Extra Credit/Grading Note:

In this assignment we have an opportunity for extra credit. The assignment in Gradescope and Canvas will be out of 115 to reflect this. Your final assignments tab grade will be calculated as: min(assignments_score, 800). This means that you can make up for a maximum of 15 lost points in previous assignments using the extra credit from this assignment. **Note: It is possible to achieve a perfect score in the overall assignments section *without* doing the extra credit.**

Assignment Goals

- Familiarize yourself with solving games using AI.
- Practice implementing a minimax algorithm.
- Develop an internal state representation in Python and get familiarized with classes in Python.

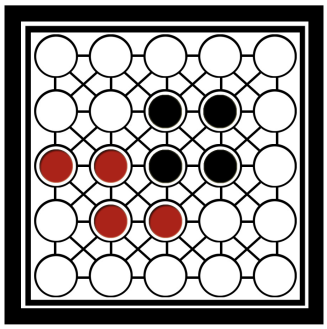
Summary

In this assignment, you'll be developing an AI game player for a game called Teeko.

As you're probably aware, there are certain kinds of games that computers are very good at, and others where even the best computers will routinely lose to the best human players. The class of games for which we can predict the best move from any given position (with enough computing power) is called Solved Games. Teeko is an example of such a game, and this week you'll be implementing an AI player for it.

How to play Teeko

Teeko is very simple:



It is a game between two players on a 5x5 board. Each player has four markers of either **red** or **black**. Beginning with black, they take turns placing markers (the "drop phase") until all markers are on the board, with the goal of getting four in a row horizontally, vertically, or diagonally, or in a 2x2 box as shown above. If after the drop phase neither player has won, they continue taking turns moving one marker at a time -- to an adjacent space only! (this includes diagonals, not just left, right, up, and down one space.) -- until one player wins. Note, the game has no “wrap-around” similar to other board games, so a player cannot move off of the board or win using pieces on the other side of the board. You can also refer to this link as an interactive demo.

Win conditions summarized for Teeko:

- Four same colored markers in a row horizontally, vertically, or diagonally.
- Four same colored markers that form a 2x2 box.

Program Specification

This week we're providing a basic Python class and some driver code, and it's up to you to finish it so that your player is actually intelligent. The skeleton code appears in **game.py**.

If you run the game as it stands, you can play as a human player against a very stupid AI. This sample game currently works through the drop phase, and the AI player only plays randomly.

First, familiarize yourself with the comments in the code and classes in Python (if you are new to this, then you can refer to this tutorial). There are several TODOs that you will complete to make a more "intelligent" player. You are allowed to implement helper functions but please do not change the signature (parameters/name etc) of the functions given in the starter code.

Make Move

The `make_move(self, state)` method begins with the current state of the board. It is up to you to generate the subtree of depth *d* under this state, create a heuristic scoring function to evaluate the "leaves" at depth *d* (as you may not make it all the way to a terminal state by depth *d* so these may still be internal nodes) and propagate those scores back up to the current state, and select and return the best possible next move using the minimax algorithm.

The following section will provide you with the steps that you should be implementing as a part of this exercise. You will be implementing several helper functions for your make_move method to work (for the helper functions, the parameters do not have to be the exact same as the write-up shows because none of those functions are directly tested).

You may assume that your program is always the **max** player.

1. Generate Successors

Define a successor function (e.g. `succ(self, state)`) that takes in a board state and returns a list of the legal successors (i.e., states that are one marker-adjacent to the current states). During the drop phase, this simply means adding a new piece of the current player's type to the board; during continued gameplay, this means moving any one of the current player's pieces to an unoccupied location on the board, adjacent to that piece.

Note: wrapping around the edge is NOT allowed when determining "adjacent" positions.

2. Evaluate Successors

Using `game_value(self, state)` as a starting point, create a function to score each of the successor states. A terminal state where your AI player wins should have the maximal positive score (1), and a terminal state where the opponent wins should have the minimal negative score (-1).

Finish coding the diagonal and 2x2 win condition checks for the game_value method.

Define a `heuristic_game_value(self, state)` function to evaluate non-terminal states. For some hints, check out the lecture slides on Game Theory, Minimax and Alpha-Beta Pruning (you should call the game_value method from this function to determine whether the **state** is a terminal state before you start evaluating it heuristically.) This function should return some floating-point value between 1 and -1.

3. Implement Minimax

Follow the pseudocode recursive functions on the slides of our class lecture, incorporating the depth cutoff to ensure you terminate in under 5 seconds.

- Define a `max_value(self, state, depth)` function where your first call will be `max_value(self, curr_state, 0)` and every subsequent recursive call will increase the value of **depth**.
- When the depth counter reaches your tested depth limit OR you find a terminal state, terminate the recursion.

We recommend timing your make_move() method (try Python's time library) to see how deep in the minimax tree you can explore in under 5 seconds. Time your function with different values for your depth and pick one that will safely terminate in under 5 seconds.

Evaluating Your Code

We will be testing your implementation of **make_move()** under the following criteria:

1. Your AI must follow the rules of Teeko as described above, including the drop phase and continued gameplay.
2. Your AI must return its move as described in the comments, without modifying the current state.
3. Your AI must select each move it makes in **five seconds or less**.

We will be timing your make_move() remotely on the gradescope environment, to be fair in terms of processing power.

Your AI will be evaluated against a set of opponents that we created with various difficulties.

1. Easy: We have 1 easy opponent. Your AI must be able to beat the easy player 75% of the time for full credit.
2. Medium: We have 1 opponent we deem to be of medium difficulty. Your AI must be able to beat the medium player 50% of the time for full credit.
3. Hard: We have 1 hard opponent that your AI must be able to beat 25% of the time for full credit.

Your AI will likely be able to beat these opponents more than this, but there is a chance of randomness so we have lowered the thresholds to account for this.

Submission Notes

Please submit your **game.py** file. As usual, make sure there is no code outside of functions and the provided main statement and no extra prints.

Extra Credit

In this assignment we have an opportunity to get extra credit! We have implemented what we believe to be a very strong Teeko AI, which we call the *Challenge Opponent*. If your implementation manages to tie (no winner after 50 moves) or beat this Challenge Opponent you can get extra points.

In game.py there is a function called run_challenge_test. If you set this function to return True, gradescope will run against the Challenge Opponent. This will slow down the grading of your assignment since this AI is much slower than our other opponents on Gradescope. We recommend you leave this as False until you are done with the assignment and want to challenge yourself.

We will play your AI against our Challenge Opponent on Gradescope and you can get a maximum of 15 points as extra credit against our Challenge Opponent.

Good luck!