**Games as Trees**

Have you ever played a game against someone and felt like they were always two steps ahead? No matter what clever move you tried, they had somehow envisioned it and had the perfect counterattack. This concept of thinking ahead is the central idea behind the *minimax algorithm*.

The minimax algorithm is a decision-making algorithm that is used for finding the best move in a two player game. It's a recursive algorithm — it calls itself. In order for us to determine if making move A is a good idea, we need to think about what our opponent would do if we made that move.

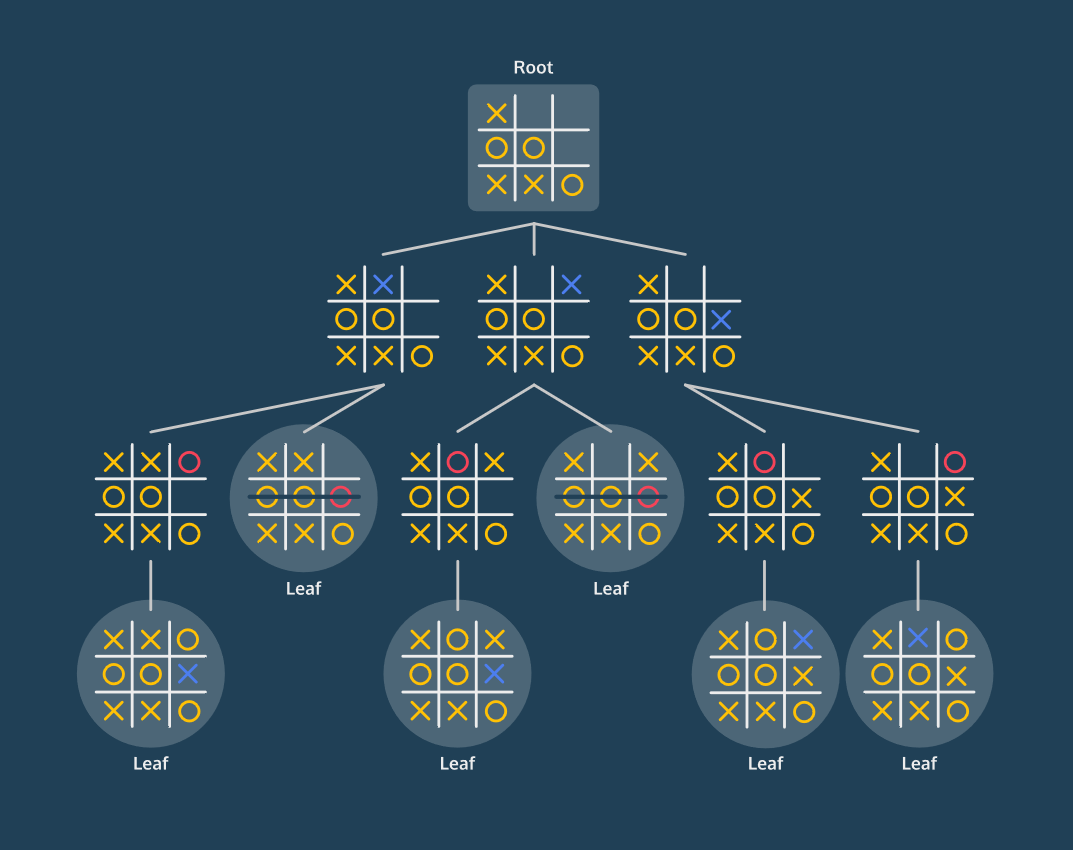
We'd guess what our opponent would do by running the minimax algorithm from our opponent's point of view. In the hypothetical world where we made move A, what would they do? Surely they want to win as badly as we do, so they'd evaluate the strength of their move by thinking about what we would do if they made move B.

As this process repeats, we can start to make a tree of these hypothetical game states. We'll eventually reach a point where the game is over — we'll reach a leaf of the tree. Either we won, our opponent won, or it was a tie. At this point, the recursion can stop. Because the game is over, we no longer need to think about how our opponent would react if we reached this point of the game.

Instructions

On this page, you'll see the game tree of a Tic-Tac-Toe game that is almost complete. At the root of the node, it is "X"'s turn.

Some of the leaves of the tree still have squares that can be filled in. Why are those boards leaves?



**Tic-Tac-Toe**

For the rest of this exercise, we're going to be writing the minimax algorithm to be used on a game of Tic-Tac-Toe. We've imported a Tic-Tac-Toe game engine in the file tic\_tac\_toe.py. Before starting to write the minimax function, let's play around with some of the Tic-Tac-Toe functions we've defined for you in tic\_tac\_toe.py.

To begin, a board is represented as a list of lists. In script.py we've created a board named my\_board where the X player has already made the first move. They've chosen the top right corner. To nicely print this board, use the print\_board() function using my\_board as a parameter.

Next, we want to be able to take a turn. The select\_space() function lets us do this. Select space takes three parameters:

* The board that you want to take the turn on.
* The space that you want to fill in. This should be a number between 1 and 9.
* The symbol that you want to put in that space. This should be a string — either an "X" or an "O".

We can also get a list of the available spaces using available\_moves() and passing the board as a parameter.

Finally, we can check to see if someone has won the game. The has\_won() function takes the board and a symbol (either "X" or "O"). It returns True if that symbol has won the game, and False otherwise.

Let's test these functions! Write your code in script.py, but feel free to take a look at tic\_tac\_toe.py if you want to look at how the game engine works.

**1.**

Call print\_board() using my\_board as a parameter.

**2.**

Call select\_space() to put an "O" in the center of the board. Print the board again after making this move.

**3.**

Make two more moves of your choice and print the available moves. You can make a move with an "X" or an "O"

Remember, you can use the available\_moves() function using my\_board as a parameter to get a list of the available moves.

**4.**

Make enough moves to win the game as "X". Use has\_won() to check to see if "X" has won. Check to see if "O" has won as well. Print both results.

from tic\_tac\_toe import \*  
  
my\_board = [  
 ["1", "2", "X"],  
 ["4", "5", "6"],  
 ["7", "8", "9"]  
]  
  
print\_board(my\_board)  
select\_space(my\_board, 5, "O")  
print\_board(my\_board)  
select\_space(my\_board, 6, "X")  
select\_space(my\_board, 8, "O")  
select\_space(my\_board, 9, "X")  
print\_board(my\_board)  
has\_won(my\_board, "X")  
available\_moves(my\_board)  
  
'''  
|-------------|  
| Tic Tac Toe |  
|-------------|  
| |  
| 1 2 X |  
| 4 5 6 |  
| 7 8 9 |  
| |  
|-------------|  
  
|-------------|  
| Tic Tac Toe |  
|-------------|  
| |  
| 1 2 X |  
| 4 O 6 |  
| 7 8 9 |  
| |  
|-------------|  
  
|-------------|  
| Tic Tac Toe |  
|-------------|  
| |  
| 1 2 X |  
| 4 O X |  
| 7 O X |  
| |  
|-------------|  
'''

# Detecting Tic-Tac-Toe Leaves

An essential step in the minimax function is evaluating the strength of a leaf. If the game gets to a certain leaf, we want to know if that was a better outcome for player "X" or for player "O".

Here's one potential evaluation function: a leaf where player "X" wins evaluates to a 1, a leaf where player "O" wins evaluates to a -1, and a leaf that is a tie evaluates to 0.

Let's write this evaluation function for our game of Tic-Tac-Toe.

First, we need to detect whether a board is a leaf — we need know if the game is over. A game of Tic-Tac-Toe is over if either player has won, or if there are no more open spaces. We can write a function that uses has\_won() and available\_moves() to check to see if the game is over.

If the game is over, we now want to evaluate the state of the board. If "X" won, the board should have a value of 1. If "O" won, the board should have a value of -1. If neither player won, it was a tie, and the board should have a value of 0.

**1.**

At the bottom of script.py, create a function called game\_is\_over() that takes a board as a parameter. The function should return True if the game is over and False otherwise.

**2.**

We've given you four different boards to test your function. Call game\_is\_over() on the boards start\_board, x\_won, o\_won, and tie. Print the result of each.

**3.**

Let's write another function called evaluate\_board() that takes board as a parameter. This function will only ever be called if we've detected the game is over. The function should return a 1 if "X" won, a -1 if "O" won, and a 0 otherwise.

**4.**

Test your function on the four different boards! For each board, write an if statement checking if the game is over. If it is, evaluate the board and print the result. You just wrote the base case of the minimax algorithm!

from tic\_tac\_toe import \*  
  
start\_board = [  
 ["1", "2", "3"],  
 ["4", "5", "6"],  
 ["7", "8", "9"]  
]  
  
x\_won = [  
 ["X", "O", "3"],  
 ["4", "X", "O"],  
 ["7", "8", "X"]  
]  
  
o\_won = [  
 ["O", "X", "3"],  
 ["O", "X", "X"],  
 ["O", "8", "9"]  
]  
  
tie = [  
 ["X", "X", "O"],  
 ["O", "O", "X"],  
 ["X", "O", "X"]  
]  
  
def game\_is\_over(board):  
 game\_over = False  
 if has\_won(board, "O"):  
 game\_over = True  
 if has\_won(board, "X"):  
 game\_over = True  
 if len(available\_moves(board)) == 0:  
 game\_over = True  
 return game\_over  
  
def evaluate\_board(board):  
 winner = 0  
 if has\_won(board, "O"):  
 winner = -1  
 if has\_won(board, "X"):  
 winner = 1  
 return winner  
  
  
if(game\_is\_over(start\_board)):  
 print(evaluate\_board(start\_board))  
if(game\_is\_over(x\_won)):  
 print(evaluate\_board(x\_won))  
if(game\_is\_over(o\_won)):  
 print(evaluate\_board(o\_won))  
if(game\_is\_over(tie)):  
 print(evaluate\_board(tie))

# Evaluating Leaves

We now know that we can evaluate the leaves of a game tree, but how does that help us? How are we going to use those values to find the best possible move for a game state that isn't a leaf?

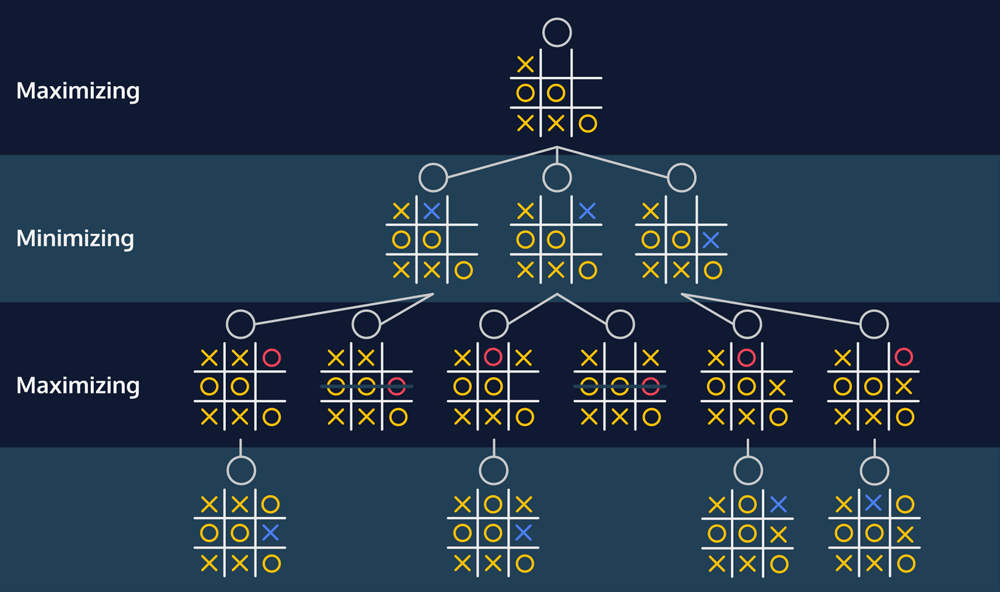
Let's imagine a situation where you're playing as the "X" player in a game of Tic-Tac-Toe and the game is almost over. The game board isn't a leaf but it's close. You have three possible moves. All three moves will immediately end the game — each of those future boards will be leaves.

Let's say picking move A will result in you winning and moves B and C will each result in a tie. You'd clearly pick move A.

By picking move A, you've picked the move that led to the board with the highest value. You were picking between a 1 (an "X" win) or two 0s (the moves that would lead to a tie). Because you picked the move with the highest value, we can say that "X" is the maximizing player.

Let's say you were playing a the "O" player under the same circumstances. Picking move A would somehow immediately lead to "X" winning, while moves B and C would lead to a tie. You'd pick one of the boards that would lead to a tie. "O" is the minimizing player. You would love to pick a board with the value of -1 (an "O" win), but unfortunately, that board doesn't exist. You'll have to settle with picking a board with the value of 0. At least you prevent "X" from winning.

Take a look at the gif below to see the minimax algorithm running on a game of Tic-Tac-Toe where the tree is slightly bigger. The leaf nodes get evaluated and those values get passed up the tree depending based on whether it was the minimizing or maximizing player's turn.



The root of the tree ends up with a value of 0. This means that the best move for the current player is the move that corresponded with the value of 0.

Instructions

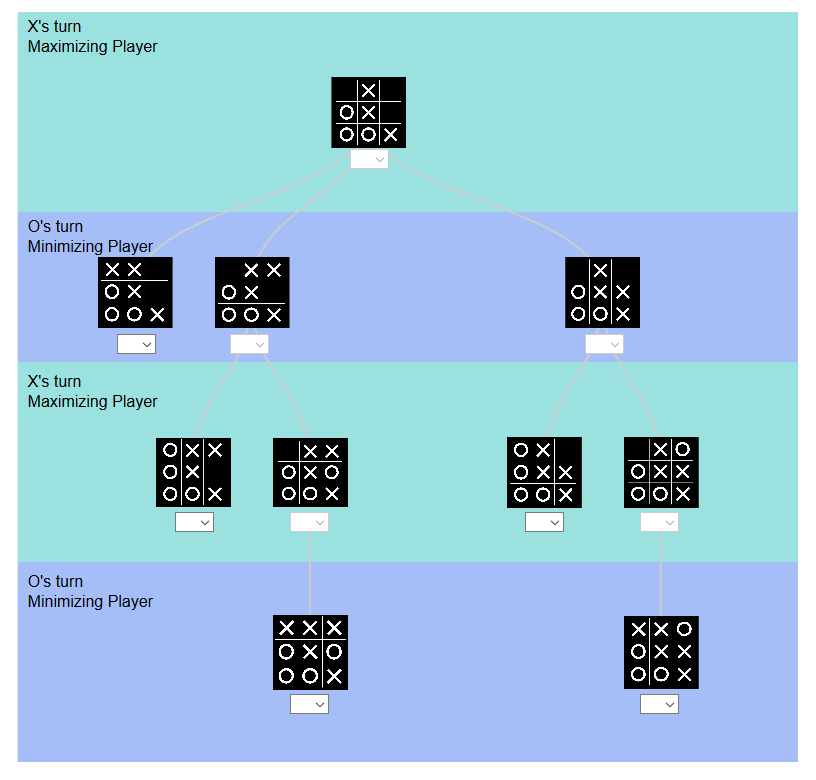
We've made an applet that will walk you through the minimax algorithm.

At the top of the tree, you'll see the current game state. You're the maximizing player ("X") trying to figure out the optimal move.

Begin by evaluating the leaf nodes of the tree. If you evaluate a node correctly, the drop down box will turn green. You can evaluate non-leaf nodes once every child of that node has been correctly evaluated.

The value of a non-leaf will depend on whether it is the maximizing player or minimizing player's turn.

Bubble the values to the top of the tree to find the value of the optimal move.



# Copying Boards

One of the central ideas behind the minimax algorithm is the idea of exploring future hypothetical board states. Essentially, we're saying if we were to make this move, what would happen. As a result, as we're implementing this algorithm in our code, we don't want to actually make our move on the board. We want to make a copy of the board and make the move on that one.

Let's look at how copying works in Python. Let's say we have a board that looks like this

my\_board = [

["X", "2", "3"],

["O", "O", "6"],

["X", "8", "9"]

]

If we want to create a copy of our board our first instinct might be to do something like this

new\_board = my\_board

This won't work the way we want it to! Python objects are saved in memory, and variables point to a location in memory. In this case, new\_board, and my\_board are two variables that point the the same object in memory. If you change a value in one, it will change in the other because they're both pointing to the same object.

One way to solve this problem is to use the deepcopy() function from Python's copy library.

new\_board = deepcopy(my\_board)

new\_board is now a copy of my\_board in a different place in memory. When we change a value in new\_board, the values in my\_board will stay the same!

Instructions

**1.**

Let's begin by failing to create a deep copy of my\_board. Create a variable named new\_board and set it equal to my\_board.

**2.**

Call the select\_space() function using new\_board as a parameter to put an "O" in the center of the board. Print out both new\_board and my\_board using print\_board() to see if making a move on the new board affected the old board.

**3.**

Change how you create new\_board. Set it equal to deepcopy(my\_board). What happens when you call select\_space() now?

from tic\_tac\_toe import \*  
from copy import deepcopy  
  
my\_board = [  
 ["1", "2", "X"],  
 ["4", "5", "6"],  
 ["7", "8", "9"]  
]  
  
new\_board = deepcopy(my\_board)  
select\_space(new\_board, 5, "O")  
print\_board(my\_board)  
print\_board(new\_board)

**The Minimax Function**

We're now ready to dive in and write our minimax() function. The result of this function will be the "value" of the best possible move. In other words, if the function returns a 1, that means a move exists that guarantees that "X" will win. If the function returns a -1, that means that there's nothing that "X" can do to prevent "O" from winning. If the function returns a 0, then the best "X" can do is force a tie (assuming "O" doesn't make a mistake).

Our minimax() function has two parameters. The first is the game state that we're interested in finding the best move. When the minimax() function first gets called, this parameter is the current state of the game. We're asking "what is the best move for the current player right now?"

The second parameter is a boolean named is\_maximizing representing whose turn it is. If is\_maximizing is True, then we know we're working with the maximizing player. This means when we're picking the "best" move from the list of moves, we'll pick the move with the *highest* value. If is\_maximizing is False, then we're the minimizing player and want to pick the minimum value.

Let's start writing our minimax() function.

**1.**

We've started the minimax() function for you and included the base case we wrote a few exercises ago.

We now need to define what should happen if the node isn't a leaf.

We'll want to set up some variables that are different depending on whether is\_maximizing is True or False.

Below the base case, write an if statement to check if is\_maximizing is True.

Inside the if statement, create a variable named best\_value. Since we're working with the maximizing player right now, this is the variable that will keep track of the highest possible value from all of the potential moves.

Right now, we haven't looked at any moves, so we should start best\_value at something lower than the lowest possible value — -float("Inf").

Write an else statement. Inside this else statement we'll be setting up variables for the minimizing player. In this case, best\_value should start at float("Inf").

Return best\_value after the else statement.

**2.**

We now want to loop through all of the possible moves of input\_board before the return statement. We're looking to find the best possible move.

Remember, you can get all of the possible moves by calling available\_moves() using input\_board as a parameter.

After the else statement, but before you return best\_value, loop through all of the possible moves and print each move.

Let's call our function to see these print statements. Outside of your function definition, call minimax() using the parameters x\_winning (the board we're using) and True (we're calling it as the maximizing player).

**3.**

Delete the print statements for move. Rather than just printing the move in this for loop, let's create a copy of the game board and make the move!

Inside the for loop, create a deepcopy of input\_board named new\_board.

Apply the move to new\_board by calling the select\_space() function. select\_space() takes three parameters.

* The board you want to use (new\_board)
* The space you're selecting (the move from the for loop)
* The symbol you're filling the space in with. This is different depending on whether we're the maximizing or minimizing player. In your if and else statements, create a variable named symbol. symbol should be "X" if we're the maximizing player and "O" if we're the minimizing player. Use symbol as the third parameter of select\_space().

To help us check if you accurately made the move, return new\_board outside the for loop (instead of returning best\_move). We'll fix that return statement soon!

from tic\_tac\_toe import \*  
from copy import deepcopy  
  
x\_winning = [  
 ["X", "2", "O"],  
 ["4", "O", "6"],  
 ["7", "8", "X"]  
]  
  
def game\_is\_over(board):  
 return has\_won(board, "X") or has\_won(board, "O") or len(available\_moves(board)) == 0  
  
def evaluate\_board(board):  
 if has\_won(board, "X"):  
 return 1  
 elif has\_won(board, "O"):  
 return -1  
 else:  
 return 0  
  
def minimax(input\_board, is\_maximizing):  
 # Base case - the game is over, so we return the value of the board  
 if game\_is\_over(input\_board):  
 return evaluate\_board(input\_board)  
 if is\_maximizing:  
 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)  
 return new\_board  
  
minimax(x\_winning, True)

# Recursion In Minimax

Nice work! We're halfway through writing our minimax() function — it's time to make the recursive call.

We have our variable called best\_value . We've made a hypothetical board where we've made one of our potential moves. We now want to know whether the value of that board is better than our current best\_value.

In order to find the value of the hypothetical board, we'll call minimax(). But this time our parameters are different! The first parameter isn't the starting board. Instead, it's new\_board, the hypothetical board that we just made.

The second parameter is dependent on whether we're the maximizing or minimizing player. If is\_maximizing is True, then the new parameter should be false False. If is\_maximizing is False, then we should give the recursive call True.

It's like we're taking the new board, passing it to the other player, and asking "what would the value of this board be if we gave it to you?"

To give the recursive call the opposite of is\_maximizing, we can give it not is\_maximizing.

That call to minimax() will return the value of the hypothetical board. We can then compare the value to our best\_value. If the value of the hypothetical board was better than best\_value, then we should make that value the new best\_value.

**1.**

Let's make that recursive call!

Inside the for loop after calling select\_space(), create a variable named hypothetical\_value and set it equal to minimax() using the parameters new\_board and not is\_maximizing.

To help us check if you did this correctly, return hypothetical\_value instead of best\_value. We'll change that return statement soon!

**2.**

Now that we have hypothetical\_value we want to see if it is better than best\_value.

Inside the for loop, write another set of if/else statements checking to see if is\_maximizing is True or False

If is\_maximizing is True, then best\_value should become the value of hypothetical\_value if hypothetical\_value is greater than best\_value.

If is\_maximizing is False, then best\_value should become the value of hypothetical\_value if hypothetical\_value is less than best\_value.

Switch your return statements back to returning best\_value.

**3.**

Wow! Great work, our minimax function is done. We've set up a couple of boards for you. Call minimax() three different times on the boards x\_winning, and o\_winning. In each of those boards, it's "X"'s turn, so set is\_maximizing to True.

Print the results of each. What does it mean if the result is a -1, 0 or 1?

You can also try running minimax() on new\_game. This might take a few seconds — the algorithm needs to traverse the entire game tree to reach the leaves!

from tic\_tac\_toe import \*  
from copy import deepcopy  
  
  
def game\_is\_over(board):  
 return has\_won(board, "X") or has\_won(board, "O") or len(available\_moves(board)) == 0  
  
  
def evaluate\_board(board):  
 if has\_won(board, "X"):  
 return 1  
 elif has\_won(board, "O"):  
 return -1  
 else:  
 return 0  
  
  
new\_game = [  
 ["1", "2", "3"],  
 ["4", "5", "6"],  
 ["7", "8", "9"]  
]  
  
x\_winning = [  
 ["X", "2", "O"],  
 ["4", "O", "6"],  
 ["7", "8", "X"]  
]  
  
o\_winning = [  
 ["X", "X", "O"],  
 ["4", "X", "6"],  
 ["7", "O", "O"]  
]  
  
  
def minimax(input\_board, is\_maximizing):  
 # Base case - the game is over, so we return the value of the board  
 if game\_is\_over(input\_board):  
 return evaluate\_board(input\_board)  
 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)  
 if is\_maximizing and hypothetical\_value > best\_value:  
 best\_value = hypothetical\_value  
 if not is\_maximizing and hypothetical\_value < best\_value:  
 best\_value = hypothetical\_value  
 return best\_value  
  
  
print(minimax(x\_winning, True)) # 1  
print(minimax(o\_winning, True)) # -1  
print(minimax(new\_game, True)) # 0

# Which Move?

Right now our minimax() function is returning the value of the best possible move. So if our final answer is a 1, we know that "X" should be able to win the game. But that doesn't really help us — we know that "X" should win, but we aren't keeping track of what move will cause that!

To do this, we need to make two changes to our algorithm. We first need to set up a variable to keep track of the best move (let's call it best\_move). Whenever the algorithm finds a new best\_value, best\_move variable should be updated to be whatever move resulted in that value.

Second, we want the algorithm to return best\_move at the very end. But in order for the recursion to work, the algorithm is dependent on returning best\_value. To fix this, we'll now return a list of two numbers — [best\_value, best\_move].

Let's edit our minimax function to keep track of which move leads to the best possible value!

**1.**

Instead of returning just the value, we're going to return a list that looks like [best\_value, best\_move].

We haven't kept track of the best move yet, so for now, let's change both of our return statements to be [best\_value, ""]. This includes the base case! The base case should return [evaluate\_board(input\_board), ""]

We also need to make sure when we're setting hypothetical\_value, we're setting it equal to the value — not the entire list. The recursive call should now look like this.

minimax(new\_board, not is\_maximizing)[0]

**2.**

Let's now keep track of which move was best.

Right after the base case, create a variable named best\_move. Set it equal to the empty string ("")

For both the maximizing case and the minimizing case, if we've found a new best\_value, we should also update best\_move. Inside those two if statements, set best\_move equal to your variable from your for loop (ours is named move). We're now remembering which move goes with the best possible value.

Change your last return statement. Instead of returning [best\_value, ""], return [best\_value, best\_move].

**3.**

Call your function on x\_winning, and o\_winning as the maximizing player. Print the results. What does the return value tell you now?

You can also try it on new\_game. This might take a few seconds.

from tic\_tac\_toe import \*  
from copy import deepcopy  
  
def game\_is\_over(board):  
 return has\_won(board, "X") or has\_won(board, "O") or len(available\_moves(board)) == 0  
  
def evaluate\_board(board):  
 if has\_won(board, "X"):  
 return 1  
 elif has\_won(board, "O"):  
 return -1  
 else:  
 return 0  
  
new\_game = [  
 ["1", "2", "3"],  
 ["4", "5", "6"],  
 ["7", "8", "9"]  
]  
  
x\_winning = [  
 ["X", "2", "O"],  
 ["4", "O", "6"],  
 ["7", "8", "X"]  
]  
  
o\_winning = [  
 ["X", "X", "O"],  
 ["4", "X", "6"],  
 ["7", "O", "O"]  
]  
  
def minimax(input\_board, is\_maximizing):  
 # Base case - the game is over, so we return the value of the board  
 if game\_is\_over(input\_board):  
 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)  
 hypothetical\_value = minimax(new\_board, not is\_maximizing)[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]  
  
# This should return [1, 7]. This means that "X" should be able to win the game if they select move 7.  
print(minimax(x\_winning, True))  
# This should return [-1, 4]. This means that no matter what "X" does, "O" will win. "X" might as well select move 4.  
print(minimax(o\_winning, True))  
  
print(minimax(new\_game, True)) # returns [0,1] ... game will end in a tie if both players play appropriately. arbitrarily choose position 1

# Play a Game

Amazing! Our minimax() function is now returning a list of [value, move]. move gives you the number you should pick to play an optimal game of Tic-Tac-Toe for any given game state.

This line of code instructs the AI to make a move as the "X" player:

select\_space(my\_board, minimax(my\_board, True)[1], "X")

Take some time to really understand all of the parameters. Why do we pass True to minimax()? Why do we use [1] at the end of minimax()?

Instructions

Take some time to play a game against the computer. If you're playing with "X"s, make your move as "X", and then call minimax() on the board using is\_maximizing = False. The second item in that list will tell you the AI's move. You can then enter the move for the AI as "O", make your next move as "X", and call the minimax() function again to get the AI's next move.

You can also try having two AIs play each other. If you uncomment the code at the bottom of the file, two AI will play each other until the game is over. What do you think the result will be? The file will run for about 15 seconds before showing you the outcome of the game.

from tic\_tac\_toe import \*  
  
my\_board = [  
 ["1", "2", "3"],  
 ["4", "5", "6"],  
 ["7", "8", "9"]  
]  
  
while not game\_is\_over(my\_board):  
 select\_space(my\_board, minimax(my\_board, True)[1], "X")  
 print\_board(my\_board)  
 if not game\_is\_over(my\_board):  
 select\_space(my\_board, minimax(my\_board, False)[1], "O")  
 print\_board(my\_board)  
  
'''  
|-------------|  
| Tic Tac Toe |  
|-------------|  
| |  
| X 2 3 |  
| 4 5 6 |  
| 7 8 9 |  
| |  
|-------------|  
  
|-------------|  
| Tic Tac Toe |  
|-------------|  
| |  
| X 2 3 |  
| 4 O 6 |  
| 7 8 9 |  
| |  
|-------------|  
  
|-------------|  
| Tic Tac Toe |  
|-------------|  
| |  
| X X 3 |  
| 4 O 6 |  
| 7 8 9 |  
| |  
|-------------|  
  
|-------------|  
| Tic Tac Toe |  
|-------------|  
| |  
| X X O |  
| 4 O 6 |  
| 7 8 9 |  
| |  
|-------------|  
  
|-------------|  
| Tic Tac Toe |  
|-------------|  
| |  
| X X O |  
| 4 O 6 |  
| X 8 9 |  
| |  
|-------------|  
  
|-------------|  
| Tic Tac Toe |  
|-------------|  
| |  
| X X O |  
| O O 6 |  
| X 8 9 |  
| |  
|-------------|  
  
|-------------|  
| Tic Tac Toe |  
|-------------|  
| |  
| X X O |  
| O O X |  
| X 8 9 |  
| |  
|-------------|  
  
|-------------|  
| Tic Tac Toe |  
|-------------|  
| |  
| X X O |  
| O O X |  
| X O 9 |  
| |  
|-------------|  
  
|-------------|  
| Tic Tac Toe |  
|-------------|  
| |  
| X X O |  
| O O X |  
| X O X |  
| |  
|-------------|  
'''

**Review**

Nice work! You implemented the minimax algorithm to create an unbeatable Tic Tac Toe AI! Here are some major takeaways from this lesson.

* A game can be represented as a tree. The current state of the game is the root of the tree, and each potential move is a child of that node. The leaves of the tree are game states where the game has ended (either in a win or a tie).
* The minimax algorithm returns the best possible move for a given game state. It assumes that your opponent will also be using the minimax algorithm to determine their best move.
* Game states can be evaluated and given a specific score. This is relatively easy when the game is over — the score is usually a 1, -1 depending on who won. If the game is a tie, the score is usually a 0.

In our next lesson on the minimax algorithm, we'll look at games that are more complex than Tic Tac Toe. How does the algorithm change if it's too computationally intensive to reach the leaves of the game tree? What strategies can we use to traverse the tree in a smarter way? We'll tackle these questions in our next lesson!

Take a look at our Connect Four AI for a sneak preview of our next minimax lesson. In the terminal type python3 minimax.py to play against the AI.

You can make your move by typing the number of the column that you want to put your piece in.

In the code, you can change the "intelligence" of the AI by changing the parameter of play\_game(). The parameter should be a number greater than 0. If you make it greater than 6 or 7, it will take the computer a *long* time to find their best move.

Make sure to click the Run button to save your code before running your file in the terminal!

You can also set up an AI vs AI game by commenting out play\_game() and calling two\_ai\_game(). This function takes two parameters — the "intelligence" of each AI players. Try starting a game with a bad X player and a smart O player by calling two\_ai\_game(2, 6) and see what happens.

Feel free to test out more games with different AIs.