Overview

We have previously seen Tic-Tac-Toe in part 1 of this class. In this assignment, we are going to revisit the game and develop an alternative strategy to play the game.

For this assignment, your task is to implement a machine player for Tic-Tac-Toe that uses a Minimax strategy to decide its next move. You will be able to use the same console-based interface and graphical user interface to play the game as you did before. Although the game is played on a 3×3 grid, your version should be able to handle any square grid (however, the time it will take to search the tree for larger grid sizes will be prohibitively slow). We will continue to use the same grid conventions that we have used previously.

This project does not require you to write a lot of code. It does, however, bring together a lot of concepts that we have previously seen in the class. We would like you to *think* about how these concepts are coming together to enable you to build a relatively complex machine player with very little code. Further, you should think about the situations in which Minimax or Monte Carlo might produce better/worse machine players for games other than Tic-Tac-Toe.

Provided Code

We have provided a `TTTBoard` class for you to use. This class keeps track of the current state of the game board. You should familiarize yourself with the interface to the `TTTBoard` class in the `poc_ttt_provided` module. The provided module also has a `switch_player(player)` function that returns the other player (`PLAYERX` or `PLAYERO`). The provided module defines the constants `EMPTY`, `PLAYERX`, `PLAYERO`, and `DRAW` for you to use in your code. The provided `TTTBoard` class and GUI use these same constants, so you will need to use them in your code, as well.

At the bottom of the provided template, there are example calls to the GUI and console game player. You may uncomment and modify these during the development of your machine player to actually use it in the game. Note that these are the same calls we used previously for your Monte Carlo strategy, so they take an `ntrials` parameter. You can pass anything you want as `ntrials`, since you will not be using it for Minimax. In order to allow us to use the same infrastructure, we have also provided a `move_wrapper` function in the template that you can pass to `play_game` and `run_gui`. This wrapper simply translates between the inputs and outputs of your function and those that were expected if you were implementing a Monte Carlo player.

Testing your mini-project

As you implement your machine player, we suggest that you build your own collection of tests using the `poc_simpletest` module that we have provided. Please review this page for an overview of the capabilities of this module. These tests can be organized into a separate test suite that you can import and run in your program.

IMPORTANT: In this project, you will use Minimax to search the entire game tree. If you start with an empty Tic-Tac-Toe board, it will take a long time for your strategy to search the tree. Therefore, we *strongly* suggest that you write tests that start with a partially full board. This will allow your code to run much faster and will lead to a more pleasant development and debugging experience.

Finally, submit your code (with the calls to `play_game` and `run_gui` commented out) to this Owltest page. This page will automatically test your mini-project. It will run faster if you comment out the calls to `play_game` and `run_gui` before submitting. Note that trying to debug your mini-project using the tests in OwlTest can be very tedious since they are slow and give limited feedback. Instead, we *strongly* suggest that you first test your program using your own test suite and the provided GUI. Programs that pass these tests are much more likely to pass the OwlTest tests.

Remember that OwlTest uses Pylint to check that you have followed the coding style guidelines for this class. Deviations from these style guidelines will result in deductions from your final score. Please read the feedback from Pylint closely. If you have questions, feel free to consult this page and the class forums.

When you are ready to submit your code to be graded formally, submit your code to the CourseraTest page for this mini-project that is linked on the main assignment page.

Machine Player Strategy

Your machine player should use a Minimax strategy to choose the next move from a given Tic-Tac-Toe position. As the objective of this assignment is to help you bring together what you have learned, we ask that you do not search for pseudo-code to implement Minimax. At this point in the class, we hope that you can use the examples in the lectures and an English language description and be able to implement Minimax.

The general idea on Minimax is to search the entire game tree alternating between minimizing and maximizing the score at each level. For this to work, you need to start at the bottom of the tree and work back towards the root. However, instead of actually building the game tree to do this, you should use recursion to search the tree in a depth-first manner. Your recursive function should call itself on each child of the current board position and then pick the move that maximizes (or minimizes, as appropriate) the score. If you do this, your recursive function will naturally explore all the way down to the bottom of the tree along each path in turn, leading to a depth first search that will implement Minimax. The following page describes the process in more detail.

As you recursively call your minimax function, you should create a copy of the board to pass to the next level. When the function returns, you no longer need that copy of the board. In this manner, you are dynamically constructing the part of the game tree that you are actively looking at, but you do not need to keep it around.

For this mini-project, you need only implement one function:

- `mm_move(board, player):` This function takes a current board and which player should move next. The function should use Minimax to return a tuple which is a score for the current board and the best move for the player in the form of a `(row, column)` tuple. In situations in which the game is over, you should return a valid score and the move `(-1, -1)`. As `(-1, -1)` is an illegal move, it should only be returned in cases where there is no move that can be made.

You should start from this code that imports the Tic-Tac-Toe class and a wrapper function to enable you to play the game. You may add extra helper functions if so desired.

Hints

You do not need to write a lot of code to implement Minimax, but it can be difficult to get everything working correctly. Here are some hints that may help you out:

- Do not forget the base case in your recursive function. Think carefully about when you can return with an answer immediately.
- Remember to make a copy of the board before you recursively call your function. If you do not, you will modify the current board and you will not be searching the correct tree.
- The `SCORES` dictionary is useful. You should use it to score a completed board. For example, the score of a board in which X has won should be `SCORES[provided.PLAYERX]`. If the game is a draw, you should score the board as `0`.
- In Minimax, you need to alternate between maximizing and minimizing. Given the `SCORES` that we have provided you with, player X is *always* the maximizing player and play O is *always* the minimizing player. You can use an `if`-`else` statement to decide when to maximize and when to minimize. But, you can also be more clever by noticing that if you multiply the score by `SCORES[player]` then you can always maximize. Why? Because this has the effect of negating player O's scores allowing you to maximize instead of minimize for player O.
- Minimax can be slow when there are a lot of moves to explore. The way we have set up the scoring, you do not always need to search everything. If you find a move that yields a winning score (+1 for X or -1 for O), you know that you cannot do any better by continuing to search the other possible moves from the current board. So, you can just return immediately with the score and move at that point. This will significantly speed up the search.

Mark as completed