Using Minimax Algorithm to Make Connect 4 AI Agent

Aya Amr Ragaa	7793
Alia Essam	7810
Jaidaa Yassin	7360

Code Analysis

1. Imports:

```
import random
import string
import math
import sys
import numpy as np
from treelib import Node, Tree
import copy
from random import randint
from ctypes import sizeof
import time
import pygame
```

- random: Provides functions for generating random numbers and performing random selections.
- string: Offers functionalities related to string manipulation.
- math: Contains mathematical functions like sine, cosine, logarithm, etc.
- sys: Provides access to system-specific parameters and functions.
- numpy (np): NumPy is a powerful library for numerical computations and array manipulation. It's often used for efficient mathematical operations in games and AI.
- treelib (from treelib import Node, Tree): This library provides tools for working with tree structures. Node and Tree classes are commonly used to represent decision trees, search trees, or game states in AI algorithms.
- copy: Offers functions for creating copies of objects. This can be useful when you want to modify a game state without affecting the original one.
- time: Provides functions for measuring time and performing timed operations. This can be helpful for profiling algorithms and measuring their execution time.
- pygame: This library is specifically designed for creating games in Python. It offers functionalities for creating graphics, handling user input, managing sounds, and more.

2. Setting Up the Game Environment

```
BLUE = (0, 0, 205)
GREY = (128, 128, 128)
RED = (255, 0, 0)
YELLOW = (238, 238, 0)

pygame.init()
myfont = pygame.font.SysFont("Times", 40)

ROW_COUNT = 6
COLUMN_COUNT = 7

COMPUTER = 1
PLAYER = 2

# Set up the screen
screen_width = 400
screen_height = 400
screen = pygame.display.set_mode((screen_width, screen_height))
pygame.display.set_caption("Choose Minimax Algorithm")

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GREEN = (0, 255, 0)
RED = (255, 0, 0)

# Fonts
font = pygame.font.Font(None, 36)

# Button properties
button_width = 150
button_height = 50
button_x = (screen_width - button_width) // 2
minimax_button_y = 50
alphabeta_button_y = 125
expectiminimax_button_y = 200
```

3. Visualizing the Connect Four Board and Player Checkers

```
def draw_button(x, y, text):
    pygame.draw.rect(screen, WHITE, (x, y, button_width, button_height))
    text_surface = font.render(text, True, BLACK)
    text_rect = text_surface.get_rect(
        center=(x + button_width // 2, y + button_height // 2)
    )
    screen.blit(text_surface, text_rect)


def draw_GUI(screen, board):
    board = int_2_array(board)
    np.flip(board, 0)
    color = GREY
    for c in range(COLUMN_COUNT):
        for r in range(ROW_COUNT):
            pygame.draw.rect(
                screen,
                BLUE,
                (c * SQUARESIZE, (r + 1) * SQUARESIZE, SQUARESIZE,
SQUARESIZE),
            )
            pygame.draw.circle(
                screen,
                color,
                (
                    (c * SQUARESIZE + SQUARESIZE // 2),
                    (r * SQUARESIZE + SQUARESIZE + SQUARESIZE // 2),
                ),
                RADIUS,
            )

    for c in range(COLUMN_COUNT):
        for r in range(ROW_COUNT):
            if board[r][c] == 2:
                pygame.draw.circle(
                    screen,
                    RED,
                    (
                        (c * SQUARESIZE + SQUARESIZE // 2),
                        (r * SQUARESIZE + SQUARESIZE // 2 + SQUARESIZE),
                    ),
                    RADIUS,
                )
            elif board[r][c] == 1:
                pygame.draw.circle(
                    screen,
                    YELLOW,
                    (
                        (c * SQUARESIZE + SQUARESIZE // 2),
                        (r * SQUARESIZE + SQUARESIZE // 2 + SQUARESIZE),
                    ),
                    RADIUS,
                )
    pygame.display.update()
    board = array_2_int(board)
```

4. Packing a Connect Four Board State into an Integer

```
def array_2_int(array):
    state_int = 0b1
    top = 0

    for j in range(7):
        top = 0

        for i in range(6):
            piece = array[i][j]

            if piece == 0:
                top = i + 1

            if piece == 1:
                piece = 0

            if piece == 2:
                piece = 1

            state_int = state_int << 1
            state_int = state_int | piece

        state_int = state_int << 3
        state_int = state_int | top
    return state_int
```

- This function efficiently encodes the state of a Connect Four into a single integer:

Initialize variables:

- state_int is set to 0b1 (binary 1), acting as a starting point for building the integer representation.
- top is initialized to 0 to keep track of the top empty row within a column.

Iterate through columns:

- For each column (j) in the board:
- Reset top to 0 for the current column.
- Iterate through rows (i) within the column:
- Get the piece at the current position (piece = array[i][j]).
- If the piece is empty (0):
- Update top to the current row index (i + 1), indicating the top empty row in that column.
- Reassign piece to 0 if it represents the computer player (1) or 1 if it represents the player (2), creating a binary representation for the pieces.

Encode piece information:

- Shift the state_int 1 bit to the left (state_int << 1) to make space for the next piece.
- Combine state_int with the current piece using a bitwise OR operation (state_int | piece), effectively packing the piece data into the integer.

Encode top row information:

- Shift the state_int 3 bits to the left (state_int << 3) to create space for encoding the top empty row.
- Combine state_int with the top value for that column using a bitwise OR operation (state_int | top), packing the top row information.

Return encoded board state:

- After processing all columns, the function returns the state_int, which now compactly represents the entire board state, including piece positions and top empty rows.

5. Decoding a Packed Connect Four Board State into an Array

```python
def int_2_array(state_int):
    # print("int_2_array")
    state_array = [
        [0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0],
    ]

    copy_state_int = state_int

    for j in range(6, -1, -1):
        top = copy_state_int & 0b111
        copy_state_int = copy_state_int >> 3

        for i in range(5, -1, -1):

            if i < top:
                state_array[i][j] = 0

            else:
                piece = copy_state_int & 0b1

                if piece == 0:
                    state_array[i][j] = 1

                else:
                    state_array[i][j] = 2
            copy_state_int = copy_state_int >> 1

    return state_array
```

- This function unpacks the integer representation of a Connect Four board state back into a 2D array representing the actual board configuration.

Initialize variables:

- state_array: A pre-defined 2D array of size (6x7) filled with zeros, representing the empty board.
- copy_state_int: A copy of the original state_int to avoid modifying the original value.

Iterate through columns (reverse order):

- Loops through columns (j) from bottom (index 6) to top (index -1) to maintain the correct placement of pieces in the 2D array.
- Extract the top row information:
- Use bitwise AND operation (&) with 0b111 (binary 7) to isolate the bottom 3 bits of copy_state_int, representing the top empty row index in the current column.
- Store the extracted top row value in top.
- Shift the remaining state information:
- Shift copy_state_int 3 bits to the right (>> 3) to discard the processed top row information and focus on piece data.

Iterate through rows (reverse order):

- Loops through rows (i) from bottom (index 5) to top (index -1) to correctly fill the board from the bottom up.
- Check if the current row (i) is less than the top row (top):
- If i is less than top, the slot is empty. Set the corresponding position in state_array to 0 (empty slot).
- Otherwise (i is greater than or equal to top):
- Extract the piece information:

- Use bitwise AND operation with 0b1 (binary 1) to isolate the least significant bit of copy_state_int, representing the piece type (0 or 1).
- Assign the piece type to the corresponding position in state_array:
- If the extracted bit is 0, set the position to 1 (computer's piece).
- If the extracted bit is 1, set the position to 2 (player's piece).
- Shift the remaining state information:
- Shift copy_state_int 1 bit to the right (>> 1) to discard the processed piece data and move to the next piece information.

Return the board array:

- After processing all columns and rows, the function returns the state_array, which now represents the unpacked board state with pieces placed according to the original integer encoding.

6. Determining the Playable Row in a Connect Four Column

```
def get_playable_row(state_int, col):
    copy_state_int = state_int
    copy_state_int = copy_state_int >> (6 - col) * 9
    top = copy_state_int & 0b111

    if top == 0:
        return None

    return top - 1
```

- This function efficiently extracts the lowest playable row within a specified column of a Connect Four board, given its integer representation (state_int):

Focus on the specified column:

- Creates a copy of state_int to avoid modifying the original value.

- Shifts copy_state_int to the right by a specific number of bits based on the column (col):

- Shifts (6 - col) * 9 bits to the right (>>), aligning the bits that encode the top empty row information for the column indicated by col.

- Isolates the top row information:

- Uses a bitwise AND operation with 0b111 to extract the bottom 3 bits of copy_state_int, representing the top empty row index for that column.

Handle a full column:

- If top is 0, it means the column is full, so the function returns None to indicate that no playable row exists in that column.

Return the playable row:

- If top is not 0, it returns top - 1 to provide the index of the lowest playable row within the specified column. This is because the top value itself indicates the first empty row, while the playable row is one below it.

7. Identifying Playable Columns on a Connect Four Board

```
def get_playable_columns(state_int):
    available = []

    for i in range(7):

        if get_playable_row(state_int, i) != None:
            available.append(i)

    return available
```

- This function identifies the columns where a piece can be played on a Connect Four board, given its integer representation (state_int):

Initialize an empty list:

- Creates an empty list named available to store the indices of playable columns.

Iterate through columns:

- Loops through each column (i) from 0 to 6:
- Calls get_playable_row(state_int, i) to check if a playable row exists within the current column.
- If the row returned by get_playable_row is not None, it means the column is not full and a piece can be placed there.
- Adds the index of the playable column (i) to the available list.

Return playable columns:

- Returns the available list containing the indices of all columns where a piece can be played based on the current board state.

8. Retrieving a Checker at a Specific Position on the Connect Four Board

```
def get_checker(state_int, row, col):
    top = get_playable_row(state_int, col)

    if top != None or row < top + 1:
        return 0

    copy_state_int = state_int
    copy_state_int = copy_state_int >> ((6 - col) * 9 + 3 + 5 - row)
    piece = copy_state_int & 0b1
    return piece + 1
```

- This function extracts the type of checker (player or computer) located at a given row and column on the Connect Four board, represented as an integer (state_int):

Check for validity:

- Uses get_playable_row(state_int, col) to determine the top empty row in the specified column.

- If the returned top row is not None (column is not full) or if the requested row (row) is lower than the top empty row plus 1, it means the requested position is not occupied by a checker. Returns 0 to indicate an empty slot.

Extract piece information:

- Creates a copy of state_int to avoid modifying the original value.
- Shifts copy_state_int to the right by a specific number of bits to align with the bits encoding the piece at the requested position:
- The specific shift amount is calculated as (6 - col) * 9 + 3 + 5 - row, taking into account both the column and row indices within the integer representation.
- Isolates the piece information:
- Uses a bitwise AND operation with 0b1 to extract the least significant bit of copy_state_int, representing the piece type (0 for computer, 1 for player).

Return player or computer:

- Adds 1 to the extracted piece value to return 1 for the computer's checker or 2 for the player's checker, aligning with the conventions used in other functions.

9. Updating the Board State After Dropping a Checker in Connect Four

```
def drop_checker(state_int, col, piece):
    copy_state_int = state_int
    masking = 0b111111111
    masking = masking << (6 - col) * 9
    column = copy_state_int & masking
    copy_state_int = copy_state_int ^ column
    column = column >> (6 - col) * 9
    top = column & 0b111

    if piece == 2:
        setting = 0b1
        setting = setting << 3 + (6 - top)
        column = column | setting

    column = column ^ top
```

```
top = top - 1
column = column | top
column = column << (6 - col) * 9
copy_state_int = copy_state_int | column
return copy_state_int
```

- This function simulates dropping a checker (player or computer) into a specified column on the Connect Four board, updating the integer representation (state_int) of the board state:

Make a copy and prepare a mask:

- Creates a copy of state_int to avoid modifying the original value.
- Defines a mask (masking) filled with 1s, used later to isolate the specific column for manipulation. It's shifted based on the column (col) to align with the relevant bits in state_int.

Isolate the target column:

- Performs a bitwise AND operation between copy_state_int and masking. This isolates the bits representing the target column (col) while setting other columns to 0.

Clear existing pieces in the column (optional):

- The commented lines (# print...) suggest there might be an optional step to clear existing pieces in the column before placing the new checker. This functionality is not explicitly implemented in the provided code snippet.

Update top row information:

- Isolates the top row information within the target column using a bitwise AND operation with 0b111.

Set the new checker based on player type:

- Checks if the piece is for the player (piece == 2).

- If so, defines a setting value (setting) with the least significant bit set to 1 (representing the player's checker).
- Shifts the setting left by 3 + (6 - top) to position it at the appropriate row within the column based on the top empty row (top).
- Performs a bitwise OR operation between the existing column data (column) and the setting to set the bit representing the new checker.

Combine top row and updated column:

- Performs a bitwise XOR operation between the updated column (column) and the top row information (top). This ensures the top row bits are not overwritten by the new checker information.
- Subtracts 1 from top to adjust the top empty row index after placing the checker.
- Performs another bitwise OR operation between the updated column (column) and the adjusted top row (top) to combine the information.

Shift and integrate changes:

- Shifts the updated column data (column) left by (6 - col) * 9 bits to place it back in the correct position within the overall state_int representation.
- Performs a bitwise OR operation between the original copy_state_int and the updated column data (column) to integrate the changes into the overall board state.

Return the updated state:

- Returns the modified copy_state_int which now reflects the updated board state after dropping the checker.

10. Counting Potential Winning Lines of Two for a Player in Connect Four

```
def get_twos(board, turn):
    twos = 0
    board = int_2_array(board)
```

```python
    for i in range(len(board)):

        for j in range(len(board[0])):

            if j < len(board[0]) - 3:
                # horizontal right
                if (
                    board[i][j] == board[i][j + 1] == turn
                    and board[i][j + 2] == board[i][j + 3] == 0
                ):
                    twos += 1

                if i < len(board) - 3:
                    # diagonal right down
                    if (
                        board[i][j] == board[i + 1][j + 1] == turn
                        and board[i + 3][j + 3] == board[i + 2][j + 2] ==0
                    ):
                        twos += 1

            if j >= 3:
                # horizontal left
                if (
                    board[i][j] == board[i][j - 1] == turn
                    and board[i][j - 3] == board[i][j - 2] == 0
                ):
                    twos += 1

                if i < len(board) - 3:
                    # diagonal left down
                    if (
                        board[i][j] == board[i + 1][j - 1] == turn
                        and board[i + 3][j - 3] == board[i + 2][j - 2] ==0
                    ):
                        twos += 1

            if i >= 3:
                # vertical
                if (
                    board[i][j] == board[i - 1][j] == turn
                    and board[i - 3][j] == board[i - 2][j] == 0
                ):
                    twos += 1

    return twos
```

- This function counts the number of patterns on the Connect Four board where the specified player (turn) has two checkers in a row, with two empty spaces for a potential win:

Initialize variables:

- Sets twos to 0 to store the count of potential winning lines.
- Converts the integer board representation (board) into a 2D array using int_2_array for easier manipulation.

Iterate through the board:

- Loops through each row (i) and column (j) of the board array.

Check for potential winning lines:

- For each position (i, j), it checks for four specific patterns:

1. Horizontal Right:

- Ensures board[i][j] and board[i][j + 1] both contain the player's piece (turn).
- Verifies that board[i][j + 2] and board[i][j + 3] are empty (0).

2. Diagonal Right Down:

- Checks if board[i][j] and board[i + 1][j + 1] match the player's piece.
- Ensures board[i + 3][j + 3] and board[i + 2][j + 2] are empty.

3. Horizontal Left:

- Similar logic as Horizontal Right, but checking towards the left (board[i][j - 1] and board[i][j - 2]).

4. Diagonal Left Down:

- Similar logic as Diagonal Right Down, but checking towards the left and down (board[i + 1][j - 1], board[i + 3][j - 3]).

5. Vertical:

- Checks for a potential vertical win (board[i][j], board[i - 1][j], board[i - 3][j]).

Increment count:

- If a pattern is found for any of these directions, it increments the twos counter.

Return the count:

- Returns the total number of potential winning lines found for the specified player.

11. Counting Winning Lines of Three for a Player in Connect Four

```
def get_threes(board, turn):
    threes = 0
    board = int_2_array(board)

    for i in range(len(board)):

        for j in range(len(board[0])):

            if j < len(board[0]) - 3:
                # horizontal right
                if (
                    board[i][j] == board[i][j + 1] == board[i][j + 2] ==
turn
```

```
                    and board[i][j + 3] == 0
                ):
                    threes += 1

                if i < len(board) - 3:
                    # diagonal right down
                    if (
                        board[i][j]
                        == board[i + 1][j + 1]
                        == board[i + 2][j + 2]
                        == turn
                        and board[i + 3][j + 3] == 0
                    ):
                        threes += 1

            if j >= 3:
                # horizontal left
                if (
                    board[i][j] == board[i][j - 1] == board[i][j - 2] ==
turn
                    and board[i][j - 3] == 0
                ):
                    threes += 1

                if i < len(board) - 3:
                    # diagonal left down
                    if (
                        board[i][j]
                        == board[i + 1][j - 1]
                        == board[i + 2][j - 2]
                        == turn
                        and board[i + 3][j - 3] == 0
                    ):
                        threes += 1

            if i >= 3:
                # vertical
                if (
                    board[i][j] == board[i - 1][j] == board[i - 2][j] ==
turn
                    and board[i - 3][j] == 0
                ):
                    threes += 1

        return threes
```

- This function counts the number of patterns on the Connect Four board where the specified player (turn) has three checkers in a row, with 1 empty spaces for a potential win:

12. Identifying Winning Lines of Four for a Player in Connect Four

```
def get_fours(board, turn):
    fours = 0
    board = int_2_array(board)

    for i in range(len(board)):

        for j in range(len(board[0])):

            if j < len(board[0]) - 3:
                # horizontal right
                if (
                    board[i][j]
                    == board[i][j + 1]
                    == board[i][j + 2]
                    == turn
                    == board[i][j + 3]
                ):
                    fours += 1

                if i < len(board) - 3:
                    # diagonal right down
                    if (
                        board[i][j]
                        == board[i + 1][j + 1]
                        == board[i + 2][j + 2]
                        == board[i + 3][j + 3]
                        == turn
                    ):
                        fours += 1

            if j >= 3:

                if i < len(board) - 3:
                    # diagonal left down
                    if (
                        board[i][j]
                        == board[i + 1][j - 1]
                        == board[i + 2][j - 2]
                        == turn
                        == board[i + 3][j - 3]
                    ):
                        fours += 1

            if i >= 3:
                # vertical
                if (
                    board[i][j]
                    == board[i - 1][j]
                    == board[i - 2][j]
                    == turn
                    == board[i - 3][j]
                ):
                    fours += 1

    return fours
```

- This function determines the number of winning lines formed by four consecutive checkers of the specified player (turn) on the Connect Four board, represented as a 2D array (board):

13. Evaluating Board State for the AI Player in Connect Four

```
def calculate_score(board):
    player_fours = get_fours(board, PLAYER)
    player_three = get_threes(board, PLAYER)
    player_two = get_twos(board, PLAYER)
    AI_fours = get_fours(board, COMPUTER)
    AI_three = get_threes(board, COMPUTER)
    AI_two = get_twos(board, COMPUTER)
    return (
        11 * (AI_fours - player_fours)
        + (AI_three - player_three) * 7
        + (AI_two - player_two) * 3
    )
```

- This function calculates a score representing the current board state's favorability for the AI player in Connect Four:

Calculate component scores:

- Calls get_fours(board, PLAYER) and get_fours(board, COMPUTER) to determine the number of winning lines of four for both players.
- Similarly, calls get_threes and get_twos for both players to find the number of potential winning lines with three and two checkers in a row, respectively.

Weighted scoring:

- Uses a weighted sum to calculate the overall score:
- Winning lines of four (both for AI and player) are weighted heavily (11 points). This emphasizes the importance of achieving or preventing four-in-a-row configurations.
- Winning lines of three (7 points) and potential winning lines of two (3 points) receive lower weights, reflecting their decreasing likelihood of leading to a win immediately.
- The score is calculated by:
- Subtracting the number of winning lines of four for the player from the AI's winning lines of four (favoring the AI if it has more fours).
- Similarly, subtracting the player's threes and twos from the AI's threes and twos, respectively.
- A positive score indicates the board state favors the AI, while a negative score suggests the player is in a better position.

Return the score:

- Returns the calculated score, which can be used by the AI to make strategic decisions during gameplay.

14. Determining the Winner and Game Over Status in Connect Four

```
def if_game_ended(board):

    if len(get_playable_columns(board)) != 0:
        return False, 0

    playerfours = get_fours(board, PLAYER)
    print(f"PLayer fours: {playerfours}")
    AIfours = get_fours(board, COMPUTER)
    print(f"COMPUTER fours: {AIfours}")

    if playerfours == AIfours:
        return True, 0

    elif playerfours > AIfours:
        return True, PLAYER

    else:
        return True, COMPUTER
```

- This function checks for the game's end state and identifies the winner (if any) in Connect Four based on the board representation (board):

Check for available moves:

- Calls get_playable_columns(board), which likely returns a list of available columns for placing checkers.
- If there are still empty columns (len(get_playable_columns(board)) != 0), the game is not over (returns False, 0). This indicates there are legal moves remaining.

Check for winning lines of four:

- If there are no playable columns, the game might be a tie or a win for one player.
- Calls get_fours(board, PLAYER) and get_fours(board, COMPUTER) to determine the number of winning lines of four for both players.
- Prints the number of winning fours for both players.

Evaluate winning conditions:

- Based on the counts of winning fours:
- If both players have the same number of fours (a tie), return True, 0. This signifies the game is over with a draw.
- If the player has more fours than the AI (playerfours > AIfours), return True, PLAYER. This indicates the player has won the game.
- Otherwise (else), return True, COMPUTER. This signifies the AI has won the game.

Return Values:

- The function returns a tuple containing two elements:
    1. True if the game is over, False otherwise.
    2. The winner (either PLAYER or COMPUTER) if the game is over, or 0 for a tie.

15. Minimax Algorithm for AI Decision-Making in Connect Four

```
def minimax(board, depth, alpha, beta, Maximizing, p, search_tree, pr):
    pruning = 0
    availabe_colomns = get_playable_columns(board)
    terminal, winner = if_game_ended(board)
    if depth == 0 or terminal:
        if terminal:
            if winner == COMPUTER:
                if pr == 1:
                    stringg = f"terminal, alpha: {alpha} beta: {beta} 
score: 1000000"
                else:
                    stringg = f"terminal, score: 1000000"
                return (None, 1000000)
            if winner == PLAYER:
                if pr == 1:
                    stringg = f"terminal, alpha: {alpha} beta: {beta} 
score: -1000000"
                else:
                    stringg = f"terminal, score: -1000000"
                return (None, -1000000)
            else:
                if pr == 1:
                    stringg = f"terminal, alpha: {alpha} beta: {beta} 
score: 0"
                else:
                    stringg = f"terminal, score: 0"

                return (None, 0)

        else:
            if pr == 1:
                stringg = f"maximum depth,alpha: {alpha} beta: {beta} 
score: {calculate_score(board)}"
            else:
                stringg = f"maximum depth, score: 
{calculate_score(board)}"

            return (None, calculate_score(board))
    if Maximizing:
        pruning = 0
        maxscore = -math.inf
        colomn = 0
        for j in availabe_colomns:
            temp_board = board
            temp_board = drop_checker(temp_board, j, COMPUTER)
            if pr == 1:
                stringg = (
                    f"Maximizing node, alpha: {alpha} beta: {beta} score: 
{maxscore}"
                )
            else:
                stringg = f"Maximizing node,score: {maxscore}"
            S = 10
            ran = "".join(random.choices(string.ascii_uppercase + 
string.digits, k=S))
            search_tree.create_node(stringg, ran, parent=p)
            new_score = minimax(
                temp_board, depth - 1, alpha, beta, False, ran, 
search_tree, pr
            )[1]
            if new_score > maxscore:
                maxscore = new_score
                colomn = j
            alpha = max(alpha, maxscore)
            if alpha >= beta and pr == 1:
                pruning = 1
                break
        if pr == 1:
            if pruning == 1:
                stringg = f"Maximizing node (pruning occured), alpha:
{alpha} beta: {beta} score: {maxscore}"

                search_tree.update_node(ran, tag=stringg)
            else:
                stringg = f"Maximizing node (no pruning occured), alpha:
{alpha} beta: {beta} score: {maxscore}"

                search_tree.update_node(ran, tag=stringg)
        else:
            stringg = f"Maximizing node, score: {maxscore}"

            search_tree.update_node(ran, tag=stringg)
        if pr == 2:
            if colomn != 0 and colomn != 6:
                if colomn - 1 in availabe_colomns and colomn + 1  not in
availabe_colomns:
                    prob = random.random()
                    if prob < 0.2:
                        colomn = colomn - 1
                        temp_board = drop_checker(temp_board, colomn,
PLAYER)

                        maxscore = calculate_score(temp_board)
                    elif prob > 0.8:
                        colomn = colomn + 1
                        temp_board = drop_checker(temp_board, colomn,
PLAYER)

                        maxscore = calculate_score(temp_board)
            elif colomn == 0 and colomn + 1  in availabe_colomns:
                prob = random.random()
                if prob < 0.4:
                    colomn = colomn + 1
                    temp_board = drop_checker(temp_board, colomn, PLAYER)
                    maxscore = calculate_score(temp_board)
            elif  colomn == 6 and colomn - 1 in availabe_colomns:
                prob = random.random()
                if prob < 0.4:
                    colomn = colomn - 1
                    temp_board = drop_checker(temp_board, colomn, PLAYER)
                    maxscore = calculate_score(temp_board)
        return colomn, maxscore
    # Minimizing
    else:
        pruning = 0
        minscore = math.inf
        colomn = 0
        for j in availabe_colomns:
            temp_board = board
            temp_board = drop_checker(temp_board, j, PLAYER)
            if pr == 1:
                stringg = (
                    f"Minimizing node, alpha: {alpha} beta: {beta} score:
{minscore}"
                )
            else:
                stringg = f"Minimizing node, score: {minscore}"
            S = 10  # number of characters in the string.
            ran = "".join(random.choices(string.ascii_uppercase +
string.digits, k=S))
            search_tree.create_node(stringg, ran, parent=p)
            new_score = minimax(
                temp_board, depth - 1, alpha, beta, True, ran,
search_tree, pr
            )[1]
            if new_score < minscore:
                minscore = new_score
                colomn = j
            beta = min(beta, minscore)
            if alpha >= beta and pr == 1:
                pruning = 1
                break
        if pr == 1:
            if pruning == 1:
                stringg = f"Minimzing node (pruning occured), alpha:
{alpha} beta: {beta} score: {minscore}"
                search_tree.update_node(ran, tag=stringg)
            else:
                stringg = f"Minimizing node (no pruning occured), alpha:
{alpha} beta: {beta} score: {minscore}"
                search_tree.update_node(ran, tag=stringg)
        else:
            stringg = f"Minimizing node, score: {minscore}"
            search_tree.update_node(ran, tag=stringg)
        if pr == 2:
            if colomn != 0 and colomn != 6:
```

```
                if colomn - 1 in availabe_colomns and colomn + 1  not in 
availabe_colomns:
                    prob = random.random()
                    if prob < 0.2:
                        colomn = colomn - 1
                        temp_board = drop_checker(temp_board, colomn, 
PLAYER)
                        minscore = calculate_score(temp_board)
                    elif prob > 0.8:
                        colomn = colomn + 1
                        temp_board = drop_checker(temp_board, colomn, 
PLAYER)
                        minscore = calculate_score(temp_board)
            elif colomn == 0 and colomn + 1  in availabe_colomns:
                prob = random.random()
                if prob < 0.4:
                    colomn = colomn + 1
                    temp_board = drop_checker(temp_board, colomn, PLAYER)
                    minscore = calculate_score(temp_board)
            elif  colomn == 6 and colomn - 1 in availabe_colomns:
                prob = random.random()
                if prob < 0.4:
                    colomn = colomn - 1
                    temp_board = drop_checker(temp_board, colomn, PLAYER)
                    minscore = calculate_score(temp_board)
        return colomn, minscore
```

- This function implements the minimax algorithm with alpha-beta pruning for the AI player in Connect Four:

Parameters:

- board: Represents the current board state as a 2D array.
- depth: The current depth of the search tree (used to limit the number of recursive calls).
- alpha: The minimum score the maximizing player (AI) is guaranteed to achieve.
- beta: The maximum score the minimizing player (player) is guaranteed to achieve.
- Maximizing: Boolean flag indicating whether the current node is a maximizing (AI) or minimizing (player) node.

- p: Parent node ID in the search tree.
- search_tree: A data structure for storing the search process and potentially visualizing it.
- pr: A flag for controlling the amount of printing done (0- minmax without pruning, 1 - minmax with pruning , 2 – Expected minmax).

Base Cases:

- If the depth reaches zero or the game ends (if_game_ended), the function returns:
- For a terminal state:
- The winner (COMPUTER or PLAYER) and a score reflecting the outcome (1000000 for AI win, -1000000 for player win, or 0 for a tie).
- For the maximum depth:
- The current board score (representing the AI's evaluation of the current state).

Available Moves:

- Calls get_playable_columns(board) to identify the legal columns where the AI can place its checker.

Maximizing Player (AI):

- Initializes maxscore to negative infinity and column to store the best move.
- Iterates through the available columns:
- Creates a temporary board (temp_board).
- Simulates the AI's move by dropping a checker in the current column using drop_checker(temp_board, j, COMPUTER).
- Recursively calls minimax on the temporary board, simulating the opponent's (player's) best move:

- Decrements the depth to explore one level deeper in the search tree.
- Sets Maximizing to False as the next level represents the minimizing player.
- Passes the updated alpha and beta values for pruning.
- The returned score (new_score) represents the minimax value from the opponent's perspective (minimizing the AI's score).
- Updates maxscore and column if the new score is better (higher) for the AI (maximizing player).
- Applies alpha-beta pruning:
- If alpha (AI's guaranteed minimum) is greater than or equal to beta (player's guaranteed maximum), it implies the opponent (player) has a move that guarantees a score worse than what the AI can already achieve. No need to explore further branches from this node. The pruning flag is set to 1 to indicate pruning occurred.

Minimizing Player (Player):

- Similar logic as the maximizing player, but with the following differences:
- Initializes minscore to positive infinity.
- Minimizes the score (new_score) returned from the recursive call, representing the AI's best move from the player's perspective.
- Updates beta (player's guaranteed maximum) if the new score is lower.

Expected Minmax (for pr=2):

- If pr is set to 2, introduces some randomness in move selection for the AI, potentially making the gameplay more unpredictable.
- With a small probability, it might choose an adjacent column instead of the recommended move from the minimax search.

Return Value:

- Returns the chosen column (column) and the corresponding minimax score, representing the AI's evaluation of the best move considering its opponent's (player's) potential responses.

16. Visualizing the Connect Four Board State

```python
def print_arr(arr):
    print("\n\n\n\n\n\n\n\n\n\n\n")

    for row in range(6):
        print("-" * 35)

        for col in range(7):

            if arr[row][col] == 0:
                print("|    |", end="")

            else:
                print("| " + str(arr[row][col]) + " |", end="")
        print()
    print("-" * 35)
```

- This function, print_arr(arr), is designed to create a visually appealing representation of the Connect Four board state for the user:

Clear the Screen:

- Prints multiple newlines at the beginning to potentially clear the console output and provide a clean slate for the board visualization.

Draw Top Border:

- Prints a horizontal line of dashes (-) repeated 35 times to represent the top border of the board.

Iterate Through Rows:

- Loops through each row (row) of the board array (arr):
- Prints another horizontal line of dashes for row separation.
- Loops through each column (col) within the current row:
- Checks the value at arr[row][col]:
- If it's 0 (empty space), prints "| |" to represent an empty slot on the board.
- Otherwise, prints "| " followed by the value (either "1" for player or "2" for AI) and another "|" to enclose the checker's representation.
- Adds a space at the end of each column representation for better formatting.

Draw Bottom Border:

- After iterating through all rows and columns, prints a final horizontal line of dashes to represent the bottom border of the board.

17. AI Selection and Game Loop for Connect Four with Minimax

```
running = True
minimax_selected = False
alphabeta_selected = False
expected_selected = False

while running:
    screen.fill(WHITE)

    # Draw buttons
    draw_button(button_x, minimax_button_y, "Minimax")
    draw_button(button_x, alphabeta_button_y, "Alpha-Beta")
    draw_button(button_x, expectiminimax_button_y, "Expectiminimax")

    # Check for events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
```

```
                pygame.quit()
                sys.exit()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                mouse_pos = pygame.mouse.get_pos()
                if button_x <= mouse_pos[0] <= button_x + button_width:
                    if minimax_button_y <= mouse_pos[1] <= minimax_button_y +
button_height:
                        minimax_selected = True
                        running = False
                    elif (
                        alphabeta_button_y
                        <= mouse_pos[1]
                        <= alphabeta_button_y + button_height
                    ):
                        alphabeta_selected = True
                        running = False
                    elif (
                        expectiminimax_button_y
                        <= mouse_pos[1]
                        <= expectiminimax_button_y + button_height
                    ):
                        expected_selected = True
                        running = False

        # Highlight selected button
        if minimax_selected:
            pygame.draw.rect(
                screen, RED, (button_x, minimax_button_y, button_width,
button_height), 3
            )
            prune = 0
        elif alphabeta_selected:
            pygame.draw.rect(
                screen, RED, (button_x, alphabeta_button_y, button_width,
button_height), 3
            )
            prune = 1
        elif expected_selected:
            pygame.draw.rect(
                screen,
                RED, (button_x, expectiminimax_button_y, button_width,
button_height), 3
            )
            prune = 2
        pygame.display.update()
```

```python
if minimax_selected or alphabeta_selected or expected_selected:
    board =
0b1000000011000000011000000011000000011000000011000000011000000011000000110

    SQUARESIZE = 90
    width = COLUMN_COUNT * SQUARESIZE
    height = (ROW_COUNT + 1) * SQUARESIZE
    RADIUS = SQUARESIZE // 2 - 5

    size = (width, height)

    screen = pygame.display.set_mode(size)
    draw_GUI(screen, board)
    pygame.display.update()
    col = 0
    game_over = False
    K = int(input("Enter number of levels: "))
    if K > 0 and (prune == 1) or (prune == 0) or (prune == 2):
        while True:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    sys.exit()

                if event.type == pygame.MOUSEMOTION:
                    pygame.draw.rect(screen, GREY, (0, 0, width,
SQUARESIZE))
                    posx = event.pos[0]
                    pygame.draw.circle(screen, RED, (posx, SQUARESIZE //
2), RADIUS)
                    pygame.display.update()

                if event.type == pygame.MOUSEBUTTONDOWN:
                    pygame.draw.rect(screen, GREY, (0, 0, width,
SQUARESIZE))
                    posx = event.pos[0]
                    col = posx // SQUARESIZE
                    status = if_game_ended(board)
                    if get_playable_row(board, col) != None:
                        board = drop_checker(board, col, 2)
                        print_arr(int_2_array(board))
                        draw_GUI(screen, board)
                        search_tree = Tree()
                        search_tree.create_node("Parent", 0)
                        start_time = time.time()
                        nextcol, score = minimax(
                            board, K, -math.inf, math.inf, True, 0,
search_tree, prune
                        )
                        end_time = time.time()
                        runtime = end_time - start_time
                        print(f"Runtime of minmax = {runtime} seconds")
                        number_of_nodes = search_tree.size() - 1
                        print(f"number of nodes {number_of_nodes}")
                        stringg = f"Max score:{score}, Next colomn:
{nextcol}"
                        search_tree.update_node(0, tag=stringg)
                        search_tree.show()
                        if nextcol == None:
                            if score == 1000000:
                                print("COMPUTER WINS")
                                label = myfont.render("COMPUTER Wins !!!",
1, YELLOW)

                                screen.blit(label, (130, 100))
                                draw_GUI(screen, board)
                                pygame.time.wait(3000)
                                quit()
                            elif score == -1000000:
                                print("PLAYER WINS")
                                label = myfont.render("Player 2 Wins !!!",
1, RED)

                                screen.blit(label, (130, 10))
                                draw_GUI(screen, board)
                                pygame.time.wait(3000)
                                quit()
                            else:
                                print("TIE")
                                label = myfont.render("TIW !!!", 1,
YELLOW)

                                screen.blit(label, (130, 10))
                                draw_GUI(screen, board)
                                pygame.time.wait(3000)
                                quit()
                        else:
                            board = drop_checker(board, nextcol, COMPUTER)
                            print_arr(int_2_array(board))
                            draw_GUI(screen, board)
                            status = if_game_ended(board)
                            if status[0] == True:
                                if status[1] == COMPUTER:
                                    print("COMPUTER WINS ")
                                    label = myfont.render(
                                        "COMPUTER Wins !!!", 1, YELLOW
                                    )
                                    screen.blit(label, (130, 10))
                                    draw_GUI(screen, board)
                                    pygame.time.wait(3000)
                                    quit()
                                elif status[1] == PLAYER:
                                    print("PLAYER WINS")
                                    label = myfont.render("Player 2 Wins
!!!", 1, RED)

                                    screen.blit(label, (130, 10))
                                    draw_GUI(screen, board)
                                    pygame.time.wait(3000)
                                    quit()
                                else:
                                    print("TIE")
                                    label = myfont.render("TIE !!!", 1,
YELLOW)

                                    screen.blit(label, (130, 10))
                                    draw_GUI(screen, board)
                                    pygame.time.wait(3000)
                                    quit()
```

- This code implements the core logic for selecting the AI search algorithm and running the Connect Four game loop:

Initialization:

- running flag controls the main game loop.
- Boolean flags (minimax_selected, alphabeta_selected, and expected_selected) track which search algorithm the user has chosen.

Main Loop:

- The loop continues as long as running is True.

- Fills the screen with white (screen.fill(WHITE)) to clear any previous visuals.

- Button Drawing:

  - Calls functions (draw_button) to visually represent the available search algorithms (Minimax, Alpha-Beta, and Expectiminimax) as buttons on the screen.

- Event Handling:

  - Iterates through Pygame events (pygame.event.get()) to capture user interaction.
  - Exits the game loop if the user closes the window (event.type == pygame.QUIT).
  - Checks for mouse clicks (event.type == pygame.MOUSEBUTTONDOWN).
  - For each button, it compares the mouse click position with the button's location and size.
  - If a button is clicked, the corresponding flag (minimax_selected, alphabeta_selected, or expected_selected) is set to True, and running is set to False to exit the selection loop.

- Highlighting Selected Button:

  - Depending on which search algorithm is selected (based on the previously set flags), a red rectangle is drawn around the corresponding button to provide visual feedback to the user.
  - A value (prune) is set based on the selection (0 for Minimax, 1 for Alpha-Beta, and 2 for Expectiminimax).
  - Updates the display (pygame.display.update()) to reflect any changes made to the screen.

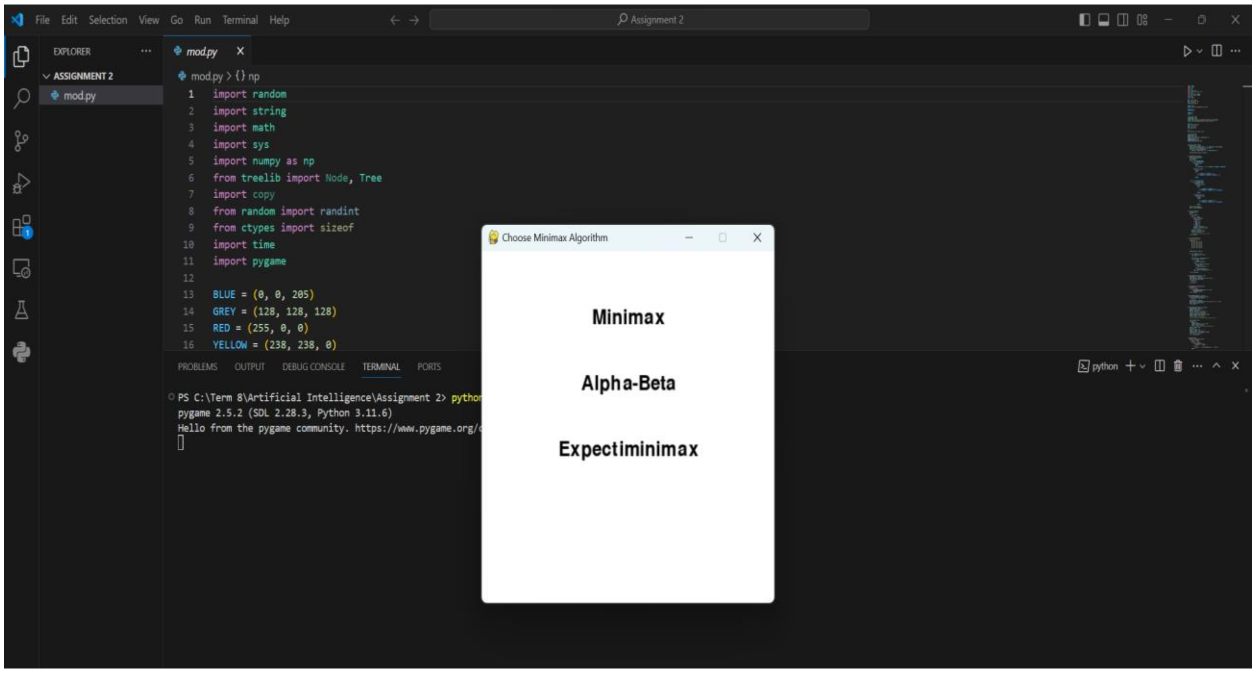
Game Loop:

- Assuming a search algorithm is selected (minimax_selected or alphabeta_selected or expected_selected is True), the game loop proceeds:

- Initializes the game board (board).
- Sets game constants like square size (SQUARESIZE), screen dimensions (width and height), and circle radius (RADIUS).
- Initializes the Pygame display (screen) and calls a function (draw_GUI) to draw the game board on the screen.
- Prompts the user to enter the number of levels (K) for the minimax search.
- If K is valid and a valid pruning option is selected (prune value), another loop starts:
- Handles mouse movement events (event.type == pygame.MOUSEMOTION) to visually indicate the potential column for the next AI move (using a red circle).
- Handles mouse click events (event.type == pygame.MOUSEBUTTONDOWN) to:
- Register the clicked column (col).
- Check if the chosen column is a valid move (get_playable_row(board, col)).
- If valid, update the game board with the AI's move (board = drop_checker(board, col, 2)).
- Print the board state (print_arr(int_2_array(board))).
- Redraw the game board (draw_GUI(screen, board)).
- Creates a search tree object (search_tree).
- Initializes the search tree with a root node (search_tree.create_node("Parent", 0)).
- Starts measuring the execution time (start_time = time.time()).
- Calls the minimax search function (minimax) to determine the best move for the AI:
- It considers the current board (board), number of levels (K), alpha and beta values (for pruning), maximizing player (True for AI), current depth (0), search tree (search_tree), and pruning option (prune).
- Stops measuring execution time (end_time = time.time()).
- Calculates the execution runtime (runtime).

- Counts the number of nodes explored in the search tree (number_of_nodes).
- Updates the root node of the search tree with additional information (search_tree.update_node(0, tag=stringg)) like the minimax score and chosen column.
- Displays the search tree (search_tree.show()).
- Checks the game state using if_game_ended(board):
- If the game has ended (status[0] == True), it determines the winner (status[1]) and displays a corresponding message on the screen.
- The game exits after a short delay (pygame.time.wait(3000)) to allow the user to see the final outcome.
- If the game hasn't ended and the chosen column (col) is not None (indicating a valid move), the AI places its checker (board = drop_checker(board, nextcol, COMPUTER)).
- Prints the board state (print_arr(int_2_array(board))).
- Redraws the game board (draw_GUI(screen, board)).
- Checks the game state again using if_game_ended(board).
- If the game has ended, similar actions are taken as before to display the winner and exit the game.

Sample Runs



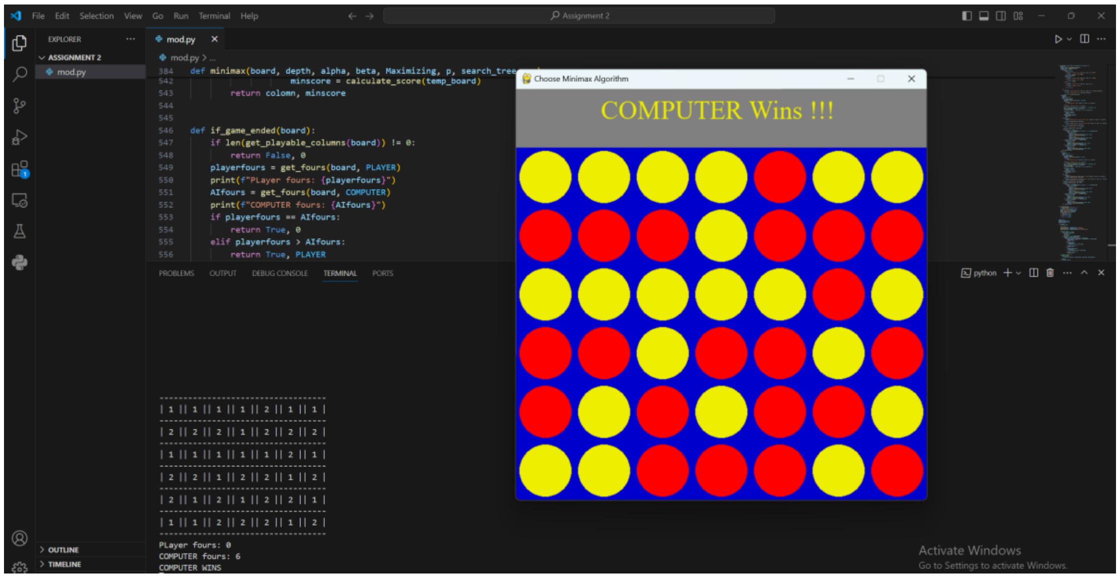
```python
import random
import string
import math
import sys
import numpy as np
from treelib import Node, Tree
import copy
from random import randint
from ctypes import sizeof
import time
import pygame

BLUE = (0, 0, 205)
GREY = (128, 128, 128)
RED = (255, 0, 0)
YELLOW = (238, 238, 0)
```

```
PS C:\Term 8\Artificial Intelligence\Assignment 2> python
pygame 2.5.2 (SDL 2.28.3, Python 3.11.6)
Hello from the pygame community. https://www.pygame.org/c
```

At first, you choose the algorithm you want to use.

A Sample Run Of MinMax (Without Pruning):

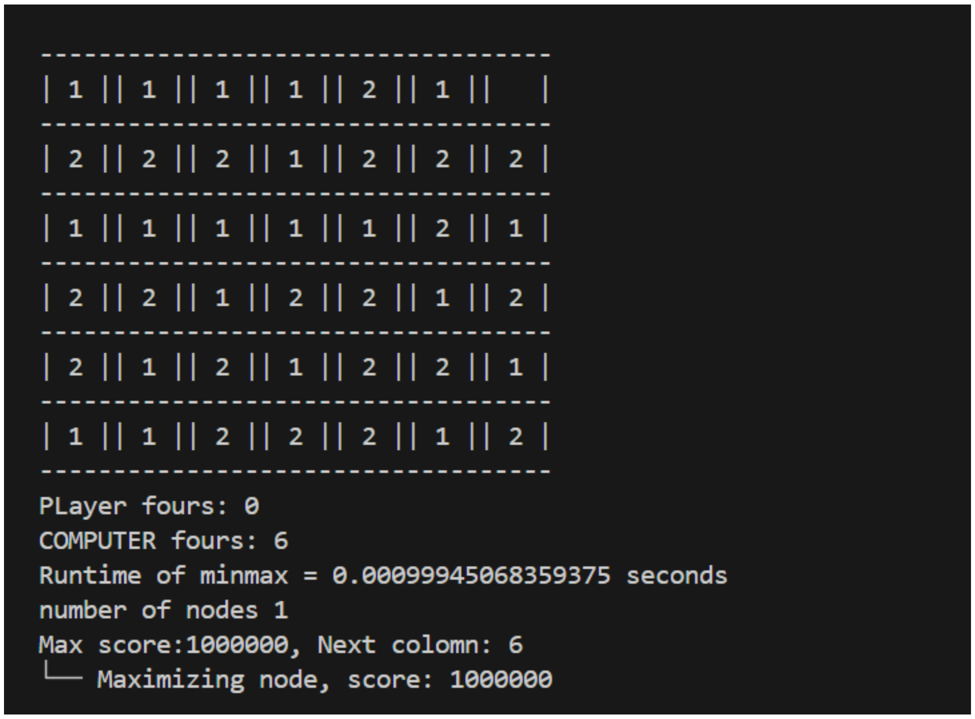


```python
def minimax(board, depth, alpha, beta, Maximizing, p, search_tree
                        minscore = calculate_score(temp_board)
            return colomn, minscore


def if_game_ended(board):
    if len(get_playable_columns(board)) != 0:
        return False, 0
    playerfours = get_fours(board, PLAYER)
    print(f"PLayer fours: {playerfours}")
    AIfours = get_fours(board, COMPUTER)
    print(f"COMPUTER fours: {AIfours}")
    if playerfours == AIfours:
        return True, 0
    elif playerfours > AIfours:
        return True, PLAYER
```

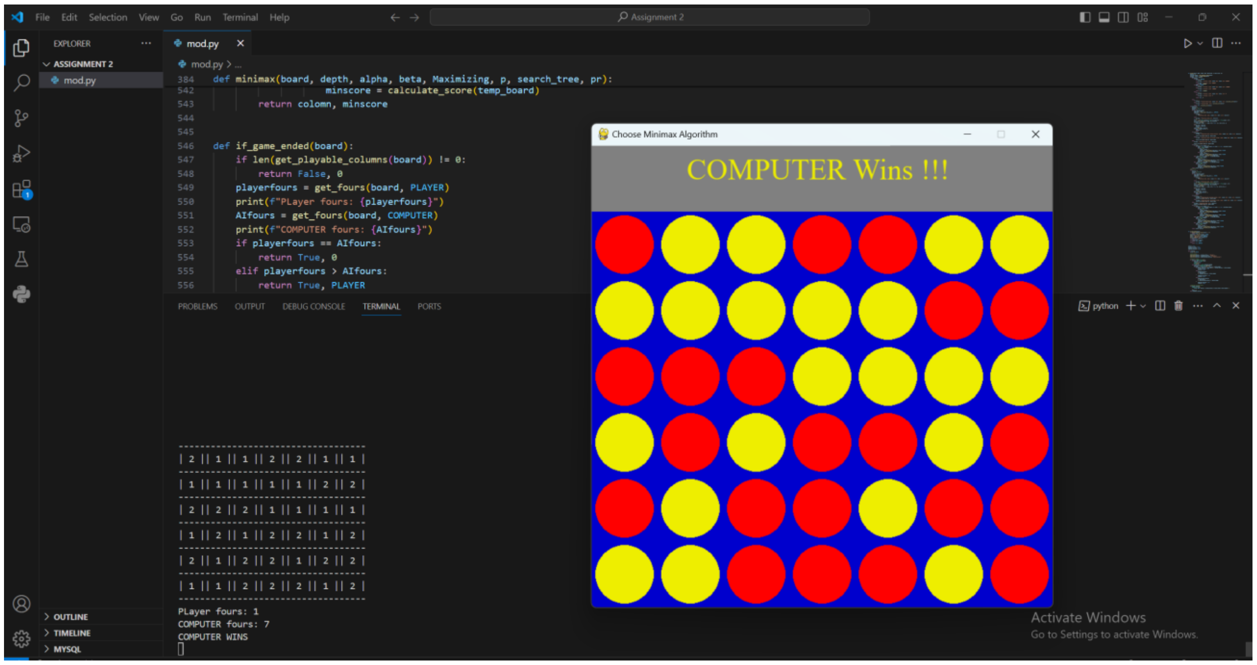
```
-----------------------------------
| 1 || 1 || 1 || 1 || 2 || 1 || 1 |
-----------------------------------
| 2 || 2 || 2 || 1 || 2 || 2 || 2 |
-----------------------------------
| 1 || 1 || 1 || 1 || 1 || 2 || 1 |
-----------------------------------
| 2 || 2 || 1 || 2 || 2 || 1 || 2 |
-----------------------------------
| 2 || 1 || 2 || 1 || 2 || 2 || 1 |
-----------------------------------
| 1 || 1 || 2 || 2 || 2 || 1 || 2 |
-----------------------------------
PLayer fours: 0
COMPUTER fours: 6
COMPUTER WINS
```

The Last Tree Before Game Ends



```
-----------------------------------
| 1 || 1 || 1 || 1 || 2 || 1 ||   |
-----------------------------------
| 2 || 2 || 2 || 1 || 2 || 2 || 2 |
-----------------------------------
| 1 || 1 || 1 || 1 || 1 || 2 || 1 |
-----------------------------------
| 2 || 2 || 1 || 2 || 2 || 1 || 2 |
-----------------------------------
| 2 || 1 || 2 || 1 || 2 || 2 || 1 |
-----------------------------------
| 1 || 1 || 2 || 2 || 2 || 1 || 2 |
-----------------------------------
PLayer fours: 0
COMPUTER fours: 6
Runtime of minmax = 0.00099945068359375 seconds
number of nodes 1
Max score:1000000, Next colomn: 6
└── Maximizing node, score: 1000000
```

A Sample Run Of Alpha-Beta Pruning (Without Pruning):

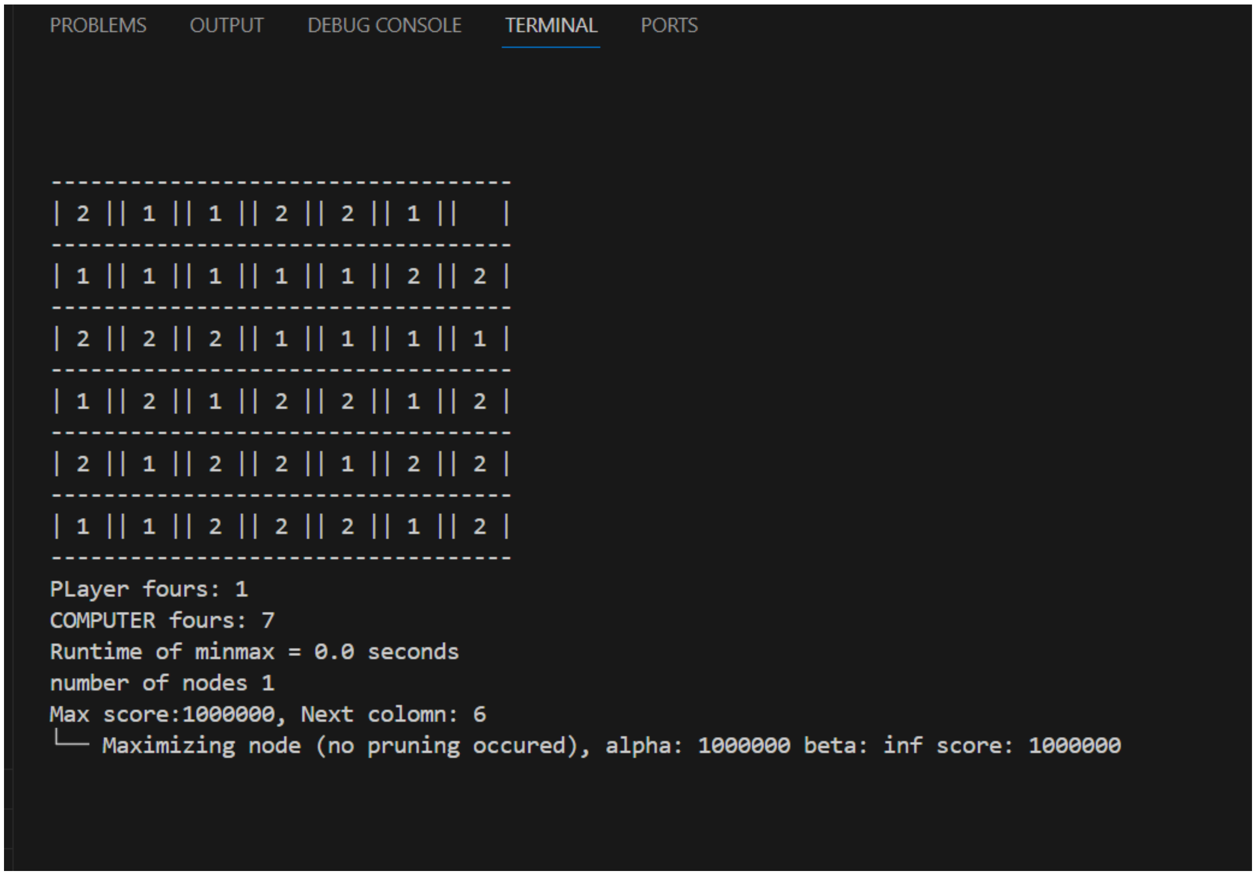


```python
def minimax(board, depth, alpha, beta, Maximizing, p, search_tree, pr):
                minscore = calculate_score(temp_board)
        return colomn, minscore


def if_game_ended(board):
    if len(get_playable_columns(board)) != 0:
        return False, 0
    playerfours = get_fours(board, PLAYER)
    print(f"PLayer fours: {playerfours}")
    AIfours = get_fours(board, COMPUTER)
    print(f"COMPUTER fours: {AIfours}")
    if playerfours == AIfours:
        return True, 0
    elif playerfours > AIfours:
        return True, PLAYER
```

```
-----------------------------------
| 2 || 1 || 1 || 2 || 2 || 1 || 1 |
-----------------------------------
| 1 || 1 || 1 || 1 || 1 || 2 || 2 |
-----------------------------------
| 2 || 2 || 2 || 1 || 1 || 1 || 1 |
-----------------------------------
| 1 || 2 || 1 || 2 || 2 || 1 || 2 |
-----------------------------------
| 2 || 1 || 2 || 2 || 1 || 2 || 2 |
-----------------------------------
| 1 || 1 || 2 || 2 || 2 || 1 || 2 |
-----------------------------------
PLayer fours: 1
COMPUTER fours: 7
COMPUTER WINS
```

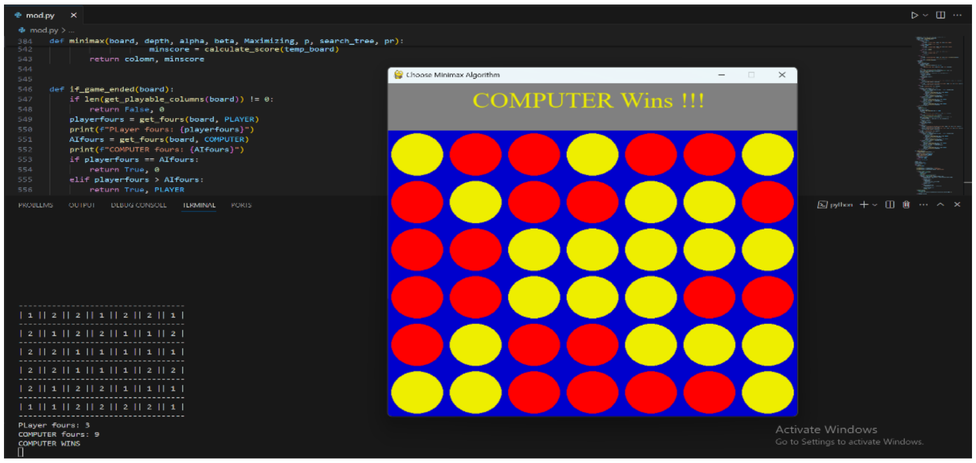
Last tree before game ends:



```
-----------------------------------
| 2 || 1 || 1 || 2 || 2 || 1 ||   |
-----------------------------------
| 1 || 1 || 1 || 1 || 1 || 2 || 2 |
-----------------------------------
| 2 || 2 || 2 || 1 || 1 || 1 || 1 |
-----------------------------------
| 1 || 2 || 1 || 2 || 2 || 1 || 2 |
-----------------------------------
| 2 || 1 || 2 || 2 || 1 || 2 || 2 |
-----------------------------------
| 1 || 1 || 2 || 2 || 2 || 1 || 2 |
-----------------------------------
PLayer fours: 1
COMPUTER fours: 7
Runtime of minmax = 0.0 seconds
number of nodes 1
Max score:1000000, Next colomn: 6
└── Maximizing node (no pruning occured), alpha: 1000000 beta: inf score: 1000000
```

As we can see, the alpha beta pruning is way faster then the min max because it doesn't have to explore the whole depth of the tree.

Sample Run for The expected MinMax:

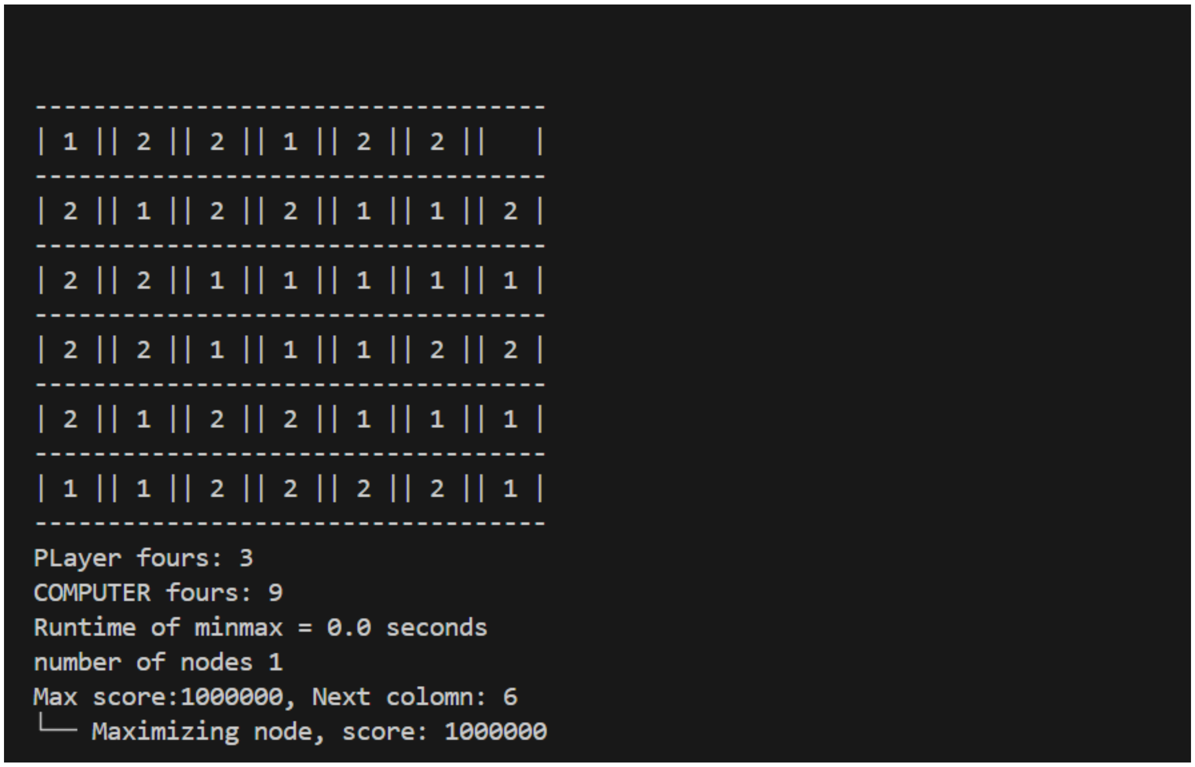


```python
def minimax(board, depth, alpha, beta, Maximizing, p, search_tree, pr):
                    minscore = calculate_score(temp_board)
        return colomn, minscore


def if_game_ended(board):
    if len(get_playable_columns(board)) != 0:
        return False, 0
    playerfours = get_fours(board, PLAYER)
    print(f"PLayer fours: {playerfours}")
    AIfours = get_fours(board, COMPUTER)
    print(f"COMPUTER fours: {AIfours}")
    if playerfours == AIfours:
        return True, 0
    elif playerfours > AIfours:
        return True, PLAYER
```

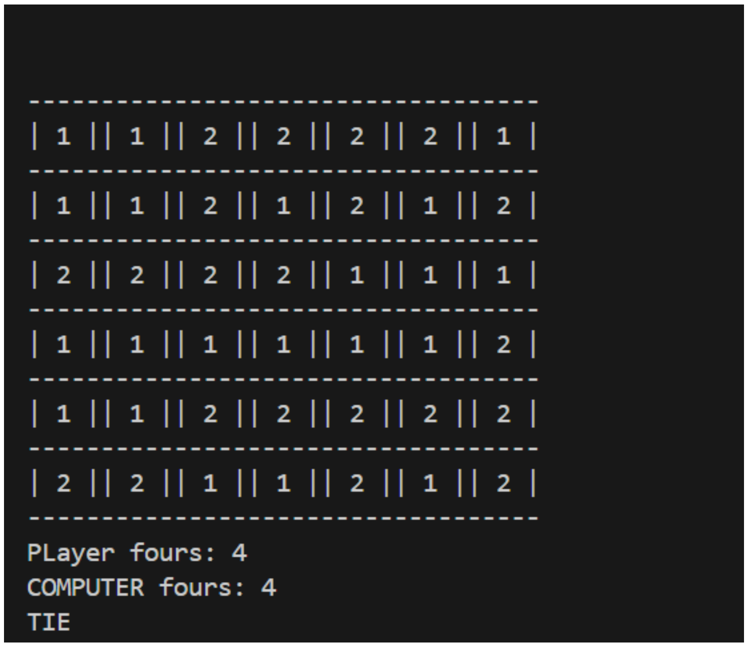
```
-----------------------------------
| 1 || 2 || 2 || 1 || 2 || 2 || 1 |
-----------------------------------
| 2 || 1 || 2 || 2 || 1 || 1 || 2 |
-----------------------------------
| 2 || 2 || 1 || 1 || 1 || 1 || 1 |
-----------------------------------
| 2 || 2 || 1 || 1 || 1 || 2 || 2 |
-----------------------------------
| 2 || 1 || 2 || 2 || 1 || 1 || 1 |
-----------------------------------
| 1 || 1 || 2 || 2 || 2 || 2 || 1 |
-----------------------------------
PLayer fours: 3
COMPUTER fours: 9
COMPUTER WINS
```

Sample of The Last Tree:



```
-----------------------------------
| 1 || 2 || 2 || 1 || 2 || 2 ||   |
-----------------------------------
| 2 || 1 || 2 || 2 || 1 || 1 || 2 |
-----------------------------------
| 2 || 2 || 1 || 1 || 1 || 1 || 1 |
-----------------------------------
| 2 || 2 || 1 || 1 || 1 || 2 || 2 |
-----------------------------------
| 2 || 1 || 2 || 2 || 1 || 1 || 1 |
-----------------------------------
| 1 || 1 || 2 || 2 || 2 || 2 || 1 |
-----------------------------------
PLayer fours: 3
COMPUTER fours: 9
Runtime of minmax = 0.0 seconds
number of nodes 1
Max score:1000000, Next colomn: 6
└── Maximizing node, score: 1000000
```

Sample for the Tie

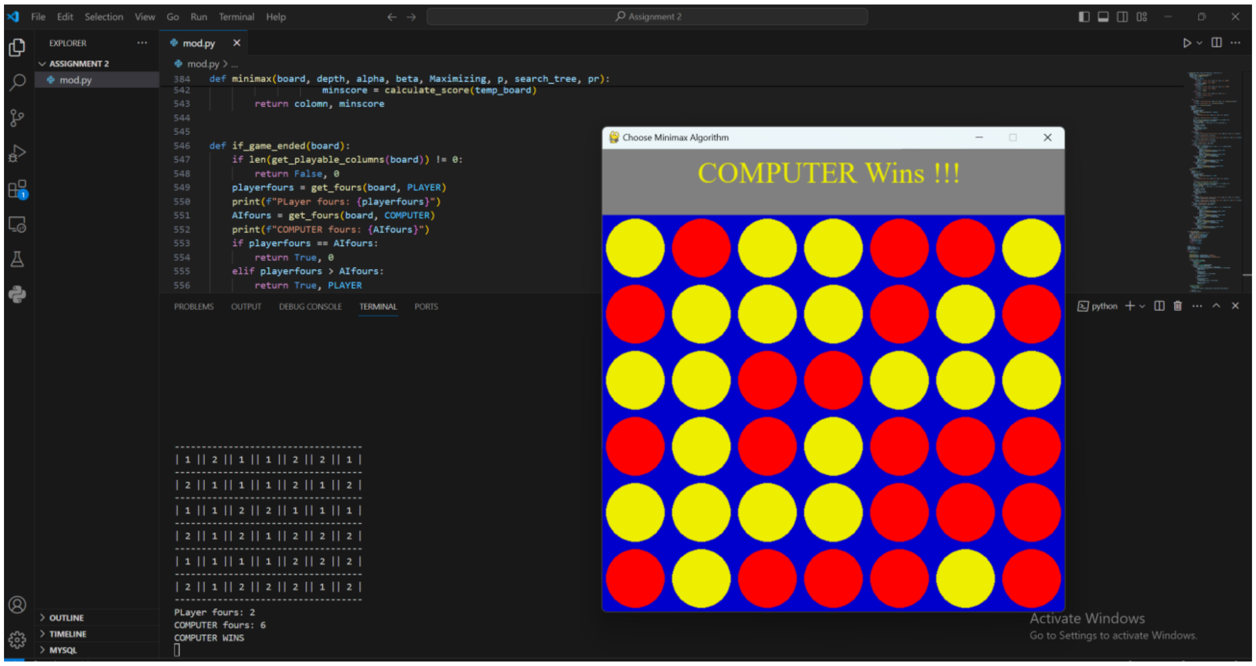


```
-----------------------------------
| 1 || 1 || 2 || 2 || 2 || 2 || 1 |
-----------------------------------
| 1 || 1 || 2 || 1 || 2 || 1 || 2 |
-----------------------------------
| 2 || 2 || 2 || 2 || 1 || 1 || 1 |
-----------------------------------
| 1 || 1 || 1 || 1 || 1 || 1 || 2 |
-----------------------------------
| 1 || 1 || 2 || 2 || 2 || 2 || 2 |
-----------------------------------
| 2 || 2 || 1 || 1 || 2 || 1 || 2 |
-----------------------------------
PLayer fours: 4
COMPUTER fours: 4
TIE
```

Depth K	MinMax	Alpha Beta Pruning
4	9ms & 280 nodes	About 0 & 98 nodes
5	0.0015 seconds & 2315 nodes	0.0016 seconds & 1807 nodes
3	0.01 seconds & 399	0.01 seconds & 119 nodes

At Depth 5:

Min Max

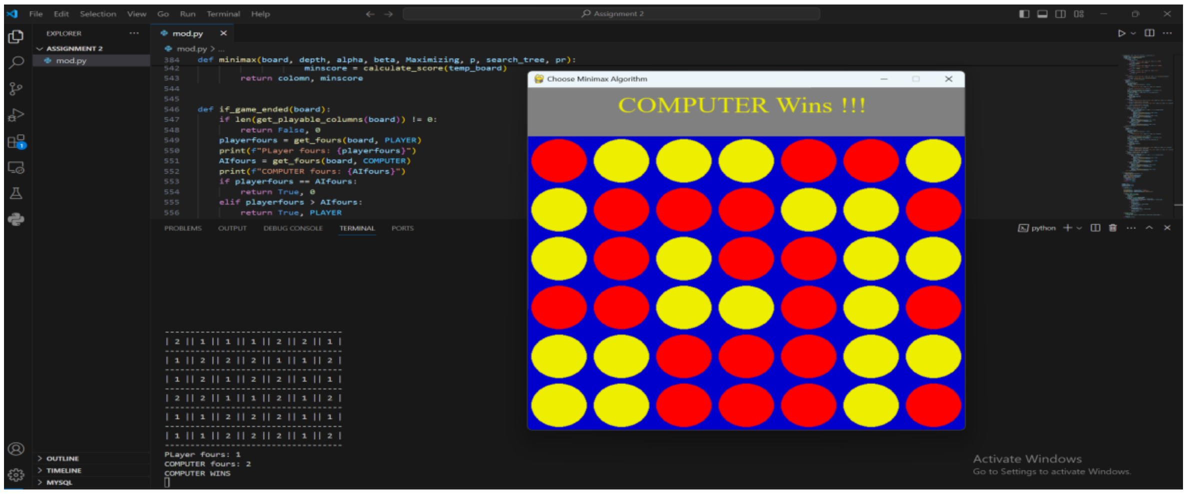


```
def minimax(board, depth, alpha, beta, Maximizing, p, search_tree, pr):
                    minscore = calculate_score(temp_board)
        return colomn, minscore


def if_game_ended(board):
    if len(get_playable_columns(board)) != 0:
        return False, 0
    playerfours = get_fours(board, PLAYER)
    print(f"PLayer fours: {playerfours}")
    AIfours = get_fours(board, COMPUTER)
    print(f"COMPUTER fours: {AIfours}")
    if playerfours == AIfours:
        return True, 0
    elif playerfours > AIfours:
        return True, PLAYER
```

```
-----------------------------------
| 1 || 2 || 1 || 1 || 2 || 2 || 1 |
-----------------------------------
| 2 || 1 || 1 || 1 || 2 || 1 || 2 |
-----------------------------------
| 1 || 1 || 2 || 2 || 1 || 1 || 1 |
-----------------------------------
| 2 || 1 || 2 || 1 || 2 || 2 || 2 |
-----------------------------------
| 1 || 1 || 1 || 1 || 2 || 2 || 2 |
-----------------------------------
| 2 || 1 || 2 || 2 || 2 || 1 || 2 |
-----------------------------------
PLayer fours: 2
COMPUTER fours: 6
COMPUTER WINS
```

Alpha Beta



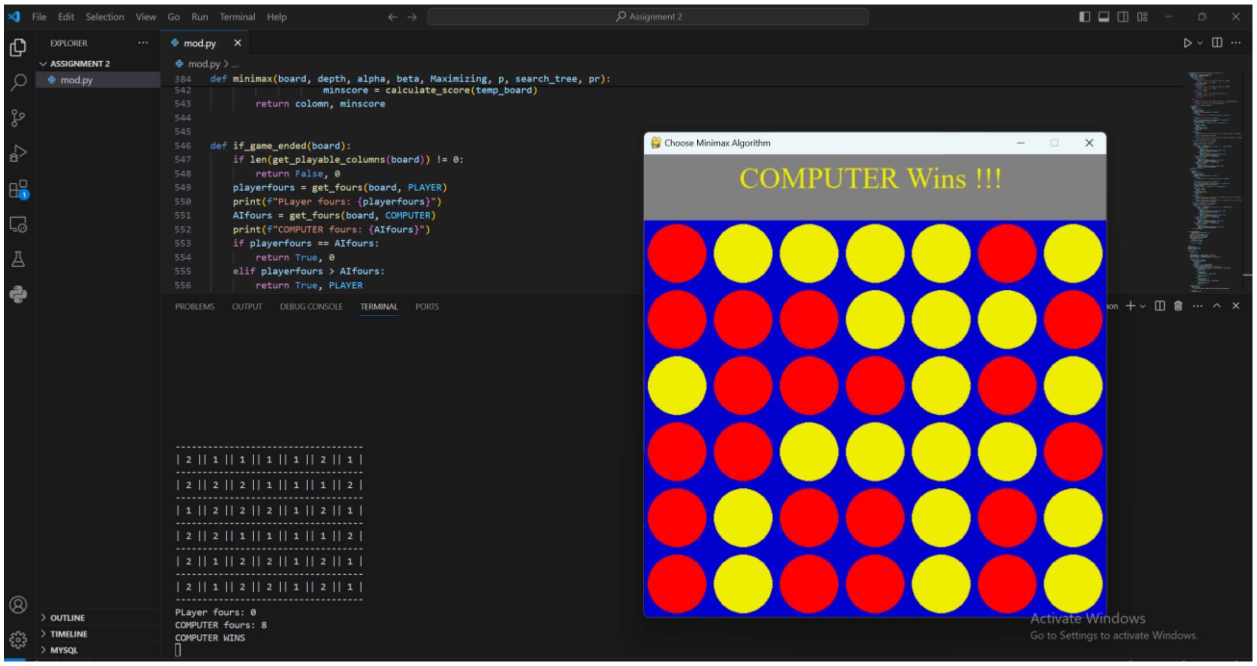
```
def minimax(board, depth, alpha, beta, Maximizing, p, search_tree, pr):
                    minscore = calculate_score(temp_board)
        return colomn, minscore


def if_game_ended(board):
    if len(get_playable_columns(board)) != 0:
        return False, 0
    playerfours = get_fours(board, PLAYER)
    print(f"PLayer fours: {playerfours}")
    AIfours = get_fours(board, COMPUTER)
    print(f"COMPUTER fours: {AIfours}")
    if playerfours == AIfours:
        return True, 0
    elif playerfours > AIfours:
        return True, PLAYER
```

```
-----------------------------------
| 2 || 1 || 1 || 1 || 2 || 2 || 1 |
-----------------------------------
| 1 || 2 || 2 || 2 || 1 || 1 || 2 |
-----------------------------------
| 1 || 2 || 1 || 2 || 2 || 1 || 1 |
-----------------------------------
| 2 || 2 || 1 || 1 || 2 || 1 || 2 |
-----------------------------------
| 1 || 1 || 2 || 2 || 2 || 1 || 1 |
-----------------------------------
| 1 || 1 || 2 || 2 || 2 || 1 || 2 |
-----------------------------------
PLayer fours: 1
COMPUTER fours: 2
COMPUTER WINS
```

At Depth 3:

Min Max

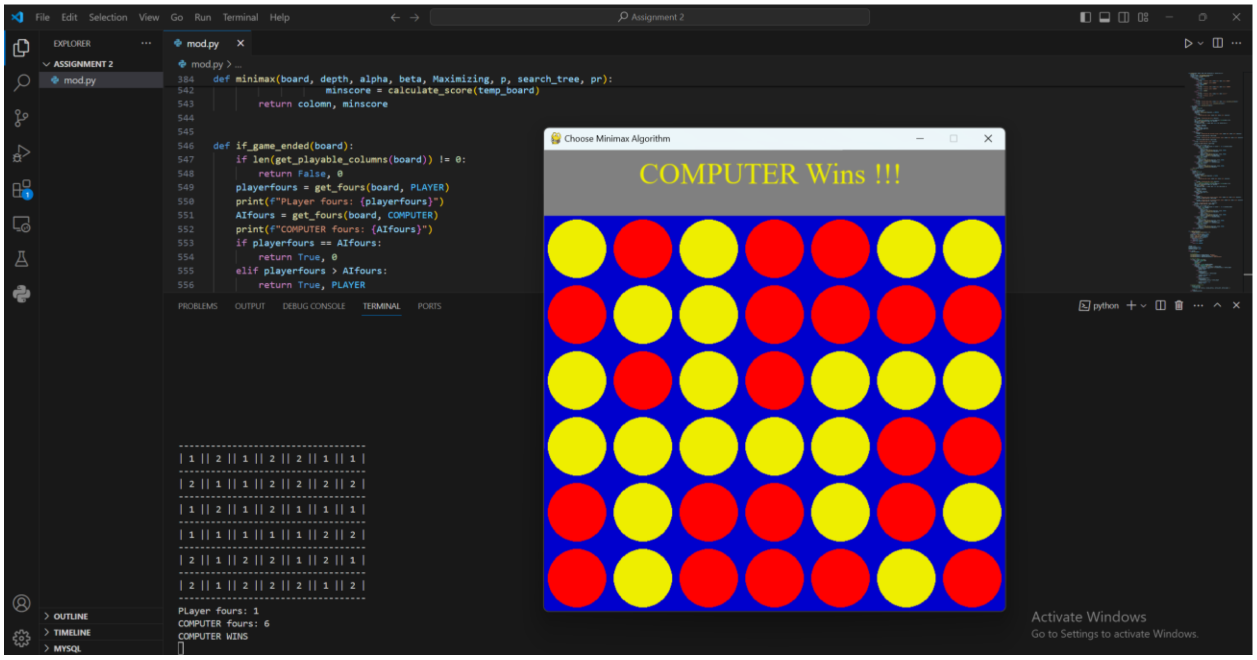


```python
def minimax(board, depth, alpha, beta, Maximizing, p, search_tree, pr):
                minscore = calculate_score(temp_board)
        return colomn, minscore


def if_game_ended(board):
    if len(get_playable_columns(board)) != 0:
        return False, 0
    playerfours = get_fours(board, PLAYER)
    print(f"PLayer fours: {playerfours}")
    AIfours = get_fours(board, COMPUTER)
    print(f"COMPUTER fours: {AIfours}")
    if playerfours == AIfours:
        return True, 0
    elif playerfours > AIfours:
        return True, PLAYER
```

```
-----------------------------------
| 2 || 1 || 1 || 1 || 1 || 2 || 1 |
-----------------------------------
| 2 || 2 || 2 || 1 || 1 || 1 || 2 |
-----------------------------------
| 1 || 2 || 2 || 2 || 1 || 2 || 1 |
-----------------------------------
| 2 || 2 || 1 || 1 || 1 || 1 || 2 |
-----------------------------------
| 2 || 1 || 2 || 2 || 1 || 2 || 1 |
-----------------------------------
| 2 || 1 || 2 || 2 || 1 || 2 || 1 |
-----------------------------------
PLayer fours: 0
COMPUTER fours: 8
COMPUTER WINS
```

Alpha Beta:



```python
def minimax(board, depth, alpha, beta, Maximizing, p, search_tree, pr):
                minscore = calculate_score(temp_board)
        return colomn, minscore


def if_game_ended(board):
    if len(get_playable_columns(board)) != 0:
        return False, 0
    playerfours = get_fours(board, PLAYER)
    print(f"PLayer fours: {playerfours}")
    AIfours = get_fours(board, COMPUTER)
    print(f"COMPUTER fours: {AIfours}")
    if playerfours == AIfours:
        return True, 0
    elif playerfours > AIfours:
        return True, PLAYER
```

```
-----------------------------------
| 1 || 2 || 1 || 2 || 2 || 1 || 1 |
-----------------------------------
| 2 || 1 || 1 || 2 || 2 || 2 || 2 |
-----------------------------------
| 1 || 2 || 1 || 2 || 1 || 1 || 1 |
-----------------------------------
| 1 || 1 || 1 || 1 || 1 || 2 || 2 |
-----------------------------------
| 2 || 1 || 2 || 2 || 1 || 2 || 1 |
-----------------------------------
| 2 || 1 || 2 || 2 || 2 || 1 || 2 |
-----------------------------------
PLayer fours: 1
COMPUTER fours: 6
COMPUTER WINS
```