Minimax Turn-Based Game AI with Tic-Tac-Toe

Eckel, TJHSST AI1, Fall 2020

Background & Explanation

One of the most famous applications of artificial intelligence is in playing games, usually against a human player. This assignment begins our explorations of game AI with an extremely simple game that we can solve completely.

The first set of questions will check to make sure you've modeled Tic Tac Toe correctly. Then, we'll implement the minimax algorithm. Be sure to watch the video on the website that introduces this algorithm & assignment. For your reference, here's the pseudocode for half of the minimax algorithm (the max part):

```
def max_step(board):
    if game over:
        return score
    results = list()
    for next_board in possible_next_boards(board, current_player):
        results.append(min_step(next_board))
    return max(results)
```

This pseudocode is a bit more of a sketch than past algorithms; you'll need to figure out many things here, including how to check if the game is over, how to score the board, and how to generate the possible next boards.

The min part of the minimax algorithm should be relatively easy to determine given the code for the max part.

Part 1: Model Tic-Tac-Toe

1) Model a 3x3 Tic-Tac-Toe board as a 9 character string where "." means empty and the players are "X" and "O". Please use capital letters to match my grading script. By convention, we will say "X" always moves first. For any board, determine if the game is over, and if so whether X wins, O wins, or there is a tie. (Note that the board does not have to be full for the game to be over.) If the game is not over, determine the set of all possible moves remaining.

2) What is the total number of distinct games that may be played? How many different final boards result?

   This question is often confusing to students, so to be clear:
   - Two different sequences of moves that result in the same final board should count as two different "games" but only one "board".
   - A board doesn't need to be full to be final. Games end as soon as anyone wins OR if the board is full

   Total games: _____   Total final boards: _____

3) Considering the distinct **final boards** (not games), how many of them are draws? In how many of them does X win in 5 steps, 7 steps, or 9 steps, respectively? In how many of them does O win in 6 or 8 steps, respectively? **Note:** these 6 numbers should add up to the same number as your total final **boards**, **not** total **games.**

   X in 5: _____   X in 7: _____   X in 9: _____
   O in 6: _____   O in 8: _____   Draws: _____

   DM me your answers to #2 and #3 on Mattermost to verify they're correct. (Or DM another student who's already done so; I'm not keeping track of this, I just want you to make sure they're right.)

Part 2: Implement Minimax

Implement the Minimax algorithm so that your AI plays perfectly.

This is a bit finickier than you might expect. Note that the Minimax algorithm does not choose a move, it just returns the expected value of a board. So, you'll need different code to choose which move to make. If the AI is playing as player X, there needs to be a function that calls the min part of minimax after generating all the possible boards, but instead of just returning the maximum available value, it returns the *move that results in* that available value. Ie, you'll need a max_step function and a max_move function, both of which call min_step on every available board, but which return different things. If that doesn't make sense to you, please don't start coding yet – ask questions of myself or a peer until the pieces all click.

In the past, the most successful students have seemed to start with the output and work backwards from there. To that end, there are four sample runs on the course website for you to compare with. Yours should match almost exactly, with one exception – if there are multiple equivalent moves, your AI does not need to choose the same move as mine. (In particular, if there is a move that wins immediately, and a different move that will guarantee a win two turns later, it's perfectly fine for your code to choose either one.)

Read the following directions **carefully**; this is one of those assignments where students tend to generally get the idea but then not meet spec because of missing directions or edge cases. **Double check this** with unusual care.

Specifically, your code should:

- Accept one **command line argument** – a 9-character Tic Tac Toe board.
- If the board is empty, **ask the user** (using an input statement) whether the computer should go first or the user should go first. X always plays first, but either the computer or the user might be player X.
- If the board is **not** empty, **assume that the computer plays next** and **figure out which token the computer should be.** (Since X plays first, if there are an equal number of Xs and Os, X should play next. Otherwise, O.)
- At each **computer** turn, your script should **print out** all possible moves and classify them as L (losing), T (tying), or W (winning), under the assumption of perfect play from both sides going forward.
- At each **player** turn, just display the board and ask the player to type in a number from 0 to 8 to move.
- Play perfectly.

Some frequent mistakes include:

- Students often assume the computer always plays X. This is not true; please read above carefully.
- Students often assume the computer always plays first. This is not true on a blank board!
- Students often print out all of the possible moves for the player as well as the AI. The AI should "show its work", so to speak, but when it is the user's turn, it should just display the current board and ask for a move.
- Make sure that you check to see if the given board is already solved. If you read in a board that already contains a victory for X or O, you should just say that, and no one should move. Students often put these steps out of order.

Check all four sample runs on the website against your code. There are four runs for a reason – many different scenarios are possible. Ensure your code responds correctly in **all** of them!

Specification

Submit a single Python script to the link on the course website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- Your code does all of the above as specified.

Specification for Outstanding Work: Negamax Tic-Tac-Toe

Watch the video about Negamax. Re-implement Minimax as Negamax. I strongly recommend you do **not** do Negamax *first* unless you frequently find writing extension-difficulty algorithms in this class easy. It is harder than you think, and having code that works using normal minimax first to compare against is really helpful.

Submit a single Python script to the link on the course website.

This extension **receives outstanding work credit** if:

- It meets the entire original specification, but using Negamax instead correctly.
- The code is commented so that I can easily open it, find the Negamax function, and verify that it is, indeed, Negamax. This means choosing good variable names and commenting not only the function but several lines within the function to explain how they're executing the algorithm.