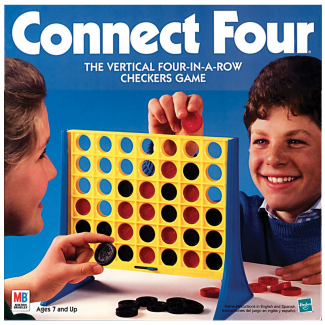
CS 380: Artificial Intelligence

Assignment 3

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 the 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. The sample code provided is written in Python, and the code for this assignment should run on tux.cs.drexel.edu with a shell script, as we have done for previous assignments.

Implementation Setup

The various parts of this assignment will require a shell script that can pass an argument to your code—thus allowing you to use **python** or **python3** while allowing us to be able to run your code with a consistent interface. Again, **you may only use built-in standard libraries (e.g., math, random, etc.); you may NOT use any external libraries or packages** (if you have any questions at all about what is allowable, please email the instructor).

Again, this is the same code as you used in the previous assignments, including the ability to accept two arguments in the general format:

```
sh run.sh <command> [<optional-argument>]
```

As before, this scheme will allow you to test your code thoroughly, and also allow us to test your code using a variety of arguments.

A sample shell script **run.sh** has been provided with this assignment. You should not need to modify this script file.

Game Implementation

Also provided with this assignment is the Python game implementation file, **connect3.py**. The code provides an implementation for a **Board** class that incorporates similar

functionality to your first assignment: read in a board from a string, represent it internally as a 2D array, print a board or boards to the screen, and so on. You can assume for this assignment that the two players are represented by 'X' and 'O'. Most importantly, the class also contains a **next()** function that (given a player label) returns the possible next board states from a given state, and a **winner()** function that returns the winner of the board if any (specifically, the winner's label, or **'TIE'** in case of a tie, or **None** if there is no winner yet). The code was left purposely uncommented; part of your task in this assignment is to read through and to understand this code provided.

Computing Next Boards

You first need to create a command to print out the next boards (i.e., successors) that can follow from a given board. The core functionality is already provided in the **next()** function in the **Board** class; you simply need to hook this functionality with a new **next** command to read in a board and print out the result. For example:

```
> sh run.sh next
 ---   ---   ---   ---
|   | |   | |   | |   |
|   | |   | |   | |   |
|X  | | X | |  X| |  X|
 ---   ---   ---   ---

> sh run.sh next "   |O  |O  |   "
 ---   ---   ---   ---
|   | |   | |   | |   |
|   | | X | |  X| |   |
|XOO| | OO| | OO| | OOX|
 ---   ---   ---   ---

> sh run.sh next "   |OXO|OOX|   "
 ---   ---
| OX| | OX|
| XO| | XO|
|XOO| | OOX|
 ---   ---
```

Playing a Random Game

Next, you should augment the code with a "**random**" command that plays a random game between two players. To do this, first create two new classes: a **Player** class that takes a label ('X' or 'O'), and a **Game** class that takes an initial board and creates two players. Then, write the code for a **RandomPlayer** class that inherits from **Player** and which plays a random game—that is, chooses a random next move from the set of possible next moves. You should assume that 'X' plays first, and the game terminates when one player has won or a tie has occurred. For example:

```
> sh run.sh random
 ---   ---   ---   ---   ---   ---
|   | |   | |   | |   | |   | |   |
|   | |   | |   | |  X| |  X| |X  X|
|   | |X  | |X  O| |X  O| |XO O| |XO O|
 ---   ---   ---   ---   ---   ---
 ---   ---
|   | |X  |
|XO X| |XO X|
|XO O| |XO O|
 ---   ---

> sh run.sh random "   |OXO|OOX|   "
 ---   ---   ---
```

```
 ____   ____   ____
| OX | | OX | | OX |
| XO | | XO | | XO |
| OO | | OOX| |OOOX|
 ----   ----   ----
```

As shown above, your code must keep track of the boards as they occur, and in the end, print the full sequence of boards from start to end (using **stringify_boards()** to print the board sequence horizontally).

Playing with Minimax

Your final and most involved task is to augment the code with a "**minimax**" command that plays a game between a random agent and a minimax agent. Implement a new class **MinimaxPlayer** that inherits from **Player** and uses minimax to analyze the current board, making the move that maximizes the player's expected benefit.

Then, create a game such that player 'X' (who moves first) is a **RandomPlayer**, whereas player 'O' is a **MinimaxPlayer**. You should see 'X' playing randomly, but 'O' playing sensibly, with 'O' winning all (or almost all) of the games—for example:

```
> sh run.sh minimax
 ____   ____   ____   ____   ____   ____
|    | |    | |    | |    | |    | |    |
|    | |    | |    | |  X | |  X | | XX |
|    | |X   | |XO  | |XO  | |XOO | |XOO |
 ----   ----   ----   ----   ----   ----
 ____
|    |
| XX |
|XOOO|
 ----
```

Among other things, you should ensure that when the minimax player can win on the next move, it does indeed play the move to win the game; if this is not the case, closely consider your utility function and whether it should be more sensitive to the depth and/or quality of its moves.

Extra Credit: Playing with Minimax + Alpha-Beta Pruning **[+5 points]**

Implement a new player **MinimaxAlphaBetaPlayer** that uses alpha-beta pruning in conjunction with minimax to play the game. You must also augment both this player and your MinimaxPlayer with timing code that outputs the real time (in seconds) needed to compute a next move. Then, you should add a shell command "alphabeta" that acts just like the "minimax" command except that it replaces the minimax player with an alpha-beta player (for 'O'; 'X' is still a random player). Please include a **README.txt** file with your submission that summarizes the timing differences you observe between the minimax player and the alpha-beta player.

Academic Honesty

Please remember that you must write all the code for this (and all) assignments by yourself, on your own, without help from anyone except the course TA or instructor.

Submission

Remember that your code must run on **tux.cs.drexel.edu**—that's where we will run the code for testing and grading purposes. Code that doesn't compile or run there will receive a grade of zero.

For this assignment, you must submit:

- Your Python code for this assignment.
- Your **run.sh** shell script that can be run as noted in the examples.

Please use a compression utility to compress your files into a single ZIP file (NOT RAR, nor any other compression format). The final ZIP file must be submitted electronically using Blackboard—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.