

CS152 - FINAL PROJECT (CONNECT 4)

OLUWAKOREDE AKANDE

FALL 2020

Problem Definition

Connect Four is a two-player connection game, similar to tic-tac-toe, in which two players select a particular colour and take turns dropping coloured discs (corresponding to their chosen colour) into a default 6 by 7 board.

Rules of the Game

- A disc necessarily occupies the lowest available row in a column.
- The first player to get four in a row either vertically, horizontally or diagonally wins. Hence, a player's goal is to connect four discs while trying to prevent their opponent from doing the same.

Clearly, success in a game against informed players will require some foresight in determining what moves the opposing player will or will not carry out and how these can be counteracted or leveraged to a player's advantage. Thus the problem can be framed as a search problem where numerous intermediate states (resulting from alternating play between the players) can be considered from the initial empty board state to lead to one of the possible goal states. This motivated my decision to tackle the problem considering the knowledge gained in class on various search algorithms. More specifically, given *Connect Four* is a two-player, zero-sum adversarial game, it served as a prime opportunity to apply the minimax adversarial search technique, which we discussed in class but did not implement with code. *Connect Four* is especially interesting given the extremely large state space of 4,531,985,219,092 positions ("John's Connect Four Playground," n.d.) for the default 6 by 7 board, providing a prime opportunity to apply the alpha-beta pruning extension to the standard minimax algorithm.

Solution Specification:

To solve the problem, I developed a game simulator using *Pygame*, where a human battles against an AI opponent. The game is started with an empty *ConnectFour* board and the first turn (AI or player) is randomly decided using the *random* module in Python. The turns will then alternate until either player wins, or the board is full. When it is the player's turn, the player utilizes the mouse to move the disc to the column intended to be dropped into, and clicks to drop the piece. This results in the disc being dropped into the lowest slot in that column. If the column is entirely full, the disc will not be dropped until a valid move is selected by the user. The AI's move is determined using Minimax with Alpha-Beta Pruning search, causing it to stop evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move (Wikipedia, 2002).

Backend Details:

The actual ConnectFour board is represented using a 2D numpy array. This is initialized as a 2D array of 0s and depending on selections made by the user or AI, array position values are updated (through indexing) to either '1' for the player or '2' for the AI agent. A ConnectFour class was created to organize and structure the major workings of the game and the evaluation of game states (see Appendix). This includes methods to determine valid moves, determine if a terminal/ winning state has been reached, and evaluate the current board state, among others (see Appendix).

Alpha-Beta Pruning with Minimax Implementation:

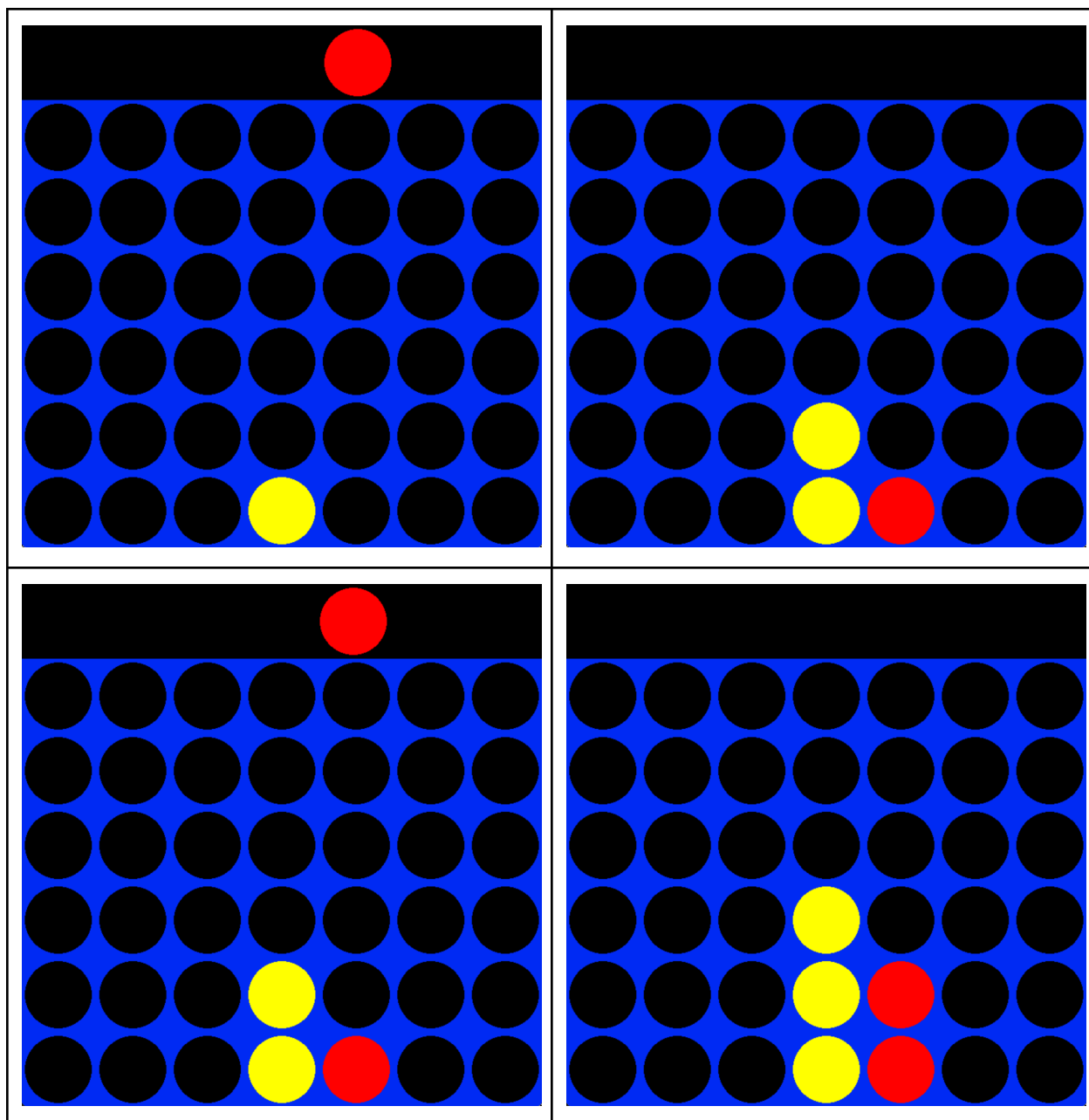
The algorithm expands on the standard minimax algorithm, which considers each possible move before choosing a column to place its disc, by considering two values: alpha and beta. Alpha corresponds to the minimum score that the maximizing player is assured of while beta refers to the maximum score that the minimizing player is assured of given the currently explored nodes. These values are repeatedly updated as the algorithm progresses, and whenever the maximum score that the minimizing player is assured of becomes less than the minimum score that the maximizing player is assured of (i.e. $\beta < \alpha$), descendants of the node are not considered, allowing the subtree to be pruned (as these are worse off than already explored nodes) and thus reducing the time taken to traverse the tree. The worst-case time complexity of Minimax with Alpha-Beta Pruning is $O(b^{d/2})$ compared to $O(b^d)$ for standard minimax (Russell & Norvig, 2002).

Test Cases:

To determine that the Connect4 game works as intended, I conducted sample runs to validate the two rules of the game as outlined above. The results are displayed below:

Rule 1: A disc necessarily occupies the lowest available row in a column

Before Action	After Action
---------------	--------------



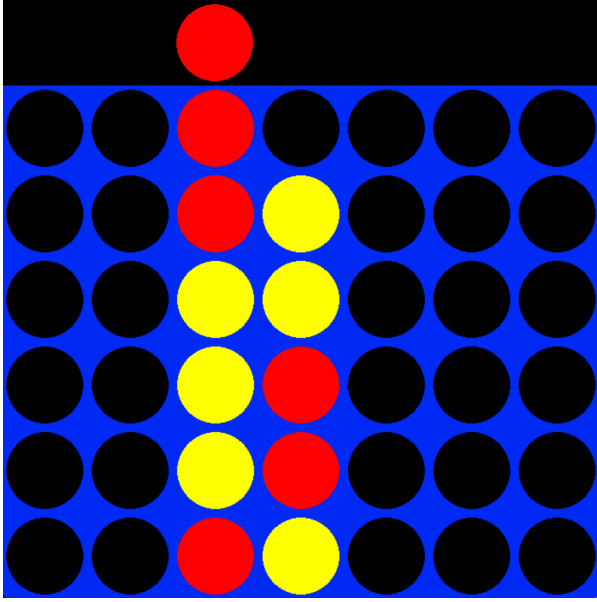
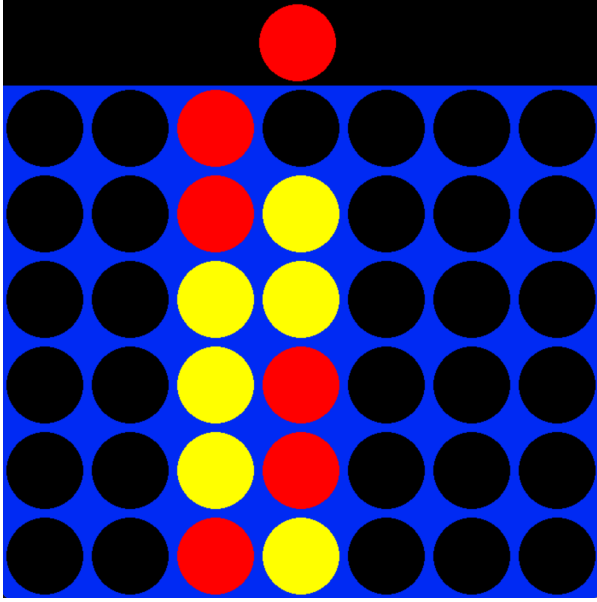
	
---	--

Table 1: Checking that a disc necessarily occupies the lowest available row in a column

This confirms that the lowest available row in a column is always occupied.

Rule 2: First player to get four in a row either vertically, horizontally or diagonally wins

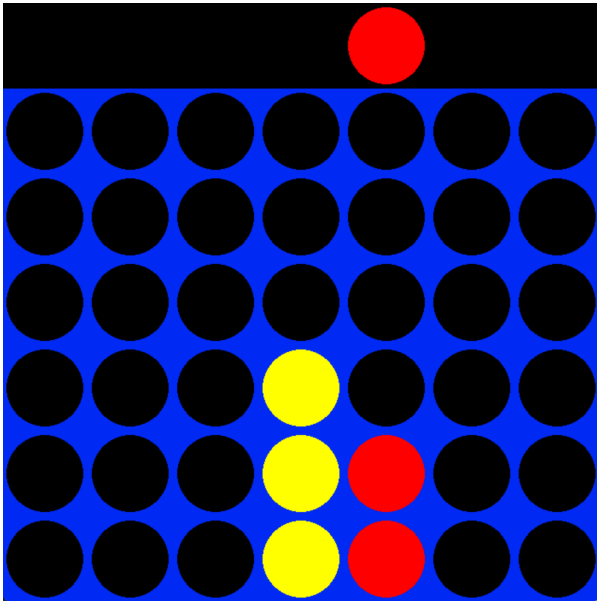
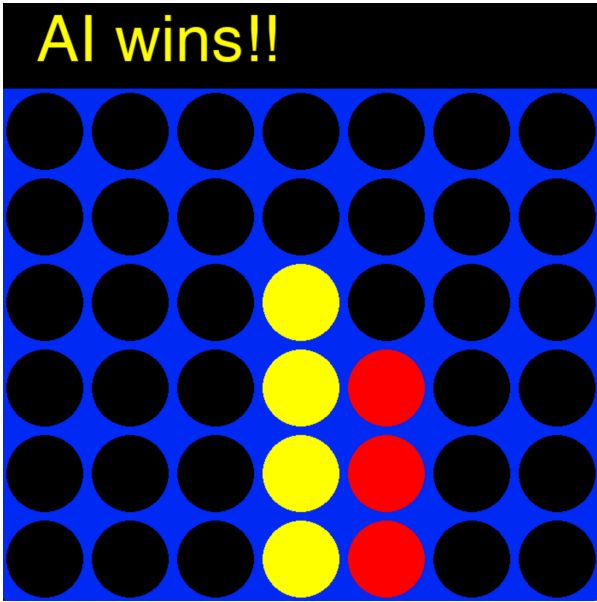
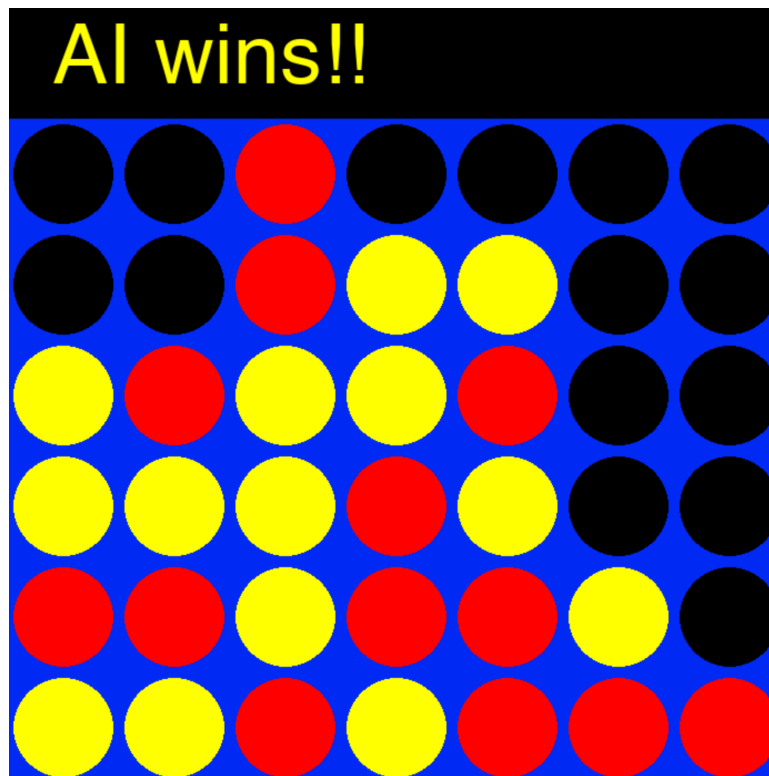
Before Action	After Action
	

Table 2: Checking that the first player to get four in a row wins

From the images above, we note that the AI agent gets to 4 consecutive vertical discs before the human agent. Hence, the AI agent wins in line with the rules of the game.

Subsequently, I ran sample games to check that the algorithm performed as expected. From the screenshot below we can see that the algorithm disrupts any chance the human player (in red) has to make meaningful progress, simultaneously working to increase the likelihood of winning the game. In particular, the left side is overloaded to increase the chances of getting four in a row either vertically, horizontally or diagonally.



Interestingly, the algorithm also incorporates/learns some notion of common sense. In Table 2, we see that rather than hindering the progress of the human player in stacking discs, the algorithm recognizes that it is ahead and instead seeks to maximize its own utility (rather than minimize its opponent's utility). This acts in accordance with what we expect.

References

John's Connect Four Playground. (n.d.). Retrieved December 21, 2019, from

<https://tromp.github.io/c4/c4.html>

Russell, S., & Norvig, P. (2002). Artificial intelligence: a modern approach.

Alpha-beta pruning. (2019). In Wikipedia . Retrieved from

https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=928332496

Appendix

Code:

```
#Import packages needed to build connect 4 pygame
import pygame
import numpy as np
import random
import sys
import math
from copy import deepcopy
```

```
pygame 1.9.6
Hello from the pygame community. https://www.pygame.org/contribute.html
```

```
#Define python/pygame colours to build connect four board
RED = (255,0,0)
YELLOW = (255,255,0)
BLUE = (0,42,243)
BLACK = (0,0,0)
```

```
#Define disc states to differentiate between player discs, AI discs and empty positions
EMPTY_SLOT= 0
PLAYER_DISC= 1
AI_DISC= 2
# ai1_disc= 2
# ai2_disc= 3

#Define player and AI turns
PLAYER= 0
AI = 1
#AI1_turn = 1
#A2_turn = 2
```

```

class ConnectFour:
    """
    Class defining connect four board and methods needed to build an adversarial game against an AI using
    minimax with alpha-beta pruning

    Adapted from Galli, K. (2019) KeithGalli/Connect4-Python [Python]. Retrieved from
    https://github.com/KeithGalli/Connect4-Python. Improvements to the code include an object-oriented implementation
    of the ConnectFour board and its properties/attributes resulting in a neater and organized code. Improved
    flexibility and generalizability of code, allowing the game to be expanded to larger board sizes.

    Alpha-Beta Pruning with Minimax adapted from Wikipedia (2002). Alpha-beta pruning Pseudocode.
    Retrieved from https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning#Pseudocode
    """

    def __init__(self, rows=6, columns=7):
        """
        Initialize board attributes
        """
        self.rows = rows
        self.columns = columns
        self.board = np.zeros((rows, columns))

    def drop_disc(self, row, col, disc):
        #Drop disc in specified location

        self.board[row][col] = disc

    def is_valid_location(self, col):
        #Check to see if column is not entirely filled (i.e. if there are empty slots)
        return self.board[self.rows-1][col] == 0

    def get_vacant_row(self, col):
        #Return vacant row
        #Note that the topmost vacant row will be returned
        #This will be rectified due to the way the board is printed in function below

        for row in range(self.rows):
            if self.board[row][col] == 0:
                return row

    def get_valid_positions(self):
        """
        Determine which columns a disc can be dropped into by players
        """
        #Initialize empty list to store valid positions
        valid_positions = []

        for col in range(self.columns):
            if self.is_valid_location(col):
                valid_positions.append(col)

        return valid_positions

```



```

def winning_move(self, disc):

    # Check horizontal locations for win
    #All columns except the last three are checked because these are the only possible starting
    #positions for a winning move that lead to four horizontal disc
    for col in range(self.columns-3):

        #Iterating over all the rows
        for row in range(self.rows):
            if self.board[row][
                col] == disc and self.board[row][col+1] == disc and self.board[
                row][col+2] == disc and self.board[row][col+3] == disc:
                return True

    #Check vertical locations for win
    for col in range(self.columns):

        #Only the first three rows are checked since the starting position for a winning move
        #can only be from those rows to have a line of four
        for row in range(self.rows-3):
            if self.board[row][
                col] == disc and self.board[row+1][
                col] == disc and self.board[row+2][col] == disc and self.board[row+3][col] == disc:
                return True

    # Check positive diagonal locations for a winning configuration

    #Starting position for a winning move are all rows and columns except the last three of each
    for col in range(self.columns-3):
        for row in range(self.rows-3):
            if self.board[row
                ][col] == disc and self.board[row+1
                ][col+1] == disc and self.board[row+2][col+2] == disc and self.b

                return True

    # Check negative diagonal locations for a winning configuration

    #Starting position for a winning move are all rows and columns except the last three columns
    #and first three rows
    for col in range(self.columns-3):
        for row in range(3, self.rows):
            if self.board[row][col] == disc and self.board[row-1][col+1] == disc and self.board[row-2][col+2] == disc
                return True

```

```

def evaluate_window(self, window, disc):

    """
    Assigning score to players depending on the state of the inputted window
    """

    #Initial score to 0
    score = 0

    #If the current disc is the player's, set opposing disc to AI disc
    if disc == PLAYER_DISC:
        opp_disc = AI_DISC
    else:
        opp_disc = PLAYER_DISC

    #If a player has successfully connected four discs, add 100 to score
    if window.count(disc) == 4:
        score += 100

    #If a player has successfully connected three discs, add 100 to score
    elif window.count(disc) == 3 and window.count(EMPTY_SLOT) == 1:
        score += 5

    #If a player has successfully connected two discs, add 100 to score
    elif window.count(disc) == 2 and window.count(EMPTY_SLOT) == 2:
        score += 2

    #If the opposing player has successfully connected three discs, subtract 4 from score
    if window.count(opp_disc) == 3 and window.count(EMPTY_SLOT) == 1:
        score -= 4

    #Return the score given the input window
    return score

```

```
def score_position(self, disc, window_length=4):  
    """  
    Score player based on board configuration using the scoring system established  
    in the evaluate_window function  
    """  
  
    #Initialize score to 0  
    score = 0  
  
    ## Score center column  
    center_window = [int(i) for i in list(self.board[:, self.columns//2])]   
    score += center_window.count(disc) * 3  
  
    ## Score rows  
    for row in range(self.rows):  
        #Get all the columns of the particular row  
        row_window = [int(i) for i in list(self.board[row,:])]   
  
        #For all the columns except the last three in the given row  
        for col in range(self.columns-3):  
            #Calculate the score using the scoring system  
            window = row_window[col:col+window_length]  
            score += self.evaluate_window(window, disc)  
  
    ## Score verticals (column discs configurations)  
    for col in range(self.columns):  
        col_window = [int(i) for i in list(self.board[:,col])]   
        for row in range(self.rows-3):  
            window = col_window[row:row+window_length]  
            score += self.evaluate_window(window, disc)  
  
    ## Score positive diagonals  
    for row in range(self.rows-3):  
        for col in range(self.columns-3):  
            window = [self.board[row+i][col+i] for i in range(window_length)]  
            score += self.evaluate_window(window, disc)  
  
    #Score negative diagonals  
    for row in range(self.rows-3):  
        for col in range(self.columns-3):  
            window = [self.board[row+3-i][col+i] for i in range(window_length)]  
            score += self.evaluate_window(window, disc)  
  
    return score
```

```

def terminal_node(self):
    """
    Check if a terminal state has been reached
    """
    #Check if a player has successfully connected four discs or
    #if there are no other empty slots on the board
    return self.winning_move(PYER_DISC) or self.winning_move(AI_DISC) or len(self.get_valid_positions()) == 0

def alpha_beta_minimax_search(self, depth, alpha, beta, maximizingPlayer):
    """
    Evaluate the best branches possible going as deep down the tree as specified by depth
    """
    #Determine possible columns that can played into given current board state
    possible_moves= self.get_valid_positions()

    if depth == 0 or self.terminal_node():
        if self.terminal_node():
            if self.winning_move(AI_DISC):
                return (None, 50000)
            elif self.winning_move(PYER_DISC):
                return (None, -50000)
            else: # Game is over, no more valid moves
                return (None, 0)
        else: # Depth is zero
            #Return board heuristic value
            return (None, self.score_position(AI_DISC))

    if maximizingPlayer:
        value = -float("inf")
        move= np.random.choice(possible_moves)

        for col in possible_moves:
            row= self.get_vacant_row(col)
            next_state= deepcopy(self)
            next_state.drop_disc(row,col,AI_DISC)
            new_score = next_state.alpha_beta_minimax_search(depth-1, alpha, beta, False)[1]

            #Update value to highest score encountered based on all explored moves
            if new_score > value:
                value= new_score
                move= col

            alpha= max(value,alpha)

            if alpha >= beta:
                break

        return move,value

```

```

#Minimizing player trying to obtain the lowest score
else:

    value = float("inf")
    move= np.random.choice(possible_moves)

    for col in possible_moves:
        row= self.get_vacant_row(col)
        next_state= deepcopy(self)
        next_state.drop_disc(row,col,PYER_DISC)
        new_score = next_state.alpha_beta_minimax_search(depth-1, alpha, beta, True)[1]

        #Update value to highest score encountered based on all explored moves
        if new_score < value:
            value= new_score
            move= col

        beta= min(value,beta)

        if alpha >= beta:
            break

    return move,value

```

```
#Initialize Connect Four board and print the empty board
#Note parameter (row,column) can be changed to create boards larger than the default 6 by 7
board = ConnectFour(6,7)

#Initialize game_over to False to signal that the game has begun and is still in progress
game_over = False

pygame.init()

#Initialize attributes for the pygame board
TILESIZE = 100
width = board.columns * TILESIZE
height = (board.rows+1) * TILESIZE

#Specify size of pygame window
size = (width, height)

#Specify radius size for player discs given size of squares
RADIUS = int(TILESIZE/2 - 5)

screen = pygame.display.set_mode(size)

draw_pygame_board(board)
pygame.display.update()

pygame.font.init()
game_font = pygame.font.SysFont('helvetica', 75)

#Randomly choose who starts first
turn = random.randint(PLAYER, AI)

while not game_over:

    #If the game is quit, exit
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    #If the mouse is moved, move the disc along with it
    if event.type == pygame.MOUSEMOTION:
        pygame.draw.rect(screen, BLACK, (0,0, width, TILESIZE))
        xpos = event.pos[0]
        if turn == PLAYER:
            pygame.draw.circle(screen, RED, (xpos, int(TILESIZE/2)), RADIUS)

    #Update the display to see changes
    pygame.display.update()

    #If the mouse is clicked (to drop)
    if event.type == pygame.MOUSEBUTTONDOWN:
        pygame.draw.rect(screen, BLACK, (0,0, width, TILESIZE))
```

```
pygame.draw.rect(screen, BLACK, (0,0, width, TILESIZE))

# If it is the player's turn
if turn == PLAYER:

    #Get the drop position and use it to estimate the intended board column
    xpos = event.pos[0]
    col = int(math.floor(xpos/TILESIZE))

    #If the selected column is a valid column
    if board.is_valid_location(col):

        #Drop the disc down into the next available row
        row = board.get_vacant_row(col)
        board.drop_disc(row, col, PLAYER_DISC)

        #If the drop results in successfully connecting four discs
        if board.winning_move(PLAYER_DISC):
            #Player 1 Wins
            label = game_font.render("Player 1 wins!!", 1, RED)
            screen.blit(label, (40,10))
            game_over = True

        turn += 1
        turn = turn % 2

        #board.print_board()
        draw_pygame_board(board)

#Ask for Player 2 Input
if turn == AI and not game_over:

    col, minimax_score = board.alpha_beta_minimax_search(4, -float('inf'), float('inf'), True)

    if board.is_valid_location(col):
        row = board.get_vacant_row(col)
        board.drop_disc(row, col, AI_DISC)

        if board.winning_move(AI_DISC):
            label = game_font.render("AI wins!!", 1, YELLOW)
            screen.blit(label, (40,10))
            game_over = True

        draw_pygame_board(board)

        turn += 1
        turn = turn % 2

if game_over:
    pygame.time.wait(5000)
    pygame.quit()
    quit()
```