Artificial Intelligence Project 2

Category: Adversarial Search and Games

Objective:

In this programming project, you will implement a Tic-Tac-Toe game with a graphical user interface (GUI) using Python and the Pygame library. The game will feature an AI opponent who plays optimally using the Minimax algorithm for adversarial search. The goal is to provide students with hands-on experience in implementing game-playing agents and understanding adversarial search algorithms.

Requirements:

1. Implement the Tic-Tac-Toe game with a graphical interface using Pygame.
2. Develop an AI opponent using the Minimax algorithm to play against the human player.
3. Display the current state of the game and allow players to make moves using mouse clicks.
4. Ensure that the AI opponent plays optimally using the Minimax algorithm.
5. Provide visual feedback to users about the game progress and outcome.
6. Include a restart button to allow users to play multiple rounds.
7. Comment your code thoroughly to explain the logic and functionality of each component.

Clearly mention the contribution of each partner in the project. Optionally, you can complete this project **individually** as well.

Step-by-Step Instructions:

1. Set up your development environment with Python and Pygame installed.
2. Create a folder named 'Project2_CISXXX_W2024' in your root directory.
3. Download the distribution Python code from the Project 2 page and unzip it into the Project2 folder.
4. Open a "Command Prompt" and switch to the directory for the project2
5. Run ***pip3 install -r requirements.txt*** t o install the required Python package (**pygame)** for this project.
6. There are two main files in this project: ***runner.py*** and ***tictactoe.py.***
7. ***runner.py*** has been implemented for you and contains all the code to run the graphical interface for the game.
8. ***tictactoe.py*** contains all the logic for playing the game (for your to-do), and for making optimal moves. You need to complete all the required functions in ***tictactoe.py***.
9. You should be able to run ***python runner.py*** to play against your AI!
10. Let's open up ***tictactoe.py*** to get an understanding for what's provided. First, we define three variables: ***X, O***, and ***EMPTY***, to represent possible moves of the board.
11. The function ***initial_state*** returns the starting state of the board. For this problem, we've chosen to represent the board (3 X 3) 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**.

Complete the implementations of ***player, actions, result, winner, terminal, utility***, and ***minimax*** functions in the file, ***tictactoe.py.***

- The ***player*** function should take 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.
    - 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 should return a new board state, without modifying the original board.
    - If ***action*** is not a valid action for the board, your program should raise an exception.
    - The returned board state should be 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 should be 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. You'll likely want to make 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.
    - If the **X** player has won the game, your function should return **X**. If the **O** player has won the game, your function should return **O**.
    - One can win the game with three of their 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 should accept 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 should return ***True***.
    - Otherwise, the function should return ***False*** if the game is still in progress.
- The ***utility*** function should accept a terminal ***board*** as input and output 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.
    - 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*** from the command line 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!)

12. Test your implementation thoroughly to ensure that the game functions correctly and the AI opponent plays optimally.
13. Add comments to your code to explain the logic and functionality of each component.
14. Prepare your project submission with the necessary documentation and any additional instructions for running the game.

Submission Guidelines:

- Submit the Python source code in zip format along with any necessary instructions for running the program.
- Include a README file with details on how to install dependencies, compile, and execute the Python code.
- Submit the report in a pdf format that shows stats of **10 runs** of the game. You will document how many times you won and how many times AI won and hown many times there was a tie.

Grading Rubric (100 points):

1. **Tic-Tac-Toe Game Implementation (70 points):**
    - ***Player*** function implemented correctly **(10 points)**
    - ***Actions*** function implemented correctly **(10 points)**
    - ***Result*** function implemented correctly **(10 points)**
    - ***Winner*** function implemented correctly **(10 points)**
    - ***terminal*** function implemented correctly **(10 points)**
    - ***utility*** function implemented correctly **(10 points)**
    - ***minimax*** function implemented correctly **(10 points)**
2. **Code Quality and Comments (10 points):**
    - The code is well-organized and easy to read (5 points).
    - Includes sufficient comments to explain the logic and functionality (5 points).
3. **Report (20 points)**
    - Document 10 runs of the game in tabular form that shows, run number, who won, who lost, was a tie.
    -

Total Points: 100