

CS50's Introduction to Artificial Intelligence with Python

 cs50.harvard.edu/ai/2020/projects/0/tictactoe

Tic-Tac-Toe

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



When to Do It

By Monday, January 1, 2024 at 10:29 AM GMT+5:30.

How to Get Help

1. Ask questions via [Ed!](#)
2. Ask questions via any of CS50's [communities!](#)

Getting Started

- Download the distribution code from <https://cdn.cs50.net/ai/2020/x/projects/0/tictactoe.zip> and unzip it.
- 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 all of the logic for playing the game, and for making optimal moves. `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!

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.

The function `initial_state` 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!

Specification

An automated tool assists the staff in enforcing the constraints in the below specification. Your submission will fail if any of these are not handled properly, if you import modules other than those explicitly allowed, or if you modify functions other than as permitted.

Complete the implementations of `player`, `actions`, `result`, `winner`, `terminal`, `utility`, and `minimax`.

- 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` 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!

How to Submit

You may not have your code in your `ai50/projects/2020/x/tictactoe` branch nested within any further subdirectories (such as a subdirectory called `tictactoe` or `project0b`). That is to say, if the staff attempts to access

<https://github.com/me50/USERNAME/blob/ai50/projects/2020/x/tictactoe/tictactoe.py> , where **USERNAME** is your GitHub username, that is exactly where your file should live. If your file is not at that location when the staff attempts to grade, your submission will fail.

1. Visit [this link](#), log in with your GitHub account, and click **Authorize cs50**. Then, check the box indicating that you'd like to grant course staff access to your submissions, and click **Join course**.
2. Install Git and, optionally, install submit50.
3. If you've installed **submit50** , execute

```
submit50 ai50/projects/2020/x/tictactoe
```

Otherwise, using Git, push your work to <https://github.com/me50/USERNAME.git> , where **USERNAME** is your GitHub username, on a branch called [ai50/projects/2020/x/tictactoe](#) .

4. Submit [this form](#).

You can then go to <https://cs50.me/cs50ai> to view your current progress!