**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.
   1. The constructor should take all three things as arguments and store them to instance variables.
   2. 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](https://docs.python.org/3/library/enum.html) called Player which can be X or O.
3. Create a class called GameState, which has one instance variable: a 2D array (list) of Players called board.
   1. The constructor should initialize the board to a 3 x 3 board full of None.
   2. Add a method called game\_over() which returns True if the game is over, and False otherwise.
      1. The game is over if either player (X or O) has three pieces in the same row, same column, or same diagonal.
      2. The game is also over if the entire board is full.
   3. Add a method called winner() which returns the Player who won the game.
      1. If the game is not yet over, it should return None.
      2. 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 Nones in the first row are equal to each other.
   4. The \_\_str\_\_() function should return a readable representation of the board state.
   5. 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).
   6. 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.
      1. 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:
   1. 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.
   2. solve\_my\_move(state: GameState, alpha: float, beta: float, player: Player) which implements this pseudocode:
      1. 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)
      2. Otherwise, keep track of a Move variable called best\_move. Initialize it to None.
         1. For each of the empty spots in state:
            1. Call solve\_opponent\_move (below) appropriately and store the result in a variable called child of type Move.
            2. If best\_move is None, or if child’s value is higher than best\_move’s value,

Make best\_move = a new Move with the given row / col of the empty spot, and the value of the child

* + - * 1. To implement alpha-beta pruning:

If best\_move’s value is higher than beta, then return best\_move

alpha should be updated to be the smallest best\_move value found so far (using the cumulative sum pattern)

* + - 1. At the end, return best\_move.
  1. solve\_opponent\_move(state: GameState, alpha: float, beta: float, player: Player), which is implemented very similarly to solve\_my\_move().
     1. 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.
     2. Don’t forget to return the opposite (negative) of the score used in the base case of solve\_my\_move()

1. Test it out using a main() function!
   1. 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.
   2. Make sure to print the GameStates along the way to see what it chose!
   3. 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.