Foundations of AI
Instructor: Rasika Bhalerao
Assignment 6
Due October 28

This assignment is to write a solver for the game of Tic Tac Toe using the Minimax algorithm.

Learning goal:

- Implement the Minimax algorithm with alpha-beta pruning

What to do:

1. Create a class called `Move` which keeps track of a row, column, and value for a move in Tic Tac Toe.
    a. The constructor should take all three things as arguments and store them to instance variables.
    b. If the client does not pass a value for the row and column, then they should both be initialized to -1 (which should be a constant named `INVALID_COORDINATE` or something that indicates that it is an invalid coordinate). The value is a required argument.
2. Create an enumerated type called `Player` which can be `X` or `O`.
3. Create a class called `GameState`, which has one instance variable: a 2D array (list) of `Player`s called `board`.
    a. The constructor should initialize the board to a 3 x 3 board full of `None`.
    b. Add a method called `game_over()` which returns `True` if the game is over, and `False` otherwise.
        i. The game is over if either player (X or O) has three pieces in the same row, same column, or same diagonal.
        ii. The game is also over if the entire board is full.
    c. Add a method called `winner()` which returns the `Player` who won the game.
        i. If the game is not yet over, it should return `None`.
        ii. If the game is over, but there was no winner (the board was full and nobody got 3 in the same row/col/diag), it should return `None`.
            1. Make sure to watch out for this common bug: an empty board saying the game is over because all three `None`s in the first row are equal to each other.
    d. The `__str__()` function should return a readable representation of the board state.
    e. Add a method called `spot(row, col)` which returns the piece that is in the given position on the board (or `None` if that spot is empty).

f. Add a method called `move(row, col, player)`. **It should not modify the current `GameState`.** Instead, it should return a new `GameState`, which is a copy of the current one, but with the additional piece placed in the provided spot on the board.
    i. If that spot was already taken, it should return `None` or raise an error (either is fine).

4. Create a class called `TicTacToeSolver`. It will do the Minimax algorithm. It should have these methods:
    a. `find_best_move(state: GameState, player: Player)` which takes a `GameState` and a `Player`, appropriately calls the `solve_my_move` method (below), and returns the best move for this player to make given the current state.
    b. `solve_my_move(state: GameState, alpha: float, beta: float, player: Player)` which implements this pseudocode:
        i. If the game is over, return the score for us (the player whose score we want to maximize):
            1. 1 if the winner was `player` (the argument)
            2. -1 if the winner was the opposite player
            3. 0 if it was a draw (no winner)
        ii. Otherwise, keep track of a `Move` variable called `best_move`. Initialize it to `None`.
            1. For each of the empty spots in `state`:
                a. Call `solve_opponent_move` (below) appropriately and store the result in a variable called `child` of type `Move`.
                b. If `best_move` is `None`, or if `child`'s `value` is higher than `best_move`'s `value`,
                    i. Make `best_move` = a new `Move` with the given row / col of the empty spot, and the `value` of the `child`
                c. To implement alpha-beta pruning:
                    i. If `best_move`'s `value` is higher than `beta`, then return `best_move`
                    ii. `alpha` should be updated to be the smallest `best_move` `value` found so far (using the cumulative sum pattern)
            2. At the end, return `best_move`.

c. `solve_opponent_move(state: GameState, alpha: float, beta: float, player: Player)`, which is implemented very similarly to `solve_my_move()`.
    i. Please adapt the pseudocode given above for `solve_my_move()` using the alpha-beta pruning pseudocode provided during lecture. Note that we do not keep track of the depth.
    ii. Don't forget to return the opposite (negative) of the score used in the base case of `solve_my_move()`

5. Test it out using a `main()` function!
    a. It should create a new empty `GameState`, and continue to find the best move for each player (from each player's point of view as the maximizing score player, respectively) in turn until the game is over.
    b. Make sure to print the `GameState`s along the way to see what it chose!
    c. If everything is implemented correctly, and both sides are playing optimally, there should be no winner.

What to turn in:
Please submit these via Gradescope:

- Your Python code
- A text or pdf file with your answers to these questions:
    - Questions specific to this assignment:
        - In `solve_opponent_move()`, why do we return the opposite (negative) of the score used in the base case of `solve_my_move()`?
        - Why is there no winner if both sides are playing optimally?
        - This pseudocode includes alpha-beta pruning. What would be the pseudocode for implementing this same algorithm, but without alpha-beta pruning?
    - The usual questions:
        - How long did this assignment take you? (1 sentence)
        - Whom did you work with, and how? (1 sentence each)
            - Discussing the assignment with others is encouraged, as long as you don't share the code.
        - Which resources did you use? (1 sentence each)
            - For each, please list the URL and a brief description of how it was useful.
        - A few sentences about:
            - What was the most difficult part of the assignment?
            - What was the most rewarding part of the assignment?
            - What did you learn doing the assignment?
            - Constructive and actionable suggestions for improving assignments, office hours, and class time are always welcome.