]:
            for dc in [-1, 0, 1]:
                if dr == 0 and dc == 0:
```

The Functions it includes:

- def __init__(self, color):
- def get_valid_moves(self, board, row, col):

## Queen:

```
class Queen(Piece):
    def __init__(self, color):
        super().__init__(color)
        self.symbol = 'Q'

    def get_valid_moves(self, board, row, col):
        moves = []
        # Rook-like moves
```

The Functions it includes:

- def __init__(self, color):
- def get_valid_moves(self, board, row, col):

## Rook:

```
class Rook(Piece):
    def __init__(self, color):
        super().__init__(color)
        self.symbol = 'R'
        self.has_moved = False

    def get_valid_moves(self, board, row, col):
        moves = []
```

The Functions it includes:

- def __init__(self, color):
- def get_valid_moves(self, board, row, col):

## Bishop:

```
class Bishop(Piece):
    def __init__(self, color):
        super().__init__(color)
        self.symbol = 'B'
```

The Functions it includes:

- def __init__(self, color):
- def get_valid_moves(self, board, row, col):

## Knight:

```
class Knight(Piece):
    def __init__(self, color):
        super().__init__(color)
        self.symbol = 'N'

    def get_valid_moves(self, board, row, col):
        moves = []
        directions = [(2, 1), (1, 2), (-1, 2), (-2, 1),
        for dr, dc in directions:
```

The Functions it includes:

- def __init__(self, color):

- def get_valid_moves(self, board, row, col):

## Pawn:

```
class Pawn(Piece):
    def __init__(self, color):
        super().__init__(color)
        self.symbol = 'P'

    def get_valid_moves(self, board, row, col):
        moves = []
        direction = -1 if self.color == WHITE else 1
        start_row = 6 if self.color == WHITE else 1
```

The Functions it includes:

- def __init__(self, color):
- def get_valid_moves(self, board, row, col):

## Player:

```
class Player:
    def __init__(self, color):
        self.color = color

    def get_move(self, board):
        raise NotImplementedError
```

The Functions it includes:

- def __init__(self, color):
- def get_move(self, board):

## HumanPlayer:

```
class HumanPlayer(Player):
    def get_move(self, board):
        while True:
            try:
                # Display the board first
                board.display()

                print(f"\n{self.color.capitalize()}'s turn (Human)")
                piece_input = input("Select a piece to move (e.g. e2): ").strip().lower()

                if len(piece_input) == 2:
                    col = ord(piece_input[0]) - ord('a')
                    row = 8 - int(piece_input[1])

                    if 0 <= row < 8 and 0 <= col < 8:
                        piece = board.get_piece(row, col)
```

The Functions it includes:

- def get_move(self, board):

## AIPlayer:

```
class AIPlayer(Player):
    def get_move(self, board):
        print(f"\n{self.color.capitalize()}'s turn (AI)")
        # Initialize alpha and beta for the root call
        _, best_move = self.minimax(board, 3, float('-inf'), float('inf'), True)
        if best_move:
            start_col = chr(best_move.start_pos[1] + ord('a'))
            start_row = str(8 - best_move.start_pos[0])
            end_col = chr(best_move.end_pos[1] + ord('a'))
            end_row = str(8 - best_move.end_pos[0])
            return f"{start_col}{start_row}{end_col}{end_row}"
        return None

    def minimax(self, board, depth, alpha, beta, maximizing_player):
        if depth == 0:
            return Evaluation.evaluate(board, self.color), None
```

The Functions it includes:

- def get_move(self, board):
- def minimax(self, board, depth, alpha, beta, maximizing_player):

## Evaluation:

```
class Evaluation:
    @staticmethod
    def evaluate(board, color):
        score = 0

        # Material score
        for row in range(8):
            for col in range(8):
                piece = board.get_piece(row, col)
                if piece:
                    value = PIECE_VALUES.get(piece.symbol.upper(), 0)
                    # King safety bonus
                    if piece.symbol.upper() == 'K':
                        value += 100  # Arbitrarily large value for king safety

                    if piece.color == color:
                        score += value
                    else:
                        score -= value

        # Positional bonuses
        for row in range(8):
            for col in range(8):
                piece = board.get_piece(row, col)
                if piece:
                    # Center control bonus
```

The Functions it includes:

- def evaluate(board, color):

## AI Implementation

Minimax algorithm and Alpha beta pruning.

```
def minimax(self, board, depth, alpha, beta, maximizing_player):
    if depth == 0:
        return Evaluation.evaluate(board, self.color), None

    valid_moves = []
    for r in range(8):
        for c in range(8):
            piece = board.get_piece(r, c)
            if piece and piece.color == (self.color if maximizing_player else (BLACK if self.color == WHITE else WHITE)):
                for move_pos in piece.get_valid_moves(board, r, c):
                    valid_moves.append(Move((r, c), move_pos, piece))

    if not valid_moves:
        # Checkmate or stalemate
        score = Evaluation.evaluate(board, self.color)
        return score, None

    best_move = None

    if maximizing_player:
        max_eval = float('-inf')
        for move in valid_moves:
            new_board = copy.deepcopy(board)
            new_board.move_piece(move)
            evaluation, _ = self.minimax(new_board, depth-1, alpha, beta, False)

            if evaluation > max_eval:
                max_eval = evaluation
                best_move = move

            alpha = max(alpha, evaluation)
            if beta <= alpha:
                break  # Beta cutoff

        return max_eval, best_move
    else:
        min_eval = float('inf')
        for move in valid_moves:
            new_board = copy.deepcopy(board)
            new_board.move_piece(move)
            evaluation, _ = self.minimax(new_board, depth-1, alpha, beta, True)

            if evaluation < min_eval:
                min_eval = evaluation
                best_move = move

            beta = min(beta, evaluation)
            if beta <= alpha:
                break  # Alpha cutoff

        return min_eval, best_move
```

This function evaluates all possible moves up to a depth of 3 and uses alpha beta pruning to skip over the moves to reduce overhead.

def minimax(self, board, depth, alpha, beta, maximizing_player):

In this:

Board: current game state

Depth: how many moves to look ahead

Alpha: best value maximizing player can guarantee

Beta: best value minimizing player can guarantee

Maximizing_player: a Boolean flag true when its AI turn

## How it works:

It uses the recursion, so the base case is that: If it reaches to the max depth, it evaluates the board and returns that score. This is the leaf node of the minmax tree.

It goes through every square of the board and generates all possible moves if no valid moves than this could be checkmate or stalemate then it evaluates in this case.

For each move it simulate the deep copy and recursively calls the minimax function for the opponents turn depth-1 updates the score.

```
_, best_move = self.minimax(board, 3, float('-inf'), float('inf'), True)
if best_move:
```

Depth or level is settled into 3.

```
# Piece values for evaluation
PIECE_VALUES = {
    'Q': 9, 'R': 5, 'B': 3, 'N': 3, 'P': 1, 'K': 0  # King's value is handled specially
}
```

Weights assigned to the pieces.

```
# King safety bonus
if piece.symbol.upper() == 'K':
    value += 100  # Arbitrarily large value for king safety
```

King safety implemented.

```
# Positional bonuses
for row in range(8):
    for col in range(8):
        piece = board.get_piece(row, col)
        if piece:
            # Center control bonus
            if (3 <= row <= 4) and (3 <= col <= 4):
                if piece.color == color:
                    score += 0.1
                else:
                    score -= 0.1
```

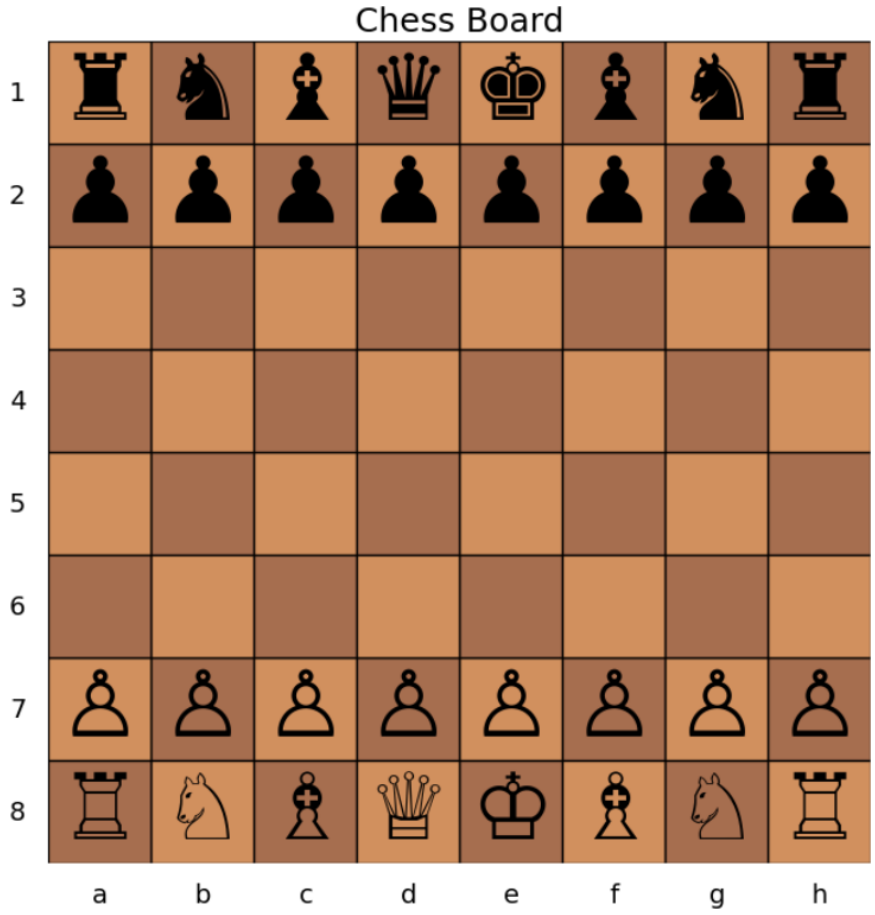
Positional advantages are implemented.

## Gameplay Features

The following are the features implemented:

### Turn based play:

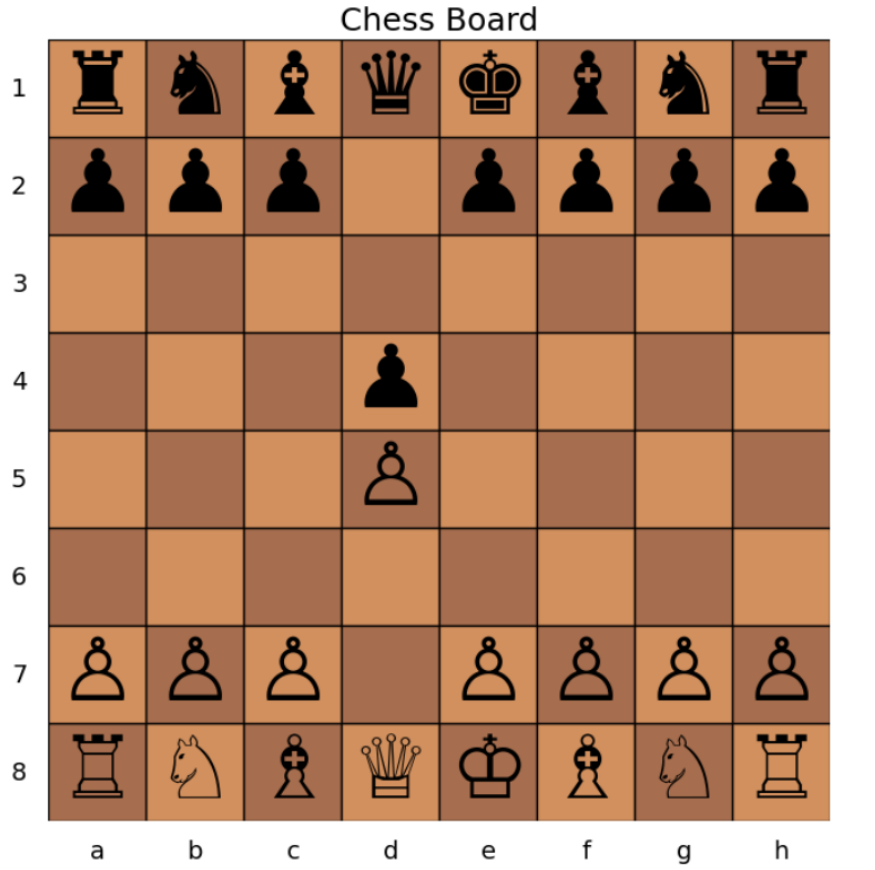
Human Turns (WHITE)




```
White's turn (Human)
Select a piece to move (e.g. e2):  d2
Valid moves for P at d2: d3 d4
Enter destination (e.g. e4):  d4
```

AI Player (BLACK)

Black's turn (AI)
AI plays: d7d5




White's turn (Human)
Select a piece to move (e.g. e2):

## Legal move generation:

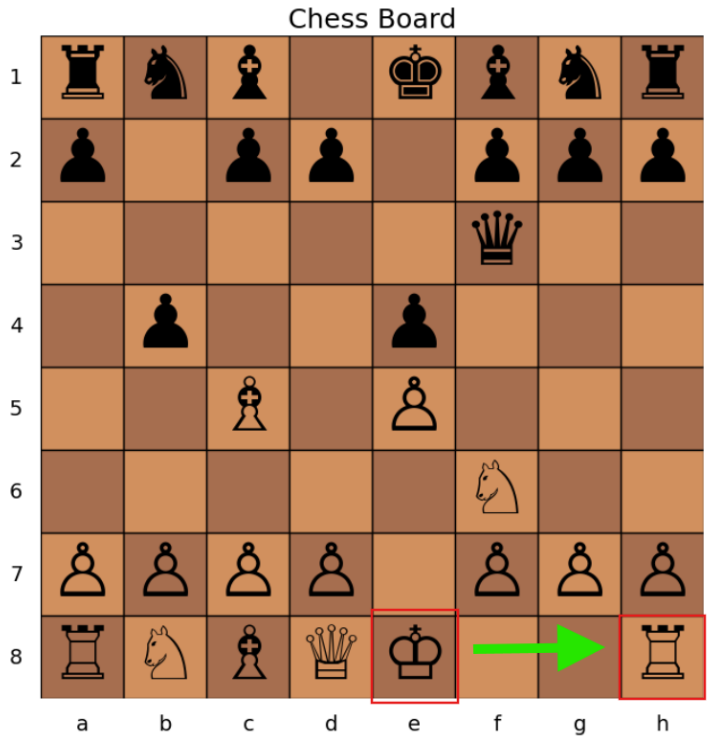
```
White's turn (Human)
Select a piece to move (e.g. e2):  e2
Valid moves for P at e2: e3 e4
Enter destination (e.g. e4): ↑↓ for history. Search history with c-↑/c-↓
```

## Special moves:

- Castling:

  It means that whenever the King is moved towards the rook which is at the distance of 2 then rook automatically moves behind the king. Like this:

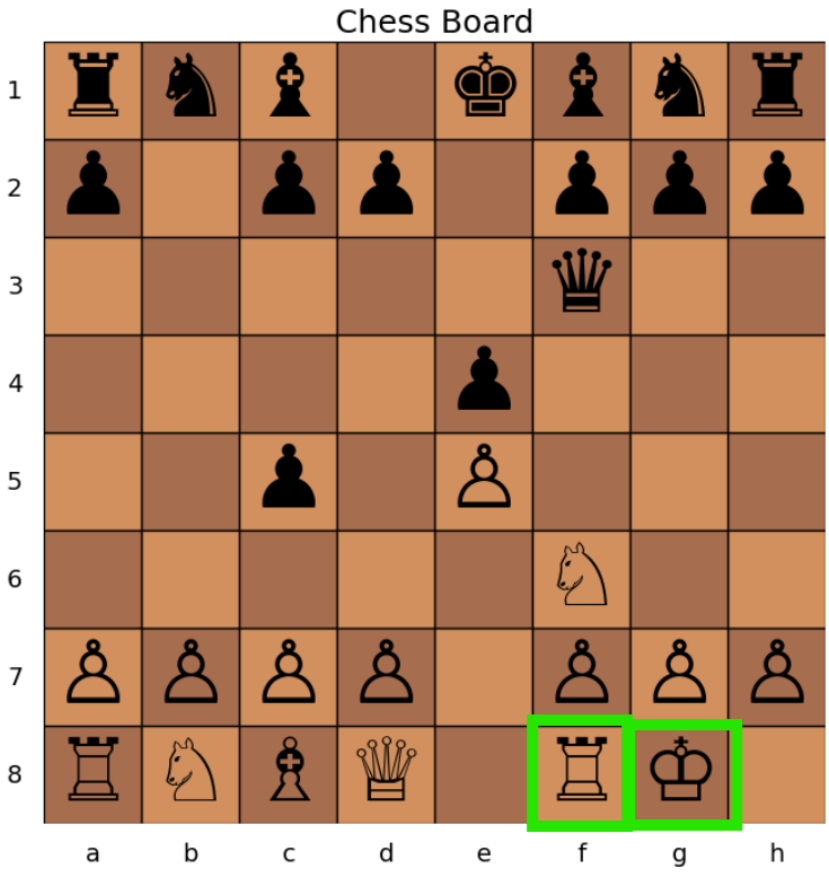
```
Black's turn (AI)
AI plays: b7b5
```




```
White's turn (Human)
Select a piece to move (e.g. e2):  e1
Valid moves for K at e1: e2 f1 g1
Enter destination (e.g. e4):  g1

Black's turn (AI)
AI plays: b5c4
Pawn (black) captured Bishop at c4!
```

```
Black's turn (AI)
AI plays: b5c4
Pawn (black) captured Bishop at c4!
```




```
White's turn (Human)
Select a piece to move (e.g. e2): ↑↓ for history. Search history with c-↑/c-↓
```

- Pawn Promotion:

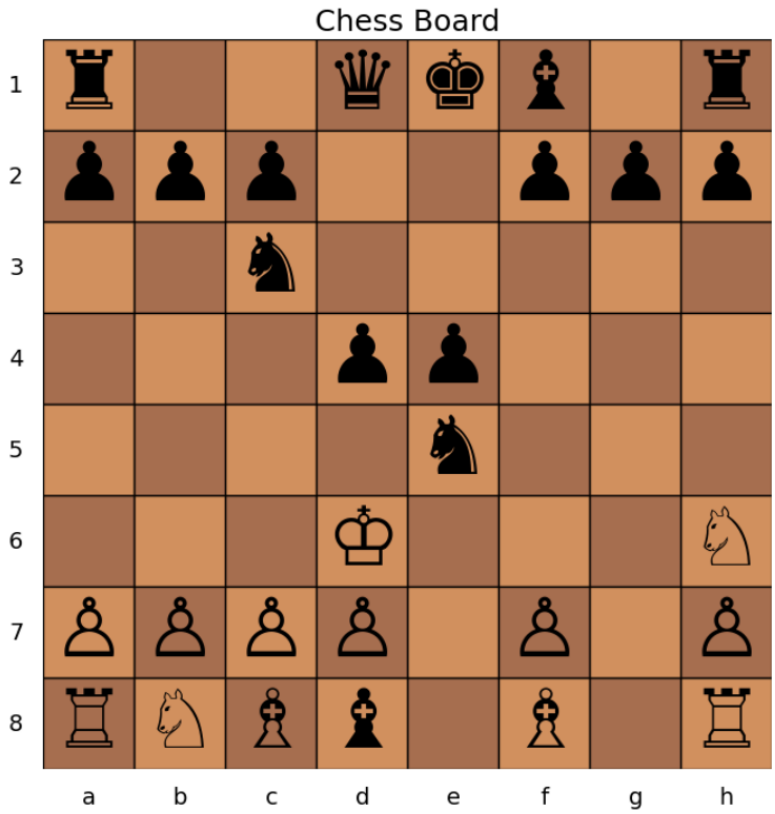
```python
# Inside Board.move_piece(move):

if isinstance(move.piece, Pawn):
    end_row = move.end_pos[0]
    if (move.piece.color == WHITE and end_row == 0) or (move.piece.color == BLACK and end_row == 7):
        # Auto promote to Queen (simplified)
        move.move_type = 'promotion'
        move.promotion_piece = 'Queen'
        self.grid[end_row][move.end_pos[1]] = Queen(move.piece.color)
        self.grid[move.start_pos[0]][move.start_pos[1]] = None
        return move.captured
```

- En Passant:

```
        # Handle en passant capture
if (isinstance(self.grid[sr][sc], Pawn) and
    abs(sc - ec) == 1 and
    captured_piece is None and
    (er, ec) == self.en_passant_target):
    direction = -1 if self.grid[sr][sc].color == WHITE else 1
    captured_row = er + direction
    captured_piece = self.grid[captured_row][ec]  # Actual pawn being captured
    self.grid[captured_row][ec] = None  # Remove the captured pawn
```

## Check detection:




```
White's turn (Human)
Select a piece to move (e.g. e2):  d3
Valid moves for K at d3: c4 d4 e4 c3 e3 e2
Enter destination (e.g. e4):  e4
Invalid move. Please try again.
```

## Checkmate/Stalemate Detection:

```
# Check if any legal moves exist
legal_moves_exist = False
for r in range(8):
    for c in range(8):
        piece = self.board.get_piece(r, c)
        if piece and piece.color == current_color:
            valid_moves = piece.get_valid_moves(self.board, r, c)
            for move_end in valid_moves:
                temp_board = copy.deepcopy(self.board)
                move = Move((r, c), move_end, piece)
                temp_board.move_piece(move)

                # Check if this move gets out of check
                test_game = ChessGame()
                test_game.board = temp_board
                test_game.turn = current_color
                if not test_game.is_king_in_check(current_color):
                    legal_moves_exist = True
                    break
            if legal_moves_exist:
                break
    if legal_moves_exist:
        break

if not legal_moves_exist:
    if is_in_check(current_color):
        print(f"Checkmate! {current_color.capitalize()} loses.")
    else:
        print("Stalemate! It's a draw.")
    return True
```

## Overall game played:

```
Invalid move. Please try again.
```