Project 2: Connect-4 Player
CSC 330

Introduction

Connect 4 is a 2-player game where the players take turns dropping discs down vertical columns that form a grid, stacking them on top of each other. The players' discs are distinguished by color, and the goal is to get four of your color in a row, either vertically, horizontally, or diagonally. To get an idea of how the game works, you can play this game against an AI program:
http://www.mathsisfun.com/games/connect4.html
Play a few times to make sure you understand the rules, and if you need more details, please let me know.

We will be writing a program to select moves for Connect 4. When you complete this project, you'll be able to play against your own program, and your program will be able to play against other programs.

I've given you some code to manage the basics of the game, so you can focus on the AI issues. Your task is to write your own player class, as a subclass of the PlayerDef class. It can then be used by the Game class just like the other player classes. Ultimately, your player class will choose moves based on the minimax algorithm. The following sections describe in detail what needs to be done.

Files

Before you do anything else, please grab a copy of the provided BlueJ project from the I: drive. Please do not change anything except your own player class! When I grade, I will use a fresh copy of the BlueJ project, so any changes you make to a personal copy of other code will not be included.

With your own copy of the BlueJ project, rename the NAME_ME_Player class to whatever player / team name you want. This is the class where you'll write your code. If you want to write other helper classes, please feel free; they should also be named after your team name, so that your class names are distinct from everyone else's.

Comments in Code

As always, please make sure that every function/method you write has at least a comment header describing what it does and the meaning of its inputs and outputs. Make sure all variable and function names give a clear idea of what they are ("self-documenting code"). Also divide your code into sections, grouping domain-independent stuff together, heuristics together, etc.

Part 1

Due: Wednesday 3/13 at 8:00 AM, by Moodle

Please spend some time reading and learning about the provided code. Just as in project 1, this investment will really pay off. This is an important skill - in the real world you won't always write everything from scratch, of course. :)

Read the rest of this assignment in its entirety, just so you have a big picture of what's coming.

For this part, you are to write the eval and expand methods for your player class. See the related class notes and the provided code documentation for details about how they should work. For ideas about what to look for in an eval function, I recommend you play several games here:
http://www.mathsisfun.com/games/connect4.html
As you play, think about what situations are helpful for you. Other than just getting 4 in a row, what kinds of situations in one part of the board set you up for a greater chance of success later?

Please note that from the PlayerDef class, your own player class will inherit the playerSymbol and totalTime fields. (Recall that "protected" essentially means "private, but inheritable".) You may ignore totalTime for now. You may need to use playerSymbol, though. It's either 'X' or 'O', indicating the symbol for the player you're writing. Your code should work correctly regardless of which symbol your player is given at construction time.

Grading for part 1

- expand should work the same for everyone, according to the specifications. Please let me know if you have any questions on that.
- eval may be different for different people. Here's what I'll do to grade it: I'll make several test cases, like having one or more of X _ X X or X _ _ X or X _ X O vertically, horizontally, or diagonally. For full credit, I'll expect most or all cases to be rated reasonably. That is, eval should capture whether something is slightly good for X, very good for X, very bad for X, etc. If you'd like to discuss your eval function before this is due, please let me know!

Part 2

Due: Thursday 4/4 at 8:00 AM, by Moodle

This is due after spring break. I'm not asking you to work on it during break, as long as you make regular progress every day or two when classes are in session. But I will be available during break for questions by email or phone, and sometimes in person, if you'd like. I strongly urge you not to wait until after break to make much progress! :)

For this part, your task is to complete your player class. This consists primarily of implementing the minimax algorithm. (In the interest of time, we'll not do alpha-beta pruning.) Here are some notes to consider:

- Pseudocode: Follow the provided pseudocode (from class) very carefully. I highly recommend even using the exact same variable names, so that you don't have to do a "translation" in your head as you work, which could lead to mistakes.

- Efficiency: Keep in mind that the slowest single act in Java is to use the `new` keyword (memory allocation, for constructing a new object). So avoid it if you can. But don't make the mistake of trying to make "global" fields to be reused, since this probably won't work properly in recursion... If you have something you want to try, feel free to chat with me about it first.

- Timing: For grading, I will always play untimed games. This will be signaled by setting the total time to -1. So be careful that your program interprets that as "infinite time", rather than "very little time".

- Timing: For grading purposes, please make sure your program doesn't take more than about 5 seconds per move.

- Depth limit: Please make a global static final variable called DEPTH_LIMIT that you use to control the depth limit of your search. I'd like to be able to adjust this easily while grading if need be. Please always refer to this DEPTH_LIMIT variable, rather than a hard-coded number in the middle of your code.

- Mimimax the first time: In implementing minimax, I personally found it helpful to write a maxValueFirstTime method, in addition to maxValue and minValue. The "first time" method returns the move to make, instead of the value. This isn't required, but I found it convenient personally. Just make sure getMove returns the column to move in.

- Expand: When you call expand, what symbol should you give it? Note that in some games, your player will be X. In other games, your player will be O. So MAX won't always be X, for example.

- Recursion: As you implement minimax recursively, you might be tempted to use some "global" variables (fields) that get updated in the recursive methods. Typically, this won't work, for any recursive method, ever. The reason is, all global variables are shared amongst all the recursive calls, so what one recursive call does may mess up what another does. For example, it wouldn't work to try to make a single global ArrayList of expanded results, since each recursive call needs its own local ArrayList to work with, and it is not the case that one recursive call would be finished with its list before another one needs to start with it. So instead, use local variables (including arguments). If you're not sure what I'm getting at, please ask me about this more.

How to test your code

In grading, I will expect the following from a properly implemented player subclass:

- Your player definition should not suggest any illegal moves.
- Your player definition should do obvious things like block a line of 3 opponent pieces to prevent a trivial opponent win. An opponent should have to work a little to be able to win.
- Similarly, your player definition should be able to take a win that is handed to it (e.g. opponent leaves your sequence of three pieces unblocked and it is now your turn).
- Your player definition should be able to easily and quickly win if it plays against a bad opponent, like RandomDef.
- In general, it should offer some amount of challenge. Exactly how good it will be will depend on your implementation.