Tic-Tac-Toe Using Minimax, implement an AI to play Tic-Tac-Toe optimally.

Given that parts of this course material were originally from 2020, the **latest** version of Python you should use in this course is **Python 3.10**.

Getting Started

- Download the distribution code from Canvas.
- Once in the directory for the project, run `pip3 install -r requirements.txt` to install the required Python package (`pygame`) for this project.

Understanding

There are two main files in this project: **runner.py and tictactoe.py**. **tictactoe.py** contains the logic for playing the game, and for making optimal moves. It is partially implemented for you. **runner.py** has been implemented for you, and contains all of the code to run the graphical interface for the game. Once you've completed all the required functions in `tictactoe.py`, you should be able to run **python runner.py** to play against your AI!

Specification

Open **tictactoe.py** to get an understanding of what's provided. First, we define three variables: `X`, `O`, and `EMPTY`, to represent possible moves of the board. You are provided with the functions:

- The **initial_state** function returns the starting state of the board. For this problem, we've chosen to represent the board as a list of three lists (representing the three rows of the board), where each internal list contains three values that are either `X`, `O`, or `EMPTY`. What follows are functions that we've left up to you to implement!

Complete the implementations of actions, winner, and minimax.

- The **player function** takes a board state as input, and return which player's turn it is (either `X` or `O`).
  - **In the initial game state, `X` gets the first move.** Subsequently, the player alternates with each additional move.
  - Any return value is acceptable if a terminal board is provided as input (i.e., the game is already over).
- The **actions function** should return a set of all of the possible actions that can be taken on a given board. **(You are to implement)**
  - Each action should be represented as a tuple `(i, j)` where `i` corresponds to the row of the move (`0`, `1`, or `2`) and `j` corresponds to which cell in the row corresponds to the move (also `0`, `1`, or `2`).
  - Possible moves are any cells on the board that do not already have an `X` or an `O` in them.
  - Any return value is acceptable if a terminal board is provided as input.

- The **result function** takes a board and an action as input, and returns a new board state, without modifying the original board.
    - If action is not a valid action for the board, an exception is raised.
    - The returned board state is the board that would result from taking the original input board, and letting the player whose turn it is make their move at the cell indicated by the input action.
    - Importantly, the original board is left unmodified: since Minimax will ultimately require considering many different board states during its computation. This means that simply updating a cell in board itself is not a correct implementation of the result function. It makes a deep copy of the board first before making any changes.
- The **winner function** should accept a board as input, and return the winner of the board if there is one. **(You are to implement)**
    - If the X player has won the game, the function returns X. If the O player has won the game, the function returns O.
    - The game is won with three of a player's moves in a row horizontally, vertically, or diagonally.
    - You may assume that there will be at most one winner (that is, no board will ever have both players with three-in-a-row, since that would be an invalid board state).
    - If there is no winner of the game (either because the game is in progress, or because it ended in a tie), the function should return None.
- The **terminal function** accepts a board as input, and return a boolean value indicating whether the game is over.
    - If the game is over, either because someone has won the game or because all cells have been filled without anyone winning, the function returns True.
    - Otherwise, the function returns False if the game is still in progress.
- The **utility function** accepts a terminal board as input and outputs the utility of the board.
    - If X has won the game, the utility is 1. If O has won the game, the utility is -1. If the game has ended in a tie, the utility is 0.
    - You may assume utility will only be called on a board if terminal(board) is True.
- The **minimax function** should take a board as input, and return the optimal move for the player to move on that board. **(You are to implement)**
    - The move returned should be the optimal action (i, j) that is one of the allowable actions on the board. If multiple moves are equally optimal, any of those moves is acceptable.
    - If the board is a terminal board, the minimax function should return None.

For all functions that accept a board as input, you may assume that it is a valid board (namely, that it is a list that contains three rows, each with three values of either X, O, or EMPTY). You should not modify the function declarations (the order or number of arguments to each function) provided.

Once all functions are implemented correctly, you should be able to run python runner.py and play against your AI. And, since Tic-Tac-Toe is a tie given optimal play by both sides, you should never be able to beat the AI (though if you don't play optimally as well, it may beat you!)

Hints

- If you'd like to test your functions in a different Python file, you can import them with lines like from tictactoe import initial_state.
- You're welcome to add additional helper functions to tictactoe.py, provided that their names do not collide with function or variable names already in the module.
- Alpha-beta pruning is **optional**, but may make your AI run more efficiently! **Clearly state in your comments for the minimax function whether or not you are using Alpha-beta pruning**.

Submit to Canvas

Only the tictactoe.py file. Make sure it will run unaltered invoked by the runner.py file