Halma Part2

Youssef Elmahdy

6398550

Instructor: Tayyaba Shaheen

CS 470 - Artificial Intelligence

Northern Arizona University (NAU)

1) **Overview:**

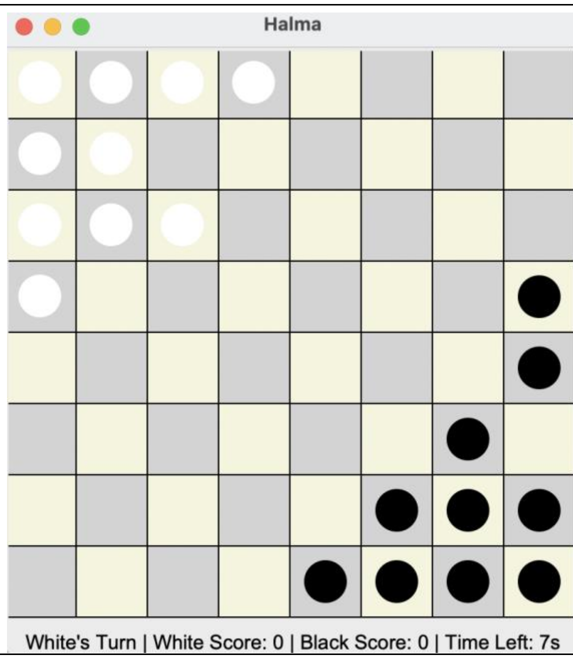
This project is an implementation of the board game Halma, featuring both a graphical user interface (GUI) and an intelligent AI agent. Part1 involves the basic setup of the game, while Part2 focuses on enhancing the game with AI capabilities.

- Part1: The game is built using Python's Tkinter library to create the GUI, where the HalmaBoard class manages the game board, initializes the pieces for both players (white and black), and handles user interactions. The Position class represents each game piece on the board, controlling its placement and movement. Players can select their pieces, view valid moves highlighted on the board, and move pieces one step in any orthogonal direction or perform a single jump over an adjacent piece into an empty square. The game includes turn management, score tracking, and win condition checking to provide a complete gameplay experience.
- Part2: To transform the program into an intelligent system, two main classes are added: AIPlayer and BoardState. The AIPlayer class introduces an AI opponent that uses the Minimax algorithm with alpha-beta pruning to make decisions. It evaluates possible moves up to a certain depth within a time limit to choose the optimal move. The BoardState class represents the current state of the game board for the AI to assess.
- A key component is the utility function within the BoardState class, which determines the "goodness" of a board state. This function calculates the sum of distances of the AI's pieces to their goal area and subtracts the sum of distances of the opponent's pieces to their goal area. By doing this, the AI aims to minimize its pieces' distance to the goal while maximizing the opponent's distance, effectively guiding its strategy to win the game.
- By integrating these components, the game now allows a human player to compete against an intelligent AI that makes strategic decisions based on the current state of the board.

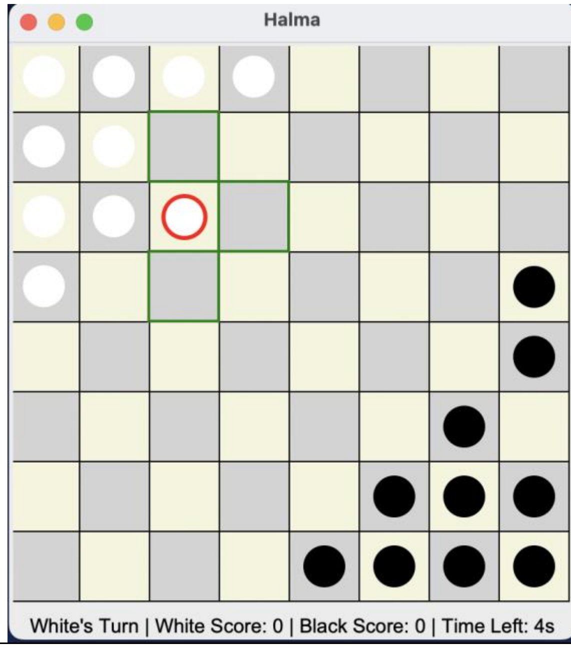
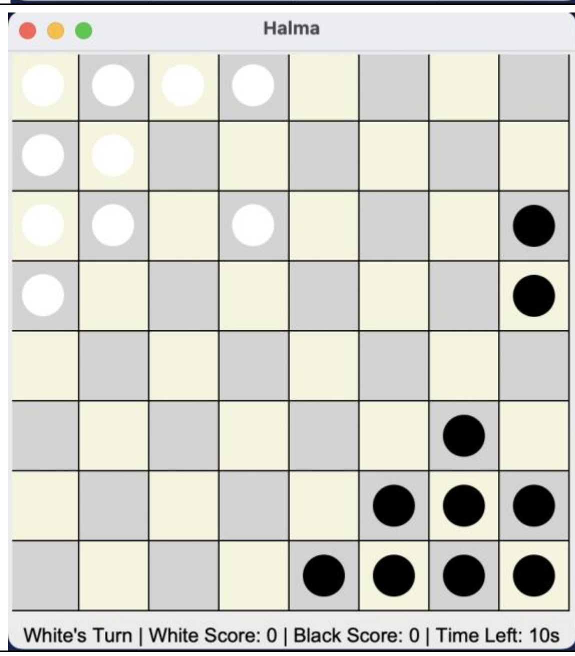
2) Functionality Table:

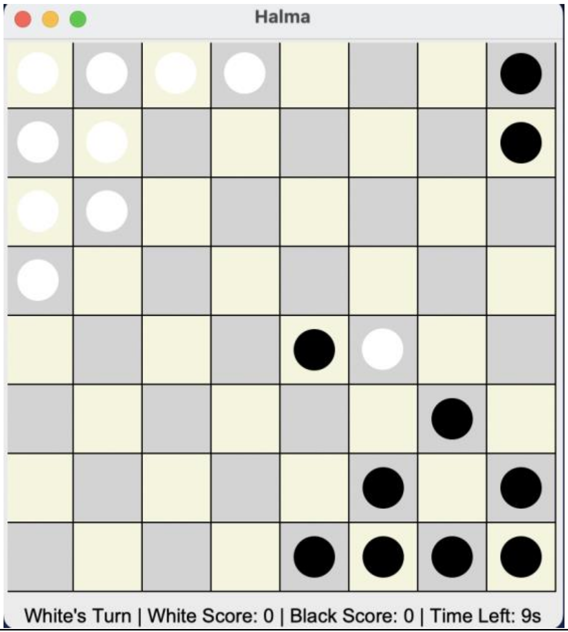
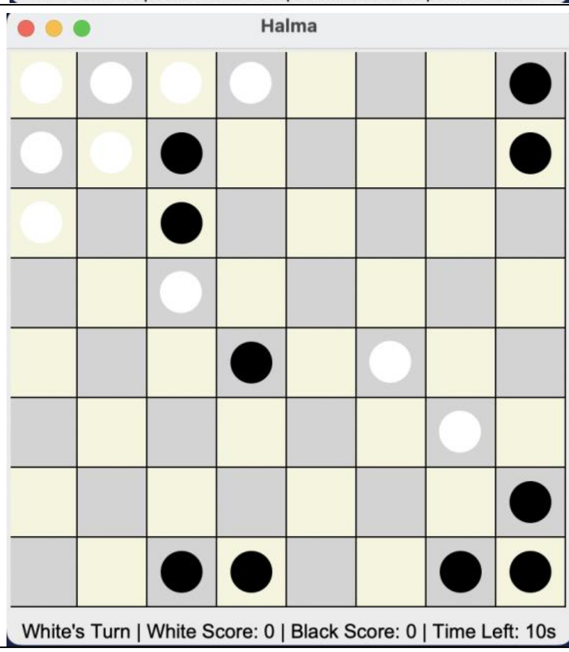
Functionality	% Complete	Notes
Utility Function	100%	Evaluates the board state by calculating the distance of pieces to goal positions. Accounts for both players' progress and handles terminal states (win conditions). Ensures strategic gameplay with a balance between offense and defense.
Minimax search	100%	Fully implemented to explore possible moves for the current player up to a defined depth. Uses the utility function for evaluation and supports both human vs. AI and AI vs. AI modes.
Alpha-beta pruning	100%	Integrated into the minimax search to reduce computational complexity. Improves efficiency by pruning unnecessary branches. Allows deeper searches within the time limit and ensures responsiveness with a timeout safeguard.
Extra Functionality	100%	Features include smooth animations for piece movement, valid move highlighting, turn management
Demos	100%	All screenshots included, with annotations and clear labeling.

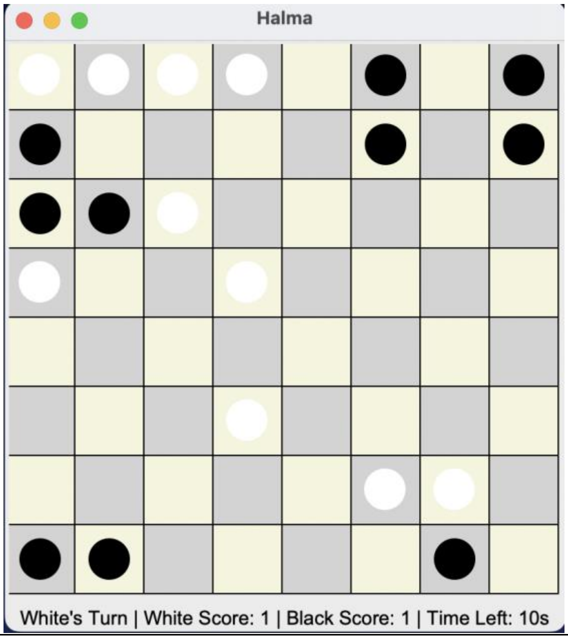
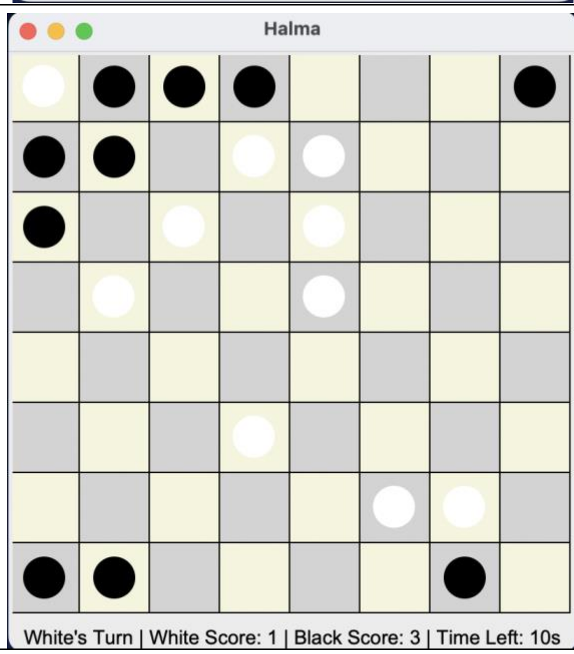
3) Demos

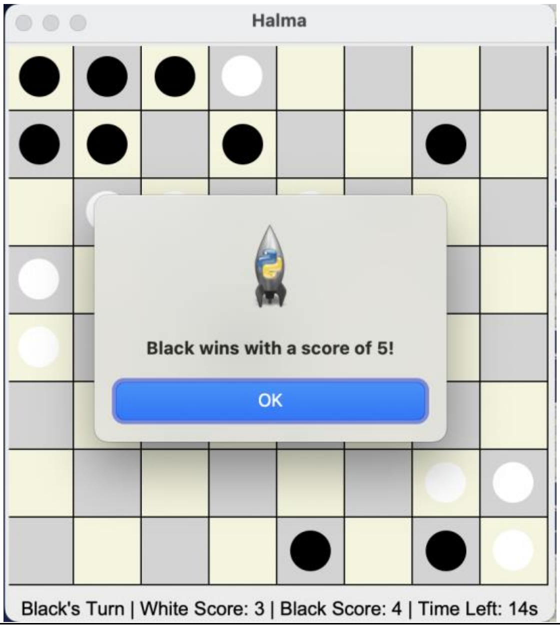
<table>
<tr><th>Screenshot</th><th>Annotation</th></tr>
<tr><td>


</td><td>The image shows a well-formatted graphical display of the Halma game board, with an 8x8 grid using alternating light gray and beige cells for easy visibility. White and black pieces are placed in their respective starting corners, distinguished by color with white being the human and Black the AI agent. Below the board, a status bar displays the current turn, each player’s score, and a countdown timer for move time limits.</td></tr>
</table>

	The human-controlled white piece, highlighted with a red outline, shows valid movement options marked by green squares. These options allow the player to either advance toward the bottom-right goal or reposition strategically to counter the AI's progression
	The AI-controlled black pieces strategically advance toward the top-left goal while positioning themselves to block the human-controlled white pieces, leveraging alpha-beta pruning for efficient decision-making, while the human player must navigate these obstacles to progress toward their bottom-right goal.

	The AI-controlled black pieces exhibit strategic movement, advancing tactically toward the top-left goal while maintaining defensive spacing to block potential pathways for the human-controlled white pieces. The AI's calculated progression includes precise jumps and single-step moves, ensuring optimal positioning for future turns while minimizing unnecessary moves that could expose vulnerabilities.
	The AI-controlled black pieces strategically employ multi-step jumps, leveraging two of its pieces to rapidly advance toward the top-left goal. By using consecutive jumps over other pieces, the AI minimizes the number of turns required to close the distance, demonstrating efficiency in movement while preserving its overall defensive formation. This calculated approach accelerates progress while maintaining pressure on the human player to respond.

	In this scenario, the AI-controlled black pieces have successfully reached one of the target goal positions in the top-left corner, as indicated by the score of 1 for black. This achievement is a result of the AI's strategic use of multi-step jumps and calculated moves to optimize progress toward the goal while maintaining its positioning across the board. The black pieces demonstrate a balance between advancing toward their objective and positioning defensively to challenge the human-controlled white pieces.
	In this stage, the AI-controlled black pieces have significantly advanced, achieving a score of 3 by successfully moving three pieces into the top-left goal area. This demonstrates the AI's efficient use of jumps and strategic movement to prioritize progress toward the goal while maintaining a distributed presence to block the human-controlled white pieces.

	In this final stage, the AI-controlled black pieces successfully achieve victory with a score of 5 by strategically advancing five pieces into the top-left goal area. The winning strategy highlights the AI's efficient use of multi-step jumps, calculated placement of pieces to maintain control over the board, and persistent blocking of white's progress. By optimizing movement and prioritizing goal completion, the black AI outmaneuvers the human-controlled white player to secure a decisive win.

4) Code:

```
import tkinter as tk
from tkinter import messagebox
import time
import threading


class Position:
    def __init__(self, row, col, color, canvas, cell_size):
        self.row = row
        self.col = col
        self.color = color
        self.canvas = canvas
        self.cell_size = cell_size
        self.piece_id = self.create_piece()

    def create_piece(self):
        x1 = self.col * self.cell_size + 10
        y1 = self.row * self.cell_size + 10
        x2 = x1 + self.cell_size - 20
        y2 = y1 + self.cell_size - 20
        return self.canvas.create_oval(x1, y1, x2, y2, fill=self.color, outline="")

    def move_to(self, new_row, new_col, duration=500, callback=None):
        start_x = self.col * self.cell_size + 10
        start_y = self.row * self.cell_size + 10
        end_x = new_col * self.cell_size + 10
        end_y = new_row * self.cell_size + 10
```

```
        dx = end_x - start_x
        dy = end_y - start_y
        steps = int(duration / 20)
        if steps == 0:
            steps = 1
        delta_x = dx / steps
        delta_y = dy / steps
        current_step = 0

        def animate():
            nonlocal current_step
            if current_step < steps:
                self.canvas.move(self.piece_id, delta_x, delta_y)
                current_step += 1
                self.canvas.after(20, animate)
            else:
                final_x1 = end_x
                final_y1 = end_y
                final_x2 = final_x1 + self.cell_size - 20
                final_y2 = final_y1 + self.cell_size - 20
                self.canvas.coords(self.piece_id, final_x1, final_y1, final_x2, final_y2)
                self.row, self.col = new_row, new_col
                if callback:
                    callback()

        animate()

    def set_outline(self, color="red", width=3):
        self.canvas.itemconfig(self.piece_id, outline=color, width=width)

    def clear_outline(self):
        self.canvas.itemconfig(self.piece_id, outline="", width=1)

    def delete(self):
        self.canvas.delete(self.piece_id)


class BoardState:
    def __init__(self, size=8, white_goal=None, black_goal=None):
        self.size = size
        self.board = [['' for _ in range(size)] for _ in range(size)]
        self.white_goal = white_goal if white_goal else []
        self.black_goal = black_goal if black_goal else []
        self.initialize_pieces()

    def initialize_pieces(self):
```

```
        white_positions = [
            (0, 0), (0, 1), (0, 2), (0, 3),
            (1, 0), (1, 1), (1, 2),
            (2, 0), (2, 1),
            (3, 0)
        ]
        black_positions = [
            (7, 7), (7, 6), (7, 5), (7, 4),
            (6, 7), (6, 6), (6, 5),
            (5, 7), (5, 6),
            (4, 7)
        ]
        for row, col in white_positions:
            self.board[row][col] = 'white'
        for row, col in black_positions:
            self.board[row][col] = 'black'

    def copy(self):
        new_board = BoardState(self.size, self.white_goal, self.black_goal)
        new_board.board = [row[:] for row in self.board]
        return new_board

    def get_pieces(self, player_color):
        pieces = []
        for row in range(self.size):
            for col in range(self.size):
                if self.board[row][col] == player_color:
                    pieces.append((row, col))
        return pieces

    def get_possible_moves(self, player_color):
        moves = []
        pieces = self.get_pieces(player_color)
        for piece in pieces:
            row, col = piece
            moves.extend(self.get_piece_moves(row, col))
        return moves

    def get_piece_moves(self, row, col):
        moves = []
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dr, dc in directions:
            nr, nc = row + dr, col + dc
            if 0 <= nr < self.size and 0 <= nc < self.size:
                if self.board[nr][nc] == '':
                    moves.append((row, col, nr, nc, []))
```

```
                elif self.board[nr][nc] != '':
                    jr, jc = nr + dr, nc + dc
                    if 0 <= jr < self.size and 0 <= jc < self.size and self.board[jr][jc] == '':
                        moves.append((row, col, jr, jc, []))
        return moves

    def make_move(self, move):
        from_row, from_col, to_row, to_col, _ = move
        self.board[to_row][to_col] = self.board[from_row][from_col]
        self.board[from_row][from_col] = ''

    def undo_move(self, move):
        from_row, from_col, to_row, to_col, _ = move
        self.board[from_row][from_col] = self.board[to_row][to_col]
        self.board[to_row][to_col] = ''

    def evaluate(self, player_color):
        total_distance = 0
        opponent_color = 'black' if player_color == 'white' else 'white'
        for row in range(self.size):
            for col in range(self.size):
                if self.board[row][col] == player_color:
                    total_distance -= self.distance_to_goal(row, col, player_color)
                elif self.board[row][col] == opponent_color:
                    total_distance += self.distance_to_goal(row, col, opponent_color)
        return total_distance

    def distance_to_goal(self, row, col, color):
        if color == 'white':
            return (self.size - 1 - row) + (self.size - 1 - col)
        else:
            return row + col

    def is_terminal(self):
        white_pieces_in_goal = sum(1 for pos in self.white_goal if
self.board[pos[0]][pos[1]] == 'white')
        black_pieces_in_goal = sum(1 for pos in self.black_goal if
self.board[pos[0]][pos[1]] == 'black')
        if white_pieces_in_goal >= 5:
            return True
        if black_pieces_in_goal >= 5:
            return True
        return False


class AIPlayer:
```

```
    def __init__(self, color, time_limit, max_depth=3):
        self.color = color
        self.time_limit = time_limit
        self.max_depth = max_depth
        self.start_time = None

    def make_move(self, board):
        self.start_time = time.time()
        best_move = None
        try:
            best_move = self.alpha_beta_search(board, self.max_depth)
        except TimeoutError:
            pass
        return best_move

    def alpha_beta_search(self, board, depth):
        def max_value(board, alpha, beta, depth):
            if time.time() - self.start_time >= self.time_limit:
                raise TimeoutError()
            if depth == 0 or board.is_terminal():
                return board.evaluate(self.color)
            v = float('-inf')
            moves = board.get_possible_moves(self.color)
            if not moves:
                return board.evaluate(self.color)
            for move in moves:
                board.make_move(move)
                v = max(v, min_value(board, alpha, beta, depth - 1))
                board.undo_move(move)
                if v >= beta:
                    return v
                alpha = max(alpha, v)
            return v

        def min_value(board, alpha, beta, depth):
            if time.time() - self.start_time >= self.time_limit:
                raise TimeoutError()
            if depth == 0 or board.is_terminal():
                return board.evaluate(self.color)
            v = float('inf')
            opponent_color = 'black' if self.color == 'white' else 'white'
            moves = board.get_possible_moves(opponent_color)
            if not moves:
                return board.evaluate(self.color)
            for move in moves:
                board.make_move(move)
```

```
            v = min(v, max_value(board, alpha, beta, depth - 1))
            board.undo_move(move)
            if v <= alpha:
                return v
            beta = min(beta, v)
        return v

    best_score = float('-inf')
    beta = float('inf')
    best_move = None
    moves = board.get_possible_moves(self.color)
    for move in moves:
        if time.time() - self.start_time >= self.time_limit:
            raise TimeoutError()
        board.make_move(move)
        v = min_value(board, best_score, beta, depth - 1)
        board.undo_move(move)
        if v > best_score:
            best_score = v
            best_move = move
    return best_move


class HalmaBoard:
  def __init__(self, root, size=8, seconds_limit=10, white_player='human',
black_player='human'):
    self.size = size
    self.cell_size = 50
    self.canvas = tk.Canvas(root, width=self.size * self.cell_size, height=self.size *
self.cell_size)
    self.canvas.pack()

    self.seconds_limit = seconds_limit
    self.time_remaining = self.seconds_limit
    self.timer_id = None

    self.create_grid()

    self.pieces = {}
    self.selected_piece = None
    self.valid_moves = []
    self.current_turn = 'white'
    self.white_score = 0
    self.black_score = 0

    self.white_goal = [(7, 7), (7, 6), (7, 5), (6, 7), (6, 6)]
```

```
        self.black_goal = [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1)]

        self.initialize_pieces()
        self.status_bar = tk.Label(root, text="White's Turn | White Score: 0 | Black Score:
0 | Time Left: 10s",
                                   font=("Arial", 14))
        self.status_bar.pack()

        self.is_human = {
            'white': white_player == 'human',
            'black': black_player == 'human'
        }

        if not self.is_human['white']:
            self.white_player = AIPlayer('white', self.seconds_limit, max_depth=3)
        else:
            self.white_player = None

        if not self.is_human['black']:
            self.black_player = AIPlayer('black', self.seconds_limit, max_depth=3)
        else:
            self.black_player = None

        self.canvas.bind("<Button-1>", self.on_click)

        self.start_timer()

        if not self.is_human[self.current_turn]:
            self.canvas.after(100, self.computer_move)

    def create_grid(self):
        for row in range(self.size):
            for col in range(self.size):
                x1 = col * self.cell_size
                y1 = row * self.cell_size
                x2 = x1 + self.cell_size
                y2 = y1 + self.cell_size
                color = 'beige' if (row + col) % 2 == 0 else 'lightgray'
                self.canvas.create_rectangle(x1, y1, x2, y2, fill=color)

    def place_piece(self, row, col, color):
        position = Position(row, col, color, self.canvas, self.cell_size)
        self.pieces[(row, col)] = position

    def initialize_pieces(self):
        white_positions = [
```

```
            (0, 0), (0, 1), (0, 2), (0, 3),
            (1, 0), (1, 1), (1, 2),
            (2, 0), (2, 1),
            (3, 0)
        ]
        black_positions = [
            (7, 7), (7, 6), (7, 5), (7, 4),
            (6, 7), (6, 6), (6, 5),
            (5, 7), (5, 6),
            (4, 7)
        ]
        for row, col in white_positions:
            self.place_piece(row, col, 'white')
        for row, col in black_positions:
            self.place_piece(row, col, 'black')

    def highlight_moves(self, row, col):
        self.clear_highlights()

        possible_moves = self.get_possible_moves(row, col)
        valid_moves = []

        for move in possible_moves:
            r, c = move[2], move[3]
            x1 = c * self.cell_size
            y1 = r * self.cell_size
            x2 = x1 + self.cell_size
            y2 = y1 + self.cell_size
            move_id = self.canvas.create_rectangle(x1, y1, x2, y2, outline='green',
width=2)
            valid_moves.append((r, c, move_id, move))

        self.valid_moves = valid_moves

    def get_possible_moves(self, row, col):
        board_state = self.create_board_state()
        return board_state.get_piece_moves(row, col)

    def clear_highlights(self):
        for move in self.valid_moves:
            self.canvas.delete(move[2])
        self.valid_moves = []

    def on_click(self, event):
        if not self.is_human[self.current_turn]:
            return
```

```
        row, col = event.y // self.cell_size, event.x // self.cell_size

        if (row, col) in self.pieces and self.pieces[(row, col)].color == self.current_turn:
            if self.selected_piece:
                self.selected_piece.clear_outline()
            self.selected_piece = self.pieces[(row, col)]
            self.selected_piece.set_outline("red", 3)
            self.highlight_moves(row, col)
        elif self.selected_piece:
            for move in self.valid_moves:
                if (row, col) == (move[0], move[1]):
                    position = self.selected_piece
                    self.apply_move(move[3])
                    break

    def move_piece(self, position, to_pos, animate=False, callback=None):
        from_pos = (position.row, position.col)

        def after_animation():
            del self.pieces[from_pos]
            self.pieces[to_pos] = position
            position.clear_outline()
            self.clear_highlights()
            self.update_score()
            self.check_for_win()
            if callback:
                callback()

        if animate:
            position.move_to(to_pos[0], to_pos[1], duration=500,
callback=after_animation)
        else:
            position.move_to(to_pos[0], to_pos[1])
            after_animation()

    def apply_move(self, move):
        from_row, from_col, to_row, to_col, path = move
        position = self.pieces[(from_row, from_col)]
        is_human_player = self.is_human[self.current_turn]

        def after_move():
            if not self.check_for_win():
                self.switch_turn()

        self.move_piece(position, (to_row, to_col), animate=True, callback=after_move)
```

```
    def switch_turn(self):
        if self.timer_id:
            self.canvas.after_cancel(self.timer_id)

        self.current_turn = 'black' if self.current_turn == 'white' else 'white'
        self.time_remaining = self.seconds_limit
        self.update_status()
        self.start_timer()

        if not self.is_human[self.current_turn]:
            self.canvas.after(100, self.computer_move)

    def start_timer(self):
        if self.time_remaining > 0:
            self.time_remaining -= 1
            self.update_status()
            self.timer_id = self.canvas.after(1000, self.start_timer)
        else:
            messagebox.showinfo("Time's up!", f"{self.current_turn.capitalize()} ran out of
time!")
            if not self.check_for_win():
                self.switch_turn()

    def update_score(self):
        self.white_score = sum(1 for pos in self.white_goal if pos in self.pieces and
self.pieces[pos].color == 'white')
        self.black_score = sum(1 for pos in self.black_goal if pos in self.pieces and
self.pieces[pos].color == 'black')

    def update_status(self):
        self.status_bar.config(text=f"{self.current_turn.capitalize()}'s Turn | White Score:
{self.white_score} | "
                                    f"Black Score: {self.black_score} | Time Left:
{self.time_remaining}s")
        self.status_bar.update_idletasks()

    def check_for_win(self):
        self.update_score()
        if self.white_score >= 5:
            messagebox.showinfo("Game Over", f"White wins with a score of
{self.white_score}!")
            self.canvas.unbind("<Button-1>")
            self.stop_timer()
            return True
        elif self.black_score >= 5:
```

```
            messagebox.showinfo("Game Over", f"Black wins with a score of
{self.black_score}!")
            self.canvas.unbind("<Button-1>")
            self.stop_timer()
            return True
        return False

    def stop_timer(self):
        if self.timer_id:
            self.canvas.after_cancel(self.timer_id)
            self.timer_id = None

    def create_board_state(self):
        board_state = BoardState(self.size, white_goal=self.white_goal,
black_goal=self.black_goal)
        board_state.board = [['' for _ in range(self.size)] for _ in range(self.size)]
        for (row, col), position in self.pieces.items():
            board_state.board[row][col] = position.color
        return board_state

    def computer_move(self):
        board_state = self.create_board_state()
        ai_player = self.white_player if self.current_turn == 'white' else self.black_player

        threading.Thread(target=self.run_ai_move, args=(ai_player, board_state)).start()

    def run_ai_move(self, ai_player, board_state):
        best_move = ai_player.make_move(board_state)
        self.canvas.after(0, self.apply_ai_move, best_move)

    def apply_ai_move(self, best_move):
        if best_move:
            self.apply_move(best_move)
        else:
            self.switch_turn()


root = tk.Tk()
root.title("Halma")
game_board = HalmaBoard(root, seconds_limit=15, white_player='human',
black_player='ai')
root.mainloop()
```