**פרויקט סיום באלגוריתמים**

**מתקדמים למערכות נבונות**

**מגישים:**

**יונתן וינשטוק - 206861148**

**יואב ימין – 319073862**

**אריאל סעדון – 208445221**

**קישורים:**

**קישור להצגת המצגת ביוטיוב -** [**https://youtu.be/nN-h1BhGzJs**](https://youtu.be/nN-h1BhGzJs)

**קישור להצגת המשחק ביוטיוב -** [**https://www.youtube.com/watch?v=lTbLrvFUMQ4**](https://www.youtube.com/watch?v=lTbLrvFUMQ4)

**קישור לגיטהאב -** [**https://github.com/ariel-dotcom/4-in-a-row-EXTREME**](https://github.com/ariel-dotcom/4-in-a-row-EXTREME)

**תיאור הפרוייקט:**

משחק 4 בשורה קלאסי כאשר הראשון להגיע לרצף של 4 מקומות צמודים(כולל

אלכסונים) מנצח, התוכנית מיושמת בשפת פייתון. במשחק שלנו יש אפשרות להשתמש בפצצות.

ישנן 2 סוגי פצצות כאשר גם לשחקן הבית וגם לשחקן המחשב יש אופציה להשתמש בהן.

פצצת עמודה: נותנת את האפשרות לשרוף ולהשמיד את כל הכלי משחק שנמצאים

בעמודה מסוימת.

פצצת קרבה: ניתן להניח בכל מקום על הלוח ולפוצץ ולהרוס את כל כלי המשחק

הנמצאים בקרבה של משבצת אחת, כולל אלכסונים, לפצצה .

שחקן המחשב יבוסס על אלגוריתם alpha beta pruning בהתאם לרמת המשחק הנקבעת.

ניתן לבחור רמת משחק כאשר רמת המשחק תקבע כמה שכבות לעומק יכנס האלגוריתם של אלפא-בטא-פרונינג.

**בעיות והתמודדות**

**מימוש הפצצות:**

היה צורך לתכנן מנגנון להפעלת פצצות רגילות (עמודה) וקרובות (מוקש):

פצצות עמודה: מנגנון הפעלת פצצות עמודה דורש זיהוי העמודה הנבחרת ומחיקת כל הכלים הנמצאים בעמודה זו. זה מצריך מחיקה פיזית של הכלים מהלוח והתאמה מחודשת של הלוח לאחר הפיצוץ. כמו כן, צריך להבטיח שהתור יעבור לשחקן הבא לאחר הפעלת הפצצה.

פצצות קרבה: פצצות קרבה דורשות זיהוי תא מסוים על הלוח והפעלת הפצצה כך שתמחק את הכלים הסמוכים אליו במרחק מוגדר. זה מצריך התאמת החוקים והלוגיקה של המשחק כדי לטפל במצבים המורכבים שנוצרים לאחר הפיצוץ.

**ההתמודדות עם מחיקת כלים מהלוח והתאמת החוקים הייתה מאתגרת:**

לאחר הפעלת הפצצות, הלוח משתנה באופן משמעותי ויש להתמודד עם ההשפעה של השינוי על המשחק. זה כולל התאמת התור לשחקן הבא, התאמת האנימציות ומעקב אחרי מצב הלוח החדש. כל שינוי קטן בלוגיקת המשחק מצריך בדיקות מקיפות כדי להבטיח שהמשחק נשאר תקין והוגן.

**התמודדות עם אלגוריתם ה-AI:**

**מימוש נכון של אלגוריתם אלפא-בטא פרונינג, כולל חישוב ערכי אלפא ובטא לשיפור הביצועים:**

אלגוריתם אלפא-בטא פרונינג משמש כדי להעריך את המהלכים האפשריים של השחקן ושל היריב (בינה מלאכותית) במטרה למצוא את המהלך האופטימלי. האלגוריתם בוחן את כל האפשרויות להנחת דיסקים, שימוש בפצצות והתגובה למהלכים של היריב.

בכל שלב של החיפוש, האלגוריתם מעדכן את ערכי האלפא והבטא בהתאם למהלכים האפשריים ומבצע "גיזום" כאשר מתברר שמהלך מסוים לא יוביל לתוצאה טובה יותר.. התמודדות עם ערכי אלפא ובטא הייתה חיונית כדי להבטיח שה-AI יוכל לבצע החלטות במהירות וביעילות.

הקוד:  
import tkinter as tk  
from tkinter import messagebox  
import random  
import copy  
  
class ConnectFour:  
 def \_\_init\_\_(self, root, rows=6, columns=7, difficulty=4):  
 self.root = root  
 self.root.title("4 in a Row")  
 self.rows = rows  
 self.columns = columns  
 self.board = [[0] \* self.columns for \_ in range(self.rows)] # Initialize the board with 0s  
 self.current\_player = 1 # Player 1 starts  
 self.bombs = [2, 2] # Each player has 2 bombs  
 self.proximity\_bombs = [2, 2] # Each player has 2 proximity bombs  
 self.proximity\_mode = False # Initially, proximity mode is off  
 self.difficulty = difficulty # Set difficulty level  
 self.highlighted\_column = -1 # No column is highlighted initially  
 self.highlighted\_cells = [] # No cells are highlighted initially  
  
 self.cell\_size = 60 # Size of each cell in the grid  
 self.canvas = tk.Canvas(root, width=self.columns \* self.cell\_size, height=self.rows \* self.cell\_size, bg="blue")  
 self.canvas.grid(row=3, column=0, columnspan=self.columns)  
  
 self.turn\_label = tk.Label(root, text="Player 1's Turn", bg="red", font=("Arial", 16), fg="white")  
 self.turn\_label.grid(row=0, column=0, columnspan=self.columns, sticky="nsew")  
  
 self.bomb\_buttons\_frame = tk.Frame(root)  
 self.bomb\_buttons\_frame.grid(row=1, column=0, columnspan=self.columns, sticky="ew")  
  
 # Create bomb buttons for each column  
 self.bomb\_buttons = [  
 tk.Button(self.bomb\_buttons\_frame, text="Bomb", command=lambda col=col: self.highlight\_column\_bomb(col),  
 bg="#8e3e41", fg="white", font=("Arial", 13)) for col in range(self.columns)]  
 for col, button in enumerate(self.bomb\_buttons):  
 button.grid(row=0, column=col, padx=3, pady=5, sticky="ew")  
  
 self.buttons\_frame = tk.Frame(root)  
 self.buttons\_frame.grid(row=2, column=0, columnspan=self.columns, sticky="ew")  
  
 # Create drop buttons for each column  
 self.buttons = [  
 tk.Button(self.buttons\_frame, text="Drop", command=lambda col=col: self.drop\_piece(col), bg="#3e8e41",  
 fg="white", font=("Arial", 14)) for col in range(self.columns)]  
 for col, button in enumerate(self.buttons):  
 button.grid(row=0, column=col, padx=1, pady=5, sticky="ew")  
  
 # Create a button to activate proximity mode  
 self.proximity\_bomb\_button = tk.Button(root, text="Prox Bomb", command=self.activate\_proximity\_mode,  
 bg="#8e7e41", fg="white", font=("Arial", 12))  
 self.proximity\_bomb\_button.grid(row=4, column=0, columnspan=self.columns, pady=5, sticky="ew")  
  
 self.canvas.bind("<Button-1>", self.canvas\_click\_handler)  
  
 self.root.bind("<Configure>", self.resize\_handler)  
 self.draw\_board()  
  
 # Handler for resizing the window  
 def resize\_handler(self, event):  
 width = self.columns \* self.cell\_size  
 height = self.rows \* self.cell\_size  
 self.canvas.config(width=width, height=height)  
  
 # Function to draw the board  
 def draw\_board(self):  
 self.canvas.delete("all")  
 for row in range(self.rows):  
 for col in range(self.columns):  
 x1 = col \* self.cell\_size  
 y1 = row \* self.cell\_size  
 x2 = x1 + self.cell\_size  
 y2 = y1 + self.cell\_size  
 color = "white"  
 if self.board[row][col] == 1:  
 color = "red"  
 elif self.board[row][col] == 2:  
 color = "yellow"  
 self.canvas.create\_oval(x1, y1, x2, y2, fill=color, outline="blue")  
  
 if self.highlighted\_column != -1:  
 self.highlight\_column(self.highlighted\_column)  
 if self.highlighted\_cells:  
 self.highlight\_cells(self.highlighted\_cells)  
  
 # Function to highlight a column  
 def highlight\_column(self, col):  
 for row in range(self.rows):  
 x1 = col \* self.cell\_size  
 y1 = row \* self.cell\_size  
 x2 = x1 + self.cell\_size  
 y2 = y1 + self.cell\_size  
 self.canvas.create\_rectangle(x1, y1, x2, y2, outline="orange", width=3)  
  
 # Function to highlight specific cells  
 def highlight\_cells(self, cells):  
 for row, col in cells:  
 x1 = col \* self.cell\_size  
 y1 = row \* self.cell\_size  
 x2 = x1 + self.cell\_size  
 y2 = y1 + self.cell\_size  
 self.canvas.create\_rectangle(x1, y1, x2, y2, outline="green", width=3)  
  
 # Function to disable all buttons  
 def disable\_buttons(self):  
 for button in self.buttons:  
 button.config(state=tk.DISABLED, disabledforeground=button.cget('fg'))  
 for button in self.bomb\_buttons:  
 button.config(state=tk.DISABLED, disabledforeground=button.cget('fg'))  
 self.proximity\_bomb\_button.config(state=tk.DISABLED, disabledforeground=self.proximity\_bomb\_button.cget('fg'))  
  
 # Function to enable all buttons  
 def enable\_buttons(self):  
 for button in self.buttons:  
 button.config(state=tk.NORMAL)  
 for button in self.bomb\_buttons:  
 button.config(state=tk.NORMAL)  
 self.proximity\_bomb\_button.config(state=tk.NORMAL)  
  
 # Function to drop a piece in a column  
 def drop\_piece(self, col):  
 if self.proximity\_mode:  
 return  
 # Check if the column is full  
 if self.board[0][col] != 0:  
 messagebox.showinfo("Invalid Move", "This column is full. Please choose another column.")  
 self.enable\_buttons()  
 return  
  
 self.disable\_buttons()  
 for row in range(self.rows - 1, -1, -1):  
 if self.board[row][col] == 0:  
 self.animate\_piece\_drop(row, col, self.current\_player)  
 self.board[row][col] = self.current\_player  
 self.draw\_board()  
 self.root.after(500, self.check\_game\_state, row, col)  
 return  
  
 # Function to animate the dropping of a piece  
 def animate\_piece\_drop(self, row, col, player):  
 color = "red" if player == 1 else "yellow"  
 for i in range(row + 1):  
 x1 = col \* self.cell\_size  
 y1 = i \* self.cell\_size  
 x2 = x1 + self.cell\_size  
 y2 = y1 + self.cell\_size  
 self.canvas.create\_oval(x1, y1, x2, y2, fill=color, outline="blue")  
 self.canvas.update()  
 self.canvas.after(50)  
 if i < row:  
 self.canvas.create\_oval(x1, y1, x2, y2, fill="white", outline="blue")  
  
 # Function to check the game state after a piece is dropped  
 def check\_game\_state(self, row, col):  
 if self.check\_winner(row, col):  
 self.highlight\_winning\_line(row, col)  
 messagebox.showinfo("Game Over", f"Player {self.current\_player} wins!")  
 self.reset\_game()  
 else:  
 self.current\_player = 3 - self.current\_player  
 self.update\_turn\_label()  
 if self.current\_player == 2:  
 self.root.after(500, self.ai\_move)  
 else:  
 self.enable\_buttons()  
  
 # Function to highlight a column for bomb usage  
 def highlight\_column\_bomb(self, col):  
 self.highlighted\_column = col  
 self.draw\_board()  
 self.root.after(1000, self.use\_bomb, col)  
  
 # Function to use a bomb on a column  
 def use\_bomb(self, col):  
 if self.proximity\_mode:  
 return  
 self.disable\_buttons()  
 if self.bombs[self.current\_player - 1] > 0:  
 for row in range(self.rows):  
 self.board[row][col] = 0  
 self.bombs[self.current\_player - 1] -= 1  
 self.current\_player = 3 - self.current\_player  
 self.update\_turn\_label()  
 self.highlighted\_column = -1  
 if self.current\_player == 2:  
 self.root.after(500, self.ai\_move)  
 else:  
 self.enable\_buttons()  
 else:  
 messagebox.showinfo("No Bombs", f"Player {self.current\_player} has no bombs left!")  
 self.enable\_buttons()  
 self.draw\_board()  
  
 # Function to activate proximity mode  
 def activate\_proximity\_mode(self):  
 if self.proximity\_bombs[self.current\_player - 1] > 0:  
 self.proximity\_mode = True  
 else:  
 messagebox.showinfo("No Proximity Bombs", f"Player {self.current\_player} has no proximity bombs left!")  
  
 # Handler for canvas click events  
 def canvas\_click\_handler(self, event):  
 if self.proximity\_mode:  
 col = event.x // self.cell\_size  
 row = event.y // self.cell\_size  
 if 0 <= row < self.rows and 0 <= col < self.columns:  
 self.highlight\_proximity\_bomb(row, col)  
 self.select\_proximity\_bomb(row, col)  
  
 # Function to highlight cells for proximity bomb usage  
 def highlight\_proximity\_bomb(self, row, col):  
 self.highlighted\_cells = [(row + dr, col + dc) for dr in range(-1, 2) for dc in range(-1, 2) if  
 0 <= row + dr < self.rows and 0 <= col + dc < self.columns]  
 self.draw\_board()  
  
 # Function to use a proximity bomb  
 def select\_proximity\_bomb(self, row, col):  
 if not self.proximity\_mode:  
 return  
 self.disable\_buttons()  
 self.clear\_adjacent\_pieces(row, col)  
 self.proximity\_bombs[self.current\_player - 1] -= 1  
 self.proximity\_mode = False  
 self.adjust\_board\_with\_animation()  
 self.highlighted\_cells = []  
 # Add delay before checking for a win  
 self.root.after(500, self.check\_board\_for\_winner\_after\_prox)  
  
 # Function to check the board for a winner after proximity bomb usage  
 def check\_board\_for\_winner\_after\_prox(self):  
 self.canvas.update()  
 self.canvas.after(500)  
 # Check for a winner for both players  
 if self.check\_board\_for\_winner():  
 return  
 self.current\_player = 3 - self.current\_player  
 self.update\_turn\_label()  
 if self.current\_player == 2:  
 self.root.after(500, self.ai\_move)  
 else:  
 self.enable\_buttons()  
  
 # Function to check the entire board for a winner  
 def check\_board\_for\_winner(self):  
 for col in range(self.columns):  
 for row in range(self.rows):  
 if self.board[row][col] != 0:  
 if self.check\_winner(row, col):  
 self.highlight\_winning\_line(row, col)  
 messagebox.showinfo("Game Over", f"Player {self.board[row][col]} wins!")  
 self.reset\_game()  
 return True  
 return False  
  
 # Function to clear adjacent pieces for proximity bomb usage  
 def clear\_adjacent\_pieces(self, row, col):  
 directions = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]  
 self.board[row][col] = 0  
 for dr, dc in directions:  
 r, c = row + dr, col + dc  
 if 0 <= r < self.rows and 0 <= c < self.columns:  
 self.board[r][c] = 0  
 self.draw\_board()  
  
 # Function to adjust the board with animation after using a proximity bomb  
 def adjust\_board\_with\_animation(self):  
 for col in range(self.columns):  
 empty\_slots = []  
 for row in range(self.rows - 1, -1, -1):  
 if self.board[row][col] == 0:  
 empty\_slots.append(row)  
 elif empty\_slots:  
 empty\_row = empty\_slots.pop(0)  
 self.animate\_piece\_fall(row, empty\_row, col, self.board[row][col])  
 self.board[empty\_row][col] = self.board[row][col]  
 self.board[row][col] = 0  
 empty\_slots.append(row)  
 self.draw\_board()  
  
 # Function to animate the fall of a piece  
 def animate\_piece\_fall(self, start\_row, end\_row, col, player):  
 color = "red" if player == 1 else "yellow"  
 for i in range(start\_row, end\_row + 1):  
 x1 = col \* self.cell\_size  
 y1 = i \* self.cell\_size  
 x2 = x1 + self.cell\_size  
 y2 = y1 + self.cell\_size  
 self.canvas.create\_oval(x1, y1, x2, y2, fill=color, outline="blue")  
 self.canvas.update()  
 self.canvas.after(50)  
 if i < end\_row:  
 self.canvas.create\_oval(x1, y1, x2, y2, fill="white", outline="blue")  
  
 # Function to check if a player has won  
 def check\_winner(self, row, col):  
 def count\_connected(r\_step, c\_step):  
 r, c = row, col  
 count = 0  
 while 0 <= r < self.rows and 0 <= c < self.columns and self.board[r][c] == self.board[row][col]:  
 count += 1  
 r += r\_step  
 c += c\_step  
 return count  
  
 directions = [(1, 0), (0, 1), (1, 1), (1, -1)]  
 for r\_step, c\_step in directions:  
 if count\_connected(r\_step, c\_step) + count\_connected(-r\_step, -c\_step) - 1 >= 4:  
 self.winning\_coords = self.get\_winning\_coords(row, col, r\_step, c\_step)  
 return True  
 return False  
  
 # Function to get the coordinates of the winning line  
 def get\_winning\_coords(self, row, col, r\_step, c\_step):  
 coords = [(row, col)]  
 # Get coordinates in the positive direction  
 r, c = row + r\_step, col + c\_step  
 while 0 <= r < self.rows and 0 <= c < self.columns and self.board[r][c] == self.board[row][col]:  
 coords.append((r, c))  
 r += r\_step  
 c += c\_step  
 # Get coordinates in the negative direction  
 r, c = row - r\_step, col - c\_step  
 while 0 <= r < self.rows and 0 <= c < self.columns and self.board[r][c] == self.board[row][col]:  
 coords.insert(0, (r, c))  
 r -= r\_step  
 c -= c\_step  
 return coords  
  
 # Function to highlight the winning line  
 def highlight\_winning\_line(self, row, col):  
 for r, c in self.winning\_coords:  
 x1 = c \* self.cell\_size  
 y1 = r \* self.cell\_size  
 x2 = x1 + self.cell\_size  
 y2 = y1 + self.cell\_size  
 self.canvas.create\_oval(x1, y1, x2, y2, fill="green", outline="blue")  
  
 # Function to update the turn label  
 def update\_turn\_label(self):  
 self.turn\_label.config(text=f"Player {self.current\_player}'s Turn",  
 bg="red" if self.current\_player == 1 else "yellow")  
  
 # Function to reset the game  
 def reset\_game(self):  
 self.board = [[0] \* self.columns for \_ in range(self.rows)]  
 self.current\_player = 1  
 self.bombs = [2, 2]  
 self.proximity\_bombs = [2, 2]  
 self.proximity\_mode = False  
 self.highlighted\_column = -1  
 self.highlighted\_cells = []  
 self.draw\_board()  
 self.update\_turn\_label()  
 self.enable\_buttons()  
  
 # Function to make an AI move  
 def ai\_move(self):  
 if self.current\_player == 2:  
 if self.difficulty == 2:  
 self.easy\_ai()  
 elif self.difficulty == 4:  
 self.moderate\_ai()  
 elif self.difficulty == 6:  
 self.hard\_ai()  
  
 # Function for easy AI moves  
 def easy\_ai(self):  
 if random.random() < 0.5 and self.bombs[1] > 0:  
 self.highlight\_column\_bomb\_ai()  
 else:  
 self.make\_move\_ai(2)  
  
 # Function for moderate AI moves  
 def moderate\_ai(self):  
 if random.random() < 0.3 and self.bombs[1] > 0:  
 self.highlight\_column\_bomb\_ai()  
 else:  
 self.make\_move\_ai(4)  
  
 # Function for hard AI moves  
 def hard\_ai(self):  
 if self.can\_win\_with\_proximity\_bomb():  
 self.highlight\_proximity\_bomb\_ai()  
 elif random.random() < 0.2 and self.bombs[1] > 0:  
 self.highlight\_column\_bomb\_ai()  
 else:  
 self.make\_move\_ai(6)  
  
 # Function to highlight a column for AI bomb usage  
 def highlight\_column\_bomb\_ai(self):  
 col = random.randint(0, self.columns - 1)  
 if any(self.board[row][col] != 0 for row in range(self.rows)):  
 self.highlighted\_column = col  
 self.draw\_board()  
 self.root.after(1000, self.use\_bomb\_ai, col)  
 else:  
 self.make\_move\_ai(6)  
  
 # Function for AI to use a bomb  
 def use\_bomb\_ai(self, col):  
 if self.proximity\_mode:  
 return  
 if self.bombs[1] > 0:  
 for row in range(self.rows):  
 self.board[row][col] = 0  
 self.bombs[1] -= 1  
 self.highlighted\_column = -1  
 self.draw\_board()  
 self.root.after(500, self.check\_board\_for\_winner\_after\_prox)  
  
 # Function for AI to highlight proximity bomb  
 def highlight\_proximity\_bomb\_ai(self):  
 for row in range(self.rows):  
 for col in range(self.columns):  
 if self.proximity\_bombs[1] > 0 and any(  
 self.board[row + dr][col + dc] != 0 for dr in range(-1, 2) for dc in range(-1, 2) if  
 0 <= row + dr < self.rows and 0 <= col + dc < self.columns):  
 temp\_board = copy.deepcopy(self.board)  
 self.clear\_adjacent\_pieces\_for\_check(temp\_board, row, col)  
 if self.check\_for\_ai\_win(temp\_board):  
 self.highlight\_proximity\_bomb(row, col)  
 self.root.after(1000, self.select\_proximity\_bomb\_ai, row, col)  
 return  
 self.make\_move\_ai(6)  
  
 # Function for AI to select proximity bomb  
 def select\_proximity\_bomb\_ai(self, row, col):  
 self.clear\_adjacent\_pieces(row, col)  
 self.proximity\_bombs[1] -= 1  
 self.adjust\_board\_with\_animation()  
 self.highlighted\_cells = []  
 # Add delay before checking for a win  
 self.root.after(500, self.check\_board\_for\_winner\_after\_prox)  
  
 # Function to clear adjacent pieces for checking  
 def clear\_adjacent\_pieces\_for\_check(self, board, row, col):  
 directions = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]  
 board[row][col] = 0  
 for dr, dc in directions:  
 r, c = row + dr, col + dc  
 if 0 <= r < self.rows and 0 <= c < self.columns:  
 board[r][c] = 0  
  
 # Function to check if AI can win with a proximity bomb  
 def can\_win\_with\_proximity\_bomb(self):  
 for row in range(self.rows):  
 for col in range(self.columns):  
 temp\_board = copy.deepcopy(self.board)  
 self.clear\_adjacent\_pieces\_for\_check(temp\_board, row, col)  
 if self.check\_for\_ai\_win(temp\_board):  
 return True  
 return False  
  
 # Function to check if AI can win  
 def check\_for\_ai\_win(self, board):  
 for col in range(self.columns):  
 for row in range(self.rows):  
 if board[row][col] == 0:  
 continue  
 if self.check\_winner\_with\_board(board, row, col, 2):  
 return True  
 return False  
  
 # Function to check if a player has won with a given board  
 def check\_winner\_with\_board(self, board, row, col, player):  
 def count\_connected\_with\_board(board, r\_step, c\_step):  
 r, c = row, col  
 count = 0  
 while 0 <= r < self.rows and 0 <= c < self.columns and board[r][c] == player:  
 count += 1  
 r += r\_step  
 c += c\_step  
 return count  
  
 directions = [(1, 0), (0, 1), (1, 1), (1, -1)]  
 for r\_step, c\_step in directions:  
 if count\_connected\_with\_board(board, r\_step, c\_step) + count\_connected\_with\_board(board, -r\_step,  
 -c\_step) - 1 >= 4:  
 return True  
 return False  
  
 # Function for AI to make a move using alpha-beta pruning  
 def make\_move\_ai(self, depth):  
 \_, col = self.alpha\_beta(self.board, depth, float('-inf'), float('inf'), True)  
 if col is not None:  
 self.drop\_piece(col)  
  
 # Alpha-beta pruning algorithm  
 def alpha\_beta(self, board, depth, alpha, beta, maximizing\_player):  
 valid\_moves = [col for col in range(self.columns) if board[0][col] == 0]  
 if depth == 0 or not valid\_moves:  
 return self.evaluate\_board(board), None  
 if maximizing\_player:  
 max\_eval = float('-inf')  
 best\_col = random.choice(valid\_moves)  
 for col in valid\_moves:  
 temp\_board = copy.deepcopy(board)  
 self.make\_move(temp\_board, col, 2)  
 eval, \_ = self.alpha\_beta(temp\_board, depth - 1, alpha, beta, False)  
 if eval > max\_eval:  
 max\_eval = eval  
 best\_col = col  
 alpha = max(alpha, eval)  
 if beta <= alpha:  
 break  
 return max\_eval, best\_col  
 else:  
 min\_eval = float('inf')  
 best\_col = random.choice(valid\_moves)  
 for col in valid\_moves:  
 temp\_board = copy.deepcopy(board)  
 self.make\_move(temp\_board, col, 1)  
 eval, \_ = self.alpha\_beta(temp\_board, depth - 1, alpha, beta, True)  
 if eval < min\_eval:  
 min\_eval = eval  
 best\_col = col  
 beta = min(beta, eval)  
 if beta <= alpha:  
 break  
 return min\_eval, best\_col  
  
 # Function to make a move on the board  
 def make\_move(self, board, col, player):  
 for row in range(self.rows - 1, -1, -1):  
 if board[row][col] == 0:  
 board[row][col] = player  
 break  
  
 # Function to evaluate the board  
 def evaluate\_board(self, board):  
 score = 0  
  
 # Score center column  
 center\_array = [int(board[row][self.columns // 2]) for row in range(self.rows)]  
 center\_count = center\_array.count(2)  
 score += center\_count \* 3  
  
 # Score horizontal, vertical, and diagonal lines  
 for row in range(self.rows):  
 for col in range(self.columns):  
 if board[row][col] == 2:  
 score += self.score\_position(board, row, col, 2)  
 elif board[row][col] == 1:  
 score -= self.score\_position(board, row, col, 1)  
  
 return score  
  
 # Function to score a position  
 def score\_position(self, board, row, col, player):  
 score = 0  
 opponent = 1 if player == 2 else 2  
  
 # Scoring directions  
 directions = [(1, 0), (0, 1), (1, 1), (1, -1)]  
  
 for direction in directions:  
 score += self.score\_line(board, row, col, player, direction)  
  
 return score  
  
 # Function to score a line  
 def score\_line(self, board, row, col, player, direction):  
 score = 0  
 line = []  
 for i in range(-3, 4):  
 r = row + i \* direction[0]  
 c = col + i \* direction[1]  
 if 0 <= r < self.rows and 0 <= c < self.columns:  
 line.append(board[r][c])  
 else:  
 line.append(None)  
  
 # Check line for scoring  
 for i in range(len(line) - 3):  
 window = line[i:i + 4]  
 score += self.evaluate\_window(window, player)  
  
 return score  
  
 # Function to evaluate a window of 4 cells  
 def evaluate\_window(self, window, player):  
 score = 0  
 opponent = 1 if player == 2 else 2  
  
 if window.count(player) == 4:  
 score += 100  
 elif window.count(player) == 3 and window.count(0) == 1:  
 score += 10  
 elif window.count(player) == 2 and window.count(0) == 2:  
 score += 5  
  
 if window.count(opponent) == 3 and window.count(0) == 1:  
 score -= 80  
  
 return score  
  
  
# Function to show the difficulty menu  
def show\_difficulty\_menu():  
 difficulty\_window = tk.Tk()  
 difficulty\_window.title("Select Difficulty Level")  
  
 tk.Label(difficulty\_window, text="Select Difficulty Level:", font=("Arial", 14)).pack(pady=5)  
  
 tk.Button(difficulty\_window, text="Beginner", command=lambda: start\_game(difficulty\_window, 2), bg="lightgreen",  
 font=("Arial", 14)).pack(pady=5)  
 tk.Button(difficulty\_window, text="Moderate", command=lambda: start\_game(difficulty\_window, 4), bg="lightblue",  
 font=("Arial", 14)).pack(pady=5)  
 tk.Button(difficulty\_window, text="Hard", command=lambda: start\_game(difficulty\_window, 6), bg="red",  
 font=("Arial", 14)).pack(pady=5)  
  
 difficulty\_window.mainloop()  
  
  
# Function to start the game  
def start\_game(difficulty\_window, difficulty):  
 difficulty\_window.destroy()  
 root = tk.Tk()  
 game = ConnectFour(root, rows=6, columns=7, difficulty=difficulty)  
 root.mainloop()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 show\_difficulty\_menu()