Homework #2[1]

Due by 11:59pm 10/18/2016

Objective

This assignment has two parts; both involve understanding how the Minimax Algorithm works and how alpha-beta pruning works. The first part of the assignment is a stand-alone exercise in which you will write a function that performs minimax with alpha-beta pruning. In the second part of the assignment, you will develop an automatic player for a two-player game called Game of the Amazons. Both parts are due at the same time.

Part 1 Basic Minimax and alpha-beta pruning

For this part of the assignment, you need to write a program called **minimax_a_b.py** that takes an input file as its argument. The input file contains a complete "game tree" (the format is described below). Your program will have to process the tree twice -- once with a vanilla minimax algorithm, and once using minimax with alpha beta pruning (in both cases, it is assumed that the tree is traversed in a *left-to-right* order). Print the utility value of the root node (assuming the root is MAX) as well as the names of all the nodes visited for each case.

The input tree is represented as a list where the zeroth element gives the name of the parent and the rest of the list elements are that node's children. The leaf nodes of the tree will have their utility values specified. Leaf nodes are represented as a tuple ('name', value). For example, a simple tree with root node named A that has two children named B (with a value of 5) and C (with a value of 10) would be represented as:

```
['A', ('B', 5), ('C',10)]
```

Please run your program on the three supplied test case files. You can import the ast library and use the "literal_eval" function to turn a string into a nested list structure.

Part 2: An Automatic Player for The Game of the Amazons

Modify and incorporate your Minimax_a_b function from Part 1 into the game of Amazons. An implementation of the basic game functionality has been written for you (look for amazonsXX.py, where XX is either 27 or 32 -- note: the 32 version seems to run OK on Python 3.4, but please let me know if you encounter issues). To run the program (currently configured for two human players), use the command: **python amazonsXX.py <config.file>**

---

[1] Shared Google directory (all files are initially committed in github repository)

Here is the format of the configuration file:
Line 1: time limit per turn (for the 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.

Program Specifications

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 from part 1 to this part, you will needs to:

- generate children states. Unlike Part 1, 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.
- 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

Part 1

- **minimax_ab.py**: your source code for minimax with alpha-beta pruning
- **transcript.part1**: show your program's output for the supplied test case files.
- **readme.part1**: Specify the python version you are using. Document anything that's not working, or any other information that you think the grader would need to know in order to test your program.

Part 2

- all relevant source codes (specify which version of python in the readme)
- **transcript.part2**: show a game between you and your computer player.
- **readme.part2**: Specify the python version you are using. 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.
- **report**: This report should 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 Guideline

Assignments are graded qualitatively on a non-linear five point scale. Below is a rough guideline:

- 5 (100%): Both parts are correct; Part 2 player consistently beats the "random" player; has a clear enough README for the grader; includes a transcript; complete report with insightful observations.
- 4 (93%): Part 1 is correct; Part 2 is correct but doesn't work well in practice (e.g., cannot beat the "random" player); the READMEs are clear enough for the grader; includes transcripts; complete report..
- 3 (80%): Part 1 is correct; some problems in Part 2; the READMEs are clear enough for the grader; includes transcripts; complete report.
- 2 (60%): Part 1 is mostly correct, but Part 2 is severely flawed or incomplete. Or, even though the programs are working at a "grade 3" level, the report is missing or extremely scant.
- 1 (40%): Both parts have major problems, but a serious attempt has been made on the assignment; the READMEs are clear enough for the grader.