CS 380: Artificial Intelligence

Assignment 2

Connect 3 (20 pts)

Connect 4 [https://shop.hasbro.com/en-us/product/connect-4-game:80FB5BCA-5056-9047-F5F4-5EB5DF88DAF4] is a 2-player adversarial game in which a player drops pieces into a grid and tries to get the pieces aligned to win the game:

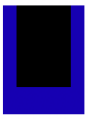
At each turn, the player drops a piece (a round disk) into a slot, where it travels to the lowest unoccupied space. Each player aims to place their pieces to form a row, column, or diagonal of 4 consecutive pieces; if they are successful, they are declared the winner.

In this assignment, we will use minimax to make an AI agent that plays the simplified game of Connect 3 — that is, the same game, except that each player tries to align 3 pieces instead of 4. You are given base code to represent the board, handle the placement of pieces, and determine whether there is a winner. Your task will be to extend this code into a full-fledged game agent that can play and win the game. As before, you may only use built-in standard libraries (e.g., math, random, etc.); you may NOT use any external libraries or packages.

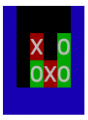
Connect3 Module

As part of the assignment, you have been given a file **connect3.py** that provides an implementation of the state representation for the game, including computation of possible legal actions. And as before, we use a simple string representation for a state, with rows of characters—'X' for player 1, 'O' for player 2, ' ' for an empty space. The **connect3.py** file includes various functions you need for adversarial search: **is_game_over()** to determine whether the game has ended, **get_winner()** to return the game winner for a finished game, **get_actions(char)** to return legal actions for player <char>, **execute(action)** to execute the action on the board, and many other utility functions (e.g., **__str__()**, **__eq__()**, **clone()**, and so on). The code was left purposely uncommented; part of your task in this assignment is to read through and to understand the code provided. You can test out parts of the game itself from the command line, like the previous assignment:

```
> python3 connect3.py print
```



```
> python3 connect3.py print "    | X O| OXO"
```




```
> python3 connect3.py over "    | XXX|OOXO"
X
> python3 connect3.py over "    | X O| OXO"
False
> python3 connect3.py winner "    | XXX|OOXO"
X
> python3 connect3.py actions X "    |   O| XXO"
drop(X,0)
drop(X,1)
drop(X,2)
drop(X,3)
> python3 connect3.py actions X "   X|  OO| XXO"
drop(X,0)
drop(X,1)
drop(X,2)
```

Game Module

Please start the assignment by creating a module **game.py** with two classes, **Player** and **Game**. The Player class should note the player's character ('X' or 'O') and should have one method: **choose_action(state)**; we will implement this method for subclasses below. The Game class should allow for creating a new game with an initial state and two players, and then a **play()** method that plays out the game. At this point, a skeletal implementation of these classes will suffice while we build out the rest of the assignment.

Human Player Module

Next, create a module **human.py** with one class, **HumanPlayer**, which is a subclass of **game.Player**. For its **choose_action(state)** method, it should determine the possible actions from the given state and print the index, ':', and the action as below. It should then present the user with the option to select one of the actions (with index 0 for the first action). For example:

```
0: drop(X,0)
1: drop(X,1)
2: drop(X,2)
3: drop(X,3)
Please choose an action: <user types a number 0-3 here>
```

Agent Player Module

The next task is the core part of the assignment, namely to build the agents to play the game. The first agent is a random agent that simply plays any of the possible actions at random. In a new file **agent.py**, implement a **RandomPlayer** class as a subclass of **game.Player** that has this functionality.

Then, in the same **agent.py** file, implement a **MinimaxPlayer** class (again, as a subclass of **game.Player**) that uses the minimax algorithm to play the game. You will need to include an evaluation function that assigns a value to any final game states. Note that it's a good idea to also include diminishing returns for nodes deeper in the tree, thus giving higher values for nodes with shorter paths.

Main Module

Putting everything together, complete your Game implementation in game.py such that it starts from an initially blank board and alternates between players, asking each to choose an action and then updating the game state. A game's **play()** method can return a tuple with the winner and the list of visited states during game play.

Then, create a file **main.py** that holds the code for handling command-line arguments. Specifically, your code should allow for the following syntax from the command line:

```
> python3 main.py <player1> <player2>
```

… where player1 or player2 can be one of `[human, random, minimax]`

Assume that player1 is assigned the character 'X' and player2 is assigned the character 'O', and that player1 always goes first. This syntax allows you to mix the various human/agent players for testing. During the game, print each new state as each player makes a move, and then at the end, print "<*winner*> wins" and print the entire sequence of states taken in the game. (State printing can be done with the **util.pprint()** function provided.)

Here is an example with two human players:

```
> python3 main.py human human
0: drop(X,0)
1: drop(X,1)
2: drop(X,2)
3: drop(X,3)
Please choose an action: 2
```

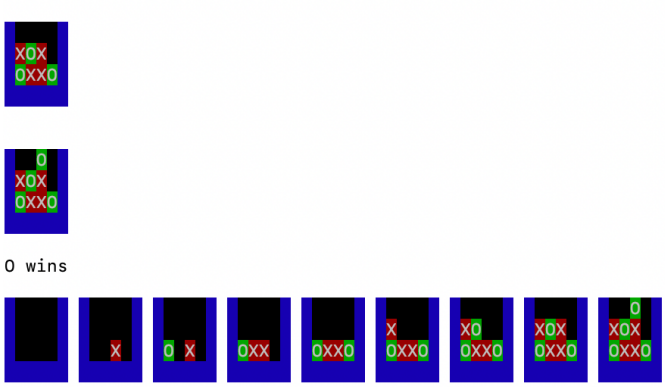
```
0: drop(O,0)
1: drop(O,1)
2: drop(O,2)
3: drop(O,3)
Please choose an action: 0
```

… etc.

And here is an example with two agent players:

```
> python3 main.py random minimax
```

… and all individual states until the end of the game…



Of course, we could also mix human and agent players if desired.

When you play your minimax agent against the random agent, you should see that the minimax agent wins the vast majority of the time. It could be that, due to lucky random plays, the random agent wins occasionally, but in general you should see your minimax agent behaving reasonably intelligently as opposed to the random agent behaving randomly.

Just for Fun: Minimax + Alpha-Beta Pruning [optional!]

If you finished the above without too much trouble, feel free to try adding a **MinimaxAlphaBetaPlayer** that uses alpha-beta pruning in conjunction with minimax to play the game. This is an optional task with no extra credit, just for fun. ☺

Submission

For this assignment, you should submit the following Python files:

- **connect3.py** – the original game code given to you for this assignment (you should not modify this file in any way!)
- **util.py** – the original utilities code given to you for this assignment (you should not modify this file in any way!)
- **human.py** – your code for the human agent
- **agent.py** – your code for the search agent
- **game.py** – your code for representing a player and running a game
- **main.py** – your code for running everything with command-line arguments

Please use a compression utility to compress your files into a single ZIP file (not RAR or any other compression format). The final ZIP file must be submitted electronically using Blackboard—please do not email your assignment to a TA or instructor. If you are having difficulty with your Blackboard account, you are responsible for resolving these problems with a TA or someone from IRT before the assignment it due. If you have any doubts, complete your work early so that someone can help you.

Academic Honesty

Please remember that all material submitted for this assignment (and all assignments for this course) must be created by you, on your own, without help from anyone except the course TA(s) or instructor(s). Any material taken from outside sources must be appropriately cited.