**Connect Four**

In our first lesson on the minimax algorithm, we wrote a program that could play the perfect game of Tic-Tac-Toe. Our AI looked at all possible future moves and chose the one that would be most beneficial. This was a viable strategy because Tic Tac Toe is a small enough game that it wouldn't take *too* long to reach the leaves of the game tree.

However, there are games, like Chess, that have much larger trees. There are 20 different options for the first move in Chess, compared to 9 in Tic-Tac-Toe. On top of that, the number of possible moves often increases as a chess game progresses. Traversing to the leaves of a Chess tree simply takes too much computational power.

In this lesson, we'll investigate two techniques to solve this problem. The first technique puts a hard limit on how far down the tree you allow the algorithm to go. The second technique uses a clever trick called pruning to avoid exploring parts of the tree that we know will be useless.

Before we dive in, let's look at the tree of a more complicated game — Connect Four!

If you've never played Connect Four before, the goal is to get a streak of four of your pieces in any direction — horizontally, vertically, or diagonally. You can place a piece by picking a column. The piece will fall to the lowest available row in that column.

**1.**

We've imported a Connect Four game engine along with a board that's in the middle of a game.

To start, let's call the print\_board() function using half\_done as a parameter.

**2.**

Call the tree\_size() function using half\_done and "X" as parameters. Print the result. This will show you the number of game states in the tree, assuming half\_done is the root of the tree and it is "X"'s turn.

**3.**

Let's make a move and see how the size of the tree changes. Let's place an "X" in column 6. Before calling the tree\_size() function, call the select\_space() function with the following three parameters:

* half\_done — The board that you're making the move on.
* 6 — The column you're selecting.
* "X" — The type of piece you're putting in column 6.

Since "X" has taken their turn, it is now "O"'s turn. Change the second parameter in the tree\_size() function to be "O".

from connect\_four import \*  
  
print\_board(half\_done)  
print(tree\_size(half\_done, "X")) # 3034  
select\_space(half\_done, 6, "X")  
print(tree\_size(half\_done, "O")) # 442  
  
'''  
 1 2 3 4 5 6 7   
+---+---+---+---+---+---+---+  
| | | | | | | |  
| O | O | O | | | | O |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| X | X | X | | | X | X |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| O | O | O | | | O | O |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| X | X | X | | | X | X |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| O | O | O | | | O | O |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| X | X | X | | | X | X |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
'''

# Depth and Base Case

The first strategy we'll use to handle these enormous trees is stopping the recursion early. There's no need to go all the way to the leaves! We'll just look a few moves ahead.

Being able to stop before reaching the leaves is critically important for the efficiency of this algorithm. It could take literal days to reach the leaves of a game of chess. Stopping after only a few levels limits the algorithm's understanding of the game, but it makes the runtime realistic.

In order to implement this, we'll add another parameter to our function called depth. Every time we make a recursive call, we'll decrease depth by 1 like so:

def minimax(input\_board, minimizing\_player, depth):

# Base Case

if game\_is over(input\_bopard):

return ...

else:

# …

# Recursive Call

hypothetical\_value = minimax(new\_board, True, depth - 1)[0]

We'll also have to edit our base case condition. We now want to stop if the game is over (we've hit a leaf), or if depth is 0.

**1.**

We've given you a minimax() function that recurses to the leaves. Edit it so it has a third parameter named depth.

Change the recursive call to decrease depth by 1 each time.

Change your base case — the recursion should stop when the game is over or when depth is 0.

**2.**

Outside the function, call minimax() on new\_board as the maximizing player with a depth of 3 and print the results. Remember, minimax() returns a list of two numbers. The first is the value of the best possible move, and the second is the move itself.

from connect\_four import \*  
import random  
random.seed(108)  
  
new\_board = make\_board()  
  
# Add a third parameter named depth  
def minimax(input\_board, is\_maximizing, depth):  
 # Change this if statement to also check to see if depth = 0  
 if game\_is\_over(input\_board) or depth == 0:  
 return [evaluate\_board(input\_board), ""]  
 best\_move = ""  
 if is\_maximizing == True:  
 best\_value = -float("Inf")  
 symbol = "X"  
 else:  
 best\_value = float("Inf")  
 symbol = "O"  
 for move in available\_moves(input\_board):  
 new\_board = deepcopy(input\_board)  
 select\_space(new\_board, move, symbol)  
 #Add a third parameter to this recursive call  
 hypothetical\_value = minimax(new\_board, not is\_maximizing, depth-1)[0]  
 if is\_maximizing == True and hypothetical\_value > best\_value:  
 best\_value = hypothetical\_value  
 best\_move = move  
 if is\_maximizing == False and hypothetical\_value < best\_value:  
 best\_value = hypothetical\_value  
 best\_move = move  
 return [best\_value, best\_move]  
  
print(minimax(new\_board, True, 3)) # [0, 1] == [best\_value, best\_move]

# Evaluation Function

By adding the depth parameter to our function, we've prevented it from spending days trying to reach the end of the tree. But we still have a problem: our evaluation function doesn't know what to do if we're not at a leaf. Right now, we're returning positive infinity if Player 1 has won, negative infinity if Player 2 has won, and 0 otherwise. Unfortunately, since we're not making it to the leaves of the board, neither player has won and we're always returning 0. We need to rewrite our evaluation function.

Writing an evaluation function takes knowledge about the game you're playing. It is the part of the code where a programmer can infuse their creativity into their AI. If you're playing Chess, your evaluation function could count the difference between the number of pieces each player owns. For example, if white had 15 pieces, and black had 10 pieces, the evaluation function would return 5. This evaluation function would make an AI that prioritizes capturing pieces above all else.

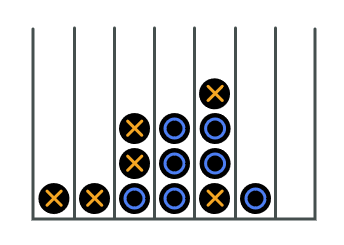
You could go even further — some pieces might be more valuable than others. Your evaluation function could keep track of the value of each piece to see who is ahead. This evaluation function would create an AI that might really try to protect their queen. You could even start finding more abstract ways to value a game state. Maybe the position of each piece on a Chess board tells you something about who is winning.

It's up to you to define what you value in a game. These evaluation functions could be incredibly complex. If the maximizing player is winning (by your definition of what it means to be winning), then the evaluation function should return something greater than 0. If the minimizing player is winning, then the evaluation function should return something less than 0.

**1.**

Let's rewrite our evaluation function for Connect Four. We'll be editing the part of the evaluation function under the else statement. We need to define how to evaluate a board when nobody has won.

Let's write a slightly silly evaluation function that prioritizes having the top piece of a column. If the board looks like the image below, we want our evaluation function to return 2 since the maximizing player ("X") has two more "top pieces" than "O".



For now, inside the else statement, delete the current return statement. Create two variables named num\_top\_x and num\_top\_o and start them both at 0. Return num\_top\_x - num\_top\_o.

**2.**

Before this new return statement, loop through every column in board. Inside that loop, loop through every square in column. You're now looking at every square in every column going from top to bottom.

If square equals "X" add one to num\_top\_x and then break the inner for loop to go to the next column.

**3.**

If square equals "O" add one to num\_top\_o and then break the inner for loop to go to the next column.

**4.**

We've imported three boards for you to test this function. We should first get an understanding of what these three boards look like.

Note that these boards aren't game states you'd find in real games of Connect Four — "X" has been taking some extra turns. Nevertheless, we can still evaluate them!

Call print\_board once per board — board\_one, board\_two, and board\_three. What should the evaluation function return for those three boards?

**5.**

Call evaluate\_board once on each board and print the results. Did we trick you with board\_three?

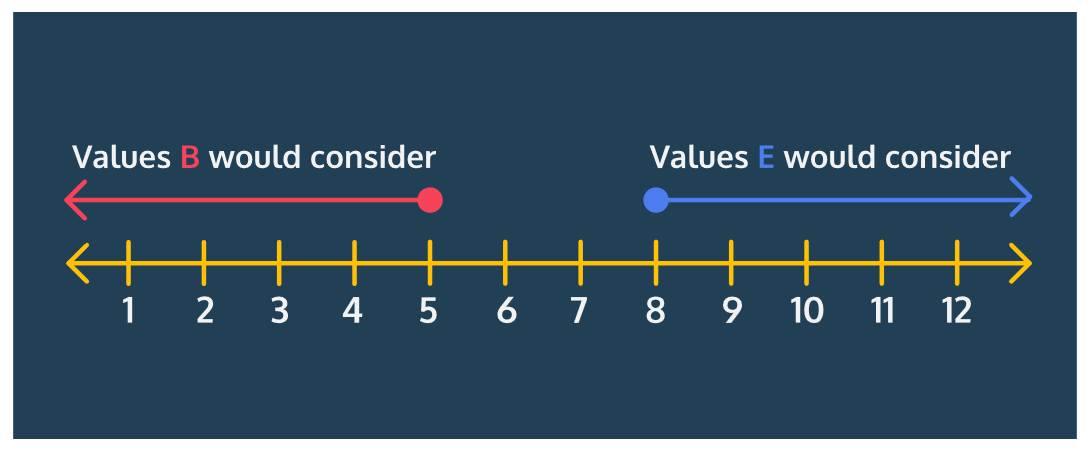
from connect\_four import \*  
import random  
  
random.seed(108)  
  
  
def evaluate\_board(board):  
 if has\_won(board, "X"):  
 return float("Inf")  
 elif has\_won(board, "O"):  
 return -float("Inf")  
 else:  
 num\_top\_x = 0  
 num\_top\_o = 0  
 for column in board:  
 for square in column:  
 if square == 'X':  
 num\_top\_x += 1  
 break  
 if square == 'O':  
 num\_top\_o += 1  
 break  
 return num\_top\_x - num\_top\_o  
  
  
print\_board(board\_one)  
print\_board(board\_two)  
print\_board(board\_three)  
  
print(evaluate\_board(board\_one)) # -1  
print(evaluate\_board(board\_two)) # 5  
print(evaluate\_board(  
 board\_three)) # Inf # "X" has won the game in board\_three. So even though "O" "owns" several more columns, the function will return infinity.  
  
'''  
 1 2 3 4 5 6 7   
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | O | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | X | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| O | X | X | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
  
 1 2 3 4 5 6 7   
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | O | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | X | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | O | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | X | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | X | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| X | X | X | O | X | X | X |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
  
 1 2 3 4 5 6 7   
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | O | O | O | X |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| O | O | O | X | X | X | X |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
-1  
5  
inf  
'''

# Alpha-Beta Pruning

By writing an evaluation function that works on non-leaf game states, we can stop the recursion early. But in a perfect world, we'd still want to reach the leaves of the tree. In order to traverse farther down the tree without dramatically increasing the runtime, we can implement a technique called alpha-beta pruning.

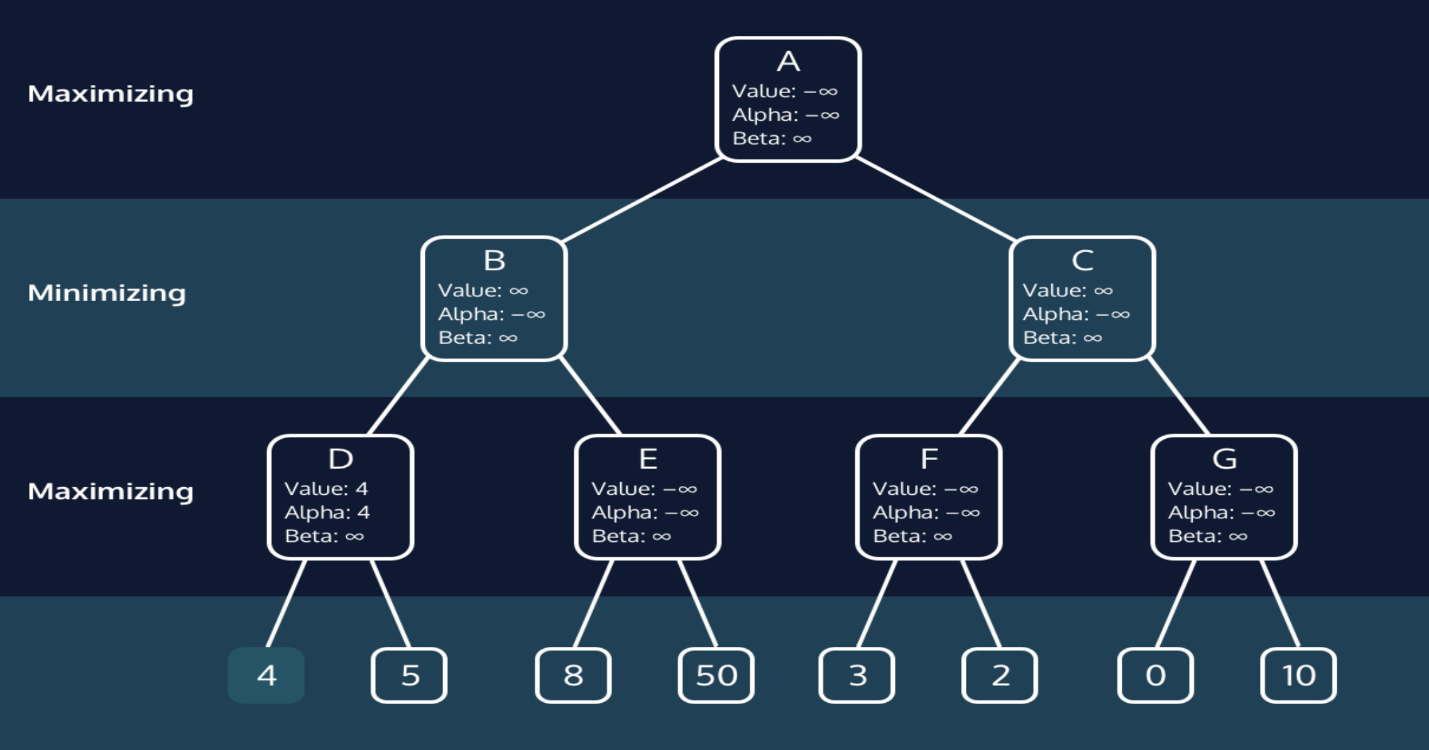
The core idea behind alpha-beta pruning is to ignore parts of the tree that we know will be dead ends. For example, consider the gif on this page. When determining the "value" of the node labeled E, we can stop looking at possible moves as soon as it discovers the 8 node.

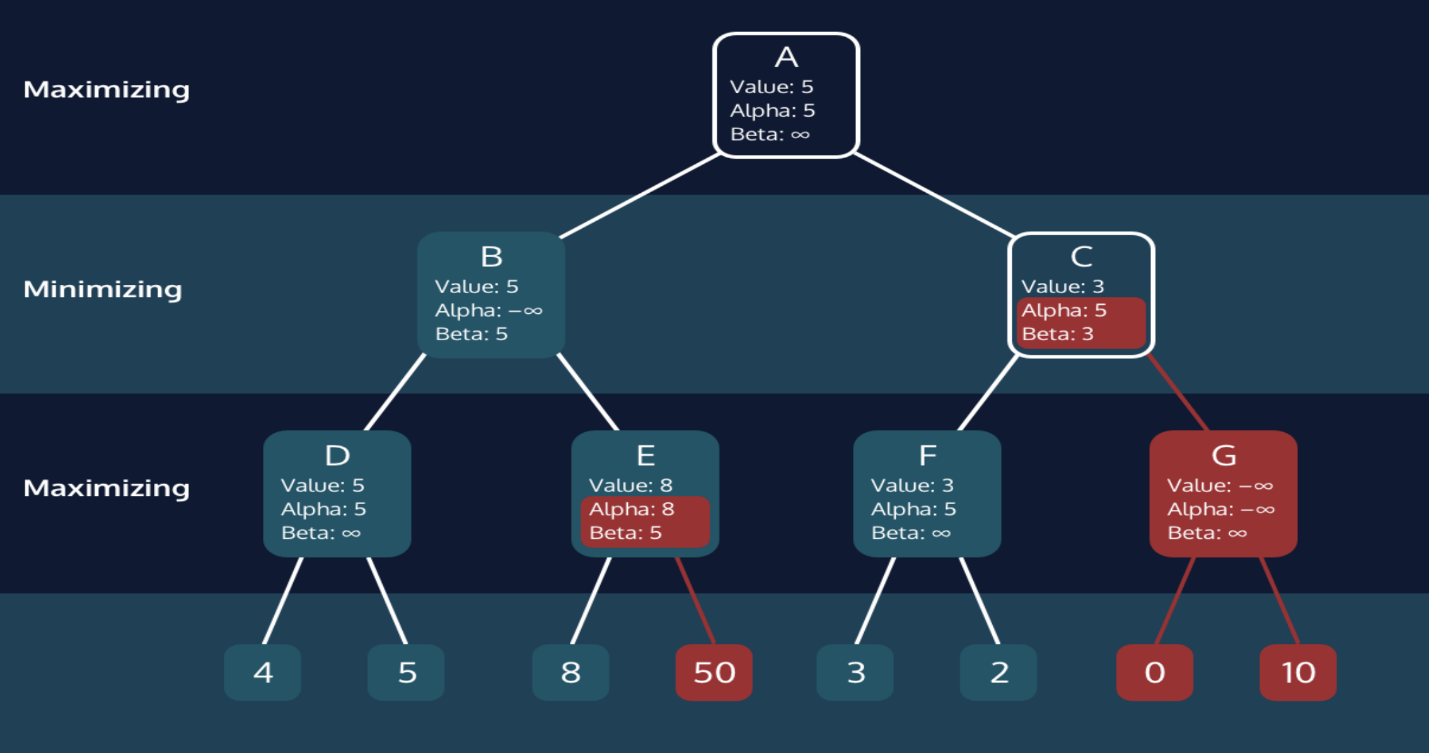
We know that B will only consider values less than or equal to 5, and E will only consider values greater than or equal to 8. We know that node B will never care about E's value and we can stop looking through E's moves.



Similarly, we can prune a large section from the right half of the tree. There comes a point where node A will only consider values greater than or equal to 5 and node C will only consider values 3 and below. We can stop looking at all of the C's children because we know they will never be relevant.

In the next lesson, we'll implement alpha-beta pruning, but for now, take the time to study the gif on this page to understand how alpha-beta pruning works conceptually.





# Implement Alpha-Beta Pruning

Alpha-beta pruning is accomplished by keeping track of two variables for each node — alpha and beta. alpha keeps track of the minimum score the maximizing player can possibly get. It starts at negative infinity and gets updated as that minimum score increases.

On the other hand, beta represents the maximum score the minimizing player can possibly get. It starts at positive infinity and will decrease as that maximum possible score decreases.

For any node, if alpha is greater than or equal to beta, that means that we can stop looking through that node's children.

To implement this in our code, we'll have to include two new parameters in our function — alpha and beta. When we first call minimax() we'll set alpha to negative infinity and beta to positive infinity.

We also want to make sure we pass alpha and beta into our recursive calls. We're passing these two values down the tree.

Next, we want to check to see if we should reset alpha and beta. In the maximizing case, we want to reset alpha if the newly found best\_value is greater than alpha. In the minimizing case, we want to reset beta if best\_value is less than beta.

Finally, after resetting alpha and beta, we want to check to see if we can prune. If alpha is greater than or equal to beta, we can break and stop looking through the other potential moves.

**1.**

Add two new parameters named alpha and beta to your minimax() function as the final two parameters. Inside your minimax() function, when you the recursive call, add alpha and beta as the final two parameters.

**2.**

After resetting the value of best\_value if is\_maximizing is True, we want to check to see if we should reset alpha. Set alpha equal to the maximum of alpha and best\_value. You can do this quickly by using the max() function. For example, the following line of code would set a equal to the maximum of b and c.

a = max(b, c)

Change both returns statements to include alpha as the last item in the list. For example, the base case return statement should be [evaluate\_board(input\_board), "", alpha].

Note that this third value in the return statement is not necessary for the algorithm — we need the value of alpha so we can check to see if you did this step correctly!

**3.**

If we reset the value of best\_value and is\_maximizing is False, we want to set beta to be the minimum of beta and best\_value. You can use the min() function this time.

In both return statements, add beta as the last item in the list. This is once again unnecessary for the algorithm, but we need it to check your code!

**4.**

At the very end of the for loop, check to see if alpha is greater than or equal to beta. If that is true, break which will cause your program to stop looking through the remaining possible moves of the current game state.

**5.**

We're going to call minimax() on board, but before we do let's see what board looks like. Call print\_board using board as a parameter.

**6.**

Call minimax() on board as the maximizing player and print the result. Set depth equal to 6. alpha should be -float("Inf") and beta should be float("Inf").

from connect\_four import \*  
import random  
  
random.seed(108)  
  
  
def minimax(input\_board, is\_maximizing, depth, alpha, beta):  
 # Base case - the game is over, so we return the value of the board  
 if game\_is\_over(input\_board) or depth == 0:  
 return [evaluate\_board(input\_board), "", alpha, beta]  
 best\_move = ""  
 if is\_maximizing == True:  
 best\_value = -float("Inf")  
 symbol = "X"  
 else:  
 best\_value = float("Inf")  
 symbol = "O"  
 for move in available\_moves(input\_board):  
 new\_board = deepcopy(input\_board)  
 select\_space(new\_board, move, symbol)  
 hypothetical\_value = minimax(new\_board, not is\_maximizing, depth - 1, alpha, beta)[0]  
 if is\_maximizing == True and hypothetical\_value > best\_value:  
 best\_value = hypothetical\_value  
 best\_move = move  
 alpha = max(alpha, best\_value)  
 if is\_maximizing == False and hypothetical\_value < best\_value:  
 best\_value = hypothetical\_value  
 best\_move = move  
 beta = min(beta, best\_value)  
 if alpha >= beta:  
 break  
 return [best\_value, best\_move, alpha, beta]  
  
  
print\_board(board)  
print(minimax(board, True, 6, -float("Inf"), float("Inf")))  
  
'''  
 1 2 3 4 5 6 7   
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | O | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| | | X | | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
| | | | | | | |  
| O | X | X | O | | | |  
| | | | | | | |  
+---+---+---+---+---+---+---+  
[-2, 1, -2, inf]  
'''

**Review**

Great work! We've now edited our minimax() function to work with games that are more complicated than Tic Tac Toe. The core of the algorithm is identical, but we've added two major improvements:

* We wrote an evaluation function specific to our understanding of the game (in this case, Connect Four). This evaluation function allows us to stop the recursion before reaching the leaves of the game tree.
* We implemented alpha-beta pruning. By cleverly detecting useless sections of the game tree, we're able to ignore those sections and therefore look farther down the tree.

Now's our chance to put it all together. We've written most of the function two\_ai\_game() which sets up a game of Connect Four played by two AIs. For each player, you need to call fill in the third parameter of their minimax() call.

Remember, right now our evaluation function is using a pretty bad strategy. An AI using the evaluation function we wrote will prioritize making sure its pieces are the top pieces of each column.

Do you think you could write an evaluation function that uses a better strategy? In the project for this course, you can try to write an evaluation function that can beat our AI!

Instructions

**1.**

Fill in the third parameter of both minimax() function calls. This parameter is the depth of the recursive call. The higher the number, the "smarter" the AI will be.

What happens if they have equal intelligence? What happens if one is significantly smarter than the other?

We suggest keeping these parameters under 7. Anything higher and the program will take a while to complete!

from connect\_four import \*  
  
def two\_ai\_game():  
 my\_board = make\_board()  
 while not game\_is\_over(my\_board):  
 # Fill in the third parameter for the first player's "intelligence"  
 result = minimax(my\_board, True, 3, -float("Inf"), float("Inf"))  
 print( "X Turn\nX selected ", result[1])  
 print(result[1])  
 select\_space(my\_board, result[1], "X")  
 print\_board(my\_board)  
  
 if not game\_is\_over(my\_board):  
 #Fill in the third parameter for the second player's "intelligence"  
 result = minimax(my\_board, False, 4, -float("Inf"), float("Inf"))  
 print( "O Turn\nO selected ", result[1])  
 print(result[1])  
 select\_space(my\_board, result[1], "O")  
 print\_board(my\_board)  
 if has\_won(my\_board, "X"):  
 print("X won!")  
 elif has\_won(my\_board, "O"):  
 print("O won!")  
 else:  
 print("It's a tie!")  
  
two\_ai\_game()