Chess AI

In this project, you will write a program for playing chess.

Grab the provided code. First, you should run test_chess.py. You (presumably, human) will go first, as the white player. You can type a move into the bottom console. The second player is a computer-controlled "A.I.". The "A.I." cleverly picks a random move and makes it. I hope you win the first game.

We will use this Python chess package. You should use it primarily to represent board states (`Board`), find transitions, and make moves. I recommend installing it using the `pip3` command as described in the documentation for the library.

Experimental GUI. I've also provided you with a simple display functions based on PyQt with the chess library .svg output function, in `gui_chess.py`. To make it work, you'll need pyqt. I'd install that using homebrew, as described on our main course web page (under installation instructions). You don't need to use the gui if you are happy with the text output from test_chess.py.

Be aware the design of `gui_chess.py` is simple, but works imperfectly. Specifically, the Qt library is event-driven. This means that Qt runs a main event loops, with which you may register events like mouse clicks, timers, or keyboard input. In response to one of those events, Qt will call a *callback* function that you specify. Since Qt is running on a single thread, Qt will stop responding to input while it waits for that callback function to return. As a consequence, callback functions must execute quickly. Chess AIs are not quick: the UI will freeze during computation. The fix is to create a separate thread to run your AI in. I don't think that's a good use of your time – so for now, you can exit the GUI with a control-c or by pushing the hard stop button in PyCharm.

Use the source, Luke

Once you have played the game to exhaustion, **read** the code I've provided you carefully. I do not hold it up as a paragon of coding excellence, but read it anyway; you'll need to be familiar with it to write the assignment.

RandomAI should serve as a basic example to get you started.

Read the python-chess documentation and source

The documentation is sparse, but I found it sufficient to get the things done that I needed to.

Notice that there is an undo move capability, using `push()` and `pop()` on `Board`s. That's important. Creating a new object takes time and memory. It is much more efficient if your game-tree search makes moves and then undoes those moves after exploring them, so you only ever have a single Board object instantiated.

Minimax and Cutoff Test

Begin with implementing depth-limited minimax search. The code should look like the pseudo-code for minimax in the book. The minimax search should stop searching provided some `cutoff_test` method returns True because one of the following has happened:

1. We have reached a terminal state (a win or a draw)
2. We have reached the specified *maximum depth*.

Vary maximum depth to get a feeling of the speed of the algorithm. Also, have the program print the number of calls it made to minimax as well as the maximum depth. Record your observations in your document.

Evaluation function

Since we are cutting off the search, we will not always reach a terminal state. We need a heuristic evaluation function. I recommend a material value heuristic such as that described by the book. Describe the evaluation function you used in your document.

Once you have implemented the evaluation function, play the computer again, looking for signs of intelligent play. With sufficient depth and allowed time, the program should not lose stupidly and should always take a win. It should also be not

easy to beat. Use the fact that you may input any initial state to check for this: The computer should try to block wins and also take wins. Vary the allowed depth, and discuss in your document.

Iterative deepening

A good chess program should be able to give a reasonable move at any requested. A good approach to such "anytime planning" is to use iterative deepening on the game tree.

Once you have depth-limited minimax working, implement iterative deepening. At each depth, the best move might be saved in an instance variable `best_move`. Verify that for some start states, `best_move` changes (and hopefully improves) as deeper levels are searched.

Alpha-Beta Pruning

Write AlphaBetaAI.py, probably by copying your code from Minimax and extending appropriately. Now, play with the maximum depth and number of states to see if you are able to search deeper.

Add some simple move-reordering and see if this helps you search deeper (or make fewer calls to the alphabeta function when searching to specified depth.) Record your observations in your document.

The challenge here is testing. **I have seen many examples of code in the past where minimax outplays alpha-beta, because alpha-beta was implemented incorrectly.** First, test both at the same depth. Ensure that given the same initial position, each returns a move with exactly the same value. If they do not, **there is a bug**. Show some results (briefly) in your document, demonstrating that for the same depth, for various positions, alpha-beta explored fewer nodes and yet gave a move (leading to a position satisfying the cut-off test) with the same value.

Transposition table

Implement a transposition table. You'll need a hash function. Be careful: calling `hash` on Board objects directly looks to me like it always returns the same value. You could try something like this as the hash function: `hash(str(game.board))`. A Python set, frozenset, or dictionary uses the **hash** method of an object to compute the needed hash value. You might need to write a class that wraps game boards and has such a method in order to use the built-in set data structure.

Demonstrate that your code uses the transposition table to prevent some calls in minimax or alpha-beta, and discuss the number of calls made in your report.

Possible extensions

Some of these extensions are tricky, and the discussions below are not in depth. For extensions, I expect you to spend a bit of time first figuring out what needs to be done and how to do it. You are welcome to discuss with others, look up pseudocode (but not real code), etc.

Zobrist hash function

Zobrist hash functions are not discussed in the text, but are quite interesting and very useful for rapid incremental construction of hash values as the game tree is searched. Implement Zobrist hashing and your own transposition table.

Opening book

One significant weakness of your program is that it won't play well at the beginning of the game. The material value of the board only changes with a capture, and captures are rare at the beginning. Most chess programs (and human beings) solve this by knowing some standard openings; there are books of these standards, called "opening books". Add an opening book.

Profiling

Amdahl's law suggests that it's usually best to improve the things that are slowest. Profile your AI and report the results. Where is most of the execution time being spent? Where might it be improved, either by reducing the number of calls (through more clever heuristics), or by improving the speed of the code (for example, the heuristic function)?

Advanced move reordering

Improve your move ordering approach for alpha-beta by using computed state values from a previous iteration of IDS.

Null-move heuristic and forward pruning

Implement a null-move heuristic or some method of forward pruning to further speed your alpha-beta search.

Improved cut-off test

Implement a quiescence search or singular extensions to mitigate horizon effects, and show the effectiveness for certain starting positions.

Related work section

There is no required additional part of this assignment for graduate students, but an interesting component of the report would be a brief review of a somewhat-recent paper or two on Chess AI.

Grading rubric

Your code should be efficient and well-designed, with excellent style, formatting, comments. It should be brief if possible; a long chain of if statements is not good code if it can be replaced with something terser and easier to read. Follow good style guidelines, although we will not be strict here. For example, Python style guidelines suggest that there should be spaces after '#', spaces around operators like '+'. Use whitespace! Group lines of code into logical units that are no more than 3-8 lines long using blank lines. Factor code out into functions. Make sure to claim authorship of your code in a comment at the top of the file (include date as well.)

- Minimax and cutoff test: 3
- Evaluation function: 1
- Alpha-beta: 2
- Iterative deepening: 3
- Transposition table: 3
- Overall code style and efficiency: 3
- Discussion questions and discussion of implementation (report): 5
- Extensions beyond the basic requirements: up to 5 points

The assignment is out of 25 points, but 20 points is a pretty good score, corresponding to about an A-. (To get an A, you'll need at least 22 points.)