Chess: By Ryan Ong – made in python using only pygame

Some chess experts say that you can’t make an engine better than yourself. However, this one player chess program can consistently beat me. This doesn’t sound like a good thing, but I promise you it is! (or maybe its just because I’m bad at chess). Have a quick match and have your say!

Currently I am looking for chess players to test the program, and programmers to improve it. If you would like to be part of this project, you can find me \_\_\_\_\_\_\_\_.

# Simplified explanation

When trying to choose the “best” move, the program simulates all its possible moves and gives a rating to each one (based on material, checks, pins, etc.). By choosing the move with the best rating, it will get the “best” move, although what is considered best would now rely on the rating algorithm. As you might be thinking, simulating many moves in advance it a tedious task, even for a computer.

Read on over the page as I explain some nifty tips!

# Full explanation

Before making my algorithm, I have to make the basic infrastructure of chess, so that the computer can “understand” how the game[[1]](#footnote-1) works. My basic definitions included:

* storing the board object
* printing boards
* getting all possible moves (including castle rules and pawn promotions)
* getting the state of the board (e.g. won, lost)
* moving pieces

More complicated functions were also useful, such as:

* assessing a pieces safety
* determining the value of pieces and checking for legal moves

One all these functions were perfected, I could easily create a two player chess program. All this was done in 2017, when I first embarked on a goal to create a working two player chess UI (short for User Interface).

In term 1 week 5 this year, Kalaish, Richard and Androsh were attempting to make a two player chess program. Thus, I decided to flex on them by making a one player chess program[[2]](#footnote-2). To do this, I have used a minimax algorithm[[3]](#footnote-3), which basically looks through all future moves, as previously said. This is done many times, in my case 3 times (also known as depth 3 or 3 ply). Thus, the program is able to predict moves in advance to choose the best move that will benefit in the future (not just in the present). The minimax algorithm works by simulating future play, where you (the computer) tries to maximise the rating, while the opponent tries to minimise the rating. Thus the name minimax. For example, after some move, if the opponents best move is “bad”, then that means that your move is good. It is optimal to look as far into future moves a possible, to get the best understanding for the possible outcome’s for a move.

However, there is a computational limit, where exceeding depth 3 with my code on my algorithm will take ridiculous amounts of time. As you can probably tell, the search space and computational time will increase exponentially[[4]](#footnote-4).Thus, I am using a pruning algorithm called alpha-beta[[5]](#footnote-5). This essentially calculates the rating until is hits a upper/lower bound. This means that it can get a move with about ½ to sample space. To further cut down on time, instead of looking through every move, I can ignore “bad” moves. I define “bad moves as either being not defended or not serving any purpose (e.g. attacking, moving to centre, developing, castling, protecting piece).

I would like to increase the depth of the search tree. I plan to do this my making the algorithm even more efficient, by using bitboards, transposition tables, \_\_\_, \_\_\_\_. Another limitation is python. It is a interpreter language, which means that is reads my code one line at a time and executes it. I would rather have it compiled to machine code, which I have heard, will dramatically decrease computation time. I plan to do this though the python module “cython”.

1. I lost the game [↑](#footnote-ref-1)
2. I think they have given up on their chess project as a result [↑](#footnote-ref-2)
3. see <https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm> for a full explanation [↑](#footnote-ref-3)
4. there are approximately 11 moves from each board position, so the size of the search tree would be n11 with n being the depth of search. [↑](#footnote-ref-4)
5. see <https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm> for more detail [↑](#footnote-ref-5)