TEMPLE UNIVERSITY

# AI Course Project Document: Artificial Intelligence-Based Connect 4 Player Using Python

Done by: Abdalaziz Sawwan

May. 05. 2021

Supervised by: Dr. Pei Wang

## Contents

# I. Introduction

Connect Four is a board game that is played by exactly two players, players in it are assigned to different colors and then take turns dropping colored discs into the suspended grid. The game's grid has seven columns and six rows. The pieces fall straight down, occupying the lowest available space. Figure 1 illustrates the game panel.

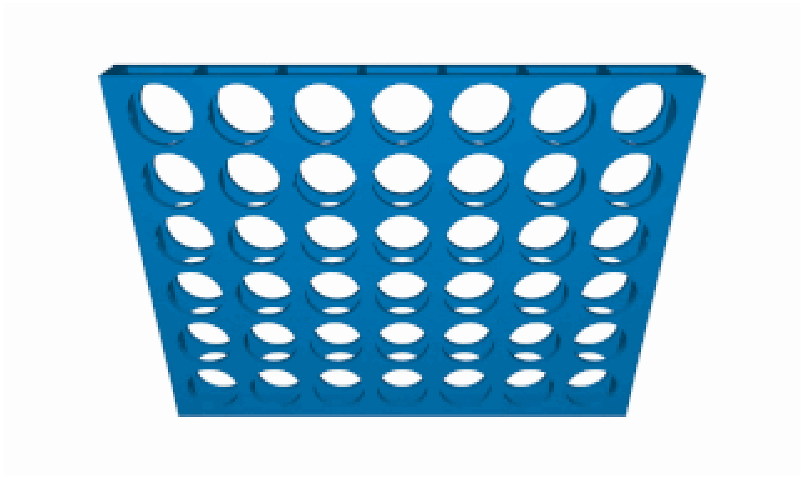


*Figure 1: An illustration of the game panel (Source in [1].)*

The main goal of the game is to be the first player to have either horizontal, vertical, or diagonal line of four same-colored discs. It is well known that Connect Four is a solved game, i.e. there is a specific known strategy by which the first player can always win by playing the correct plays. Hence, in this project, we try to play "semi-perfectly" against the AI and observe the results.

Furthermore, unlike most of the card games, Connect Four does provide full information, where when a player at a time plays, all the two players get all the information regarding moves that have deterministically already taken place and regarding all moves that can take place for the next step, in a given game state. This makes implementing an AI player of the game more feasible for the purpose of this project.

The project includes first, implementing the game environment itself in a suitable and user-friendly way, and second, an implantation of an AI player who we can configure its parameters and hardness. Finally, a general evaluation of the AI is presented.

## II. Theoretical Analysis

Starting from the standard game, we will have a panel of $6 \times 7 = 42$ locations, each location has three possible states that can have; being empty, having the disc of the first player, or having the disc of the second player. That would directly give a rough upper-bound of $3^{42} \approx 1.1 \times 10^{20}$ possible game situations that the search space of the game may have.

However, we can compute a more accurate and tight upper bound of the possible situations by evaluating the sequence of possible games corresponding to a specific number of discs played so far, and then, just summing up the elements of this sequence of numbers.

This sequence of possible situations after a number of turns will start from zero situations after zero turns, then seven situations after one turn (the first player has seven possible positions to place their disc on), then $7 \times 7 = 49$ positions after the second turn.

After the second turn, it becomes a little bit trickier, calculating the number of possible situations after three turns is not trivial; the number of possible situations at that point will be $7 \times 7 \times 1 + 7 \times 6 \times \frac{5}{2} + 7 \times 6 \times 2 = 238$ situations. Which is derived after considering that in some scenarios, different orders of placing the discs of the first player may produce the same situation of the game. Here, we consider all the cases.

The whole sequence of possible situations after a specific number of turns, and starting from zero is given by the following sequence: $1,\ 7,\ 49,\ 238,\ 1120,\ 4263,\ 16422,\ 54859,\ 184275,\ \ldots$ [2].

After calculating the number of situations after considering all 42 possible turns as well as the aero turn, we will end up with a total number of situations that is around $4.5 \times 10^{12}$ situations.

For a usual board game, an order of trillions of possible states in the search space is relatively in too large (like the search space in Chess or Go board games), and it is not too low that a simple brute-force algorithm that exhausts all the situations is feasible alone to do that. This proves that our choice to consider this game specifically was a good choice because implementing an AI agent that does not trivially exhausts all the possible scenarios would be useful in order to play against in this game.

## III. Related Work

Tommy *et al.* [3] have demonstrated that a suitable optimal AI algorithm is still unknown for the Connect Four game, they applied two different AI algorithms; one that mainly utilizes the alpha-beta pruning algorithm, and one utilizes the MTD(f) Algorithm (Memory-enhanced Test Driver Algorithm).

They consider in their research the optimality, which is the winning percentage in general, the speed, which is related to the execution time, and finally the number of leaf nodes. Their results state that their AI models win in less than 50% of games with a reasonable execution time.

Sarhan *et al.* [4] have presented a design that converts the standard Connect Four game into a real-time game by incorporating time restrains. They have designed their artificial intelligence model based on influence mapping. In their work, a waterfall-based AI software has been developed for the game. Their main result was to successfully design their software using C++ programming language.

Finally, Galli [5], who is a prominent online instructive videos maker about different topics in computer science has discussed an implementation of an AI model for the Connect Four problem and the implementation of the game itself. In our project here, we in some instances, use similar techniques that are used in his lectures. This is because of their relatively-high efficiency.

## IV. The General Employed AI Techniques

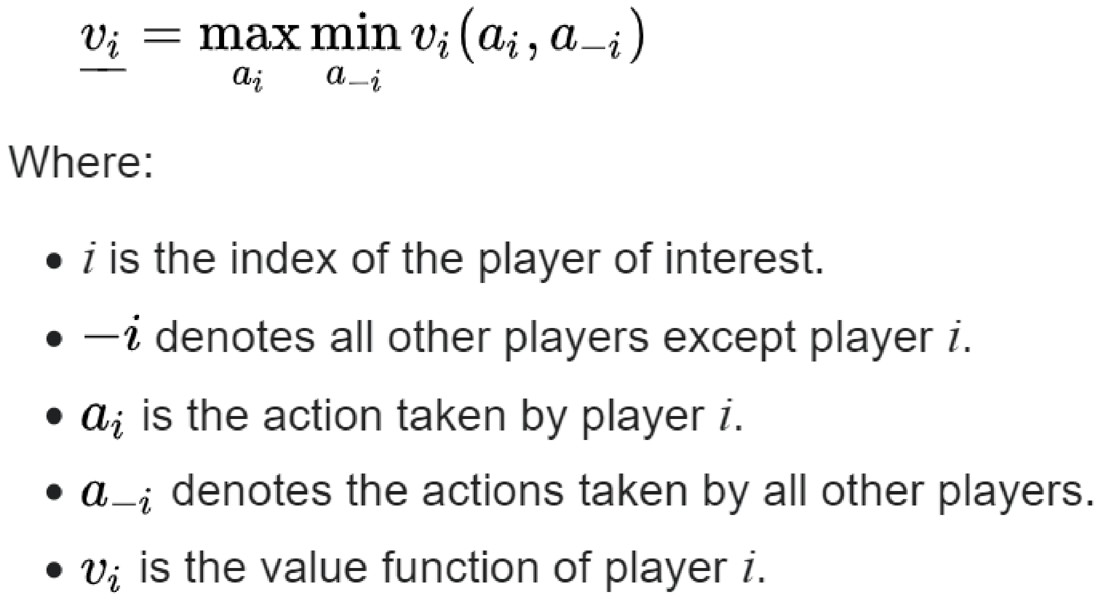
In this section, we demonstrate a brief summary of the theory in of the main techniques used in our AI model for this project.

### IV.I The Minimax Algorithm

We implement an AI that mainly employs the minimax algorithm that we have learned in the class. The algorithm is simply implemented by making the players try to optimize some utility function that they have.

One player will try to maximize their utility function, which we will call the maximizing player. And the other player will try to minimize their utility function, which we will call the minimizing player.

We can choose the utility functions for each player to be the complement of the other, so that, we would have a zero-sum game that is easier and more feasible to be implemented using minimax algorithm. Figure 2 shows the algorithm from a game-theoretic perspective.



$$\underline{v_i} = \max_{a_i} \min_{a_{-i}} v_i(a_i, a_{-i})$$

Where:

- $i$ is the index of the player of interest.
- $-i$ denotes all other players except player $i$.
- $a_i$ is the action taken by player $i$.
- $a_{-i}$ denotes the actions taken by all other players.
- $v_i$ is the value function of player $i$.

*Figure 2: Game theoretic perspective of the minimax objective function. (Source [6].)*

For the exact way the algorithm works, it is better to illustrate using an example; consider the simple example in Figure 3.

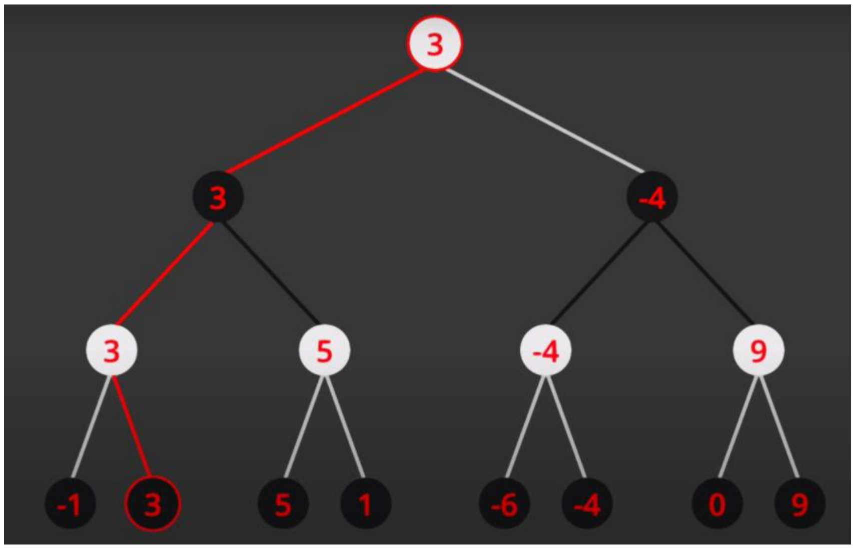
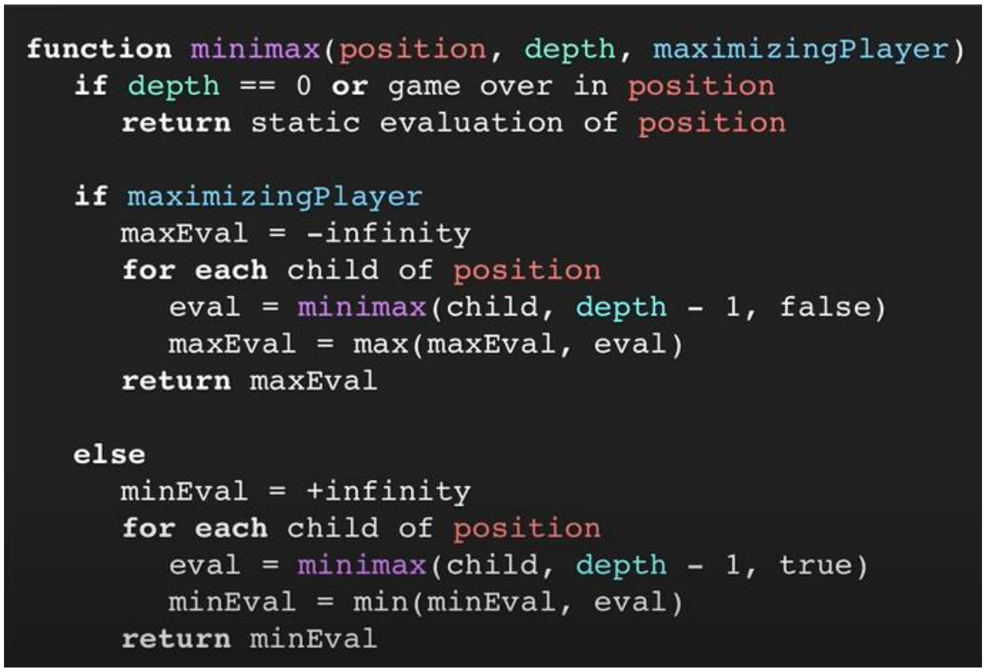


*Figure 3: A simple example for the minimax algorithm (Source [7].)*

Here, we consider a simple game with a simple case where there are eight possible final situations of the game, where each one is assigned to a utility function that the first player (the white player) tried to maximize, and the second player (the black player tried to minimize.)

The algorithm evaluates each leaf node using a heuristic evaluation function that we call the utility function, obtaining the values shown. The moves where the maximizing player has advantage are assigned with positive numbers, while the moves that lead to a situation where the minimizing player has advantage are assigned with negative numbers. As much the magnitudes of the numbers become larger, as much the advantage prevails towards one side. A pseudocode of the algorithm is provided in Figure 4.



```
function minimax(position, depth, maximizingPlayer)
  if depth == 0 or game over in position
    return static evaluation of position

  if maximizingPlayer
    maxEval = -infinity
    for each child of position
      eval = minimax(child, depth - 1, false)
      maxEval = max(maxEval, eval)
    return maxEval

  else
    minEval = +infinity
    for each child of position
      eval = minimax(child, depth - 1, true)
      minEval = min(minEval, eval)
    return minEval
```

*Figure 4: Pseudocode for the minimax algorithm (Source [7].)*

In our Connect Four case, the search tree grows exponentially with a base of nearly seven, this makes it very computationally expensive to make our algorithm explore the tree until it reaches the last leaf nodes. Hence, the new idea in section IV.II is introduced.

## IV.II The Alpha-Beta Pruning Algorithm

As clear from our previous analysis of the minimax algorithm, it would be very computationally expensive to search through the whole depth of the tree because of the exponentially-increasing number of leafs. Hence a very feasible way to reduce the search space is by trimming a part of the tree of the possible situations of the game.

Alpha–Beta Pruning can be considered as a general search algorithm which has an objective of minimizing the number of explored nodes that have utility-function values calculated by the main heuristic function evaluator that is employed in the minimax algorithm.

Alpha-Beta Pruning algorithm is an adversarial search algorithm that is usually employed to implement an agent playing two-player game, which makes it a very great fit for the purpose of implementing and AI for the Connect Four board game.

The idea behind the algorithm is that it terminated the evaluation of the utility function values of the nodes when at least one possibility gets found that proves that the play in inquiry is not better than a play that has already been previously examined. Such plays do not need to have their utility function values calculated further.

When applied the Alpha-Beta Pruning algorithm on a standard minimax search tree, it returns a result of the same move as the minimax would return without applying the algorithm, but a new trimmed tree will be considered that has some branches, which are impossible to possibly affect the final decision of the algorithm, cut out. Hence, a more tractable problem will be considered rather than the main computationally-expensive original one. Figure 5 depicts an example of an application for the Alpha-Beta Pruning algorithm.

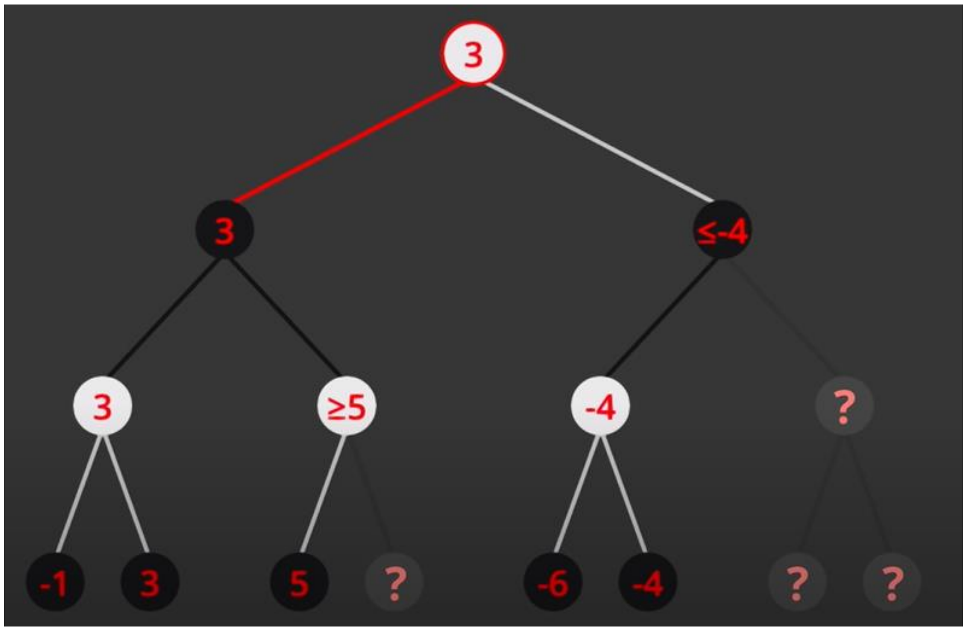
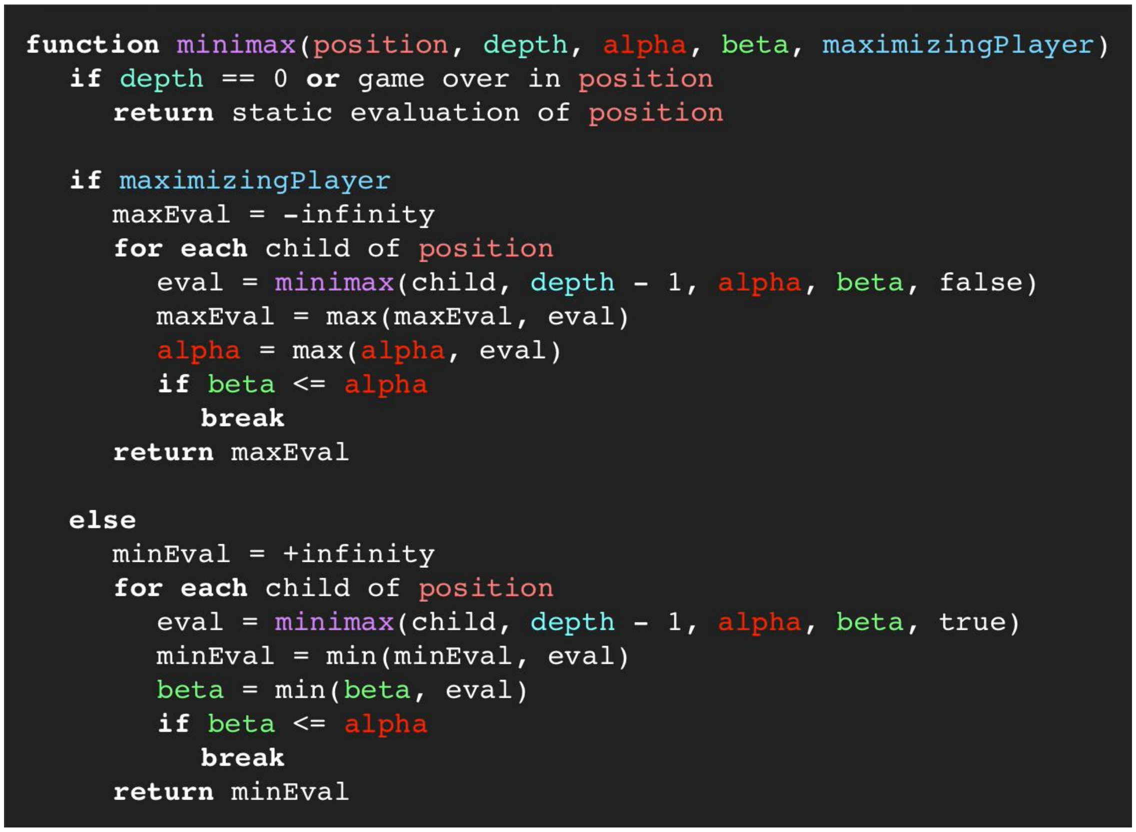


*Figure 5: A simple example for the alpha-beta pruning algorithm (Source [7].)*

In Figure 5, an example where it is not important to calculate the utility function of three possible situations out of eight because, after we have evaluated the other five already, we became sure that whatever values those leafs may have, that will not affect the final result of the main minimax algorithm in anyway.

We can easily observe that, when looking at the left subtree, whatever the value of the fourth leaf is assigned, the weight node above it will not ever have a value less than five. This means surely that the black node above it will inherit the value from the left child, which has a value of three, instead of inheriting it from the child on the right, which will not have a value less than three, let alone five. The same exact principle is applied on the right subtree that gets more nodes trimmed at once after the information of the first five leafs have already been discovered.

Figure 6 shows the pseudocode of the standard alpha-beta pruning algorithm applied to the minimax algorithm.



```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

*Figure 6: Pseudocode for the Alpha-Beta Pruning algorithm (Source [7].)*

## IV.III The Utility-Function Heuristic Evaluator

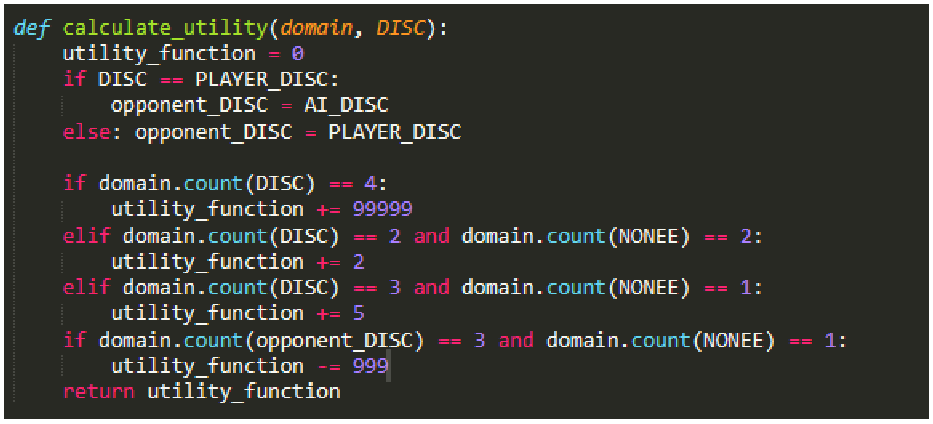
Determining the utility function, or the function's value evaluator, is the hardest part of the solution for the game because of its heuristic nature. This major hindrance could have been avoided if we can efficiently reach the end nodes (i.e., the leafs). In this case, we would just assign the end nodes to either $+\infty$ or $-\infty$ depending on who wins on those end leafs.

However, in our case here, we have to have some kind of heuristic evaluation of whose advantage some state of the game is, and in what degree roughly it is advantageous (or disadvantageous). Hence, constructing the heuristic utility-function has some degree of arbitrariness.

In the implementation of our AI, I started firstly considering the number of discs of each color within a specific domain (a $4 \times 4$ window in this case here) that may end up to form a winning situation, whether by lying down horizontally, vertically, or diagonally.

Furthermore, a very great addition to this heuristic function that I have done was made by distinguishing the middle column and giving it more advantage over other columns. This addition has another benefit too, which is breaking the initial symmetry that we have in the start of the game. That is it, whenever the AI starts with the first turn, it chooses the middle column always, and it gives the middle column more advantage over other columns.

Now, consider Figure 7 next:



```
def calculate_utility(domain, DISC):
    utility_function = 0
    if DISC == PLAYER_DISC:
        opponent_DISC = AI_DISC
    else: opponent_DISC = PLAYER_DISC

    if domain.count(DISC) == 4:
        utility_function += 99999
    elif domain.count(DISC) == 2 and domain.count(NONEE) == 2:
        utility_function += 2
    elif domain.count(DISC) == 3 and domain.count(NONEE) == 1:
        utility_function += 5
    if domain.count(opponent_DISC) == 3 and domain.count(NONEE) == 1:
        utility_function -= 999
    return utility_function
```

*Figure 7: The primitive heuristic utility function to evaluate the score of a board.*

This is my main chosen way to calculate the utility function. As clear from the code, when the count of the number of the player's discs within a certain given domain of spaces to be considered (our certain domain here is the $4 \times 4$ grid around the added disc.)

Our affecting parts and their degrees are chosen as following:

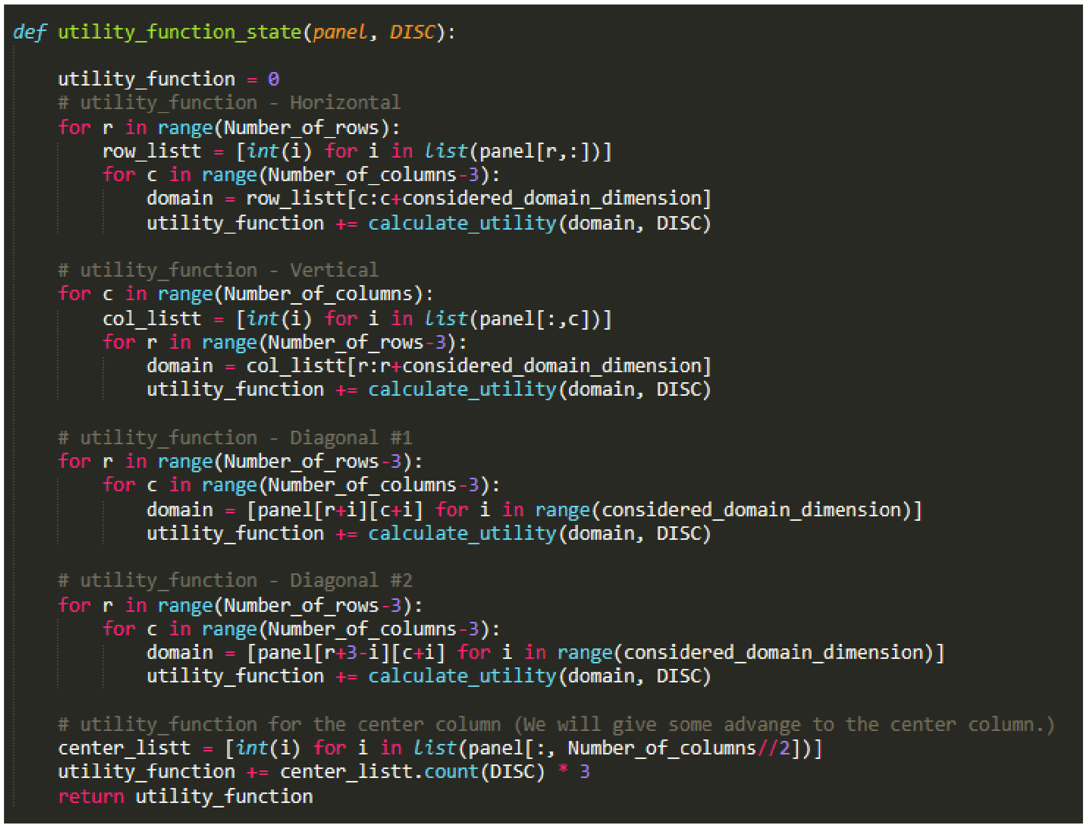
- You get +99999 points of utility whenever your disc in your hand completes a count of exactly four discs of yours within the considered domain, which is $4 \times 4$ grid in this case (i.e., after performing a winning situation.)
- You get +5 points of utility whenever your disc in your hand completes a count of exactly three discs of yours and one empty space within the considered domain, which is $4 \times 4$ grid in this case.
- You get +2 points of utility whenever your disc in your hand completes a count of exactly two discs of yours and two empty spaces within the considered domain, which is $4 \times 4$ grid in this case.
- You get $-999$ points of utility whenever your disc in your hand leaves behind it a count of exactly three discs of your opponent's and one empty spaces within the considered domain, which is $4 \times 4$ grid in this case.

The factor in arbitrariness in our choices is apparent in the choice of the exact values and in the exact scenarios. However, the experimental results shown in section VI verify that they are acceptable to some extent.

Furthermore, we did not have to consider more involved cases and assign values to them, like for example considering $5 \times 5$ grids of domain with more cases so that the heuristic function becomes more sophisticated.

Our choice to +99999 points of utility is obvious to be happening in case that the move is a winning move, and the choice of $-999$ points of utility is too obvious to be happening in case that the move is a losing move (i.e., the opponent can win in a single move after it.) In the same time, it is important to make the AI prefer playing a winning move over avoiding a losing move. This is why we chose $|99999| > |999|$ of utility points in the two cases.

Now, considering Figure 8:



```python
def utility_function_state(panel, DISC):

    utility_function = 0
    # utility_function - Horizontal
    for r in range(Number_of_rows):
        row_listt = [int(i) for i in list(panel[r,:])]
        for c in range(Number_of_columns-3):
            domain = row_listt[c:c+considered_domain_dimension]
            utility_function += calculate_utility(domain, DISC)

    # utility_function - Vertical
    for c in range(Number_of_columns):
        col_listt = [int(i) for i in list(panel[:,c])]
        for r in range(Number_of_rows-3):
            domain = col_listt[r:r+considered_domain_dimension]
            utility_function += calculate_utility(domain, DISC)

    # utility_function - Diagonal #1
    for r in range(Number_of_rows-3):
        for c in range(Number_of_columns-3):
            domain = [panel[r+i][c+i] for i in range(considered_domain_dimension)]
            utility_function += calculate_utility(domain, DISC)

    # utility_function - Diagonal #2
    for r in range(Number_of_rows-3):
        for c in range(Number_of_columns-3):
            domain = [panel[r+3-i][c+i] for i in range(considered_domain_dimension)]
            utility_function += calculate_utility(domain, DISC)

    # utility_function for the center column (We will give some advange to the center column.)
    center_listt = [int(i) for i in list(panel[:, Number_of_columns//2])]
    utility_function += center_listt.count(DISC) * 3
    return utility_function
```

*Figure 8: The second piece of code regarding the utility function.*

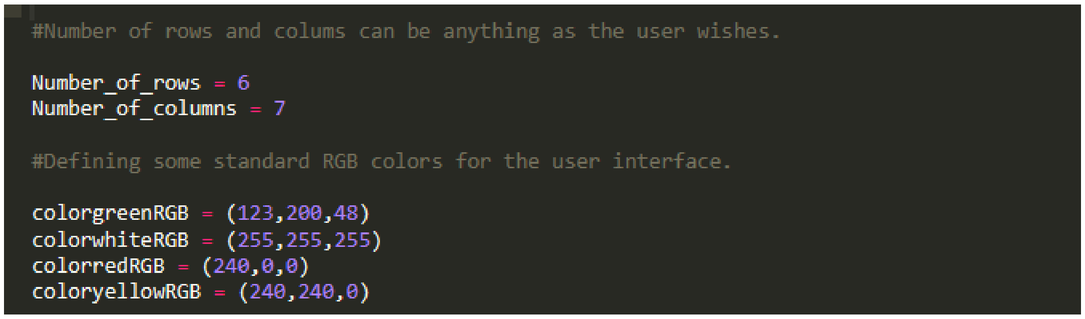
Here, the practical implementation of the considered domain considering all the possible orientations of the discs; the horizontal, vertical, and the two diagonals. And a very important part is the one that gives more significance of reserving the middle column (I assumed that the number of columns here is odd.)

## V. The Implementation

In this section, we discuss briefly the high level of some parts of our implementation of the code, I did write the code heavily commented, and I made the names of the variables representative to what they mean. Some ideas from the code were inspired or applied from [5 – 9].

### V.I The Basic Panel and Its User-Friendly Interface

Starting with a very brief description of the user-friendly interface to build the panel itself, we start, by simply and directly applying the straightforward instructions in the documentation of the pygame library [10]. Figure 9 shows a part from that implementation.



```
#Number of rows and colums can be anything as the user wishes.

Number_of_rows = 6
Number_of_columns = 7

#Defining some standard RGB colors for the user interface.

colorgreenRGB = (123,200,48)
colorwhiteRGB = (255,255,255)
colorredRGB = (240,0,0)
coloryellowRGB = (240,240,0)
```

*Figure 9: A piece of code for the initialization of the panel and the game.*

This part is mainly about defining the simple RGB arrays of each color to represent it later in the user-friendly interface.

Furthermore, as Dr. Wang has suggested, I made my code flexible to accept more general cases of choosing whatever number of rows and columns. Figure 10 shows the user-interface after setting the Number_of_rows variable to 4, and the Number_of_columns variable to 5.

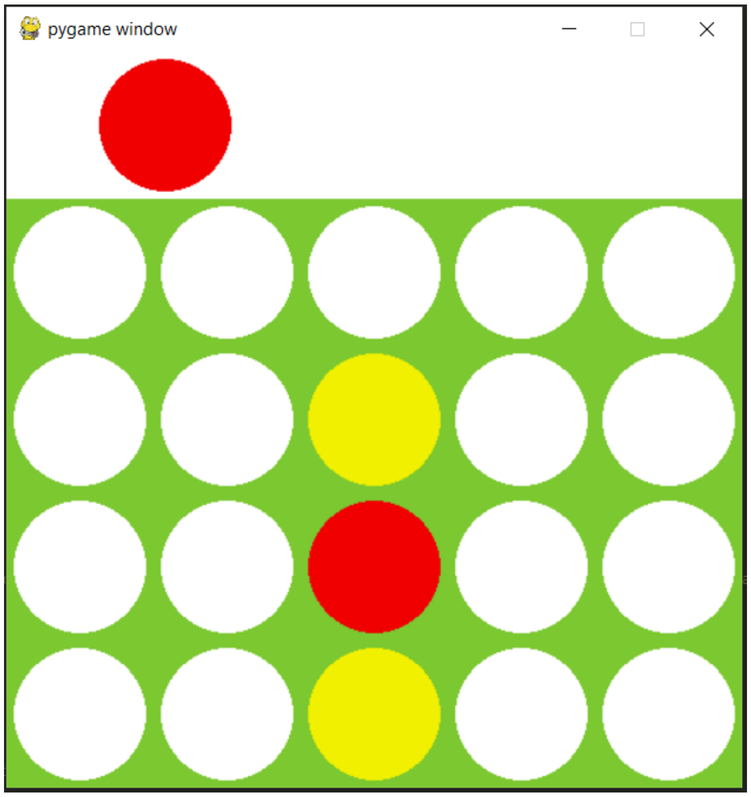
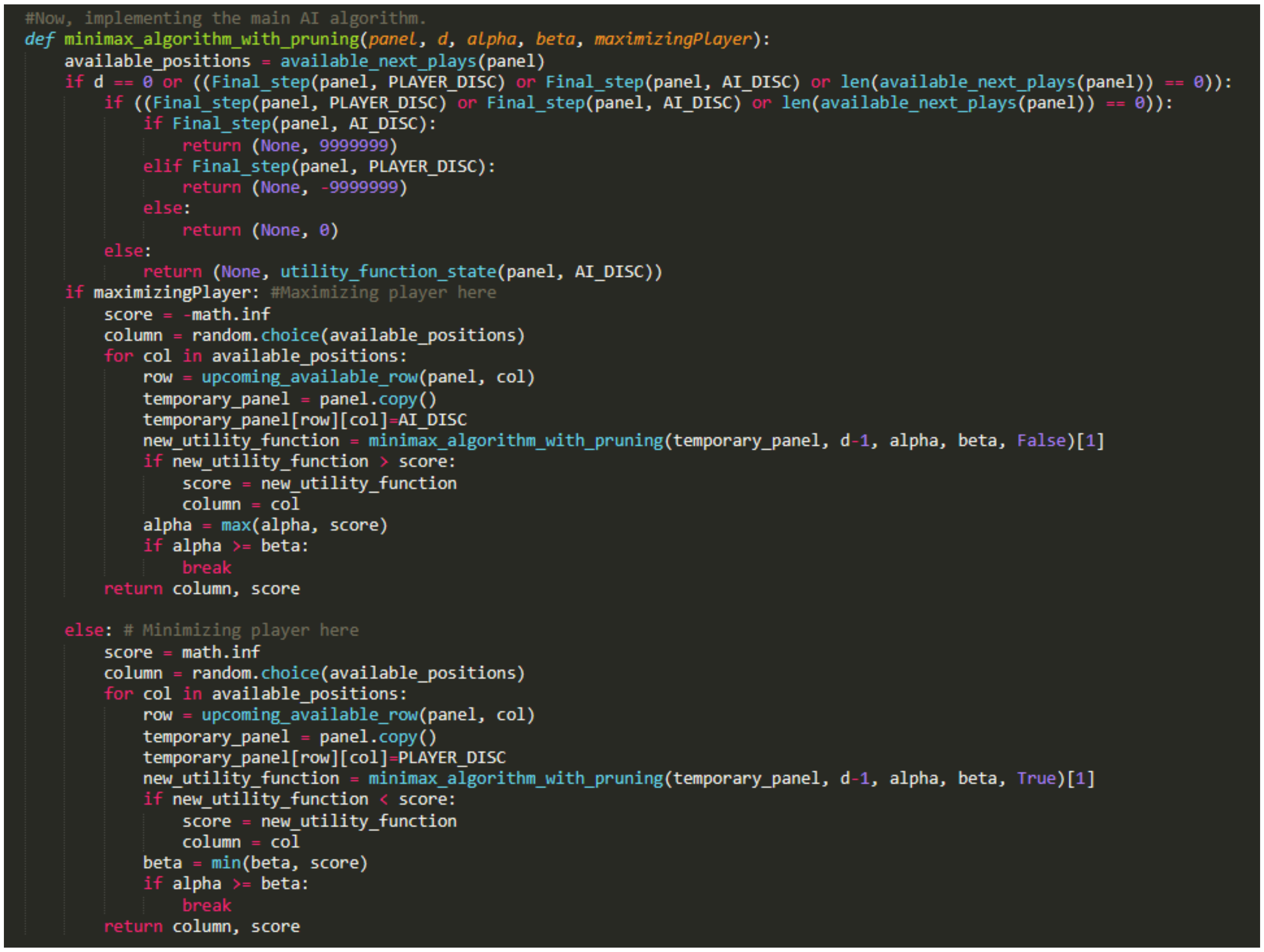


*Figure 10: A general panel that can be produced by setting the Number_of_rows and Number_of_columns variables.*

It is worth mentioning here, that the AI works very perfectly in different settings of the number of columns and rows, because the number of columns and rows that are set in the initialization designated variables are directly linked in the implementation of the AI itself. In addition to that, it is worth to be highlighted that the game at the first place was just a command line game that just prints the matrix of the occupation of the slots of the panel after each step rather than showing our final colored panel.

## V.II The Implementation of the AI Agent and Some Other Parts of the Code

A brief clarification of the implementation of the AI player. Figure 11 shows the main part of the AI algorithm which constitutes the main body of the AI.



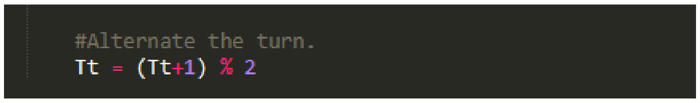
```python
#Now, implementing the main AI algorithm.
def minimax_algorithm_with_pruning(panel, d, alpha, beta, maximizingPlayer):
    available_positions = available_next_plays(panel)
    if d == 0 or ((Final_step(panel, PLAYER_DISC) or Final_step(panel, AI_DISC) or len(available_next_plays(panel)) == 0)):
        if ((Final_step(panel, PLAYER_DISC) or Final_step(panel, AI_DISC) or len(available_next_plays(panel)) == 0)):
            if Final_step(panel, AI_DISC):
                return (None, 9999999)
            elif Final_step(panel, PLAYER_DISC):
                return (None, -9999999)
            else:
                return (None, 0)
        else:
            return (None, utility_function_state(panel, AI_DISC))
    if maximizingPlayer: #Maximizing player here
        score = -math.inf
        column = random.choice(available_positions)
        for col in available_positions:
            row = upcoming_available_row(panel, col)
            temporary_panel = panel.copy()
            temporary_panel[row][col]=AI_DISC
            new_utility_function = minimax_algorithm_with_pruning(temporary_panel, d-1, alpha, beta, False)[1]
            if new_utility_function > score:
                score = new_utility_function
                column = col
            alpha = max(alpha, score)
            if alpha >= beta:
                break
        return column, score

    else: # Minimizing player here
        score = math.inf
        column = random.choice(available_positions)
        for col in available_positions:
            row = upcoming_available_row(panel, col)
            temporary_panel = panel.copy()
            temporary_panel[row][col]=PLAYER_DISC
            new_utility_function = minimax_algorithm_with_pruning(temporary_panel, d-1, alpha, beta, True)[1]
            if new_utility_function < score:
                score = new_utility_function
                column = col
            beta = min(beta, score)
            if alpha >= beta:
                break
        return column, score
```

*Figure 11: The main function of the AI agent.*

As clear, the function is recursive, and was taken from the pseudocodes implemented in Figure 4 and Figure 6. The adjustable parameters are the depth, which is the variable d. The variables alpha and beta, as stated in the standard pseudocode, are $+\infty$ and $-\infty$.

As obvious in the function, each invoking of it reduces the depth by one (as obvious because the algorithm goes up level by level as depicted in Figure 3.) Furthermore, when invoking the function itself each time, it alternates between two subfunctions within it, one for the maximizing player, which is flagged by a true value of the maximizingPlayer variable, and one for the minimizing player, which is flagged by a false value of the maximizingPlayer variable.

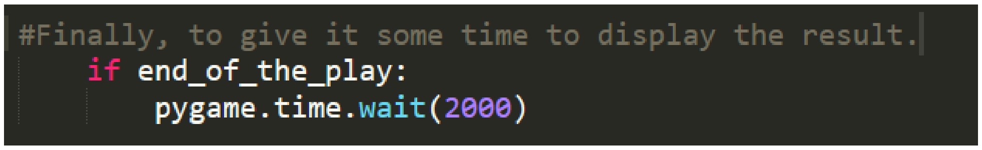
Another small piece of code that I would like to state here is the way I chose to alternate between the turns of the two players. Figure 12 shows the part of code that alternated the Tt variable between 0 and 1 after each time the line in the figure is invoked.



```
    #Alternate the turn.
    Tt = (Tt+1) % 2
```

*Figure 12: How the Tt variable alternates between the two values of 0 and 1.*

The last small part of the code to be highlighted here is from the pygame library, which holds the pygame window for 2 seconds after the game ends to avoid the sudden diminish of the window right after the last move. Figure 13 shows this part of the code.



```
#Finally, to give it some time to display the result.
    if end_of_the_play:
        pygame.time.wait(2000)
```

*Figure 13: The piece of code that is responsible on holding the pygame window for 2000 milliseconds after the game ends.*

## VI. The Efficiency of the AI Model

This section gives a very brief feedback of the efficiency of our AI model considering many criteria:

- The AI with any depth assigned defeats a random player who chooses their next column randomly always.
- The AI without the Alpha-Beta Pruning needs unreasonable time to play when assigned to depth more than 4.

- The AI with the Alpha-Beta Pruning needs unreasonable time to play when assigned to depth more than 6.
- The AI with depth value of 5 was never defeated by rational players.
- The AI with depth value of 3 was defeated in roughly around one game out of ten played against rational players.
- An AI with higher depth always defeats the AI with lower depth.
- The AI who starts first always wins against the same AI-modeled agent with the same depth.
- The AI with depth value of 2 was defeated in roughly around two times out of ten games played against rational players.

Figure 14 shows the end of a sample game played against an AI with a depth value of four. The screenshot was captured within the two seconds freezing after it finishes.

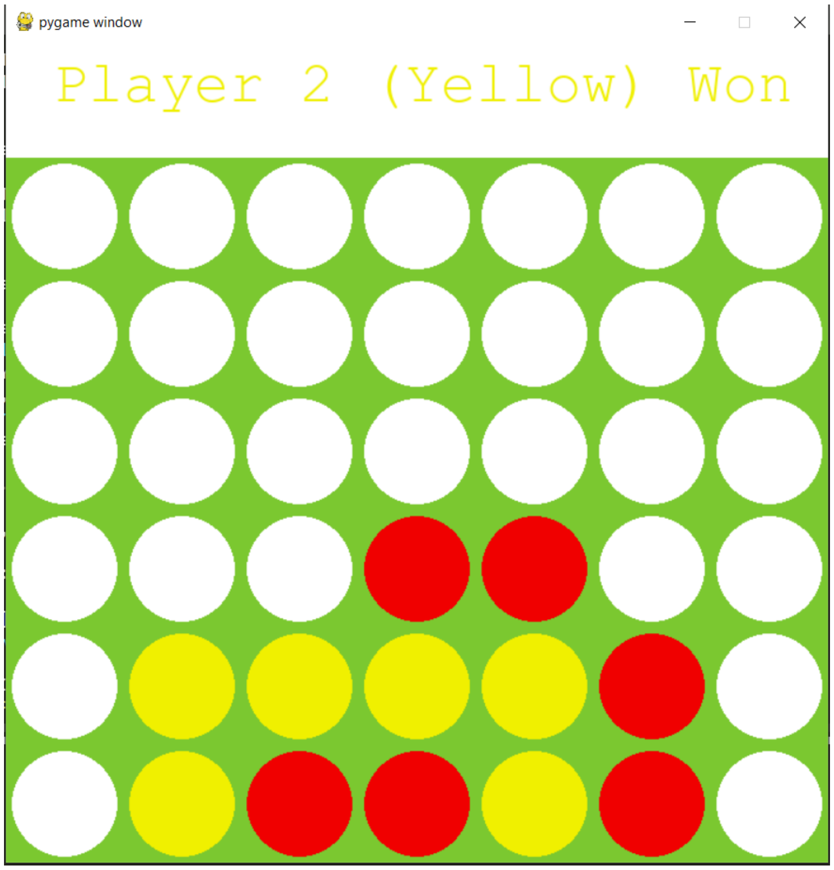


*Figure 14: The end of a sample game against the AI with depth value of four.*

## VII. Conclusion, Personal Comments, and Final Thoughts

We have, in this project, explored the Connect Four board game with some extension for it, and have built a new AI model to play against after building a user-friendly interface for the game. The novelty of our AI model is actually in its heuristic utility-function that we have used.

We have used the minimax algorithm and made it more efficient by applying the Alpha-Beta Pruning method on it. Testing the powerfulness of our AI model shows that, even without the Alpha-Beta Pruning, is smart in playing the game and easily defeats reasonable players.

Doing this project was a very great experience for me, and took me time all throughout the semester, and it is the first time for me to program a game let alone an AI agent who plays a code. This has improved my skills and got me exposed to a new interesting Python library that has an extensive comprehensive documentation.

I was surprised, at first, to find this game specifically explored by the research community even until very recently [3]. Which shows that having a game solved does not necessarily mean that there is no usefulness that can be extracted from exercising on trying to build an AI player for it.

## VIII. References

[1]https://en.wikipedia.org/wiki/Connect_Four#/media/File:Connect_Four.gif

[2]https://oeis.org/A212693

[3] Tommy, L.; Hardjianto, M.;Agani, N. (2017). "The Analysis of Alpha Beta Pruning and MTD(f) Algorithm to Determine the Best Algorithm to be Implemented at Connect Four Prototype." IOP Conf. Ser.: Mater. Sci. Eng. 190 012044

[4] Sarhan, A.; Shaout, A.; Michele, S. (2009). "Real-Time Connect 4 Game Using Artificial Intelligence." Journal of Computer Science. 5. 10.3844/jcs.2009.283.289.

[5] https://www.youtube.com/channel/UCq6XkhO5SZ66N04IcPbqNcw

[6] https://en.wikipedia.org/wiki/Minimax

[7] https://www.youtube.com/watch?v=l-hh51ncgDI&t=37s

[8] https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

[9] https://www.pygame.org/docs/

[10] Yamaguchi, Y.; K. Yamaguchi; T. Tanaka; T. Kaneko (2012). "Infinite Connect-Four is solved: Draw". Advances in Computer Games, ACG 2011. LNCS 7168: 208–219.

[11] https://archive.org/details/isbn_9781402756214

[12]https://tsdr.uspto.gov/#caseNumber=73019915&caseType=SERIAL_NO&searchType=statusSearch