```
import tkinter as tk
import random
import tkinter.messagebox as messagebox
import time
from typing import List, Tuple

BOARD_SIZE = 8
SQUARE_SIZE = 60
window = tk.Tk()
window.withdraw()
menu = tk.Menu(window)

board = [[None] * 8 for _ in range(8)]
selected_piece = None
ai_difficulty = "Easy"
game_over = False
PLAYER_COLOR = "white"
AI_COLOR = "black"
PLAYER_KING_COLOR = "green"
AI_KING_COLOR = "yellow"

NORMAL_VALUE = 10  # 普通棋子的价值
KING_VALUE = 20  # 升王棋子的价值
EDGE_VALUE = 5  # 边缘棋子的价值
CENTER_VALUE = 3  # 中心棋子的价值

def draw_board():
    # Clear the canvas
    canvas.delete("all")
    # Draw the squares of the checkerboard
    for row in range(8):
        for col in range(8):
            # Calculate the coordinates of the square based on the row and
column numbers
            x1, y1 = col * 60, row * 60
            x2, y2 = x1 + 60, y1 + 60
            # Fill the square with gray if it's a black square
            if (row + col) % 2 == 0:
                canvas.create_rectangle(x1, y1, x2, y2, fill="#8B4513")
            # If there's a piece on the square, draw the piece with its color
and shape
            piece = board[row][col]
            if piece is not None:
                # Draw a circle with the color of the piece
                color = "white" if piece == "white" else "black" if piece ==
"black" else "blue" if piece == "green" else "red"
                canvas.create_oval(x1 + 5, y1 + 5, x2 - 5, y2 - 5, fill=color)
            # If the square is a selected piece or a valid move, mark it with a
yellow border
            if selected_piece is not None and (row, col) in
get_valid_moves(board, selected_piece[0], selected_piece[1]):
                canvas.create_rectangle(x1 + 2, y1 + 2, x2 - 2, y2 - 2,
outline="black", width=4)
    # Update the canvas
    canvas.update()
```

```
def is_valid_move(game_board, start_row, start_col, end_row, end_col):
    piece_color = game_board[start_row][start_col]

    # Define a list to store all possible colors of the opponent's pieces
    enemy_piece_colors = []
    if piece_color in ["white", "green"]:
        enemy_piece_colors = ["black", "yellow"]
    elif piece_color in ["black", "yellow"]:
        enemy_piece_colors = ["white", "green"]

    # If the target position is not within the range of the board, or is not
empty, return False
    if not (0 <= end_row < 8 and 0 <= end_col < 8) or game_board[end_row]
[end_col] is not None:
        return False

    # If the absolute difference between the target position and the starting
position is 1, it means it is a normal move
    if abs(end_row - start_row) == 1:
        # If the piece is a king, it can move one square in any direction
        if piece_color in ["green", "blue"]:
            # If the target position is the opponent's piece, return False
            if game_board[end_row][end_col] in enemy_piece_colors:
                return False
            else:
                return True
        # If the piece is a normal player piece, it can only move one square up
        elif piece_color == "white" and end_row < start_row:
            # If the target position is the opponent's piece, return False
            if game_board[end_row][end_col] in enemy_piece_colors:
                return False
            else:
                return True
        # If the piece is a normal AI piece, it can only move one square down
        elif piece_color == "black" and end_row > start_row:
            # If the target position is the opponent's piece, return False
            if game_board[end_row][end_col] in enemy_piece_colors:
                return False
            else:
                return True

    # If the absolute difference between the target position and the starting
position is 2, it means it is a jump move
    elif abs(end_row - start_row) == 2:
        # Calculate the row and column numbers of the middle position of the
jump
        mid_row = (start_row + end_row) // 2
        mid_col = (start_col + end_col) // 2

        # If there is an opponent's piece in the middle position, it can jump
over it
        if game_board[mid_row][mid_col] is not None and game_board[mid_row]
[mid_col] in enemy_piece_colors:
            return True

        # If there is own piece in the middle position, it cannot jump over it
```

```
        # Define a list to store all possible colors of own pieces
        own_piece_colors = [piece_color, piece_color.upper()]
        if piece_color in ["white", "green"]:
            own_piece_colors.append("green")
        elif piece_color in ["black", "yellow"]:
            own_piece_colors.append("yellow")

        if game_board[mid_row][mid_col] is not None and game_board[mid_row]
[mid_col] in own_piece_colors:
            return False

    # In all other cases, return False
    return False

# Returns all legal moves for a piece, parameters are the board, row number, and
column number
# Function to get all possible legal moves for a given piece
def get_valid_moves(board, row, col):
    piece = board[row][col]
    moves = []

    # Function to check and add valid moves
    def check_and_add_moves(directions):
        for dx, dy in directions:
            new_row, new_col = row + dx, col + dy
            if is_valid_move(board, row, col, new_row, new_col):
                moves.append((new_row, new_col))
            new_row, new_col = row + 2 * dx, col + 2 * dy
            if is_valid_move(board, row, col, new_row, new_col):
                moves.append((new_row, new_col))

    # King pieces can move or jump in any direction
    if piece in ["green", "yellow"]:
        check_and_add_moves([(-1, -1), (-1, 1), (1, -1), (1, 1)])

    # Regular player pieces can only move or jump upwards
    elif piece == "white":
        check_and_add_moves([(-1, -1), (-1, 1)])

    # Regular AI pieces can only move or jump downwards
    elif piece == "black":
        check_and_add_moves([(1, -1), (1, 1)])

    # Return the list of possible moves
    return moves


def can_jump(row, col):
    # Check if a piece has a chance to jump
    piece = board[row][col]

    if piece is None:
        return False

    directions = [(-1, -1), (-1, 1), (1, -1), (1, 1)]
    for dx, dy in directions:
        move_row, move_col = row + dx, col + dy
        jump_row, jump_col = row + 2 * dx, col + 2 * dy
```

```
        if is_on_board(jump_row, jump_col) and board[jump_row][jump_col] is
None:
            # If the cell after the jump is empty
            if (piece == PLAYER_COLOR or piece == "white king") and move_row <
row and board[move_row][move_col] in [AI_COLOR, "black king"]:
                return True
            elif (piece == AI_COLOR or piece == "black king") and move_row > row
and board[move_row][move_col] in [PLAYER_COLOR, "white king"]:
                return True

    return False

def is_on_board(x, y):
    # Check if a coordinate is on the board
    return 0 <= x < 8 and 0 <= y < 8

def make_king(x, y):
    # Check if a piece can be promoted to king, and update its status
    piece = board[x][y]

    if piece is None:
        return False

    if piece == PLAYER_COLOR and row == 0:
        board[row][col] = "white king"
        return True

    elif piece == AI_COLOR and row == 7:
        board[row][col] = "black king"
        return True

    return False

# Move a piece, with parameters for the starting position and the target
position
def make_move(board, start_row, start_col, end_row, end_col, is_crowning=False):
    # Get the color of the piece
    piece_color = board[start_row][start_col]

    # Check if it's a jump move
    is_jump_move = abs(end_row - start_row) > 1

    if is_jump_move:
        # Calculate the position of the captured piece
        middle_row = (start_row + end_row) // 2
        middle_col = (start_col + end_col) // 2

        # Remove the captured piece
        board[middle_row][middle_col] = None

    # Move the piece to the target position
    board[end_row][end_col] = piece_color

    # Check if it needs to be crowned
    is_player_piece = piece_color == PLAYER_COLOR
    is_ai_piece = piece_color == AI_COLOR
```

```
    reached_opponent_end = (is_player_piece and end_row == 0) or (is_ai_piece
and end_row == 8 - 1)

    if reached_opponent_end:
        board[end_row][end_col] = PLAYER_KING_COLOR if is_player_piece else
AI_KING_COLOR

    # Clear the starting position
    board[start_row][start_col] = None

    return board

def get_all_moves(board: List[List[str]], color: str) -> List[Tuple[int, int,
int, int]]:
    return [(row, col, *move)
            for row in range(8)
            for col in range(8)
            if board[row][col] in {color, color.upper()}
            for move in get_valid_moves(board, row, col)]

def evaluate(board: List[List[str]], color: str) -> int:
    color_map = {PLAYER_KING_COLOR: 7, AI_KING_COLOR: -7, PLAYER_COLOR: 1,
AI_COLOR: -1}
    return sum(color_map.get(board[row][col], 0) for row in range(8) for col in
range(8) if board[row][col])

def deep_copy(obj):
    return [deep_copy(item) for item in obj] if isinstance(obj, list) else obj

def alpha_beta_search(board, depth, alpha, beta, is_ai_turn):
    if is_game_over() or depth == 0:
        return evaluate(board, AI_COLOR), None

    moves = get_all_moves(board, AI_COLOR if is_ai_turn else PLAYER_COLOR)
    if not moves:
        return (-float('inf'), None) if is_ai_turn else (float('inf'), None)

    best_score = -float('inf') if is_ai_turn else float('inf')
    best_move = None

    for move in moves:
        new_board = deep_copy(board)
        make_move(new_board, *move)

        score, _ = alpha_beta_search(new_board, depth - 1, alpha, beta, not
is_ai_turn)

        if is_ai_turn and score > best_score:
            best_score = score
            best_move = move
            alpha = max(alpha, best_score)

        if not is_ai_turn and score < best_score:
            best_score = score
            best_move = move
            beta = min(beta, best_score)

        if beta <= alpha:
```

```
            break

    return best_score, best_move

# AI移动，根据ai_difficulty的值来调用不同的搜索算法
def ai_move():
    time.sleep(0.2)
    best_move = None

    # 获取AI移动后的行列号和是否升王
    row = None
    col = None
    is_king = None

    # 如果ai_difficulty是简单，就随机选择一个走法
    if ai_difficulty == "Easy":
        moves = []
        for row in range(8):
            for col in range(8):
                if board[row][col] == AI_COLOR or board[row][col] ==
AI_KING_COLOR:
                    valid_moves = get_valid_moves(board, row, col)
                    moves.extend([(row, col, move[0], move[1]) for move in
valid_moves])
        if moves:
            # 检查所有可能的移动，以确定是否有必吃的情况
            must_jump = any(abs(move[0] - move[2]) > 1 for move in moves)
            if must_jump:
                moves = [move for move in moves if abs(move[0] - move[2]) > 1]
            random_move = random.choice(moves)
            make_move(board, random_move[0], random_move[1], random_move[2],
random_move[3])
            draw_board()

    elif ai_difficulty == "Medium":
        _, best_move = alpha_beta_search(board, 2, -float('inf'), float('inf'),
True)
        if best_move:
            make_move(board, best_move[0], best_move[1], best_move[2],
best_move[3])
            draw_board()

    elif ai_difficulty == "Hard":
        _, best_move = alpha_beta_search(board, 4, -float('inf'), float('inf'),
True)
        if best_move:
            make_move(board, best_move[0], best_move[1], best_move[2],
best_move[3])
            draw_board()

    if best_move:
        row = best_move[2]
        col = best_move[3]
        is_king = make_king(row, col)

    # 这里检查AI是否可以继续跳
    # 检查是否可以继续跳
    while best_move and not is_king and can_jump(row, col):
```

```
        best_move = alpha_beta_search(board, 4, -float('inf'), float('inf'),
True)
        if best_move:
            make_move(board, best_move[0], best_move[1], best_move[2],
best_move[3])
            draw_board()
            row = best_move[2]
            col = best_move[3]
            is_king = make_king(row, col)

    if is_game_over():
        show_game_over_message()
        return True

    return False



def set_difficulty(difficulty):
    # 设置AI难度
    global ai_difficulty
    ai_difficulty = difficulty


# 定义一个变量，用来存储当前选中的棋子的有效移动
valid_moves = []


# 处理鼠标点击事件
def handle_click(event):
    # 获取鼠标点击的位置
    x = event.x
    y = event.y

    row = y // 60
    col = x // 60

    # 使用global关键字，声明要使用全局变量
    global selected_piece
    global valid_moves

    # 检查game_over是否为True，如果是，就直接返回
    if game_over:
        return
    # 如果鼠标点击的位置不在棋盘上，就直接返回
    if not is_on_board(row, col):
        return

    # 如果没有选中棋子，就尝试选中一个玩家的棋子
    if selected_piece is None:
        # 如果方格有玩家的棋子，就选中它，并获取它的有效移动
        if board[row][col] in (PLAYER_COLOR, PLAYER_KING_COLOR):
            selected_piece = (row, col)
            valid_moves = get_valid_moves(board, row, col)  # 获取有效移动

            # 检查玩家是否有必须跳过对方棋子的情况
            must_jump = False
            for r in range(8):
```

```
                for c in range(8):
                    if board[r][c] in (PLAYER_COLOR, PLAYER_KING_COLOR):
                        moves = get_valid_moves(board, r, c)
                        for move in moves:
                            if abs(move[0] - r) > 1:
                                must_jump = True
                                break

            # 如果有，就只允许玩家选择可以跳过对方棋子的走法，并弹出提示信息
            if must_jump:
                valid_moves = [move for move in valid_moves if abs(move[0] -
row) > 1]


    # 如果已经选中棋子，就尝试移动它
    else:
        # 如果方格是有效的移动目标，就移动棋子，并取消选中
        if (row, col) in valid_moves:
            is_king = make_move(board, selected_piece[0], selected_piece[1],
row, col)
            draw_board()

            # 检查玩家是否可以继续跳跃，如果可以，就不要让AI移动，而是让玩家继续选择目标位置
            # 只有当移动的距离大于1时，才表示吃掉了对方的棋子，才需要判断是否能继续跳跃
            if not is_king and abs(row - selected_piece[0]) > 1 and
can_jump(row, col):
                selected_piece = (row, col)
                valid_moves = get_valid_moves(board, row, col)
                valid_moves = [move for move in valid_moves if abs(move[0] -
row) > 1]
                tk.messagebox.showinfo("提示", "你可以继续跳跃。")
            else:
                selected_piece = None
                # 让AI移动
                ai_move()


        # 如果方格是另一个玩家的棋子，就取消之前的选中，并选中这个棋子
        elif board[row][col] in (PLAYER_COLOR, PLAYER_KING_COLOR):
            selected_piece = (row, col)
            valid_moves = get_valid_moves(board, row, col)  # 获取有效移动

            # 检查玩家是否有必须跳过对方棋子的情况
            must_jump = False
            for r in range(8):
                for c in range(8):
                    if board[r][c] in (PLAYER_COLOR, PLAYER_KING_COLOR):
                        moves = get_valid_moves(board, r, c)
                        for move in moves:
                            if abs(move[0] - r) > 1:
                                must_jump = True
                                break

            # 如果有，就只允许玩家选择可以跳过对方棋子的走法，并弹出提示信息
            if must_jump:
                valid_moves = [move for move in valid_moves if abs(move[0] -
row) > 1]
                tk.messagebox.showinfo("提示", "有必吃，请移动该棋子。")
```

```
            # 如果方格是其他情况，就取消选中
            else:
                selected_piece = None

        # 重新画出棋盘
        draw_board()


# 判断游戏是否结束，返回True或False
def is_game_over():
    # 初始化两个变量，用来存储玩家和AI的棋子数量
    player_pieces = 0
    ai_pieces = 0

    # 遍历棋盘，统计棋子数量
    for row in range(8):
        for col in range(8):
            # 如果是玩家的棋子，就增加玩家的棋子数量
            if board[row][col] in (PLAYER_COLOR, PLAYER_KING_COLOR):
                player_pieces += 1

            # 如果是AI的棋子，就增加AI的棋子数量
            elif board[row][col] in (AI_COLOR, AI_KING_COLOR):
                ai_pieces += 1

    # 如果玩家或者AI的棋子数量为0，就返回True，表示游戏结束
    if player_pieces == 0 or ai_pieces == 0:
        return True

    # 如果玩家或者AI没有有效的走法，就返回True，表示游戏结束
    for row in range(8):
        for col in range(8):
            # 如果是玩家的棋子，并且有有效的走法，就返回False，表示游戏未结束
            if board[row][col] in (PLAYER_COLOR, PLAYER_KING_COLOR) and
get_valid_moves(board, row, col):
                return False

            # 如果是AI的棋子，并且有有效的走法，就返回False，表示游戏未结束
            elif board[row][col] in (AI_COLOR, AI_KING_COLOR) and
get_valid_moves(board, row, col):
                return False

    # 如果一方棋子设法抓住了对方的国王，它会立即加冕为国王，随后游戏结束
    for row in range(8):
        for col in range(8):
            # 如果是玩家的普通棋子，并且可以跳过对方的国王，就返回True，表示游戏结束
            if board[row][col] == PLAYER_COLOR and is_valid_move(board, row,
col, row - 2, col - 2) and board[row - 1][
                col - 1] == AI_KING_COLOR:
                return True
            if board[row][col] == PLAYER_COLOR and is_valid_move(board, row,
col, row - 2, col + 2) and board[row - 1][
                col + 1] == AI_KING_COLOR:
                return True

            # 如果是AI的普通棋子，并且可以跳过对方的国王，就返回True，表示游戏结束
```

```
            if board[row][col] == AI_COLOR and is_valid_move(board, row, col,
row + 2, col - 2) and board[row + 1][
                col - 1] == PLAYER_KING_COLOR:
                return True
            if board[row][col] == AI_COLOR and is_valid_move(board, row, col,
row + 2, col + 2) and board[row + 1][
                col + 1] == PLAYER_KING_COLOR:
                return True

    # 如果以上都不满足，就返回True，表示游戏结束
    return True


def show_rules():
    # 显示规则说明
    tk.messagebox.showinfo("规则说明", "这是一个国际64格跳棋游戏。\n"
                                       "游戏规则如下：\n"
                                       "1. 每人用一种颜色棋子占一个角。\n"
                                       "2. 棋子可以在有直线连接的相邻六个方向移动或跳过其他
棋子。\n"
                                       "3. 谁先把正对面的阵地全部占领，谁就取得胜利。\n"
                                       "4. 如果一方棋子设法抓住了对方的国王，它会立即加冕为
国王，随后游戏结束。\n"
                                       "5. 国王是在自己的棋子到达对面的最后一行时才产生的。
\n"
                                       "6. 国王可以在任意方向移动或跳过其他棋子。\n"
                                       "祝你玩得开心！")


def restart_game():
    # 使用global关键字，声明要使用全局变量
    global board
    global game_over
    # 重置棋盘状态
    board = [[None] * 8 for _ in range(8)]
    for row in range(3):
        for col in range(8):
            if (row + col) % 2 == 1:
                board[row][col] = AI_COLOR

    for row in range(5, 8):
        for col in range(8):
            if (row + col) % 2 == 1:
                board[row][col] = PLAYER_COLOR

    # 设置game_over为False
    game_over = False

    # 重新绘制棋盘
    draw_board()

    # 清除游戏结束信息
    canvas.delete("all")
    # 重新绘制棋盘
    draw_board()

def show_game_over_message():
    # 显示游戏结束信息
```

```
    player_pieces = 0
    ai_pieces = 0
    for row in range(8):
        for col in range(8):
            if board[row][col] == PLAYER_COLOR or board[row][col] == 'white
king':
                player_pieces += 1
            elif board[row][col] == AI_COLOR or board[row][col] == 'black king':
                ai_pieces += 1

    if player_pieces == 0:
        message = 'AI赢了!'
    else:
        message = '你赢了!'
    # 创建一个新的顶级窗口
    top = tk.Toplevel()
    top.title('游戏结束')

    # 在顶级窗口中添加标签显示游戏结束的消息
    label = tk.Label(top, text=message, font=('Arial', 24))
    label.pack(padx=20, pady=20)

    # 定义关闭窗口的函数
    def close_window():
        top.destroy()
        window.destroy()

    # 在顶级窗口中添加一个按钮用于关闭窗口
    button = tk.Button(top, text='关闭窗口', command=close_window)
    button.pack(pady=10)
    # 在顶级窗口中添加一个按钮用于重新开始游戏

    def game_close_restart():
        top.destroy()
        restart_game()
    restart_button = tk.Button(top, text='重新开始游戏',
command=game_close_restart)
    restart_button.pack(pady=10)
    # # 使用messagebox显示游戏结束信息
    # messagebox.showinfo('游戏结束', message)

    # canvas.create_text(8 * 60 / 2, 8 * 60 / 2, text=message, fill='yellow',
    #                    font=('Arial', 24))
    window.quit()  # 终止主循环
    window.deiconify()
    window.mainloop()

    # 设置game_over为True
    global game_over
    game_over = True

    canvas.create_text(8 * 60 / 2, 8 * 60 / 2, text=message, fill='yellow',
                       font=('Arial', 24))
    window.quit()  # 终止主循环
    window.deiconify()
    window.mainloop()
```

```
window = tk.Tk()
window.title("Checkers")

canvas = tk.Canvas(window, width=8 * 60, height=8 * 60)
canvas.pack()

# 初始化棋盘状态
for row in range(3):
    for col in range(8):
        if (row + col) % 2 == 1:
            board[row][col] = AI_COLOR

for row in range(5, 8):
    for col in range(8):
        if (row + col) % 2 == 1:
            board[row][col] = PLAYER_COLOR

# 绑定事件
canvas.bind("<Button-1>", handle_click)

# 显示菜单
menu = tk.Menu(window)
window.config(menu=menu)

difficulty_menu = tk.Menu(menu)
# 添加菜单项
rules_menu = tk.Menu(menu)

menu.add_cascade(label="规则说明", menu=rules_menu)
rules_menu.add_command(label="查看规则", command=show_rules)

menu.add_cascade(label="难度选择", menu=difficulty_menu)
difficulty_menu.add_command(label="简单", command=lambda: set_difficulty("Easy"))
difficulty_menu.add_command(label="中等", command=lambda:
set_difficulty("Medium"))
difficulty_menu.add_command(label="困难", command=lambda: set_difficulty("Hard"))
menu.add_command(label="重开游戏", command=restart_game)

# 绘制棋盘
draw_board()

window.mainloop()
```