HW6: Game Theory

Due Nov 4, 2020 by 11:05am **Points** 100 **Submitting** a file upload **File Types** zip
Available Oct 27, 2020 at 11am - Nov 4, 2020 at 11:05am 8 days

This assignment was locked Nov 4, 2020 at 11:05am.

Assignment Goals

- Practice implementing a minimax algorithm
- Develop an internal state representation

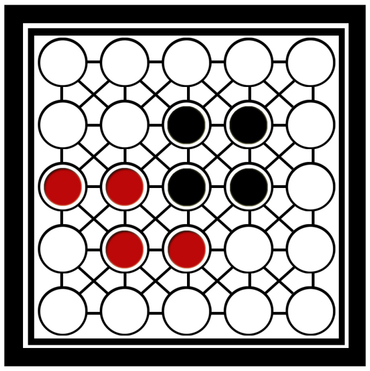
Summary

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 **(https://xkcd.com/1002/)**, 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** **(https://en.wikipedia.org/wiki/Solved_game)**. 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 diamond winning condition. Any diamond is defined by an empty center position surrounded by 4 markers on the spaces above, below, to the right, and to the left of the center. Mathematically, if (i,j) is the center of a diamond, 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), (i-1,j), (i,j+1), and (i,j-1). Visually, this makes a diamond shape on the board.

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: **teeko2_player.py (https://canvas.wisc.edu/courses/205182/files/15890515/download?download_frd=1)**

(If your computer doesn't like downloading .py files, grab **teeko2_player.py.txt (https://canvas.wisc.edu/courses/205182/files/15890517/download?download_frd=1)** 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. There are several TODOs that you will complete to make a more "intelligent" player.

Make Move

The `make_move(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.

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

Generate Successors

Define a successor function (e.g. `succ(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.

Evaluate Successors

Using `game_value(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).

1. Finish coding the diagonal and diamond checks for `game_value(state)`.
2. Define a `heuristic_game_value(state)` function to evaluate non-terminal states. (You should call `game_value(state)` from this function to determine whether **state** is a terminal state before you start evaluating it heuristically.) This function should return some float value between 1 and -1.

Implement Minimax

Follow the pseudocode recursive functions **on slide 43 of this presentation (https://happyharrycn.github.io/CS540-Fall20/lectures/games_part1.pdf)**, incorporating the depth cutoff to ensure you terminate in under 5 seconds.

1. Define a `Max_Value(state, depth)` function where your first call will be `Max_Value(curr_state, 0)` and every subsequent recursive call will increase the value of **depth**.
2. 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 (https://docs.python.org/3/library/time.html#time.time)**) 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 **hw6_<netid>.zip**, where you replace <netid> with your netID (your wisc.edu login). Inside your zip file, there should be **only** one file named: **teeko2_player.py**. Do not submit a Jupyter notebook .ipynb file.

Be sure to **remove all debugging output** before submission; your functions should run silently (except for the image rendering window). Failure to remove debugging output will be **penalized (10pts)**.

If a regrading request isn't justifiable (the initial grade is correct and clear, subject to the instructors' judgment), the request for regrading will be **penalized (10 pts)**.

This assignment due at 11/04/2020 11:00am. Submitting right at 11:00am will result in a late submission. 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.

Changelog

- Updated the teeko2_player.py file to do valid move checking on the opposing player and to include print statements to see the game progress. Also, clarified that adjacent spots include diagonals.
- Extended the deadline to Wednesday November 4 at 11 AM.

Game AI

Criteria	Ratings		Pts
Repeated calls to make_move() result in legal moves	**25 to >0.0 pts Full Marks**	**0 pts No Marks**	25 pts
The current board state is not modified in make_move()	**10 pts Full Marks**	**0 pts No Marks**	10 pts
The result of make_move() is correctly formatted in both the drop phase and continued gameplay	**10 to >0.0 pts Full Marks**	**0 pts No Marks**	10 pts
make_move() returns a move in under 5 seconds	**10 to >0.0 pts Full Marks**	**0 pts No Marks**	10 pts
make_move() wins against a random opponent in at least 2 out of 3 games	**25 pts Full Marks**	**0 pts No Marks**	25 pts
Manual inspection: make_move() uses a recursive minimax algorithm	**20 to >0.0 pts Full Marks**	**0 pts No Marks**	20 pts
			Total Points: 100