CS4300 Spring 2018 Hauschild

Project 2: Numerical Tictactoe 4 rows x 4 columns
Due Date: Mar 8th, 2018

Submission: Please submit your completed source files and any documentation through MyGateway submission system. If you have any questions, please feel free to send me emails or stop by my office. **Please do not email your actual submission to me though.**

In this project your task is to implement a computer numerical tic-tac-toe player on a board of size 4.

Numerical tic-tac-toe is similar to normal tic-tac-toe except instead of X's and Os', the two players are given the numbers {1,3,5,7,9,11,13,15} and {2,4,6,8,10,12,14,16} respectively. The players take turns (with the odd player going first) and at each round the players put one of their unused numbers on an open spot on the board. The goal is to have the 4 numbers in a line total to 34 (which is actually 4 times the average of the numbers 1 through 16).

I want you to implement two different ways to play your game:

1) The computer against a human player, with the player allowed to choose to go first or second.
2) Your computer player against itself, taking alternating moves, stopping between moves to allow a human watcher to hit a key to continue.

Specifications:

1) You must use minimax (or alpha-beta pruning). You are not allowed to simply use lookup rules or something similar.

2) Each turn, display the board and let them either hit a space to move on to the next board in the case of computer vs computer or to enter a number 0-15, 0 being the top left position, 15 being the bottom right. Do not prompt this choice by row and column, if necessary convert it yourself.

3) In all cases display minimax result from all possible moves and then the board state resulting from the move.

4) You must include a README file specifying what you have implemented, your evaluation function for cutoff of iterative deepening later on, and any outstanding problems you still have or managed to solve.

The search space for this problem is fairly large, so you will definitely not want to fully create your state space tree before doing minimax on it. You should implement minimax with iterative deepening search, so start searching for a depth of 1, then a depth of 2, then a depth of 3. If during your search you realize that more than 2 seconds have passed, you should terminate the current search and use the result from the previous depth that was completely searched. For example, if while searching depth of 5, you find that two total seconds have passed since trying to make your move, you should terminate that search and do the best move you found in the previous depth limited search of 4.

Implement this incrementally! I would suggest doing something along these lines:

1) Implement a node that stores a state and a function to evaluate whether it is a win.
2) Write a stub function that given a current state, returns a move and the set of minimax values that it had to choose from. Implement this as a stub, so just return a random move and some dummy minimax values.
3) Implement it playing against itself and against a human player. With the stub implemented, this will be the equivalent of playing against a random player. Make sure the game works at this point.
4) Start implementing your stub, I would suggest implementing depth limited search, so your stub always does depth-limited search to some n, where n=2 fixed. Do this with minimax first, even if you want to do alpha-beta pruning later.
    1) Note that his will require some evaluation function for a state that is not a win, loss or tie.
5) Implement iterative deepening search based on iteratively calling the depth-limited search as above.

Extra credit: 10 points for using alpha-beta pruning, though you will not get 10 points just for "trying" it and not having something I can test. That is, make sure minimax works before trying to get extra credit!