

# Homework 2 Coding

Due Tuesday, October 12<sup>th</sup> at 11:59pm ET

*You are encouraged to discuss the assignment in general with your classmates, and may optionally collaborate with one other student. If you choose to do so, you must indicate with whom you worked. Multiple teams (or non-partnered students) submitting the same code will be considered plagiarism.*

*Code must be written in a reasonably current version of Python (>3.6), and be executable from a Unix command line. You are free to use Python's standard modules for data structures and utilities, as well as the pandas, scipy, and numpy modules. Do not use any modules or libraries that implement Minimax or Alpha-Beta pruning.*

## Greetings, Professor Falken. Shall we play a game?

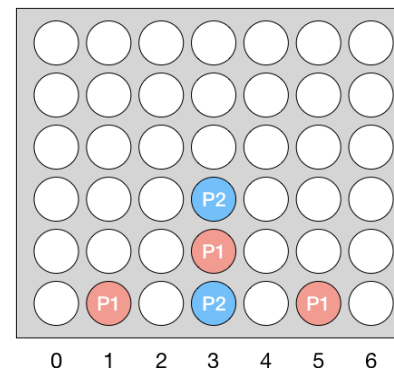
For this assignment, you will implement a game playing agent for a **modified** version of “Connect 4”, a vertical tic-tac-toe-like game. Your agent will be able to play against humans or other automated agents, and we will pit them against each other in a class-wide tournament.

### Classic Connect 4

In the basic game, the board is made up of six rows and seven columns, which are filled with tokens. Players take turns dropping tokens into vertical columns until one player has 4-in-a-row (vertically, horizontally, or diagonally) or the board is full.

The board shown on the right illustrates a game in progress after the following five moves have been made:

Player 1 to column 1  
Player 2 to column 3  
Player 1 to column 3  
Player 2 to column 3  
Player 1 to column 5

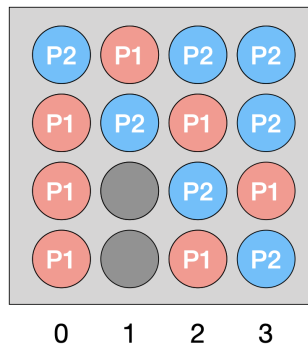


You can play an online version [here](#).

### Connect 383

Our variant, Connect 383, will have three main differences from the basic game. First, we will not be limited to 6x7 boards. Secondly, games may start with one or more “obstacle” pieces already on the board which do not belong to either player. Lastly, play will always continue

until the board is completely full (even after a player has achieved 4-in-a-row), at which point scores for each player will be calculated.



Points are awarded as follows: for each run of length three or greater, the player will receive points equal to the square of the length of that run. For example, 3-in-a-row is worth 9 points, 4-in-a-row 16, 5-in-a-row 25, etc.

For the 4x4 board on the left, Player 1 scores 18 points (one vertical and one diagonal 3-in-a-row), while Player 2 receives 16 points (one diagonal 4-in-a-row). The two obstacles (gray) don't contribute to either player's score.

When calculating the value of different moves, your game-playing agent will use the scores as utility values for the terminal games states, and seek to maximize the delta between its score and that of its opponent using Minimax.

## Implementing Adversarial Search for Connect 383

### Step 0: Running the Game

We've created a Python module that will handle representing the game state and manage play called `connect383.py`. *You do not need to change any of the code in this file*, but will need to understand how it works to program your game playing agent.

To play a game, you run `connect383.py` from a command line (or within an IDE) and supply it with arguments that determine the parameters of the game and which type of agents are playing. The required arguments are, in order:

- `player1` – one of {'r', 'h', 'c', 'o'} specifying the agent type (see code for details)
- `player2` – one of {'r', 'h', 'c', 'o'}
- `rows` – the number of rows in the board
- `columns` – the number of columns in the board

For example, to play against a random computer agent on a 4x5 board, you would type:

```
> python connect383.py r h 4 5
```

Try playing to make sure you understand the game – later, you'll replace the `r` with a `c` if you want to play against your bot.

## Step 1: Implement Minimax

Once you're comfortable running the game, it's time to implement your computer agent. The `agents.py` file contains code for different agent types, each defined by a class. Your task at this step is to implement the `minimax()` method for the `MinimaxAgent` class. Given a game state, `minimax()` should recursively traverse the game tree, eventually determining the value of that state for use in the `get_move()` method. The utility of the *terminal* states is calculated using the `GameState.utility()` method, with positive numbers corresponding to Player 1 outscoring Player 2, and negative numbers for Player 2 winning.

## Step 2. Testing Your Minimax Code

To facilitate debugging and allow you to test your code for correctness, you can optionally supply a starting game state when you invoke `connect383.py`. To do this, you must supply an additional command line argument, using the `--board` flag. The value of that argument must match one of the keys of the `boards` dictionary defined in `test_boards.py`. For example, to start the game with a state labeled "awesome":

```
> python connect383.py c h 4 5 --board awesome
```

Note that when you use this option, the dimensions of the board you pass in will be ignored and set to those found in the test case. Additionally, the `--board` flag is the only way to specify obstacle spots in the board.

N.B.: Assuming you don't have a quantum supercomputer at your disposal, you will not be able to run Minimax on a full-sized (6x7) or larger board, as the game tree will be much too big (over 13 decillion game states). Instead, you should test your Minimax on much smaller boards, where you can easily calculate the value of the state by hand and verify correctness.

## Step 3: Create an Evaluation Heuristic

In order to be able to play on larger boards, your computer agent must be able to invoke a heuristic evaluation function when Minimax reaches a certain depth limit. The code for this should be put in `HeuristicAgent.evaluation()` and get called by `MinimaxHeuristicAgent.minimax()` in accordance with the `depth_limit` parameter given to the constructor. Note that if the depth is larger than the number of total moves for a game, this agent should behave identically to the `MinimaxAgent`.

The evaluation will contain the "brains" of your Connect 383 bot, and the quality of your utility estimation is what will differentiate your agent from others. Your method must compute a utility based on static features of the game state, and should not generate new states in its calculations (we will check for this!).

The evaluation function should return an estimated utility value (positive or negative) for any game state based on features that you construct. For example, you might count all the open-ended 2-in-a-rows for each player for a given state, and return the difference. To make your life easier, we've added some convenience accessors to the GameState class (`get_rows()`, `get_cols()`, and `get_diags()`). Note that you cannot simply duplicate the `score()` method found in `connect383.py` for use as an evaluation function.

To run your heuristic code using the evaluation function, you must supply a depth limit using the `--depth` flag:

```
> python connect383.py r c 6 7 --depth 3
```

## Step 4: Alpha-Beta Pruning

Finally, you will create a computer agent that uses alpha-beta pruning within the Minimax traversal by implementing `MinimaxPruneAgent.minimax()` in `agents.py`. Whenever possible, this implementation of Minimax should ignore parts of the tree that don't need to be traversed (see the AIMA Sec. 5.3 and the slides and videos from class for examples).

## The Tournament

We will be pitting all of the submitted computer agent bots against each other in a round-robin tournament, and will be publishing a top-twenty ranking. Note that since Minimax is optimal, your agents can only be differentiated when using a depth limit – your “secret sauce” will be in your implementation of the `evaluation()` method. For the tournament, agents will play on a 6x7 board with a lookahead depth limit of 3. Agents that come in the top 3 will receive extra credit and a can of Moxie.

## Additional Notes

See the comments in the code for more details. Make sure that you *do not change the API* (number of arguments or return values) for any of the above methods, as we'll be calling them directly to evaluate your code.

There is a stub for an additional class `OtherMinimaxHeuristicAgent` in `agents.py`. You are not required to implement this class, but may find it useful for testing and improving your heuristic agent's evaluation function. This class will be used if you specify player type 'o' on the command line.

The `connect383.GameState` class does some bookkeeping to help you test your pruning code. Specifically, it counts the number states generated by each player (see the code for more details). Pay attention to this output when testing your lookahead limit and pruning code.

You should test your agent on a variety of boards to ensure that it's functioning correctly. To help with this, we've included a file called `test_boards.py` to hold test cases. You should modify the test boards that are there and/or create more.

We will be checking your individual method implementations, as well as verifying that your agent can successfully beat some simple agents. Note that your solutions will only be tested on well-formed boards – we are not going to feed your program incorrectly formatted game states.

## What to Submit

You should submit your versions of the two Python files (`agents.py`, and `test_boards.py`) required to run a game, along with a `readme.txt`, which should contain:

- Your name(s)
- The name of your tournament bot
- A short description of your thinking and the design behind your `evaluation()` function, along with some thoughts on its effectiveness
- Short descriptions of the cases that your boards in `test_boards.py` are meant to test
- Notes or warnings about what you got working, what is partially working, and what is broken