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.

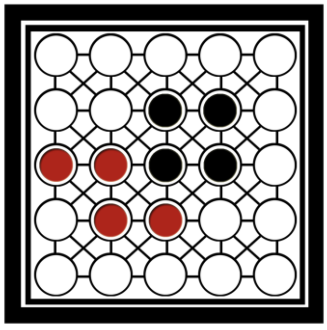
Summary

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 can not move off of the board or win using pieces on the other side of the board.

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 SURE TO USE PYTHON VERSION 3.10.6

(You can check your python version by running `python3 -version` on the terminal)

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. 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 Games II Lecture Slides (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 five 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**.
4. Your AI must be able to beat a random player in 2 out of 3 matches.

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

Using the minigrader

You have been provided with a minigrader in the form of a python executable. It is named `minigrader.pyc`. Make sure that the minigrader and the game.py file is in the same folder and run the following command on the terminal:
`python3 minigrader.pyc`

Note that scoring 100 points on the minigrader **does not** guarantee 100 points on the actual grader.

Submission Notes

Please submit your files in a zip file named **hw9_<netid>.zip**, where you replace <netid> with your netID (your wisc.edu login). Inside your zip file, there should be **only** one file named: **game.py**.

As usual make sure there is no code outside of functions and the provided main statement and no extra prints. The standard regrade policy for the course as specified in HW4 applies. Make sure to test your submission on the **CS Linux** machines! The minigrader we've provided contains most of the tests this time, so your performance on the minigrader will be highly accurate and so regrade requests will be more strict. If your code does not pass the minigrader, you likely won't get a regrade.