HW9: Game AI

Due Dec 7, 2021 by 11am **Points** 100 **Submitting** a file upload **File Types** zip **Available** Nov 23, 2021 at 11am - Dec 7, 2021 at 11am

This assignment was locked Dec 7, 2021 at 11am.

Update

Update Dec 6 9:00PM: Fixed a win condition in test script.

Update Dec 4 2:30PM: New test file which includes scores here.

Update Dec 1 12:30PM: MakeMove formatting should be a list of 1 tuple in drop phase and a list of 2 tuples in move phase.

Update Nov 29 6:30PM: HW9 due date extended to next Tuesday (12/7). We are still planning to release HW10 this Thursday though.

Update Nov 29: To clarify, the helper functions you need to implement are not directly tested, so they do not need to have the same parameters as the writeup.

Update Nov 27: Sample testing script is available here. This is just a simulation of a game against a random opponent; beating the random opponent does not guarantee full credit on the assignment. See the HW9 rubric for the grading breakdown.

Assignment Goals

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

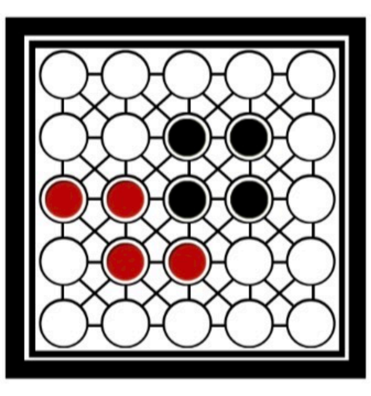
Summary

In this assignment, you'll be developing an AI game player for a modified version of the game called Teeko. We call this modified version Teeko2.

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 a computer player for a modified version of 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! (Note this includes diagonals, not just left, right, up, and down one space.) -- until one player wins.

How to play Teeko2

The Teeko2 rules are almost identical to those of Teeko but we will exchange a rule. Specifically, we remove the 2x2 box winning condition and replace it with a 3x3 box winning condition -- same colored markers at the four corners of a 3x3 square. Mathematically, if (i,j) is the center of a 3x3 board, then it must be that (i,j) is empty and that there is a marker of the appropriate color on each of (i-1,j-1), (i-1,j+1), (i+1,j-1), and (i+1,j+1).

Win conditions summarized for Teeko2:

- Four same colored markers in a row horizontally, vertically, or diagonally.
- Four same colored markers at the corners of a 3x3 square and the center of the square is empty.

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.

Here is our partially-implemented game: game.py (game.py)

(If your computer doesn't like downloading .py files, grab game.py.txt(game.py.txt) and remove the .txt extension.)

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. 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 3x3 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 Slides 56 through 59 of the Games I 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 slide #58 of our Game I 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 (use 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.

Testing Your Code

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

1. Your AI must follow the rules of Teeko2 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.

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**. Do not submit a Jupyter notebook .ipynb file.

Be sure to **remove all debugging output** before submission. Failure to remove debugging output will be **penalized (10pts)**.

Make sure to test your submission on the **CS Linux** machines!

This assignment is due on **December 2nd at 11:00 AM**. It is preferable to first submit a version well before the deadline (at least one hour before) and check the content/format of the submission to make sure it's the right version. Then, later update the submission until the deadline if needed.

HW9 Rubric

Criteria	Ratings				Pts
AI follows all rules of Teeko2, both during and after the drop phase.	25 pts Full Marks AI only makes legal moves during all game phases.	15 pts Partial Marks AI makes illegal moves in the drop phase.	10 pts Partial Marks AI makes illegal moves in the continued gameplay phase.	0 pts No Marks AI makes illegal moves in both the phases.	25 pts
AI correctly returns move without modifying state in make_move	10 pts Full Marks make_move does not modify the state of its Teeko2Player instance.	5 pts Partial Marks AI modifies the state in either drop or continued gameplay phases.	0 pts No Marks make_move does modify the state of its Teeko2Player instance.		10 pts
AI correctly returns move in make_move which is formatted in both the drop phase and continued gameplay	10 pts Full Marks make_move returns the move in a correct format in both drop and continued gameplay phases	5 pts Partial Marks AI returns an incorrect move format in either of the phases.	0 pts No Marks make_move returns a move in incorrect format for both the phases.		10 pts
AI selects a move in 5 or fewer seconds	25 pts Full Marks AI selects a move in 5 or fewer seconds.	13 pts Partial Marks AI selects a move in greater than 5 seconds and lesser than 10 seconds.	0 pts No Marks AI takes more than 10 seconds to select a move.		25 pts
AI wins at least 2 out of 3 games against a random player.	30 pts Full Marks AI wins at least 2 out of 3 games against a random player.	15 pts Partial Marks AI wins 1 out of 3 games against a random player.	0 pts No Marks AI wins no games against a random player.		30 pts
					Total Points: 100