CSE 537 Project 2
Game Search
Due: Oct. 8 23:59

In this homework you will practice adversarial game search techniques, e.g. minimax search, alpha-beta pruning and evaluation function design.
You will need to get the code framework from Blackboard. This project is adapted from one from MIT, so there may be some inconsistency between the comment in code and instructions here but **please stick to THIS DOCUMENT**. You are required to develop your own solution. You cannot copy solutions from elsewhere. We will run plagiarism software. Academic dishonesty will not be tolerated. You will be asked about specific algorithm details when you do the demo.

Submission:
Pack your code and project report in a ZIP file and submit through Blackboard.

Game Info:
This project focuses on the game Connect Four (https://en.wikipedia.org/wiki/Connect_Four).
In this game, the board is a 7x6 grid of possible positions

```
  0 1 2 3 4 5 6
0 * * * * * * *
1 * * * * * * *
2 * * * * * * *
3 * * * * * * *
4 * * * * * * *
5 * * * * * * *
6 * * * * * * *
```

Two players take turns alternately adding tokens to the board. Tokens can be added to any column that is not full (ie., does not already contain 6 tokens). When a token is added, it immediately falls to the lowest unoccupied cell in the column. The game is won by the first player to have four tokens lined up in a row, either vertically or horizontally, or along a diagonal.

What to do:

0. Playing the game

You can get a feel for how the game works by playing it against a baby AI player. For example, by uncommenting this line in `lab3.py`, you can play black, while a computer player that randomly picks a column plays white:

```
run_game(random_player, human_player)
```

For each move, the program will prompt you to make a choice, by choosing what column to add a token to.

1. (20pts) Minimax Search

In this part you will implement the minimax search algorithm. You need to fill in the `minimax` function in the file `basicplayer.py`, which must take the following arguments:
`board` -- The ConnectFourBoard instance representing the current state of the game
`depth` -- The maximum depth of the search tree to scan.
`eval_fn` -- The "evaluate" function to use to evaluate board positions. For this question you will be using `basic_evaluate`, as specified by the function header.

And optionally it takes two more function arguments:
`get_next_moves_fn` -- a function that given a board/state, returns the successor board/states. By default `get_next_moves_fn` takes on the value of `basicplayer.get_all_next_moves`
`is_terminal_fn` -- a function that given a depth and board/state, returns True or False. True if the board/state is a terminal node, and that static evaluation should take place. By default `is_terminal_fn` takes on the value of `basicplayer.is_terminal`

You should use these functions in your implementation to find next board/states and check termination conditions. The search should return the **column** number that you want to add a token to. If you are experiencing massive tester errors, make sure that you're returning the column number and not the entire board!

After you're done with the part, replace the start command in `lab3.py` with the following to test it out:

```
run_game(basic_player, human_player)
```

or have 2 AIs play the game if you like.

TIP: We've added a file called `tree_searcher.py` to help you debug problems with your implementation; it's for alpha-beta-search, but you can easily modify it for testing minimax search. It contains code that will test your search on static game trees of the kind that you can work out by hand. To debug your search, you should run: `python tree_searcher.py`; and visually check the output to see if your code return the correct expected next moves on simple game trees. Only after you've passed tree_searcher then should you go on and run the full tester.

2. (20pts) Better Evaluation Function
In this part you will devise a better evaluation function to aggressively increase the streak of the AI player. You need to create a `new_evaluate` function in `basicplayer.py` and uncomment the following line:

```
new_player = lambda board: minimax(board, depth=4,
eval_fn=new_evaluate)
```

And add `new_player` to the game in `lab3.py` with the start command:

```
run_game(new_player, ...)
```

3. (30pts) Alpha-Beta Search

In this part you will implement the alpha-beta-search algorithm. You need to fill in the `alpha_beta_search` function in the file `lab3.py`. Alpha-beta-search should give you the same result as minimax. You may still use tree_searcher to test it out before adding `alphabeta_player` to the game:

```
run_game(alphabeta_player, ...)
```

4. Generalization of the Game

For the following 2 parts you will be using alpha-beta-search but modify the game setting. Notice that the changes made are twofold: (a) the program needs to decide when it is a win instead of Connect Four; (b) the search AI needs a new evaluation function to pick the best move, as in question 2.

a. (10pts) Connect-k Problem

Extend the Connect Four problem to Connect-k (eg. k = 3, 4, 5).

TIP: the winning number k can be considered a variable of the game. To reuse the code we already have, you can make this number a parameter of the constructor of some game object, and set the default value to 4; so if not explicitly specified, it is a conventional Connect Four game:

```
run_game(alphabeta_player, human_player, ConnectFourBoard(k))
```

b. (10pts) Longest-streak-to-win Problem

The game does not stop at a fixed number k. Instead, each player has 10 tokens in total, and in turn they put the tokens all in. After 20 rounds, the one who has the longest horizontal, vertical or diagonal streak wins.

5. (10pts) Project Report and Comment/Documentation in code

Please print the number of nodes expanded and running time for minimax (question 1) and alpha-beta-search (question 3) in your code and your report.