Amit bouton 207005323

Noam Churi 207652744

Ron Azar 314779448

Youtube link presentation: https://www.youtube.com/watch?v=qhlzDiCvEOo

Moodle link presentation: <https://moodle.ruppin.ac.il/pluginfile.php/183195/assignsubmission_file/submission_files/198544/presentation.rar?forcedownload=1>

Youtube link Gameplay: <https://youtu.be/9gozPqJcZg8>

Moodle link Gameplay: <https://moodle.ruppin.ac.il/pluginfile.php/183195/assignsubmission_file/submission_files/198544/gameplay.rar?forcedownload=1>

Git link: https://github.com/AmitBouton98/Checkers-Complex-Game

Readme : <https://moodle.ruppin.ac.il/pluginfile.php/183195/assignsubmission_file/submission_files/198544/readme.txt?forcedownload=1>

**תיאור כללי**

המשחק הוא גרסה מורחבת של דמקה הכולל לוח 12x12, סוס עם תנועות מיוחדות, ומכניקת הנחת קופסאות חסימה. המשחק כולל שני מצבי משחק:

1. שחקן מול שחקן - PlayerVsPlayer
2. שחקן מול מחשב – ComputerVsPlayer

**דרישות פונקציונליות**

1. **לוח משחק**:
   * גודל לוח: 12x12 משבצות
   * כל שחקן מתחיל עם 24 חיילים רגילים
   * לכל שחקן יש סוס אחד שניתן להניח בתחילת המשחק
2. **חוקים ותנועות**:
   * החיילים הרגילים נעים קדימה בלבד ואוכלים שעוברים את היריב
   * הסוס נע כמעת סוס בשחמט
   * אפשרות להניח קופסאות חסימה בלוח המשחק
3. **מצבי משחק**:
   * שחקן מול שחקן
   * שחקן מול מחשב
4. **זכייה**:
   * המשחק מסתיים כאשר אחד השחקנים משיג 3 נקודות, אוכל את כל חיילי היריב, או כאשר הזמן מסתיים והשחקן עם הכי הרבה נקודות זוכה

**תכנון ממשק המשתמש**

* **מסך ראשי**: מציג את האפשרויות לשים את הסוס של היריב
* **לוח המשחק**: מציג את הלוח, החיילים, הסוסים והקופסאות
* **פאנל צדדי**: מציג מידע על התור הנוכחי, מספר נקודות, זמן משחק ולשים קופסא

**בעיות ופתרונות**

במהלך יצירת המשחק היו מספר בעיות עיקריות:

1. **העדפת אכילה על פני תנועה**:
   * בתחילה המחשב לא העדיף לאכול חיילים של היריב אלא רק לנוע.
   * הפתרון היה שיפור הפונקציה minmax ושיפור פונקציית הניקוד.

**תיעוד האלגוריתמים**

האלגוריתמים המרכזיים בהם השתמשנו הם:

* **Minimax**:
  + מחפש את המהלך הטוב ביותר על ידי הערכת מהלכים אפשריים עד לעומק מסוים ( בחרנו 3 ).
  + שיפור על ידי הוספת Alpha-Beta Pruning להפחתת מספר המהלכים הנבדקים.
* **Alpha-Beta Pruning**:
  + משפר את אלגוריתם Minimax על ידי חיתוך מהלכים לא רלוונטיים.
* **פונקציית הערכה**:
  + מעריכה את מצב הלוח כדי לתת ציון לכל מהלך אפשרי.

בקוד יש שתי ישויות עיקריות:

* **Board**:
  + מכילה את כל הלוגיקה של הלוח, תנועות החיילים והסוסים, ובדיקות הצעדים האפשריים.
* **Piece**:
  + מייצגת חייל או סוס, כולל מיקום וצבע.

קוד :

import pygame  
import sys  
import time  
import copy  
  
# Initialize Pygame  
pygame.init()  
  
# Screen dimensions  
BOARD\_WIDTH, BOARD\_HEIGHT = 700, 700  
PANEL\_WIDTH = 400  
WIDTH, HEIGHT = BOARD\_WIDTH + PANEL\_WIDTH, BOARD\_HEIGHT  
ROWS, COLS = 12, 12  
SQUARE\_SIZE = BOARD\_HEIGHT // ROWS  
  
# Colors  
WHITE = (245, 245, 245)  
BLACK = (40, 40, 40)  
RED = (235, 64, 52)  
BLUE = (52, 152, 235)  
GREY = (128, 128, 128)  
GREEN = (34, 139, 34)  
YELLOW = (255, 255, 0)  
SPECIAL\_RED = (255, 105, 180) # Pink for red player knight  
SPECIAL\_BLUE = (135, 206, 250) # Light blue for blue player knight  
HIGHLIGHT = (173, 216, 230) # Light blue for highlighting valid cells  
LIGHT\_RED = (255, 182, 193)  
LIGHT\_BLUE = (173, 216, 230)  
  
# Screen setup  
WIN = pygame.display.set\_mode((WIDTH, HEIGHT))  
pygame.display.set\_caption('12x12 Checkers Game')  
  
  
class Piece:  
 PADDING = 15  
 OUTLINE = 2  
  
 def \_\_init\_\_(self, row, col, color, is\_king=False, is\_knight=False):  
 self.row = row  
 self.col = col  
 self.color = color  
 self.king = is\_king  
 self.knight = is\_knight  
 self.x = 0  
 self.y = 0  
 self.calc\_pos()  
  
 def calc\_pos(self):  
 self.x = SQUARE\_SIZE \* self.col + SQUARE\_SIZE // 2  
 self.y = SQUARE\_SIZE \* self.row + SQUARE\_SIZE // 2  
  
 def draw(self, win):  
 radius = SQUARE\_SIZE // 2 - self.PADDING  
 pygame.draw.circle(win, GREY, (self.x, self.y), radius + self.OUTLINE)  
 pygame.draw.circle(win, self.color, (self.x, self.y), radius)  
 if self.king:  
 pygame.draw.circle(win, YELLOW, (self.x, self.y), radius // 2)  
 if self.knight:  
 pygame.draw.circle(win, GREEN, (self.x, self.y), radius // 3)  
  
 def move(self, row, col):  
 self.row = row  
 self.col = col  
 self.calc\_pos()  
  
  
class Board:  
 def \_\_init\_\_(self):  
 self.board = []  
 self.selected\_piece = None  
 self.turn = RED  
 self.valid\_moves = {}  
 self.red\_captures = 0  
 self.blue\_captures = 0  
 self.red\_points = 0  
 self.blue\_points = 0  
 self.red\_knight\_set = False  
 self.blue\_knight\_set = False  
 self.setup\_phase = True  
 self.winner = None  
 self.red\_boxes = []  
 self.blue\_boxes = []  
 self.placing\_box = False  
 self.create\_board()  
 self.start\_time = time.time() # Start the timer  
  
 def draw\_squares(self, win):  
 win.fill(BLACK)  
 for row in range(ROWS):  
 for col in range(row % 2, COLS, 2):  
 pygame.draw.rect(win, WHITE, (row \* SQUARE\_SIZE, col \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))  
  
 def create\_board(self):  
 self.board = []  
 for row in range(ROWS):  
 self.board.append([])  
 for col in range(COLS):  
 if row % 2 == ((col + 1) % 2):  
 if row < 4:  
 self.board[row].append((Piece(row, col, RED), RED))  
 elif row > 7:  
 self.board[row].append((Piece(row, col, BLUE), BLUE))  
 else:  
 self.board[row].append((0, None))  
 else:  
 self.board[row].append((0, None))  
  
 self.red\_knight = Piece(-1, -1, SPECIAL\_RED, is\_knight=True)  
 self.blue\_knight = Piece(-1, -1, SPECIAL\_BLUE, is\_knight=True)  
  
 def draw(self, win):  
 self.draw\_squares(win)  
 for row in range(ROWS):  
 for col in range(COLS):  
 piece, color = self.board[row][col]  
 if isinstance(piece, int) and piece == 1:  
 if color == RED:  
 pygame.draw.rect(win, LIGHT\_RED, (col \* SQUARE\_SIZE, row \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))  
 else:  
 pygame.draw.rect(win, LIGHT\_BLUE, (col \* SQUARE\_SIZE, row \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))  
 elif piece != 0:  
 piece.draw(win)  
 if self.red\_knight\_set:  
 self.red\_knight.draw(win)  
 if self.blue\_knight\_set:  
 self.blue\_knight.draw(win)  
 if self.setup\_phase:  
 self.highlight\_valid\_cells(win)  
 self.draw\_valid\_moves(win)  
 self.draw\_panel(win)  
  
 def move(self, piece, row, col):  
 self.board[piece.row][piece.col] = (0, None)  
 if (piece.color == RED and row == ROWS - 1) or (piece.color == BLUE and row == 0):  
 if piece.color == RED:  
 self.red\_points += 1  
 else:  
 self.blue\_points += 1  
 piece.move(-1, -1)  
 self.check\_winner()  
 else:  
 self.board[row][col] = (piece, piece.color)  
 piece.move(row, col)  
  
 def get\_piece(self, row, col):  
 piece, color = self.board[row][col]  
 return piece  
  
 def draw\_valid\_moves(self, win):  
 for move in self.valid\_moves:  
 row, col = move  
 pygame.draw.circle(win, GREEN, (col \* SQUARE\_SIZE + SQUARE\_SIZE // 2, row \* SQUARE\_SIZE + SQUARE\_SIZE // 2), 15)  
  
 def highlight\_valid\_cells(self, win):  
 if self.turn == RED and not self.blue\_knight\_set:  
 for row in range(ROWS - 3, ROWS):  
 for col in range(COLS):  
 if self.board[row][col] == (0, None):  
 pygame.draw.rect(win, HIGHLIGHT, (col \* SQUARE\_SIZE, row \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))  
 elif self.turn == BLUE and not self.red\_knight\_set:  
 for row in range(3):  
 for col in range(COLS):  
 if self.board[row][col] == (0, None):  
 pygame.draw.rect(win, HIGHLIGHT, (col \* SQUARE\_SIZE, row \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))  
  
 def select(self, row, col):  
 if self.winner:  
 return  
  
 if self.setup\_phase:  
 if self.turn == RED and not self.blue\_knight\_set:  
 if row >= ROWS - 3 and self.board[row][col] == (0, None):  
 self.board[row][col] = (self.blue\_knight, BLUE)  
 self.blue\_knight.move(row, col)  
 self.blue\_knight\_set = True  
 self.turn = BLUE  
 self.computer\_place\_enemy\_knight()  
 return True  
 elif self.turn == BLUE and not self.red\_knight\_set:  
 if row < 3 and self.board[row][col] == (0, None):  
 self.board[row][col] = (self.red\_knight, RED)  
 self.red\_knight.move(row, col)  
 self.red\_knight\_set = True  
 self.turn = RED  
 self.setup\_phase = False  
 return True  
 return False  
  
 if self.placing\_box:  
 if self.board[row][col] == (0, None):  
 self.board[row][col] = (1, self.turn)  
 if self.turn == RED:  
 self.red\_boxes.append(((row, col), 6))  
 else:  
 self.blue\_boxes.append(((row, col), 6))  
 self.placing\_box = False  
 self.change\_turn()  
 return True  
  
 if self.selected\_piece:  
 result = self.\_move(row, col)  
 if not result:  
 self.selected\_piece = None  
 self.select(row, col)  
  
 piece = self.get\_piece(row, col)  
 if isinstance(piece, Piece) and (  
 piece.color == self.turn or (piece.color == SPECIAL\_RED and self.turn == RED) or (  
 piece.color == SPECIAL\_BLUE and self.turn == BLUE)):  
 self.selected\_piece = piece  
 self.valid\_moves = self.get\_valid\_moves(piece)  
 return True  
  
 return False  
  
 def computer\_place\_enemy\_knight(self):  
 if self.turn == BLUE and not self.red\_knight\_set:  
 for row in range(3):  
 for col in range(COLS):  
 if self.board[row][col] == (0, None):  
 self.board[row][col] = (self.red\_knight, RED)  
 self.red\_knight.move(row, col)  
 self.red\_knight\_set = True  
 self.turn = RED  
 self.setup\_phase = False  
 return  
  
 def \_move(self, row, col):  
 if self.selected\_piece and (row, col) in self.valid\_moves:  
 skipped = self.valid\_moves[(row, col)]  
 self.move(self.selected\_piece, row, col)  
 if skipped:  
 self.remove(skipped)  
 self.change\_turn()  
 self.check\_winner()  
 else:  
 return False  
  
 return True  
  
 def change\_turn(self):  
 self.valid\_moves = {}  
 self.selected\_piece = None  
 if self.turn == RED:  
 self.turn = BLUE  
 else:  
 self.turn = RED  
 self.update\_boxes()  
  
 def update\_boxes(self):  
 new\_red\_boxes = []  
 for position, turns in self.red\_boxes:  
 if turns > 1:  
 new\_red\_boxes.append((position, turns - 1))  
 else:  
 row, col = position  
 self.board[row][col] = (0, None)  
 self.red\_boxes = new\_red\_boxes  
  
 new\_blue\_boxes = []  
 for position, turns in self.blue\_boxes:  
 if turns > 1:  
 new\_blue\_boxes.append((position, turns - 1))  
 else:  
 row, col = position  
 self.board[row][col] = (0, None)  
 self.blue\_boxes = new\_blue\_boxes  
  
 def remove(self, pieces):  
 for piece in pieces:  
 if isinstance(piece, Piece):  
 if piece.color == RED or piece.color == SPECIAL\_RED:  
 self.blue\_captures += 1  
 else:  
 self.red\_captures += 1  
  
 try:  
 if self.selected\_piece.knight:  
 # Place the knight in the captured piece's position  
 self.board[piece.row][piece.col] = (self.selected\_piece, self.selected\_piece.color)  
 if piece.knight:  
 # If the captured piece is a knight, reset its position to -1, -1  
 piece.move(-1, -1)  
 else:  
 self.board[piece.row][piece.col] = (0, None)  
 if piece.knight:  
 # If the captured piece is a knight, reset its position to -1, -1  
 piece.move(-1, -1)  
 except AttributeError:  
 # If self.selected\_piece is not defined or doesn't have the knight attribute  
 self.board[piece.row][piece.col] = (0, None)  
 if piece.knight:  
 # If the captured piece is a knight, reset its position to -1, -1  
 piece.move(-1, -1)  
  
 def get\_valid\_moves(self, piece):  
 moves = {}  
 if piece.knight:  
 moves.update(self.\_knight\_moves(piece))  
 else:  
 row = piece.row  
 col = piece.col  
 if piece.color == BLUE or piece.color == SPECIAL\_BLUE:  
 moves.update(self.\_traverse\_forward(row - 1, max(row - 3, -1), -1, piece.color, col))  
 if piece.color == RED or piece.color == SPECIAL\_RED:  
 moves.update(self.\_traverse\_forward(row + 1, min(row + 3, ROWS), 1, piece.color, col))  
 return moves  
  
 def \_traverse\_forward(self, start, stop, step, color, col, skipped=[]):  
 moves = {}  
 last = []  
 knight\_color = SPECIAL\_BLUE if color == BLUE else SPECIAL\_RED  
 for r in range(start, stop, step):  
 if col < 0 or col >= COLS:  
 break  
  
 current, current\_color = self.board[r][col]  
  
 if current == 1:  
 break  
  
 if current == 0:  
 if skipped and not last:  
 break  
 elif skipped:  
 moves[(r, col)] = last + skipped  
 else:  
 moves[(r, col)] = last  
  
 if last:  
  
 if step == -1:  
 row = max(r - 3, -1)  
 else:  
 row = min(r + 3, ROWS)  
 moves.update(self.\_traverse\_forward(r + step, row, step, color, col, skipped=last))  
 break  
 elif isinstance(current, Piece) and (current.color == color or current.color == knight\_color):  
 break  
 elif isinstance(current, Piece) and (current.color != color or current.color != knight\_color) and not current.knight:  
 last = [current]  
 else:  
 break  
  
 return moves  
  
 def \_knight\_moves(self, piece):  
 knight\_color = SPECIAL\_BLUE if piece.color == BLUE else SPECIAL\_RED  
 moves = {}  
 directions = [  
 (2, 1), (1, 2), (-1, 2), (-2, 1),  
 (-2, -1), (-1, -2), (1, -2), (2, -1)  
 ]  
 for dr, dc in directions:  
 new\_row, new\_col = piece.row + dr, piece.col + dc  
 if 0 <= new\_row < ROWS and 0 <= new\_col < COLS:  
 target, target\_color = self.board[new\_row][new\_col]  
 if target == 1:  
 continue  
 if target == 0 or (isinstance(target, Piece) and target.color != piece.color and not (piece.color == SPECIAL\_RED and target.color == RED) and not (piece.color == SPECIAL\_BLUE and target.color == BLUE)):  
 if isinstance(target, Piece) and (target.color == piece.color or (piece.color == SPECIAL\_RED and target.color == RED) or (piece.color == SPECIAL\_BLUE and target.color == BLUE)):  
 continue  
 if isinstance(target, Piece) and target.color == knight\_color:  
 continue  
 moves[(new\_row, new\_col)] = [target] if isinstance(target, Piece) else []  
 return moves  
  
 def draw\_panel(self, win):  
 panel\_x = BOARD\_WIDTH  
 pygame.draw.rect(win, GREY, (panel\_x, 0, PANEL\_WIDTH, HEIGHT))  
 font = pygame.font.SysFont(None, 40)  
  
 turn\_text = font.render("Turn:", True, BLACK)  
 win.blit(turn\_text, (panel\_x + 20, 20))  
  
 color\_rect = pygame.Rect(panel\_x + 20, 70, PANEL\_WIDTH - 40, 50)  
 pygame.draw.rect(win, self.turn, color\_rect)  
  
 red\_captures\_text = font.render(f"Red Captures: {self.red\_captures}", True, BLACK)  
 blue\_captures\_text = font.render(f"Blue Captures: {self.blue\_captures}", True, BLACK)  
 win.blit(red\_captures\_text, (panel\_x + 20, 140))  
 win.blit(blue\_captures\_text, (panel\_x + 20, 200))  
  
 red\_points\_text = font.render(f"Red Points: {self.red\_points}", True, BLACK)  
 blue\_points\_text = font.render(f"Blue Points: {self.blue\_points}", True, BLACK)  
 win.blit(red\_points\_text, (panel\_x + 20, 260))  
 win.blit(blue\_points\_text, (panel\_x + 20, 320))  
  
 if self.setup\_phase:  
 if self.turn == RED:  
 setup\_text = font.render("Red, place Blue's knight", True, BLACK)  
 else:  
 setup\_text = font.render("Blue, place Red's knight", True, BLACK)  
 win.blit(setup\_text, (panel\_x + 20, 380))  
  
 if not self.winner:  
 elapsed\_time = time.time() - self.start\_time  
 remaining\_time = max(0, int(300 - elapsed\_time))  
 minutes = int(remaining\_time // 60)  
 seconds = int(remaining\_time % 60)  
 time\_text = font.render(f"Time: {minutes:02}:{seconds:02}", True, BLACK)  
 win.blit(time\_text, (panel\_x + 20, 440))  
 else:  
 winner\_text = font.render(f"{self.winner} Wins!", True, BLACK)  
 win.blit(winner\_text, (panel\_x + 20, 440))  
  
 if not self.winner:  
 if (self.turn == RED and not any(turns > 0 for \_, turns in self.red\_boxes)) or (self.turn == BLUE and not any(turns > 0 for \_, turns in self.blue\_boxes)):  
 box\_button = pygame.Rect(panel\_x + 20, 500, PANEL\_WIDTH - 40, 50)  
 pygame.draw.rect(win, GREEN, box\_button)  
 box\_text = font.render("Put Box", True, BLACK)  
 win.blit(box\_text, (panel\_x + 40, 510))  
 return box\_button  
 else:  
 winner\_text = font.render(f"{self.winner} Wins!", True, BLACK)  
 win.blit(winner\_text, (panel\_x + 20, 440))  
 reset\_button = pygame.Rect(panel\_x + 20, 500, PANEL\_WIDTH - 40, 50)  
 pygame.draw.rect(win, GREEN, reset\_button)  
 reset\_text = font.render("Reset", True, BLACK)  
 win.blit(reset\_text, (panel\_x + 40, 510))  
 return reset\_button  
  
 return None  
  
 def check\_winner(self):  
 if self.red\_points >= 3:  
 self.winner = "Red"  
 if self.blue\_points >= 3:  
 self.winner = "Blue"  
  
 red\_pieces = sum(1 for row in self.board for piece, color in row if isinstance(piece, Piece) and (color == RED or color == SPECIAL\_RED))  
 blue\_pieces = sum(1 for row in self.board for piece, color in row if isinstance(piece, Piece) and (color == BLUE or color == SPECIAL\_BLUE))  
  
 if red\_pieces == 0:  
 self.winner = "Blue"  
 if blue\_pieces == 0:  
 self.winner = "Red"  
  
 if red\_pieces == 1 and blue\_pieces == 1:  
 self.winner = "Tie"  
  
 elapsed\_time = time.time() - self.start\_time  
 if elapsed\_time > 300:  
 if self.red\_points > self.blue\_points:  
 self.winner = "Red"  
 elif self.blue\_points > self.red\_points:  
 self.winner = "Blue"  
 else:  
 if self.red\_captures > self.blue\_captures:  
 self.winner = "Red"  
 elif self.blue\_captures > self.red\_captures:  
 self.winner = "Blue"  
 else:  
 self.winner = "Tie"  
  
 return self.winner  
  
 def reset(self):  
 self.\_\_init\_\_()  
 self.start\_time = time.time()  
  
 def get\_all\_valid\_moves(self, color):  
 moves = []  
 knight\_color = SPECIAL\_BLUE if color == BLUE else SPECIAL\_RED  
 for row in self.board:  
 for piece, \_ in row:  
 if isinstance(piece, Piece) and (piece.color == color or piece.color == knight\_color):  
 valid\_moves = self.get\_valid\_moves(piece)  
 for move, skipped in valid\_moves.items():  
 moves.append((piece, move, skipped))  
 return moves  
  
 def is\_piece\_in\_danger(self, piece):  
 directions = [(-1, -1), (-1, 1), (1, -1), (1, 1)]  
 for dr, dc in directions:  
 new\_row, new\_col = piece.row + dr, piece.col + dc  
 if 0 <= new\_row < ROWS and 0 <= new\_col < COLS:  
 opponent\_piece, color = self.board[new\_row][new\_col]  
 if isinstance(opponent\_piece, Piece) and opponent\_piece.color != piece.color:  
 capture\_row, capture\_col = new\_row + dr, new\_col + dc  
 if 0 <= capture\_row < ROWS and 0 <= capture\_col < COLS:  
 target\_piece, \_ = self.board[capture\_row][capture\_col]  
 if target\_piece == 0:  
 return True  
 return False  
  
 def is\_future\_move\_safe(self, piece, move\_pos):  
 row, col = move\_pos  
 opponent\_color = RED if piece.color == BLUE else BLUE  
  
 if self.\_is\_threat\_from\_diagonals(row, col, opponent\_color):  
 return False  
  
 if self.\_is\_threat\_from\_knight(row, col, opponent\_color):  
 return False  
  
 return True  
  
 def \_is\_threat\_from\_diagonals(self, row, col, opponent\_color):  
 directions = [(-1, -1), (-1, 1), (1, -1), (1, 1)]  
  
 for dr, dc in directions:  
 opp\_row, opp\_col = row + dr, col + dc  
 if 0 <= opp\_row < ROWS and 0 <= opp\_col < COLS:  
 opponent\_piece, color = self.board[opp\_row][opp\_col]  
 if isinstance(opponent\_piece, Piece) and color == opponent\_color:  
 capture\_row, capture\_col = opp\_row + dr, opp\_col + dc  
 if 0 <= capture\_row < ROWS and 0 <= capture\_col < COLS:  
 target\_piece, \_ = self.board[capture\_row][capture\_col]  
 if target\_piece == 0:  
 return True  
  
 for dr, dc in directions:  
 opp\_row, opp\_col = row + 2 \* dr, col + 2 \* dc  
 if 0 <= opp\_row < ROWS and 0 <= opp\_col < COLS:  
 opponent\_piece, color = self.board[opp\_row][opp\_col]  
 if isinstance(opponent\_piece, Piece) and color == opponent\_color:  
 middle\_row, middle\_col = row + dr, col + dc  
 middle\_piece, \_ = self.board[middle\_row][middle\_col]  
 if isinstance(middle\_piece, Piece) and middle\_piece.color == opponent\_color:  
 return True  
  
 return False  
  
 def \_is\_threat\_from\_knight(self, row, col, opponent\_color):  
 knight\_directions = [  
 (2, 1), (1, 2), (-1, 2), (-2, 1),  
 (-2, -1), (-1, -2), (1, -2), (2, -1)  
 ]  
  
 for dr, dc in knight\_directions:  
 opp\_row, opp\_col = row + dr, col + dc  
 if 0 <= opp\_row < ROWS and 0 <= opp\_col < COLS:  
 opponent\_piece, color = self.board[opp\_row][opp\_col]  
 if isinstance(opponent\_piece, Piece) and color == opponent\_color and opponent\_piece.knight:  
 return True  
  
 return False  
  
 def is\_knight\_capture\_possible(self, piece):  
 directions = [  
 (2, 1), (1, 2), (-1, 2), (-2, 1),  
 (-2, -1), (-1, -2), (1, -2), (2, -1)  
 ]  
 for dr, dc in directions:  
 new\_row, new\_col = piece.row + dr, piece.col + dc  
 if 0 <= new\_row < ROWS and 0 <= new\_col < COLS:  
 target, target\_color = self.board[new\_row][new\_col]  
 if isinstance(target, Piece) and target.color != piece.color:  
 return True  
 return False  
  
 def should\_place\_box(self):  
 opponent\_color = RED if self.turn == BLUE else BLUE  
 row\_n\_minus\_1 = ROWS - 2 if opponent\_color == RED else 1  
 row\_n = ROWS - 1 if opponent\_color == RED else 0  
  
 for col in range(COLS):  
 opp\_piece, opp\_color = self.board[row\_n\_minus\_1][col]  
 my\_piece, my\_color = self.board[row\_n][col]  
 if isinstance(opp\_piece, Piece) and opp\_color == opponent\_color and my\_piece == 0:  
 if self.board[row\_n][col] == (0, None):  
 return row\_n, col  
  
 return None  
  
 def copy(self):  
 new\_board = copy.deepcopy(self)  
 return new\_board  
  
  
def evaluate(board):  
 score = board.blue\_points - board.red\_points  
  
 for row in board.board:  
 for piece, color in row:  
 if isinstance(piece, Piece):  
 if piece.color == BLUE:  
 score += 1  
 if piece.knight:  
 score += 5  
 if not board.is\_future\_move\_safe(piece, (piece.row, piece.col)):  
 score -= 3  
 if board.is\_knight\_capture\_possible(piece):  
 score += 5  
 valid\_moves = board.get\_valid\_moves(piece)  
 for move, skipped in valid\_moves.items():  
 if skipped:  
 score += 10  
 elif piece.color == RED:  
 score -= 1  
 if piece.knight:  
 score -= 5  
 if not board.is\_future\_move\_safe(piece, (piece.row, piece.col)):  
 score += 3  
 if board.is\_knight\_capture\_possible(piece):  
 score -= 5  
 valid\_moves = board.get\_valid\_moves(piece)  
 for move, skipped in valid\_moves.items():  
 if skipped:  
 score -= 10  
  
 return score  
  
  
def minmax(board, depth, alpha, beta, maximizing\_player):  
 if depth == 0 or board.winner:  
 return evaluate(board), None  
  
 valid\_moves = board.get\_all\_valid\_moves(board.turn)  
 safe\_moves = []  
  
 for move in valid\_moves:  
 piece, move\_pos, skipped = move  
 if board.is\_future\_move\_safe(piece, move\_pos):  
 safe\_moves.append(move)  
  
 moves\_to\_consider = safe\_moves if safe\_moves else valid\_moves  
  
 final\_moves = []  
 for move in valid\_moves:  
 piece, move\_pos, skipped = move  
 if skipped:  
 final\_moves.append(move)  
 moves\_to\_consider = final\_moves if final\_moves else moves\_to\_consider  
  
 if maximizing\_player:  
 max\_eval = float('-inf')  
 best\_move = None  
 for move in moves\_to\_consider:  
 temp\_board = board.copy()  
 piece, move\_pos, skipped = move  
 temp\_board.change\_turn()  
 eval, \_ = minmax(temp\_board, depth - 1, alpha, beta, False)  
 if eval > max\_eval:  
 max\_eval = eval  
 best\_move = move  
 alpha = max(alpha, eval)  
 if beta <= alpha:  
 break  
 return max\_eval, best\_move  
 else:  
 min\_eval = float('inf')  
 best\_move = None  
 for move in moves\_to\_consider:  
 temp\_board = board.copy()  
 piece, move\_pos, skipped = move  
 temp\_board.change\_turn()  
 eval, \_ = minmax(temp\_board, depth - 1, alpha, beta, True)  
 if eval < min\_eval:  
 min\_eval = eval  
 best\_move = move  
 beta = min(beta, eval)  
 if beta <= alpha:  
 break  
 return min\_eval, best\_move  
  
  
def place\_enemy\_knight(board):  
 if board.turn == RED and not board.blue\_knight\_set:  
 for row in range(ROWS - 3, ROWS):  
 for col in range(COLS):  
 if board.board[row][col] == (0, None):  
 board.board[row][col] = (board.blue\_knight, BLUE)  
 board.blue\_knight.move(row, col)  
 board.blue\_knight\_set = True  
 board.turn = BLUE  
 return  
 elif board.turn == BLUE and not board.red\_knight\_set:  
 for row in range(3):  
 for col in range(COLS):  
 if board.board[row][col] == (0, None):  
 board.board[row][col] = (board.red\_knight, RED)  
 board.red\_knight.move(row, col)  
 board.red\_knight\_set = True  
 board.turn = RED  
 board.setup\_phase = False  
 return  
  
  
def main():  
 run = True  
 clock = pygame.time.Clock()  
 board = Board()  
 action\_button = None  
  
 while run:  
 clock.tick(60)  
  
 board.check\_winner()  
  
 if board.turn == BLUE and board.setup\_phase and not board.blue\_knight\_set:  
 board.computer\_place\_enemy\_knight()  
  
 if board.turn == BLUE and not board.setup\_phase and not board.winner:  
 box\_position = board.should\_place\_box()  
 if box\_position and len(board.blue\_boxes) == 0:  
 row, col = box\_position  
 board.board[row][col] = (1, BLUE)  
 board.blue\_boxes.append(((row, col), 6))  
 board.change\_turn()  
 else:  
 \_, best\_move = minmax(board, 3, float('-inf'), float('inf'), True)  
 if best\_move:  
 piece, move\_pos, skipped = best\_move  
 board.move(piece, move\_pos[0], move\_pos[1])  
 if skipped:  
 board.remove(skipped)  
 board.change\_turn()  
  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 run = False  
  
 if event.type == pygame.MOUSEBUTTONDOWN:  
 pos = pygame.mouse.get\_pos()  
 if board.winner:  
 if action\_button and action\_button.collidepoint(pos):  
 board.reset()  
 continue  
 elif pos[0] < BOARD\_WIDTH and board.turn == RED:  
 row, col = pos[1] // SQUARE\_SIZE, pos[0] // SQUARE\_SIZE  
 board.select(row, col)  
 elif action\_button and action\_button.collidepoint(pos) and not board.winner:  
 board.placing\_box = True  
  
 board.draw(WIN)  
 action\_button = board.draw\_panel(WIN)  
 pygame.display.update()  
  
 pygame.quit()  
 sys.exit()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()