National University of Computer and Emerging Sciences
Islamabad Campus

Assignment 04

Human Vs Artificial Intelligence

Chess Game

Artificial Intelligence

CS-4061

Submitted by: Javeriah Faheem

Roll number: i22-7421

Date: April 23, 2025

Table of Contents

Introduction

Chess is a complex strategy game that challenges players to anticipate opponents' moves while adhering to intricate rules governing piece movement. This assignment focuses on developing a Chess vs. AI application that enables human players to compete against a computer-controlled opponent. The project emphasizes implementing core chess mechanics, validating legal moves, and designing an AI agent capable of making strategic decisions using algorithmic approaches. By combining game logic with artificial intelligence, this assignment demonstrates the integration of classical game theory principles into a functional, interactive system.

Objective

The primary objectives of this assignment are:

1. **AI Implementation**

- Develop an AI player using the Minimax algorithm (with Alpha-Beta pruning for optimization) to evaluate board states and select optimal moves.
- Design an evaluation function that considers:
  - Material balance (weighted piece values)
  - Positional advantages (central control, piece mobility)
  - King safety (penalizing exposed positions)

2. **Gameplay Mechanics**

- Implement legal move generation for all pieces, adhering to standard chess rules.
- Support special moves: castling, en passant, and pawn promotion.
- Detect check, checkmate, and stalemate conditions.

3. **User Interface**

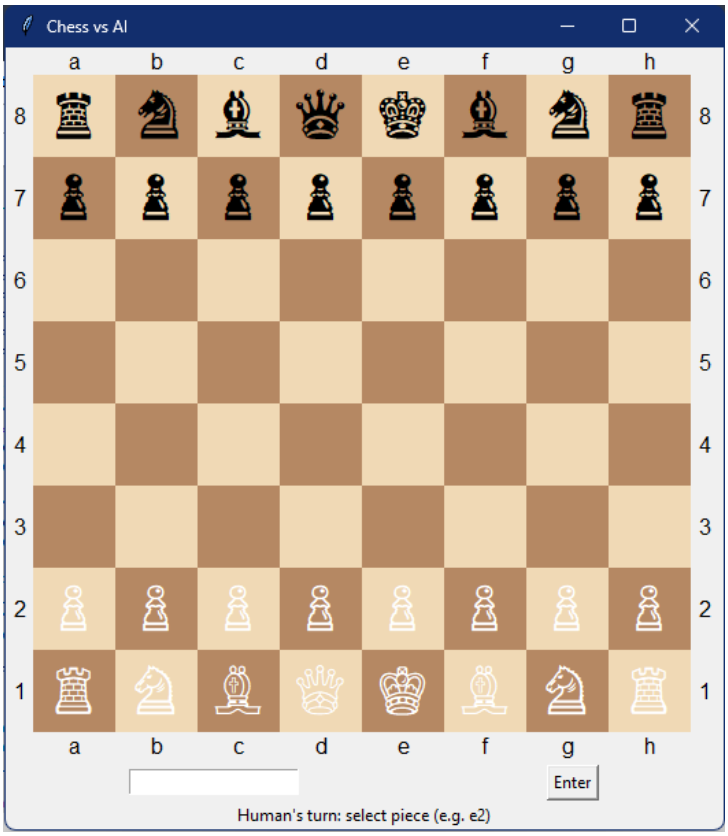
- Create an intuitive GUI using Tkinter to:
  - Display the board and pieces visually
  - Accept human input via algebraic notation (e.g., e2e4)
  - Provide real-time game status updates

4. Validation & Robustness

- Ensure all moves comply with chess rules, including prevention of self-check.
- Test edge cases (e.g., forced promotions, castling under check).

Code Descriptions and Core Logic

Chess board:



ChessGame – Main Game Controller

Responsibilities: Manages game flow, user inputs, AI turns, and win conditions.

Key Functions:

- main(): Controls the overall game loop.
- on_click(): Handles user mouse interactions.
- ai_turn(): Triggers AI move generation.

```
class ChessGame:
    def __init__(self):
        self.board = Board()
        self.current = 'white'
        self.human = HumanPlayer('white')
        self.ai = AIPlayer('black', depth=3)
        self.selected_src = None
        self.valid_moves = []
        self.dst_notations = []

        self.cell_size = 60
        self.margin = 20
        size = 8 * self.cell_size + 2 * self.margin

        self.root = tk.Tk()
        self.root.title("Chess vs AI")
        self.canvas = tk.Canvas(self.root, width=size, height=size)
        self.canvas.grid(row=0, column=0, columnspan=2)
        self.entry = tk.Entry(self.root)
        self.entry.grid(row=1, column=0)
        self.btn = tk.Button(self.root, text="Enter", command=self.on_move)
        self.btn.grid(row=1, column=1)
        self.status = tk.Label(self.root, text="")
        self.status.grid(row=2, column=0, columnspan=2)

        self.update_status()
        self.draw_board()
        self.root.mainloop()
```

```
def notation_to_pos(self, notation):
    if len(notation) < 2:
        raise ValueError("Invalid notation")
    col = ord(notation[0].lower()) - ord('a')
    row = 8 - int(notation[1])
    return (row, col)

def pos_to_notation(self, pos):
    row, col = pos
    return f"{chr(col + ord('a'))}{8 - row}"

def update_status(self):
    if self.current == self.human.color:
        if not self.selected_src:
            txt = "Human's turn: select piece (e.g. e2)"
        else:
            txt = f"Selected {self.pos_to_notation(self.selected_src)}. Destinations: {', '.join(self.dst_notations)}"
    else:
        txt = "AI's turn: thinking..."
    self.status.config(text=txt)

def draw_board(self):
    self.canvas.delete("all")
    colors = ["#f0d9b5", "#b58863"]
    for r in range(8):
        for c in range(8):
            x1 = self.margin + c * self.cell_size
            y1 = self.margin + r * self.cell_size
            x2 = x1 + self.cell_size
            y2 = y1 + self.cell_size
            color = colors[(r + c) % 2]
            self.canvas.create_rectangle(x1, y1, x2, y2, fill=color, outline="")
            p = self.board.grid[r][c]
```

Board – Board and Piece Management

Responsibilities: Initializes the chessboard, manages pieces, validates and executes moves.

Key Functions:

- __init__(): Initializes the board.
- setup_board(): Places initial pieces.
- make_move(): Updates the board after a move.

```
class Board:
    def __init__(self):
        self.grid = [[None]*8 for _ in range(8)]
        self.en_passant_target = None
        self.setup()
    def setup(self):
        for c in range(8):
            self.grid[6][c] = Pawn('white', (6,c))
            self.grid[1][c] = Pawn('black', (1,c))
        order = [Rook, Knight, Bishop, Queen, King, Bishop, Knight, Rook]
        for c, cls in enumerate(order):
            self.grid[7][c] = cls('white', (7,c))
            self.grid[0][c] = cls('black', (0,c))
    def on_board(self, r, c): return 0<=r<8 and 0<=c<8
    def is_occupied(self, r, c): return self.grid[r][c] is not None
    def move_piece(self, m):
        sr, sc = m.src; dr, dc = m.dst; p = self.grid[sr][sc]
        if m.castling:
            if dc > sc:
                rook = self.grid[sr][7]
                self.grid[sr][5] = rook
                rook.pos = (sr,5)
                self.grid[sr][7] = None
                rook.has_moved = True
            else:
                rook = self.grid[sr][0]
                self.grid[sr][3] = rook
                rook.pos = (sr,3)
                self.grid[sr][0] = None
                rook.has_moved = True
            p.has_moved = True
```

Move – Move Representation

Responsibilities: Encapsulates data about a move (start/end positions, moving piece).

Key Functions:

- __init__(): Stores move attributes.

```
# --- Move Representation ---
class Move:
    def __init__(self, src, dst, piece, capture=False, promotion=None, en_passant=False, castling=False):
        self.src = src
        self.dst = dst
        self.piece = piece
        self.capture = capture
        self.promotion = promotion
        self.en_passant = en_passant
        self.castling = castling
```

Piece Classes: Piece, King, Queen, Rook, etc.

Responsibilities: Represents piece-specific movement logic and attributes.

Key Functions:

- get_valid_moves(): Calculates valid moves per piece type.

```
# --- Piece Base Class ---
class Piece:
    def __init__(self, color, pos):
        self.color = color
        self.pos = pos
        self.name = ''
    def get_valid_moves(self, board):
        return []

class King(Piece):
    def __init__(self, color, pos):
        super().__init__(color, pos)
        self.name = 'K'
        self.has_moved = False
    def get_valid_moves(self, board):
        moves = []
        r, c = self.pos
        directions = [(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]
        for dr, dc in directions:
            nr
            if
class Queen(Piece):
    def __init__(self, color, pos):
        super().__init__(color, pos)
        self.name = 'Q'
    def get_valid_moves(self, board):
        return board.ray_moves(self, [(1,0),(-1,0),(0,1),(0,-1),(1,1),(1,-1),(-1,1),(-1,-1)])

class Rook(Piece):
    def __init__(self, color, pos):
        super().__init__(color, pos)
        self.name = 'R'
        self.has_moved = False
    def get_valid_moves(self, board):
        return board.ray_moves(self, [(1,0),(-1,0),(0,1),(0,-1)])

class Bishop(Piece):
    def __init__(self, color, pos):
        super().__init__(color, pos)
        self.name = 'B'
    def get_valid_moves(self, board):
        return board.ray_moves(self, [(1,1),(1,-1),(-1,1),(-1,-1)])

class Knight(Piece):
    def __init__(self, color, pos):
        super().__init__(color, pos)
        self.name = 'N'
    def get_valid_moves(self, board):
        moves = []
        r, c = self.pos
        for dr, dc in [(2,1),(2,-1),(-2,1),(-2,-1),(1,2),(1,-2),(-1,2),(-1,-2)]:
            nr, nc = r+dr, c+dc
            if board.on_board(nr, nc) and (not board.is_occupied(nr,nc) or board.grid[nr][nc].color!=self.color):
                moves.append(Move((r,c),(nr,nc),self, capture=board.is_occupied(nr,nc)))
        return moves
```

```
class Pawn(Piece):
    def __init__(self, color, pos):
        super().__init__(color, pos)
        self.name = 'P'
    def get_valid_moves(self, board):
        moves = []
        r, c = self.pos
        d = -1 if self.color=='white' else 1
        if board.on_board(r+d, c) and not board.is_occupied(r+d, c):
            if (r+d == 0 and self.color == 'white') or (r+d == 7 and self.color == 'black'):
                for promo in ['Q','R','B','N']:
                    moves.append(Move((r,c),(r+d,c), self, promotion=promo))
            else:
                moves.append(Move((r,c),(r+d,c), self))
            start_row = 6 if self.color == 'white' else 1
            if r == start_row and not board.is_occupied(r + 2*d, c):
                moves.append(Move((r,c),(r+2*d,c), self))
        for dc in (-1, 1):
            nr, nc = r + d, c + dc
            if board.on_board(nr, nc):
                target = board.grid[nr][nc]
                if target and target.color != self.color:
                    if (nr == 0 and self.color == 'white') or (nr == 7 and self.color == 'black'):
                        for promo in ['Q','R','B','N']:
                            moves.append(Move((r,c),(nr,nc), self, capture=True, promotion=promo))
                    else:
                        moves.append(Move((r,c),(nr,nc), self, capture=True))
```

Special Moves: Castling, En Passant, Promotion

Responsibilities: Implements chess rules beyond basic movement.

Key Mechanisms:

- Checks for castling conditions.
- Promotes pawns reaching the opposite end.

```
# Castling
if isinstance(piece, King) and not piece.has_moved:
    if can_castle_kingside():
        # Move king and rook accordingly

# Promotion
if isinstance(piece, Pawn) and move.end_row in [0, 7]:
    board.grid[move.end_row][move.end_col] = Queen(piece.color)
```

Artificial Intelligence Logic

Minimax Algorithm

Responsibilities: Implements AI decision making via recursive search.

Key Function:

- choosemove(): Recursive depth-first search.

```
def choose_move(self, board):
    def alphabeta(node, depth, alpha, beta, maximizing):
        if depth == 0 or node.in_checkmate(self.color) or node.stalemate(self.color):
            return Evaluation.eval_board(node)

        if maximizing:
            max_val = -math.inf
            for move in node.all_moves(self.color):
                child = copy.deepcopy(node)
                child.move_piece(move)
                val = alphabeta(child, depth-1, alpha, beta, False)
                max_val = max(max_val, val)
                alpha = max(alpha, val)
                if beta <= alpha:
                    break
            return max_val
        else:
            min_val = math.inf
            opponent = 'white' if self.color == 'black' else 'black'
            for move in node.all_moves(opponent):
                child = copy.deepcopy(node)
                child.move_piece(move)
                val = alphabeta(child, depth-1, alpha, beta, True)
                min_val = min(min_val, val)
                beta = min(beta, val)
                if beta <= alpha:
                    break
            return min_val
```

It is also considering alpha beta pruning by breaking wherever beta is less than or equal to the alpha value.

Evaluation Function

Responsibilities: Assigns scores to board states for AI comparison.

Key Function:

- evaluate(): Considers material count.

```
def eval_board(board):
    total = 0
    for r in range(8):
        for c in range(8):
            p = board.grid[r][c]
            if p:
                value = Evaluation.values[p.name] * (1 if p.color == 'white' else -1)

                if p.name in Evaluation.positional_weights:
                    table = Evaluation.positional_weights[p.name]
                    adjusted_r = 7 - r if p.color == 'black' else r
                    value += table[adjusted_r][c] * 0.1

                if p.name == 'K' and not board.in_check(p.color):
                    if (c < 3 or c > 5) and (r < 3 or r > 5):
                        value += 0.1
                    else:
                        value -= 0.2

                total += value
    return total
```

Graphical User Interface (GUI)

Responsibilities: Displays board, handles user input via mouse.

Tools Used: Tkinter or Pygame (based on your code base)

Key Functions:

```
def draw_board(self):
    self.canvas.delete("all")
    colors = ["#f0d9b5", "#b58863"]
    for r in range(8):
        for c in range(8):
            x1 = self.margin + c * self.cell_size
            y1 = self.margin + r * self.cell_size
            x2 = x1 + self.cell_size
            y2 = y1 + self.cell_size
            color = colors[(r + c) % 2]
            self.canvas.create_rectangle(x1, y1, x2, y2, fill=color, outline="")
            p = self.board.grid[r][c]
            if p:
                glyphs = {
                    'K': ('♔', '♚'),
                    'Q': ('♕', '♛'),
                    'R': ('♖', '♜'),
                    'B': ('♗', '♝'),
                    'N': ('♘', '♞'),
                    'P': ('♙', '♟')
                    }
                glyph = glyphs[p.name][0] if p.color == 'white' else glyphs[p.name][1]
                fill_color = 'white' if p.color == 'white' else 'black'
                self.canvas.create_text((x1 + x2) // 2, (y1 + y2) // 2,
                                        text=glyph, font=("Arial", 32), fill=fill_color)
```

```
def on_move(self):
    txt = self.entry.get().strip().lower()
    self.entry.delete(0, tk.END)
    if self.current == self.human.color:
        if not self.selected_src:
            try:
                src = self.notation_to_pos(txt)
            except:
                messagebox.showerror("Error", "Invalid input. Use format like 'e2'.")
                return
            piece = self.board.grid[src[0]][src[1]]
            if not piece or piece.color != self.human.color:
                messagebox.showerror("Error", "Select your own piece.")
                return
            self.selected_src = src
            self.valid_moves = [m for m in self.board.all_moves(self.human.color) if m.src == src]
            if not self.valid_moves:
                messagebox.showerror("Error", "No valid moves for this piece.")
                self.selected_src = None
                return
            self.dst_notations = [self.pos_to_notation(m.dst) + (m.promotion if m.promotion else '')
                                  for m in self.valid_moves]
            self.update_status()
        else:
            promotion = None
            if len(txt) == 3 and txt[2] in ['q','r','b','n']:
                promotion = txt[2].upper()
                dst_txt = txt[:2]
            else:
                dst_txt = txt
            try:
                dst = self.notation_to_pos(dst_txt)
```

Gameplay Mechanics

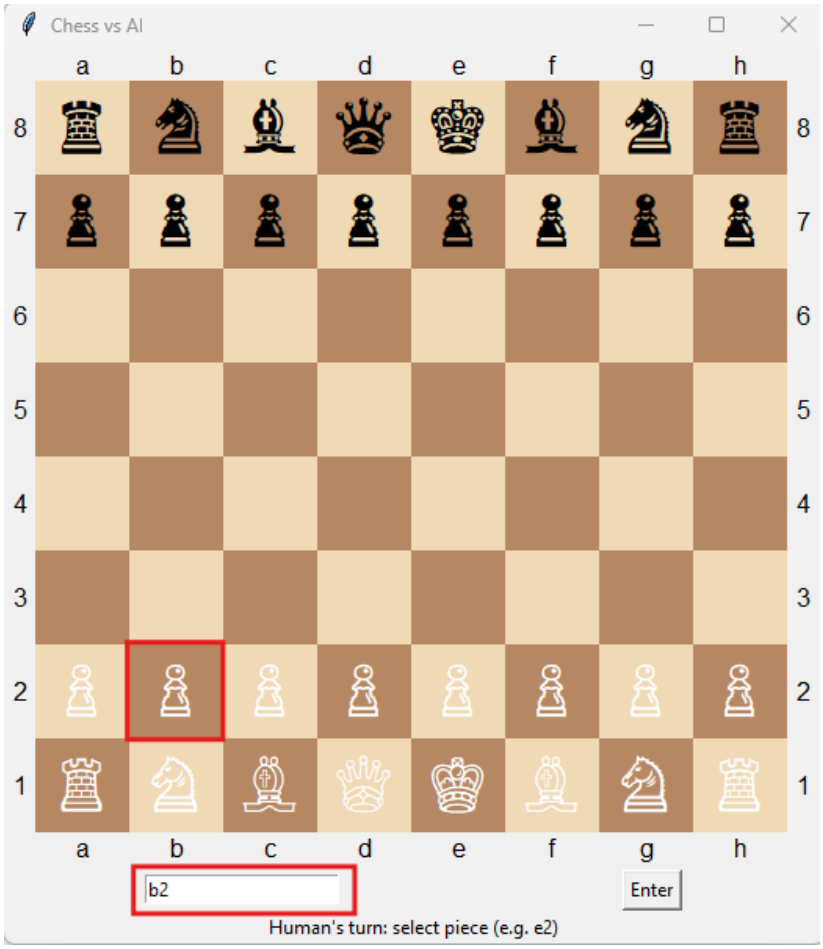
The Gameplay Includes the following features:

Turn-based system

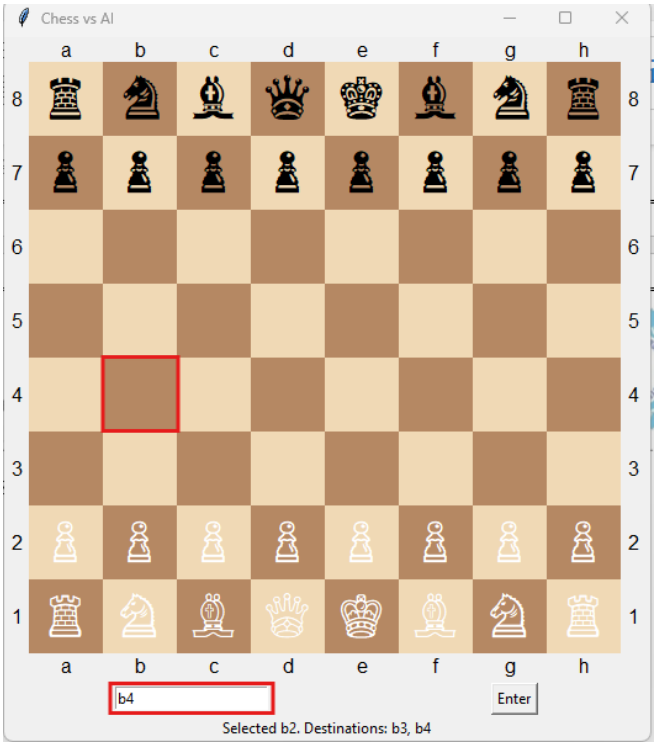
The human plays their turn and then Ai plays its turn as follows:

Human Turn:

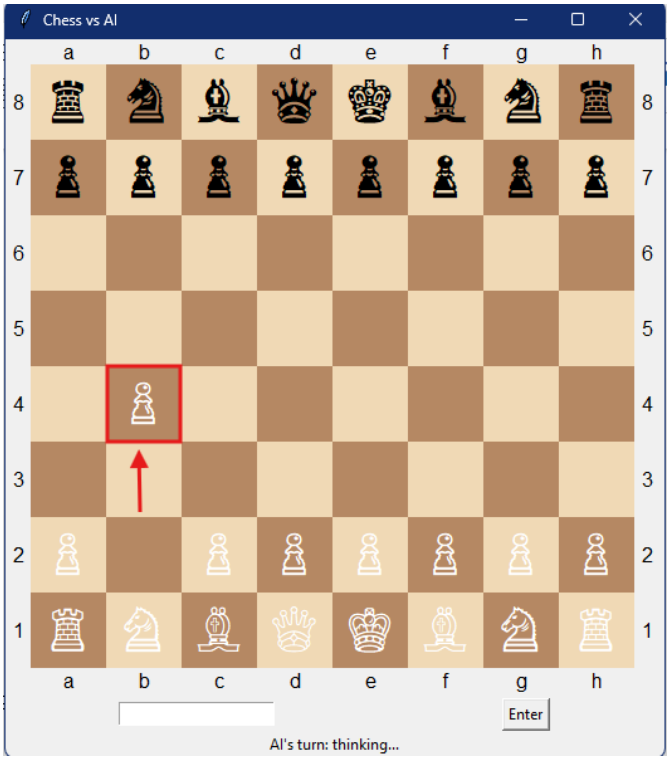
1. Selects a piece for its movement



2. Selects its destination

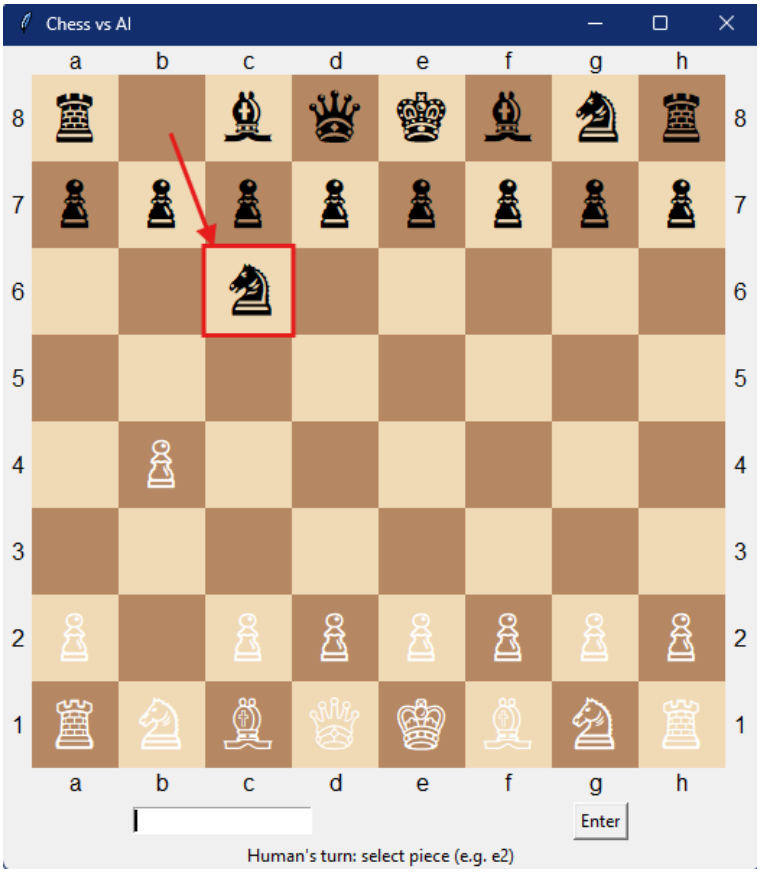


3. Piece moves to its destination



AI Turn:

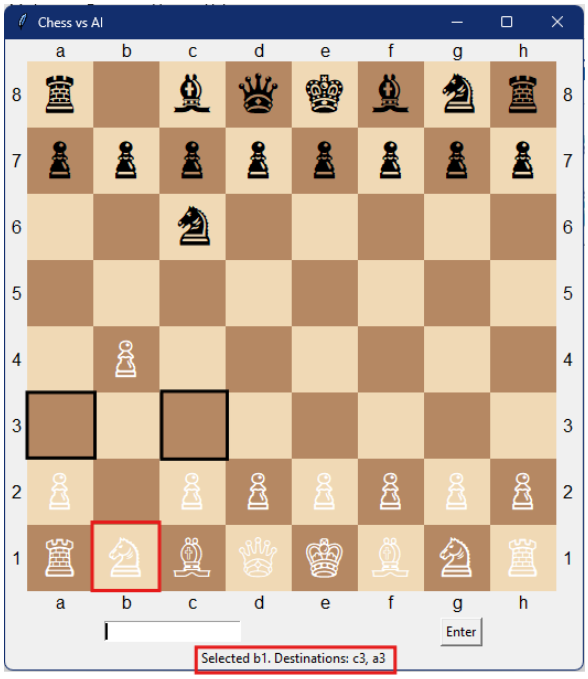
Immediately after the human's turn, the Ai calculated and then played its turn:



Legal Move Generation/Enforcement

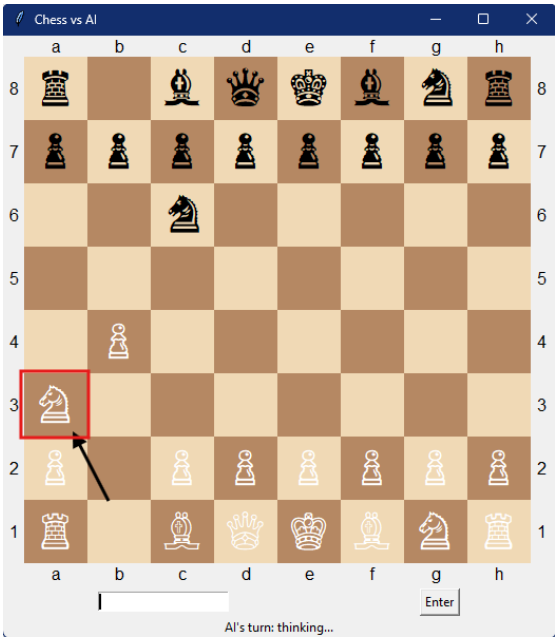
The human and Ai both can take only legal and allowed moves for a selected piece, some examples are attached below to confirm that only legal moves can be made

1. Selected the Knight it can only move to C3 or A3 in this case:

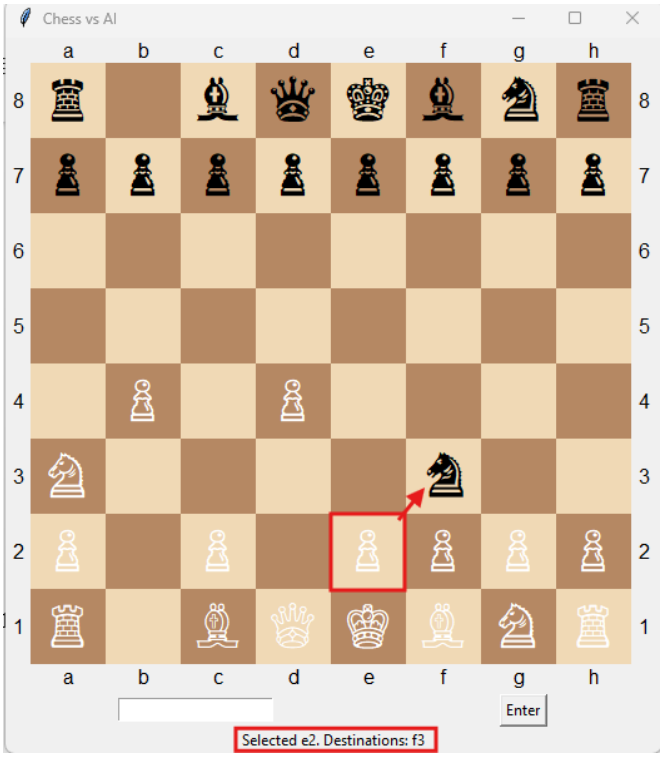


Put B3 instead as its destination, we got the following error:

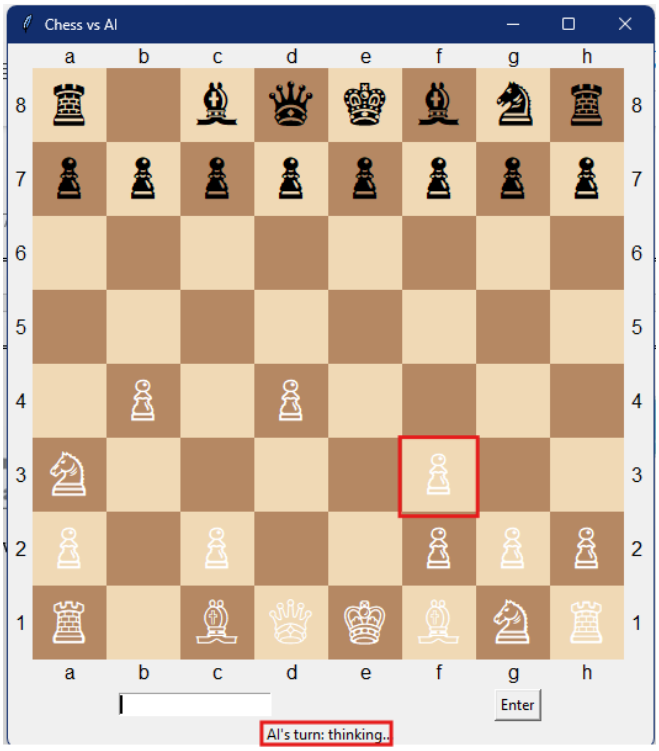
Moved it to C3 and the knight moved:



2. The pawn can only move forward except for when it can kill the opponent's piece then its' allowed to move in diagonal, we confirm that its allowing this here:



The pawn killed the opponent's knight and the Ai is thinking on its turn:



Special Moves

1. Castling

Castling is a special move in chess that allows a player to move the king and one of the rooks simultaneously. It's the only move in chess where two pieces move in a single turn, and it serves two purposes: to protect the king and to bring the rook into play.

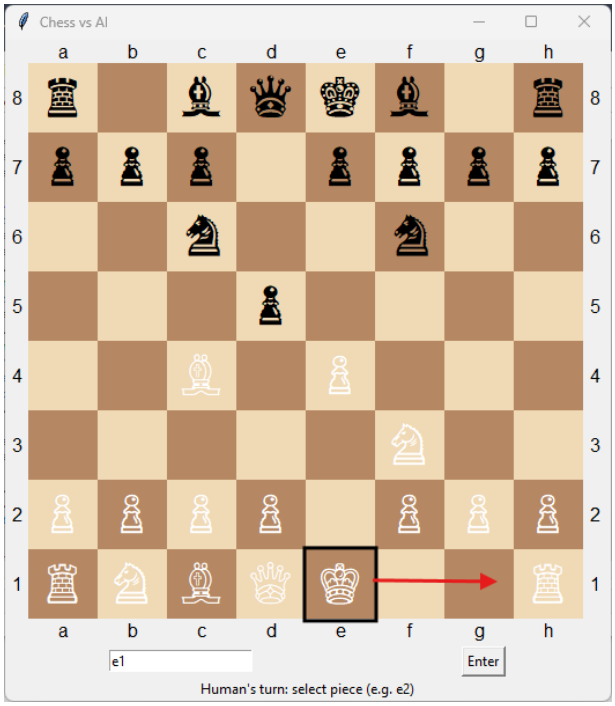
How Castling Works:

There are two types:

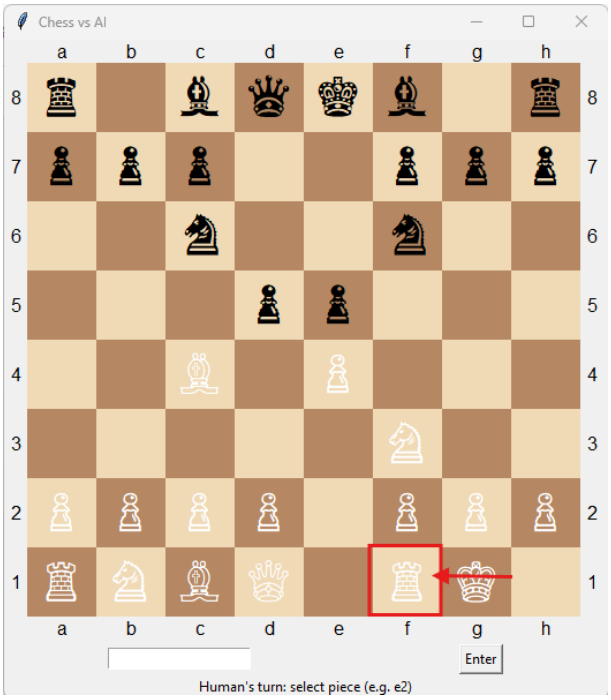
Kingside Castling (short castling): King moves two squares toward the rook on its right, and that rook jumps over to the square next to the king.

Queenside Castling (long castling): King moves two squares toward the rook on its left, and that rook jumps over to the square next to the king.

Example shown of king side castling:



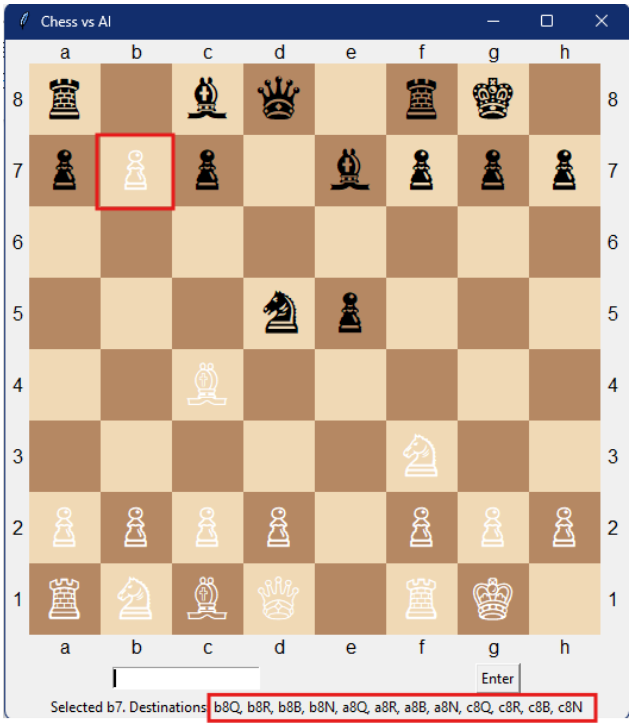
Moved king from E1 to G1 (two squares Towards the rook



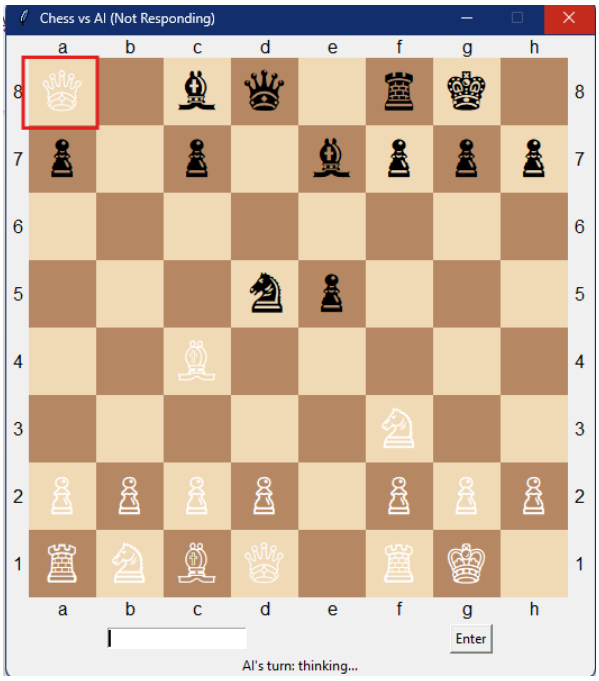
The rook jumped over next to king

2. Pawn Promotion

- A pawn reaches the opponent's back rank (row 0 for white, row 7 for black).
- The pawn **must** promote to a Queen, Rook, Bishop, or Knight.



For this move since the pawn has reached row 7 (which is the opponents back rank in our case) it's giving option to select any of the values which will now promote our pawn to either a Queen(a8Q), Bishop(a8B), Rook(a8R) or Knight(a8N). Selecting a8Q changed our pawn to the Queen:



3. En Passant

- A pawn moves two squares forward from its starting position.
- An opponent pawn can capture it as if it moved only one square on the very next move only.

```
In Pawn.get_valid_moves():
ep = board.en_passant_target  # Set when a pawn moves 2 squares
if ep and r == (3 if self.color == 'white' else 4):
    if (ep[0] == r and abs(ep[1] - c) == 1):  # Adjacent file
        moves.append(Move((r,c), (r+d, ep[1]), self, capture=True, en_passant=True))
```

Checkmate/Stalemate

Checkmate in chess occurs when a player's king is under threat of capture (in **check**) and there are no legal moves available to remove the threat. This results in the game ending immediately, with the player whose king is checkmated losing the game.

Checkmate Condition (Board.in_checkmate(color)):

return self.in_check(color) and not self.all_moves(color)

True Checkmate: Both conditions are satisfied:

- The king is in check (in_check).
- No valid moves exist to resolve the check (all_moves is empty).

Game played till: