Project Report

Artificial Intelligence

[CSE3705]

The Halma Game

By

Meka Jeyadev

210274

Burugupalli Sai Chaitanya

210288

Department of Computer Science and Engineering
School of Engineering and Technology
BML Munjal University

November 2023

Declaration by the Candidates

We hereby declare that the project entitled "**The Halma Game**" has been carried out to fulfill the partial requirements for completion of the course **Artificial Intelligence** offered in the 5th Semester of the Bachelor of Technology (B.Tech) program in the Department of Computer Science and Engineering during AY-2023-24 (odd semester). This experimental work has been carried out by us and submitted to the course instructor *Dr. Soharab Hossain Shaikh***.** Due acknowledgments have been made in the text of the project to all other materials used. This project has been prepared in full compliance with the requirements and constraints of the prescribed curriculum.

MEKA JEYADEV

BURUGUPALLI SAI CHAITANYA

Place: BML Munjal University

Date: 18th November 2023

Contents

1. Introduction & Problem Statement

Introduction:

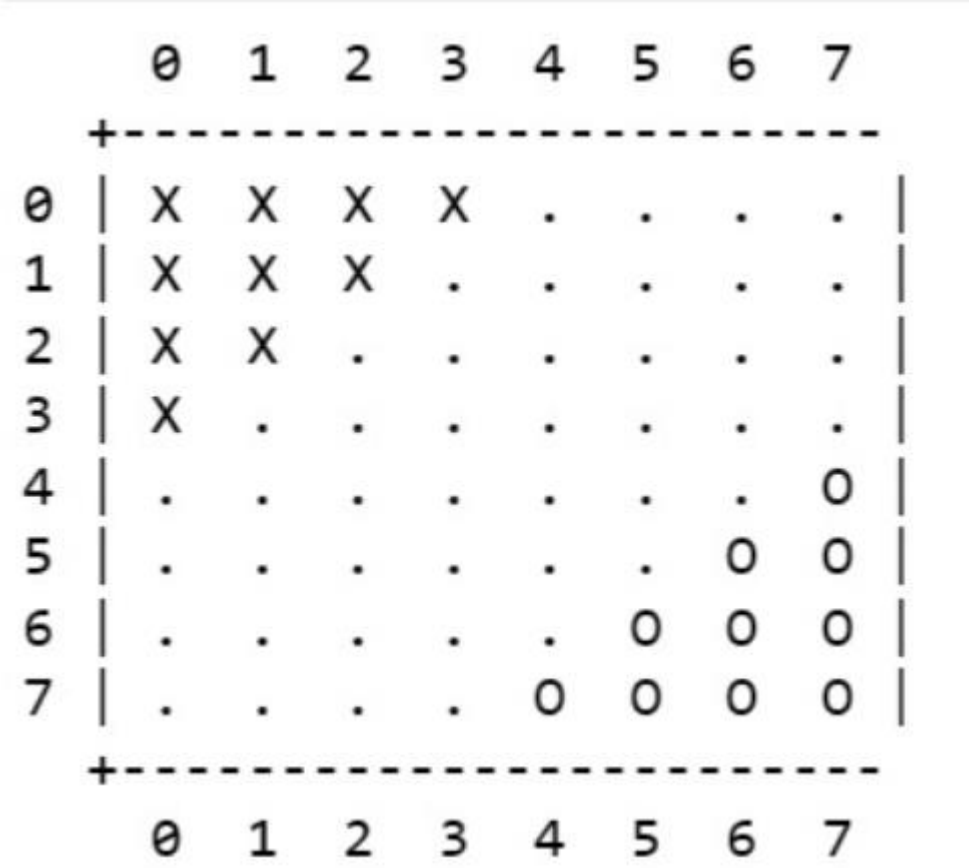
Halma is a classic strategic board game that challenges players to defeat their opponents in a quest to move their pieces from one corner of the board to the opposite corner. This project presents the development of a Halma game application using Python and Tkinter, a graphical user interface toolkit. The primary objective is to create an engaging and visually appealing platform for players to enjoy the game, featuring both a player-versus-player mode and an artificial intelligence (AI) opponent.

Problem Statement:

The development of the Halma game application involves two crucial components: implementing an agent function through a table-driven approach and incorporating intelligence into the agent function using a minimax search algorithm with alpha-beta pruning. The first part requires designing a comprehensive table-driven approach to dictate the AI's moves, while the second part involves creating a more sophisticated AI that can make intelligent decisions by considering potential future moves through a minimax search algorithm.

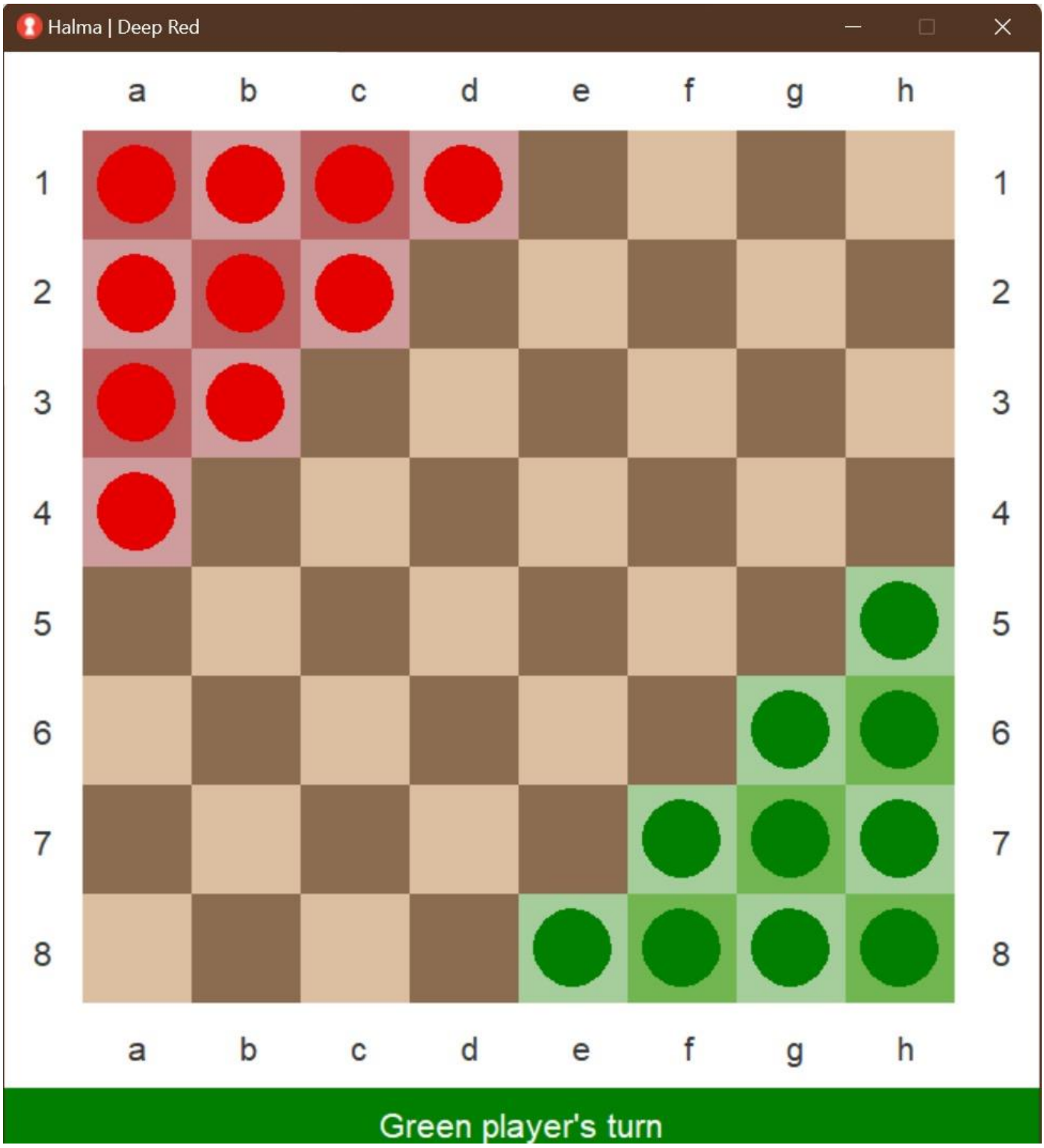
a. Table-Driven Approach :

The challenge in the table-driven approach lies in crafting an effective set of rules and strategies for the AI to follow. The AI must be capable of analysing the current state of the game, identifying possible moves, and making decisions based on pre-defined rules. The goal is to create an AI opponent that can provide a reasonable challenge to players, even if it lacks the depth of strategic thinking that the minimax algorithm can offer.



b. Minimax Search Algorithm with Alpha-Beta Pruning :

The second part of the project introduces a more intelligent approach to the AI opponent by implementing a minimax search algorithm with alpha-beta pruning. The challenge here is to develop an algorithm that explores the game tree efficiently, evaluating potential moves while minimizing unnecessary computations. The AI should strive for optimal moves, considering the game's rules and constraints, and ultimately aim to defeat the opponent through strategic decision-making.


Halma | Deep Red
a b c d e f g h
1 2 3 4 5 6 7 8
Green player's turn

Project Outcome:

1. **Fully Functional Halma Game Application:**

   The primary achievement of this project is the successful development of a fully functional Halma game application. The application provides a user-friendly interface that allows players to interact with the game effortlessly. The graphical representation of the game board and pieces enhances the overall gaming experience.

2. **Player Versus AI Modes:**

   The Halma game application, crafted using Python and Tkinter, features an immersive Player versus AI (PvAI) mode, offering a compelling solo gaming experience. In this mode, a single player engages in strategic gameplay against an intelligent AI opponent.

3. **Utilization of Minimax Algorithm for AI Decision-Making:**

   The AI opponent in the game employs the minimax algorithm, a strategic decision-making technique widely used in board games. This algorithm allows the AI to explore possible future moves by evaluating the potential outcomes of different game states. By considering the consequences of each move, the AI aims to make optimal decisions, enhancing the level of challenge for players engaging with the AI opponent.

4. **Clear Feedback on User Interactions:**

   The application provides clear and intuitive feedback to users during gameplay. This includes visual cues for valid moves, highlighting selected pieces, and displaying the progression of the game. The feedback mechanism ensures that players can easily understand the current state of the game and make informed decisions.

5. **Display of Relevant Game Statistics:**

   To enhance the gaming experience, the application includes a feature that displays relevant game statistics. This may include the number of moves made by each player, the time elapsed during the game, or any other pertinent information. Providing such statistics not only adds depth to the gaming experience but also allows players to track their progress and performance over multiple sessions.

2. Methodology

2.1 Table Driven

Percepts (Inputs)	**Actions (Outputs)**
Current state of the game board	Display the current state of the board
User input for Player 1's move	Process and validate the move
Randomly generated AI move	Make the AI move on the board
Win condition for Player 1	Display "Player 1 wins!" message
Win condition for Player 2	Display "Player 2 wins!" message
Lose condition (after 30 moves)	Display "Player 1 loses!" message
Invalid move input	Display "Invalid move. Try again." message

Percept's:

Current State of the Game Board:

Perceiving the arrangement of pieces on the game board for both Player 1 (X) and Player 2 (O). This includes the current positions of all pieces on the board.

User Input for Player 1's Move:

Receiving coordinates (x, y) entered by the human player, indicating the starting position and target position of their move.

Randomly Generated AI Move:

The AI generates random moves for Player 2 according to predefined rules and regulations, simulating the decision-making process of an AI opponent.

Actions (Outputs):

1. **Display Current State of the Board:**

   Print the current state of the game board, visually indicating the positions of Player 1's and Player 2's pieces. This action provides players with a clear representation of the ongoing game.

2. **Process and Validate Player 1's Move:**

   Check the validity of the entered move for Player 1 (human). If the move is valid, update the board accordingly. If the move is invalid, request a new move from the player. This action ensures that the game adheres to the rules and that players make legal moves.

3. **Make the AI Move on the Board:**

   The AI generates a random move for Player 2 based on predefined rules and updates the game board accordingly. This action simulates the decision-making process of an AI opponent, adding a challenging element to the game.

4. **Display "Player 1 Wins!":**

   If all of Player 1's pieces reach Player 2's starting area, display a message indicating that Player 1 wins. This action recognizes the winning condition for Player 1 and communicates the

game outcome to the player.

5. **Display "Player 2 Wins!":**

   If all of Player 2's pieces reach Player 1's starting area, display a message indicating that Player 2 wins. This action recognizes the winning condition for Player 2 and communicates the game outcome to the player.

6. **Display "Player 1 Loses!":**

   If the game reaches 30 moves without a win, display a message indicating that Player 1 loses. This action establishes a condition for determining when Player 1 loses the game.

7. **Display "Invalid Move. Try Again.":**

   If the user input for Player 1's move is invalid, display a message prompting the user to try again. This action provides feedback to the player, guiding them to make valid moves and ensuring a smooth gaming experience.

2.2 Intelligent Agent:

The intelligence of the agent in this Halma implementation comes from the use of a Min-Max search algorithm with alpha-beta pruning. Let's break down the key aspects of the strategy:

1. Minimax Search:

- The minimax algorithm is a decision-making algorithm commonly used in two-player turn-based games, such as Halma.
- The algorithm explores the game tree by considering possible moves for both the maximizing player (computer) and the minimizing player (opponent).
- It assigns utility values to terminal states (winning or losing positions) and uses these values to evaluate non-terminal states.

2. Alpha-Beta Pruning:

- It maintains two values, alpha and beta, representing the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of, respectively.
- Nodes that do not affect the final decision (i.e., will not change the outcome of the game) are pruned, reducing the overall search space.

3. Evaluation Function:

- The utility function utility_distance is used to evaluate non-terminal states. It calculates a heuristic value based on the distances of the pieces to their respective goal positions.
- The heuristic aims to guide the computer player to make moves that bring its pieces closer to its goal while hindering the opponent's progress.

4. Look-Ahead (Ply Depth):

- The algorithm looks ahead in the game tree up to a certain depth (self.ply_depth in the code) to make informed decisions.
- A deeper look-ahead allows the computer player to consider more potential future moves, but it also increases the computational cost.

5. Time Limit:

- There is a time limit (self.t_limit) for each turn, ensuring that the computer's move computation does not take an excessive amount of time.
- If the search exceeds the time limit, the algorithm returns the best move found so far.

6. Heuristic Evaluation for Piece Movement:

- The heuristic used in the utility_distance function considers the distances of the computer player's pieces to the opponent's goal and vice versa. The goal is to maximize the distance for the opponent's pieces while minimizing it for the computer player's pieces.

3. Implementation - Technology Stack

The chosen technology stack for the Halma game implementation reflects a well-rounded selection of tools and libraries to achieve a comprehensive and functional application. Let's further elaborate on the significance of each component in the stack:

1. **Graphical User Interface (GUI) Library - Tkinter:**

   Tkinter is a widely used GUI library for Python, and its selection for this project highlights its simplicity and integration capabilities. Being a built-in library, Tkinter provides a convenient way to create a user-friendly interface for the Halma game, allowing players to interact with the application seamlessly.

2. **Game Logic and Algorithms - Custom Classes and Functions:**

   The use of custom classes (Board, Tile, and Halma) signifies a well-structured and object-oriented approach to modeling the game elements. This promotes code organization, reusability, and clarity. The various functions within the Halma class encapsulate the game logic, handling crucial aspects such as tile clicks, move execution, utility calculation, and other essential game-related tasks.

3. **Artificial Intelligence (AI) Algorithm - Minimax Algorithm with Alpha-Beta Pruning:**

   The implementation of the minimax algorithm with alpha-beta pruning introduces a level of intelligence to the computer player, enabling strategic decision-making. This choice of AI algorithm aligns with the requirements of turn-based games like Halma, where considering future moves is crucial for optimal decision outcomes.

4. **Other Standard Libraries - sys and time:**

   The inclusion of the sys library indicates the use of standard output functionality, potentially for debugging or providing informative messages to the console. The time library's utilization for tracking move computation time is essential for enforcing time limits on the AI's decision-making process, preventing excessive delays during gameplay.

5. **External Resources - Game Icon:**

   The loading of the game's icon from a local file path using wm_iconbitmap in the Tkinter Board class contributes to the overall aesthetic appeal of the application. This attention to detail enhances the user experience by incorporating visual elements beyond the core gameplay.

6. **Development Environment:**

   The project's flexibility in not specifying a particular Integrated Development Environment (IDE) or code editor allows developers to use their preferred tools. This adaptability makes the code accessible and easily deployable in various development environments.

4. Conclusions

The Halma game implementation demonstrates the integration of several key components to create a playable desktop application. The technology stack includes Python as the programming language, Tkinter for the graphical user interface, and custom classes and functions for game logic. The implementation features an AI opponent using the minimax algorithm with alpha-beta pruning to make intelligent moves.

Key Takeaways:

1. **Minimax Algorithm with Alpha-Beta Pruning:**

   The intelligence of the computer player is derived from the minimax algorithm, which explores potential moves and evaluates their utility. Alpha-beta pruning enhances the efficiency of the search, making it suitable for practical use.

2. **Tkinter for GUI:**

   The graphical user interface is built using Tkinter, the standard GUI library for Python. It provides the necessary tools to create windows, labels, canvases, and other GUI elements.

3. **Game Logic:**

   The custom classes (`Board`, `Tile`, and `Halma`) encapsulate the game logic, including tile movements, player turns, and the execution of the computer player's moves.

4. **Heuristic Evaluation:**

   The implementation includes a heuristic evaluation function (`utility_distance`) to guide the computer player's decisions based on the distances of pieces to their respective goal positions.

5. **User Interaction:**

   The game allows user interaction through the GUI, with the ability to click on tiles to make moves.

6. **Time Limit:**

   A time limit is set for each turn to ensure that the AI's move computation does not take an excessive amount of time, providing a balance between computational cost and responsiveness.

The Halma game implementation showcases the synergy of AI algorithms, GUI development, and game logic using Python and Tkinter. It serves as a practical example of how these technologies can be combined to create an interactive and intelligent desktop application. The use of the minimax algorithm with alpha-beta pruning enhances the computational efficiency of the AI opponent's decision-making process.

References

Python Documentation:

- Python Software Foundation. (n.d.). Python Documentation. Retrieved from https://docs.python.org/

Tkinter Documentation:

- Python Software Foundation. (n.d.). Tkinter - Python Interface to Tcl/Tk. Retrieved from https://docs.python.org/3/library/tkinter.html

AI Algorithms and Minimax:

- Russell, S. J., & Norvig, P. (2010). Artificial Intelligence: A Modern Approach. Prentice Hall. ISBN-13: 978-0136042594.

Appendix:

Part-1: Implement the agent function by using the table-driven approach.

In [4]:
```
import random

# Constants
EMPTY, PLAYER1, PLAYER2 = 0, 1, 2
BOARD_SIZE = 8

# Define player yards as external tables
PLAYER1_YARD = [(0, 0), (1, 0), (0, 1), (0, 2), (2, 1), (2, 0), (1, 1), (3, 0), (1, 2), (0, 3)]
PLAYER2_YARD = [(7, 7), (6, 7), (7, 6), (7, 5), (5, 6), (5, 7), (6, 6), (4, 7), (6, 5), (7, 4)]

# Initialize the board using external tables
board = [[EMPTY for _ in range(BOARD_SIZE)] for _ in range(BOARD_SIZE)]
for x, y in PLAYER1_YARD:
    board[x][y] = PLAYER1
for x, y in PLAYER2_YARD:
    board[x][y] = PLAYER2

# Define move directions as external table
MOVES = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]

# Define rules and configurations as external tables
WIN_CONDITION_PLAYER1 = lambda b: all(b[x][y] == PLAYER2 for x, y in PLAYER1_YARD)
WIN_CONDITION_PLAYER2 = lambda b: all(b[x][y] == PLAYER1 for x, y in PLAYER2_YARD)
LOSE_CONDITION = lambda t: t >= 30

# Function to print the board with axis labels
def print_board(board):
    print("   " + " ".join(str(i) for i in range(BOARD_SIZE)))
    print("  " + " ".join(["-" for _ in range(2 * BOARD_SIZE + 1)]))
    for i, row in enumerate(board):
        print(f"{i} |", end=" ")
        print(" ".join(str(cell) if cell == EMPTY else 'X' if cell == PLAYER1 else 'Y' for cell in row))
    print()

# Function to check if a move is valid using external tables
def is_valid_move(x1, y1, x2, y2, player, board):
    if 0 <= x1 < BOARD_SIZE and 0 <= y1 < BOARD_SIZE and 0 <= x2 < BOARD_SIZE and 0 <= y2 < BOARD_SIZE:
        if board[x1][y1] == player and board[x2][y2] == EMPTY:
            return True
    return False

# Function to make a move using external tables
def make_move(x1, y1, x2, y2, player, board):
    if is_valid_move(x1, y1, x2, y2, player, board):
        board[x2][y2] = player
        board[x1][y1] = EMPTY
```

```python
# Function for AI player's move (one step either straight or diagonally) using external tables
def ai_move(player, board):
    while True:
        x1, y1 = random.randint(0, BOARD_SIZE - 1), random.randint(0, BOARD_SIZE - 1)
        random.shuffle(MOVES)  # Randomize move directions
        for dx, dy in MOVES:
            x2, y2 = x1 + dx, y1 + dy
            if is_valid_move(x1, y1, x2, y2, player, board):
                make_move(x1, y1, x2, y2, player, board)
                return

# Main game loop
turns = 0
while True:
    print_board(board)
    turns += 1

    # Player 1 (human) move
    print("Player 1's turn (Human)")
    x1, y1 = map(int, input("Enter starting position (x y): ").split())
    x2, y2 = map(int, input("Enter target position (x y): ").split())
    if is_valid_move(x1, y1, x2, y2, PLAYER1, board):
        make_move(x1, y1, x2, y2, PLAYER1, board)
    else:
        print("Invalid move. Try again.")
        turns -= 1
        continue

    # Check win condition for Player 1
    if WIN_CONDITION_PLAYER1(board):
        print("Player 1 wins!")
        break

    # AI player 2 move
    print("Player 2's turn (AI)")
    ai_move(PLAYER2, board)

    # Check win condition for Player 2
    if WIN_CONDITION_PLAYER2(board):
        print("Player 2 wins!")
        break

    # Check lose condition
    if LOSE_CONDITION(turns):
        print("Player 1 loses!")
        break
```

Output:

```
   0 1 2 3 4 5 6 7
 - - - - - - - - - - - - - - - -
0 | X X X X 0 0 0 0
1 | X X X 0 0 0 0 0
2 | X X 0 0 0 0 0 0
3 | X 0 0 0 0 0 0 0
4 | 0 0 0 0 0 0 0 Y
5 | 0 0 0 0 0 0 Y Y
6 | 0 0 0 0 0 Y Y Y
7 | 0 0 0 0 Y Y Y Y

Player 1's turn (Human)
Enter starting position (x y): 3 0
Enter target position (x y): 4 1
Player 2's turn (AI)
   0 1 2 3 4 5 6 7
 - - - - - - - - - - - - - - - -
0 | X X X X 0 0 0 0
1 | X X X 0 0 0 0 0
2 | X X 0 0 0 0 0 0
3 | 0 0 0 0 0 0 0 0
4 | 0 X 0 0 0 0 0 Y
5 | 0 0 0 0 0 Y 0 Y
6 | 0 0 0 0 0 Y Y Y
7 | 0 0 0 0 Y Y Y Y
```

```
  - - - - - - - - - - - - - - - -
0 | X X X X 0 0 0 0
1 | X X X 0 0 0 0 0
2 | X X 0 0 0 0 0 0
3 | 0 0 0 0 0 0 0 0
4 | 0 0 0 0 0 0 0 Y
5 | 0 0 X 0 Y Y 0 Y
6 | 0 0 0 0 0 0 Y Y
7 | 0 0 0 0 Y Y Y Y

Player 1's turn (Human)
Enter starting position (x y): 5 2
Enter target position (x y): 6 2
Player 2's turn (AI)
   0 1 2 3 4 5 6 7
  - - - - - - - - - - - - - - - -
0 | X X X X 0 0 0 0
1 | X X X 0 0 0 0 0
2 | X X 0 0 0 0 0 0
3 | 0 0 0 0 0 0 0 0
4 | 0 0 0 0 0 0 0 Y
5 | 0 0 0 0 Y Y Y 0
6 | 0 0 X 0 0 0 Y Y
7 | 0 0 0 0 Y Y Y Y

Player 1's turn (Human)
Enter starting position (x y): 1 2
Enter target position (x y): 3 2
Player 2's turn (AI)
   0 1 2 3 4 5 6 7
  - - - - - - - - - - - - - - - -
0 | X X X X 0 0 0 0
1 | X X 0 0 0 0 0 0
2 | X X 0 0 0 0 0 0
3 | 0 0 X 0 0 0 0 0
4 | 0 0 0 0 0 0 0 Y
5 | 0 0 0 0 Y Y Y 0
6 | 0 0 X 0 0 Y Y Y
7 | 0 0 0 0 Y Y 0 Y

Player 1's turn (Human)

Enter starting position (x y):
```

Part-2: Implement the agent function by using an intelligent technique.

Minimax Algorithm with Alpha-Beta Pruning

In [1]:
```
#board Standard Library imports
import tkinter as tk


class Board(tk.Tk):

    def __init__(self, init_board, *args, **kwargs):

        # Initialize parent tk class
        tk.Tk.__init__(self, *args, **kwargs)

        # Save metadata
        self.title("Halma | Deep Red")
        self.wm_iconbitmap("C:/Users/jeyad/OneDrive/Desktop/AI Project/deep_red.ico")
        self.resizable(False, False)
        self.configure(bg="#fff")

        # Save tracking variables
        self.tiles = {}
        self.board = init_board
        self.b_size = len(init_board)

        # Create column/row labels
        label_font = "Helvetica 16"
        label_bg = "#fff"
        label_fg = "#333"
        for i in range(self.b_size):

            row_label1 = tk.Label(self, text=i + 1, font=label_font,
                bg=label_bg, fg=label_fg)
            row_label1.grid(row=i + 1, column=0)

            row_label2 = tk.Label(self, text=i + 1, font=label_font,
                bg=label_bg, fg=label_fg)
            row_label2.grid(row=i + 1, column=self.b_size + 2)

            col_label1 = tk.Label(self, text=chr(i + 97), font=label_font,
                bg=label_bg, fg=label_fg)
            col_label1.grid(row=0, column=i + 1)

            col_label2 = tk.Label(self, text=chr(i + 97), font=label_font,
                bg=label_bg, fg=label_fg)
            col_label2.grid(row=self.b_size + 2, column=i + 1)

        # Create grid canvas
        self.canvas = tk.Canvas(self, width=550, height=550, bg="#fff",
            highlightthickness=0)
        self.canvas.grid(row=1, column=1,
            columnspan=self.b_size, rowspan=self.b_size)
```

```
        # Create status label
        self.status = tk.Label(self, anchor="c", font=(None, 16),
            bg="#212121", fg="#fff", text="Green player's turn")
        self.status.grid(row=self.b_size + 3, column=0,
            columnspan=self.b_size + 3, sticky="ewns")

        # Bind the drawing function and configure grid sizes
        self.canvas.bind("<Configure>", self.draw_tiles)
        self.columnconfigure(0, minsize=48)
        self.rowconfigure(0, minsize=48)
        self.columnconfigure(self.b_size + 2, minsize=48)
        self.rowconfigure(self.b_size + 2, minsize=48)
        self.rowconfigure(self.b_size + 3, minsize=48)

    # Public Methods #

    def add_click_handler(self, func):
        self.click_handler = func

    def set_status(self, text):
        self.status.configure(text=text)

    def set_status_color(self, color):
        self.status.configure(bg=color)

    def draw_tiles(self, event=None, board=None):

        if board is not None:
            self.board = board

        # Delete old rectangles and save properties
        self.canvas.delete("tile")
        cell_width = int(self.canvas.winfo_width() / self.b_size)
        cell_height = int(self.canvas.winfo_height() / self.b_size)
        border_size = 5

        # Recreate each rectangle
        for col in range(self.b_size):
            for row in range(self.b_size):

                board_tile = self.board[row][col]
                tile_color, outline_color = board_tile.get_tile_colors()

                # Calculate pixel positions
                x1 = col * cell_width + border_size / 2
                y1 = row * cell_height + border_size / 2
                x2 = (col + 1) * cell_width - border_size / 2
                y2 = (row + 1) * cell_height - border_size / 2
```

```
                # Render tile
                tile = self.canvas.create_rectangle(x1, y1, x2, y2,
                    tags="tile", width=border_size, fill=tile_color,
                    outline=outline_color)
                self.tiles[row, col] = tile
                self.canvas.tag_bind(tile, "<1>", lambda event, row=row,
                    col=col: self.click_handler(row, col))

        self.draw_pieces()

    def draw_pieces(self, board=None):

        if board is not None:
            self.board = board

        self.canvas.delete("piece")
        cell_width = int(self.canvas.winfo_width() / self.b_size)
        cell_height = int(self.canvas.winfo_height() / self.b_size)
        border_size = 20

        for col in range(self.b_size):
            for row in range(self.b_size):

                # Calculate pixel positions
                x1 = col * cell_width + border_size / 2
                y1 = row * cell_height + border_size / 2
                x2 = (col + 1) * cell_width - border_size / 2
                y2 = (row + 1) * cell_height - border_size / 2

                if self.board[row][col].piece == 2:
                    piece = self.canvas.create_oval(x1, y1, x2, y2,
                        tags="piece", width=0, fill="#E50000")
                elif self.board[row][col].piece == 1:
                    piece = self.canvas.create_oval(x1, y1, x2, y2,
                        tags="piece", width=0, fill="#007F00")
                else:
                    continue

                self.canvas.tag_bind(piece, "<1>", lambda event, row=row,
                    col=col: self.click_handler(row, col))

        self.update()
```

```
class Tile():

    # Goal constants
    T_NONE = 0
    T_GREEN = 1
    T_RED = 2

    # Piece constants
    P_NONE = 0
    P_GREEN = 1
    P_RED = 2

    # Outline constants
    O_NONE = 0
    O_SELECT = 1
    O_MOVED = 2

    def __init__(self, tile=0, piece=0, outline=0, row=0, col=0):
        self.tile = tile
        self.piece = piece
        self.outline = outline

        self.row = row
        self.col = col
        self.loc = (row, col)

    def get_tile_colors(self):

        # Find appropriate tile color
        tile_colors = [
            ("#8C6C50", "#DBBFA0"),  # Normal tiles
            ("#71b651", "#a6ce9d"),  # Red goal tiles
            ("#ba6262", "#ce9d9d")   # Green goal tiles
        ]
        tile_color = tile_colors[self.tile][(self.loc[0] + self.loc[1]) % 2]

        # Find appropriate outline color
        outline_colors = [
            tile_color,
            "yellow",  # TODO: Change
            "#1100BB"
        ]
        outline_color = outline_colors[self.outline]

        return tile_color, outline_color

    def __str__(self):
        return chr(self.loc[1] + 97) + str(self.loc[0] + 1)
```

```
    def __repr__(self):
        return chr(self.loc[1] + 97) + str(self.loc[0] + 1)
```

```
# halma Python Standard Library imports
import sys
import time
import math

# Custom module imports



class Halma():

    def __init__(self, b_size=8, t_limit=60, c_player=Tile.P_RED):

        # Create initial board
        board = [[None] * b_size for _ in range(b_size)]
        for row in range(b_size):
            for col in range(b_size):

                if row + col < 4:
                    element = Tile(2, 2, 0, row, col)
                elif row + col > 2 * (b_size - 3):
                    element = Tile(1, 1, 0, row, col)
                else:
                    element = Tile(0, 0, 0, row, col)

                board[row][col] = element

        # Save member variables
        self.b_size = b_size
        self.t_limit = t_limit
        self.c_player = c_player
        self.board_view = Board(board)
        self.board = board
        self.current_player = Tile.P_GREEN
        self.selected_tile = None
        self.valid_moves = []
        self.computing = False
        self.total_plies = 0

        self.ply_depth = 3
        self.ab_enabled = True

        self.r_goals = [t for row in board
                        for t in row if t.tile == Tile.T_RED]
        self.g_goals = [t for row in board
                        for t in row if t.tile == Tile.T_GREEN]
```

```
        self.board_view.set_status_color("#E50000" if
            self.current_player == Tile.P_RED else "#007F00")

        if self.c_player == self.current_player:
            self.execute_computer_move()

        self.board_view.add_click_handler(self.tile_clicked)
        self.board_view.draw_tiles(board=self.board)  # Refresh the board

        # Print initial program info
        print("Halma Solver Basic Information")
        print("==============================")
        print("AI opponent enabled:", "no" if self.c_player is None else "yes")
        print("A-B pruning enabled:", "yes" if self.ab_enabled else "no")
        print("Turn time limit:", self.t_limit)
        print("Max ply depth:", self.ply_depth)
        print()

        self.board_view.mainloop()  # Begin tkinter main loop

    def tile_clicked(self, row, col):

        if self.computing:  # Block clicks while computing
            return

        new_tile = self.board[row][col]

        # If we are selecting a friendly piece
        if new_tile.piece == self.current_player:

            self.outline_tiles(None)  # Reset outlines

            # Outline the new and valid move tiles
            new_tile.outline = Tile.O_MOVED
            self.valid_moves = self.get_moves_at_tile(new_tile,
                self.current_player)
            self.outline_tiles(self.valid_moves)

            # Update status and save the new tile
            self.board_view.set_status("Tile `" + str(new_tile) + "` selected")
            self.selected_tile = new_tile

            self.board_view.draw_tiles(board=self.board)  # Refresh the board

        # If we already had a piece selected and we are moving a piece
        elif self.selected_tile and new_tile in self.valid_moves:

            self.outline_tiles(None)  # Reset outlines
            self.move_piece(self.selected_tile, new_tile)  # Move the piece
```

```
        self.board_view.set_status_color("#E50000" if
            self.current_player == Tile.P_RED else "#007F00")

        if self.c_player == self.current_player:
            self.execute_computer_move()

        self.board_view.add_click_handler(self.tile_clicked)
        self.board_view.draw_tiles(board=self.board)  # Refresh the board

        # Print initial program info
        print("Halma Solver Basic Information")
        print("==============================")
        print("AI opponent enabled:", "no" if self.c_player is None else "yes")
        print("A-B pruning enabled:", "yes" if self.ab_enabled else "no")
        print("Turn time limit:", self.t_limit)
        print("Max ply depth:", self.ply_depth)
        print()

        self.board_view.mainloop()  # Begin tkinter main loop

    def tile_clicked(self, row, col):

        if self.computing:  # Block clicks while computing
            return

        new_tile = self.board[row][col]

        # If we are selecting a friendly piece
        if new_tile.piece == self.current_player:

            self.outline_tiles(None)  # Reset outlines

            # Outline the new and valid move tiles
            new_tile.outline = Tile.O_MOVED
            self.valid_moves = self.get_moves_at_tile(new_tile,
                self.current_player)
            self.outline_tiles(self.valid_moves)

            # Update status and save the new tile
            self.board_view.set_status("Tile `" + str(new_tile) + "` selected")
            self.selected_tile = new_tile

            self.board_view.draw_tiles(board=self.board)  # Refresh the board

        # If we already had a piece selected and we are moving a piece
        elif self.selected_tile and new_tile in self.valid_moves:

            self.outline_tiles(None)  # Reset outlines
            self.move_piece(self.selected_tile, new_tile)  # Move the piece
```

```
            # Update status and reset tracking variables
            self.selected_tile = None
            self.valid_moves = []
            self.current_player = (Tile.P_RED
                if self.current_player == Tile.P_GREEN else Tile.P_GREEN)

            self.board_view.draw_tiles(board=self.board)  # Refresh the board

            # If there is a winner to the game
            winner = self.find_winner()
            if winner:
                self.board_view.set_status("The " + ("green"
                    if winner == Tile.P_GREEN else "red") + " player has won!")
                self.current_player = None

            elif self.c_player is not None:
                self.execute_computer_move()

        else:
            self.board_view.set_status("Invalid move attempted")

    def minimax(self, depth, player_to_max, max_time, a=float("-inf"),
                b=float("inf"), maxing=True, prunes=0, boards=0):

        # Bottomed out base case
        if depth == 0 or self.find_winner() or time.time() > max_time:
            return self.utility_distance(player_to_max), None, prunes, boards

        # Setup initial variables and find moves
        best_move = None
        if maxing:
            best_val = float("-inf")
            moves = self.get_next_moves(player_to_max)
        else:
            best_val = float("inf")
            moves = self.get_next_moves((Tile.P_RED
                    if player_to_max == Tile.P_GREEN else Tile.P_GREEN))

        # For each move
        for move in moves:
            for to in move["to"]:

                # Bail out when we're out of time
                if time.time() > max_time:
                    return best_val, best_move, prunes, boards
```

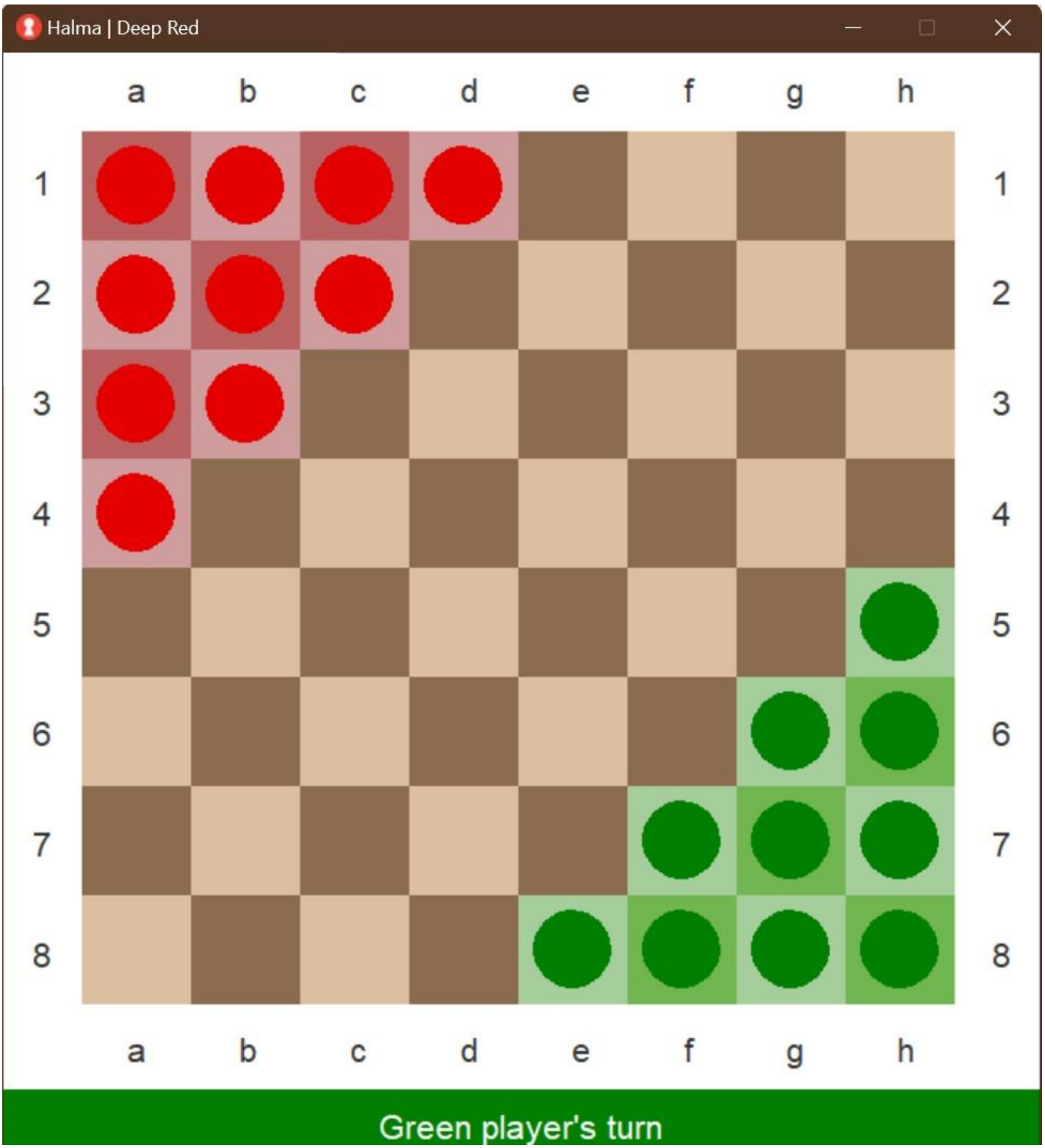
```
            # Move piece to the move outlined
            piece = move["from"].piece
            move["from"].piece = Tile.P_NONE
            to.piece = piece
            boards += 1

            # Recursively call self
            val, _, new_prunes, new_boards = self.minimax(depth - 1,
                player_to_max, max_time, a, b, not maxing, prunes, boards)
            prunes = new_prunes
            boards = new_boards

            # Move the piece back
            to.piece = Tile.P_NONE
            move["from"].piece = piece

            if maxing and val > best_val:
                best_val = val
                best_move = (move["from"].loc, to.loc)
                a = max(a, val)

            if not maxing and val < best_val:
                best_val = val
                best_move = (move["from"].loc, to.loc)
                b = min(b, val)

            if self.ab_enabled and b <= a:
                return best_val, best_move, prunes + 1, boards

    return best_val, best_move, prunes, boards

def execute_computer_move(self):

    # Print out search information
    current_turn = (self.total_plies // 2) + 1
    print("Turn", current_turn, "Computation")
    print("=================" + ("=" * len(str(current_turn))))
    print("Executing search ...", end=" ")
    sys.stdout.flush()

    # self.board_view.set_status("Computing next move...")
    self.computing = True
    self.board_view.update()
    max_time = time.time() + self.t_limit

    # Execute minimax search
    start = time.time()
    _, move, prunes, boards = self.minimax(self.ply_depth,
        self.c_player, max_time)
    end = time.time()
```

```
        # Print search result stats
        print("complete")
        print("Time to compute:", round(end - start, 4))
        print("Total boards generated:", boards)
        print("Total prune events:", prunes)

        # Move the resulting piece
        self.outline_tiles(None)  # Reset outlines
        move_from = self.board[move[0][0]][move[0][1]]
        move_to = self.board[move[1][0]][move[1][1]]
        self.move_piece(move_from, move_to)

        self.board_view.draw_tiles(board=self.board)  # Refresh the board

        winner = self.find_winner()
        if winner:
            self.board_view.set_status("The " + ("green"
                if winner == Tile.P_GREEN else "red") + " player has won!")
            self.board_view.set_status_color("#212121")
            self.current_player = None
            self.current_player = None

            print()
            print("Final Stats")
            print("===========")
            print("Final winner:", "green"
                if winner == Tile.P_GREEN else "red")
            print("Total # of plies:", self.total_plies)

        else:  # Toggle the current player
            self.current_player = (Tile.P_RED
                if self.current_player == Tile.P_GREEN else Tile.P_GREEN)

        self.computing = False
        print()

    def get_next_moves(self, player=1):

        moves = []  # All possible moves
        for col in range(self.b_size):
            for row in range(self.b_size):

                curr_tile = self.board[row][col]

                # Skip board elements that are not the current player
                if curr_tile.piece != player:
                    continue
```

```
            move = {
                "from": curr_tile,
                "to": self.get_moves_at_tile(curr_tile, player)
            }
            moves.append(move)

    return moves

def get_moves_at_tile(self, tile, player, moves=None, adj=True):

    if moves is None:
        moves = []

    row = tile.loc[0]
    col = tile.loc[1]

    # List of valid tile types to move to
    valid_tiles = [Tile.T_NONE, Tile.T_GREEN, Tile.T_RED]
    if tile.tile != player:
        valid_tiles.remove(player)  # Moving back into your own goal
    if tile.tile != Tile.T_NONE and tile.tile != player:
        valid_tiles.remove(Tile.T_NONE)  # Moving out of the enemy's goal

    # Find and save immediately adjacent moves
    for col_delta in range(-1, 2):
        for row_delta in range(-1, 2):

            # Check adjacent tiles

            new_row = row + row_delta
            new_col = col + col_delta

            # Skip checking degenerate values
            if ((new_row == row and new_col == col) or
                new_row < 0 or new_col < 0 or
                new_row >= self.b_size or new_col >= self.b_size):
                continue

            # Handle moves out of/in to goals
            new_tile = self.board[new_row][new_col]
            if new_tile.tile not in valid_tiles:
                continue

            if new_tile.piece == Tile.P_NONE:
                if adj:  # Don't consider adjacent on subsequent calls
                    moves.append(new_tile)
                continue
```

```
            # Check jump tiles

            new_row = new_row + row_delta
            new_col = new_col + col_delta

            # Skip checking degenerate values
            if (new_row < 0 or new_col < 0 or
                new_row >= self.b_size or new_col >= self.b_size):
                continue

            # Handle returning moves and moves out of/in to goals
            new_tile = self.board[new_row][new_col]
            if new_tile in moves or (new_tile.tile not in valid_tiles):
                continue

            if new_tile.piece == Tile.P_NONE:
                moves.insert(0, new_tile)  # Prioritize jumps
                self.get_moves_at_tile(new_tile, player, moves, False)

    return moves

def move_piece(self, from_tile, to_tile):

    # Handle trying to move a non-existant piece and moving into a piece
    if from_tile.piece == Tile.P_NONE or to_tile.piece != Tile.P_NONE:
        self.board_view.set_status("Invalid move")
        return

    # Move piece
    to_tile.piece = from_tile.piece
    from_tile.piece = Tile.P_NONE

    # Update outline
    to_tile.outline = Tile.O_MOVED
    from_tile.outline = Tile.O_MOVED

    self.total_plies += 1

    self.board_view.set_status_color("#007F00" if
        self.current_player == Tile.P_RED else "#E50000")
    self.board_view.set_status("Piece moved from `" + str(from_tile) +
        "` to `" + str(to_tile) + "`, " + ("green's" if
        self.current_player == Tile.P_RED else "red's") + " turn...")
```

```
    def find_winner(self):

        if all(g.piece == Tile.P_GREEN for g in self.r_goals):
            return Tile.P_GREEN
        elif all(g.piece == Tile.P_RED for g in self.g_goals):
            return Tile.P_RED
        else:
            return None

    def outline_tiles(self, tiles=[], outline_type=Tile.O_SELECT):

        if tiles is None:
            tiles = [j for i in self.board for j in i]
            outline_type = Tile.O_NONE

        for tile in tiles:
            tile.outline = outline_type

    def utility_distance(self, player):

        def point_distance(p0, p1):
            return math.sqrt((p1[0] - p0[0])**2 + (p1[1] - p0[1])**2)

        value = 0

        for col in range(self.b_size):
            for row in range(self.b_size):

                tile = self.board[row][col]

                if tile.piece == Tile.P_GREEN:
                    distances = [point_distance(tile.loc, g.loc) for g in
                                 self.r_goals if g.piece != Tile.P_GREEN]
                    value -= max(distances) if len(distances) else -50

                elif tile.piece == Tile.P_RED:
                    distances = [point_distance(tile.loc, g.loc) for g in
                                 self.g_goals if g.piece != Tile.P_RED]
                    value += max(distances) if len(distances) else -50

        if player == Tile.P_RED:
            value *= -1

        return value


if __name__ == "__main__":

    halma = Halma()
```


Halma | Deep Red
a b c d e f g h
1 2 3 4 5 6 7 8
Green player's turn