**B351 Assignment 3: Connect-Four with Minimax with Alpha-Beta Pruning**

**Due on October 23rd at 11:59pm**

**Warning**: This assignment is going to be harder than assignment 1. Start early!

Connect-Four is a two player game with zero-sum utility (meaning that all moves can be described in terms of an equal and opposite a) advantage to one player, or b) disadvantage to the other), perfect information (meaning that there are never any unknowns during the game), and alternating turns. So it’s a perfect game for us to use Minimax with!

Each player takes turns dropping a circular piece (red or black) into a 7x6 (7 columns by 6 rows) vertical board, with the goal of getting four pieces of their color in a line (either horizontally, vertically, or diagonally).

|  |  |
| --- | --- |
|  | If you haven’t played before, it will be worth trying Connect-Four out against a computer opponent for a few rounds to get a hang for how the game goes: <http://www.connectfour.org/connect-4-online.php>. |

**Part 1 (20 points):** Implement a class called Board for managing the board state of a Connect-Four game. To help with this you have been provided with a starter file board.py, with a basic class definition and 6 unimplemented functions, as described below:

init(self)

This is the constructor of the class. You should use this to create the data structures you want to hold the state of the board, along with any other information you might want to keep track of, such as whose turn it is, and the list of moves made by each player (to be used in unmake\_last\_move(self) described below). The board should be initialized to the starting state: an empty board, with player1 to move first, and no moves made by either player.

generate\_moves(self)

Returns a list of moves available in the current position. A move is represented by a number between 0 and 6 inclusive that indicates into which column the tile is inserted.

make\_move(self, move)

Accepts a move, represented as a number from 0 to 6, and updates the state accordingly.

unmake\_last\_move(self)

Restores the state of the internal board to the state before the last move was made. To implement this function you will need to keep track of the moves made by each player.

last\_move\_won(self)

This function just confirms whether or not the most recent move brought the board to a winning state. Returns true or false.

\_\_str\_\_(self)

This function just returns a string that represents the board state. If you give the function this exact name, then for all items b of type Board, then print(b), which you will find useful for debugging, will call the \_\_str\_\_ function you wrote directly.

Note that you may feel free to use additional helper functions as well, but all of the functions above must be implemented. Your solution will be graded based on correctness, and efficient use of data structures and coding clarity.

**Part 2 (60 points):** For this part, and the next, you will be implementing the functions outlined in search.py.

minimax(board, depth) and minimax\_root(board, depth)

Here’s where you start to implement minimax: minimax\_root just serves as a wrapper function for minimax. minimax accepts a board object and a maximum depth to search, and finds the best move for the current player to make (the current player turn can be found in board) using the minimax algorithm. minimax should return a single number, representing either the best move for the player to make, or -1 in the case where all moves result in a loss (a forced loss).

Then, minimax\_root should return a tuple of values, either the tuple (-1, None) if the MAX player was forced to lose, the tuple (1, None) if the MIN player was forced to lose, or the tuple (0, n), where n is the best move for the player to make, if there was a move found.

Once you have these working, copy the two functions into minimax\_alpha\_beta(board, depth) and minimax\_alpha\_beta\_root(board, depth), for functions that behave in the exact same way, but implement minimax with alpha-beta pruning rather than just minimax.

**Part 3 (20 points):** Now we’ll wrap things up with a class to represent a connect-four player, filling out the functions in player.py. You will need four simple functions:

init(self, board)

This is the constructor for your class. Use this to initialize your player, who has a name and access to the board.

make\_move(self, move)

This function should update the internal board of your player according to the move to be made.

get\_move(self)

This is where one of the search algorithms written in Part 2 will be used. This should return a legal move, identifying the case of a forced loss, or performing the move. If you don’t get alpha-beta pruning to work, you’ll want to have this function simply call the version without pruning. To re-iterate, the move returned consists of a tuple, with a code as to whether not there was a forced win in which case the tuple is either (1, None), or (-1, None), or (0, n) for some n in [0,6].

As before, you are encouraged to add extra helper functions to the Player class, but these functions are the ones you need to implement.

**How to test your player:**

We have provided two files, gamePlay.py, and randomPlayer.py, which should help you test your player's ability. gamePlay.py provides an interface for you to have two agents play against each other, and randomPlayer.py is a player that just makes random, valid moves.

Here's how to call gamePlay.py:

python gamePlay.py randomPlayer player

**Submission instructions:**

You should submit three python files: (board.py, search.py, and player.py).

Your player will be graded based on the correctness, efficiency and clarity of your coding,

and on whether or not it can consistently beat a random player.