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.

1. 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. |

1. Demos

|  |  |
| --- | --- |
| Screenshot | Annotation |
|  | 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. |
|  | 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. |

1. 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() |