Extra Credit: Participate in PittCS 2018 Adversarial Game Tournament

Due by 11:59pm 10/19/2018

The goal of this activity is to provide a more practical context for applying Minimax with alpha beta pruning. You are invited to submit an automatic player for a two player game called The Game of the Amazons. Later in the semester (probably Oct. 31) we will showcase the submissions in a bracket tournament.

Support Code

An implementation of the basic game functionality has been prepared for you. Look for amazonsXX.py in this repository (where XX=27 works for Python 2.7 and XX=32 should work for Python 3 -- I'd like to encourage everyone to use Python 2.7, though, because that will greatly simplify the logistics for running the tournament.

The program amazonXX.py takes a configuration file as its argument. To run the program, use the command:
python amazonsXX.py <config.file>

Here is the format of the configuration file:
Line 1: time limit per turn (for any automatic player) in seconds
Line 2: dimensions of the board
Line 3: name of the White player (human or your player function name)
Line 4: the initial locations of the White Queens
Line 5: name of the Black player (human or your player function name)
Line 6: the initial locations of the Black Queens
In the included config file (named "amazons.config"), the standard board setup is used (10x10 board, 4 queens each side in default configuration). You will need to modify the config file to have it choose your automatic player. It is currently configured to let two humans play each other.

More details about the game and formats are given as comments in the program.

What you need to do

Define a function in with your Pitt userid. This is the gateway function to your minimax player. The function will receive a Board object as its input argument, and it should return a valid move in the form of a tuple of three tuples, specifying the location of the queen to be played, the move-to location of the queen, and the landing site of the arrow. All three locations must be in

(row, column) format (0,0 is defined as the lower left corner of the grid). An example of move is: ((2,1),(5,1),(0,1)).

You should implement additional classes and functions to support your automatic player. All function/class names should be prefixed with your userid so that we won't have a lot of conflict when we concatenate multiple programs together. **Important:** you should not make any change to the support code. Please send me an email to report bugs. The pre-defined Board class serves as a simple interface between the game control and your player. Note that it stores only the minimal information necessary to communicate the board configuration to you. You don't have to use it directly as your state representation; you may convert it to something else that you think is the most appropriate for your implementation.
To adapt your minimax function, you will needs to:

- generate children states. Unlike the main homework, where an entire tree is given to you, here you need to generate the tree on the fly by defining the appropriate function to generate successor states (similar to Project 1).
- define a heuristic function to evaluate a game state. Since we can't store the entire tree, the computer player can only look ahead a few steps. Therefore, it needs to use a heuristic function to evaluate the goodness of the board.
- deal with time management since your player has to return a move within the allotted time (as specified in the game initialization). The base code does not cut your function off when the timer runs out, but whatever move that is returned will not be applied (in other words, if your player times out, it loses a turn).

What to submit

- Your code (i.e., the modified AmazonXX.py)
- **readme** Document anything that's not working; any external resources you have used for this part of the assignment and/or any person with whom you've discussed the project; any other information that you think the grader would need to know in order to test your program. Also, please discuss the following issues:
    - The start of the game has many possible moves. How does your program deal with the huge branching factor?
    - There is a time limit for the turn. What does your program do to make the most use out of the time you have?
    - What different heuristic functions have you tried before settling on your final choice? What ideas worked? What didn't? Be sure to describe your final heuristic function in sufficient details.
    - How well does your automatic player compete against a human player?

Grading

Since this is extra credit, not doing it will not hurt your grade. Unlike the regular assignments, however, the bonus point scheme is somewhat more exacting.

- +20 points: You've designed an excellent player. Your player is one of the two finalists in the class tournament, **and** your player beats the "random" player nearly all the time (say, 9 out of 10 games). You've also written a good discussion in the readme about how you came about your heuristic design.
- +13 points: You've designed a good player. Your player can beat the "random" player most of the time (say, 6 out of 10 games). You've also written a good discussion in the readme about how you came about your heuristic design.
- +7 points: You've designed a functional minimax player with an interesting and well-reasoned heuristic, even though it only occasionally beat the random player. You've given good justifications about your design in the readme.
- +3 points: You've correctly incorporated your core minimax_a_b code into this game, but your heuristic function is not successful (e.g., always time-out or is basically the same as random). The correctness of your player's behavior can be demonstrated on a smaller board with fewer queens. You have a discussion section in the readme.