KACCE senior project report
Jonathan Dolan

Abstract

In 1997, an IBM built supercomputer called Deep Blue beat Gary Kasparov, one of the best human chess players in history, in a chess tournament. Deep Blue leveraged raw computational power in order to accomplish this. After this pivotal moment, the field of computer chess has remained largely the same, with small variations and different approaches to the same ideas. The goal of this project was to create a chess program specifically designed to beat other chess engines, using a varying evaluation function which will leverage the horizon effect.

Introduction

KACCE (KACCE Anti Computer Chess Engine), is a chess engine designed to beat other chess engines. I believe that there is a weakness in most chess engines that can be exploited by another chess engine. This weakness is in the MiniMax (or variation of) search algorithm that the chess engine uses to choose between moves. The MiniMax algorithm searches through all of the possible moves from a given position, and chooses one based on an evaluation function. Then the algorithm guesses what its opponent will play based on the same evaluation function, and so on until it chooses the best move that will lead it down the most beneficial path. The key to the weakness is that you have no guarantee that your opponent will play in a similar manner to you. This project was meant to exploit this weakness by leveraging the horizon effect, and changing strategies quickly, which would limit the opponent's ability to predict future moves and therefore would reduce its strength.

Implementation

KACCE was implemented in the Go programming language, which is a compiled language initially written at Google. Go is built for concurrency and speed. It has type inference, static types, as well as a garbage collector.

I also used a library that was also written in Go, which handled board representation. One of the hardest things about writing chess engines from an implementation standpoint is how to represent the board in the computer. Some intuitive options are to have a 64 cell array, and put objects representing each game piece in each cell depending where they are. Another option is to have a list of pieces that store their own positions. While it makes sense from a logical standpoint, this approach is very slow, since you have to deal with the board representation for every single step from move generation and evaluation function to the search algorithm. Consequently, a small speedup in this section can reap huge benefits overall. A very fast approach is to use bitboards, or 64 bit integers for each piece and color, with each bit representing a square. Using bitboards, it is possible to use bitwise operators to generate moves very quickly. I did not want to mess with bitwise logic all semester, so I used a library to do so. This library became the cause of most of the problems with this project.

Challenges

The main challenges encountered in this project involved the board representation library. The first problem I encountered was how to license the software. I wanted to make sure that all of the licensing and copyright was correct, and that proved quite a challenge because resources and examples were hard to find. I eventually was put in contact with Amy Kelly from the Giovale Library, who was able to help. I ended up having to put the license and copyright statement in an acknowledgments file, and then I could have my own license file elsewhere, and I would keep the copyright to my own work.

The second major challenge involved with this library is that it didn't work as well as I had hoped it would, and there were many issues and bugs. The entire library was badly designed, and was

very inefficient. For example, it has methods that dealt with the moves, such as a method to parse a move from algebraic notation in the package that was meant to deal with the board and not in the moves package. Another example of its inefficiencies was that it both generated a list of moves, and then determined if a given move was legal in the same method. Instead of filtering illegal moves out at that time, it ran the entire method over again for each move in the list in order to see if the move was illegal. In addition, the code that checked to see if a move is illegal had several bugs in it. An example can be seen in that it would say that moving your king into check was a legal move. I ended up having to rewrite from scratch about two thirds of the library.

Achievements

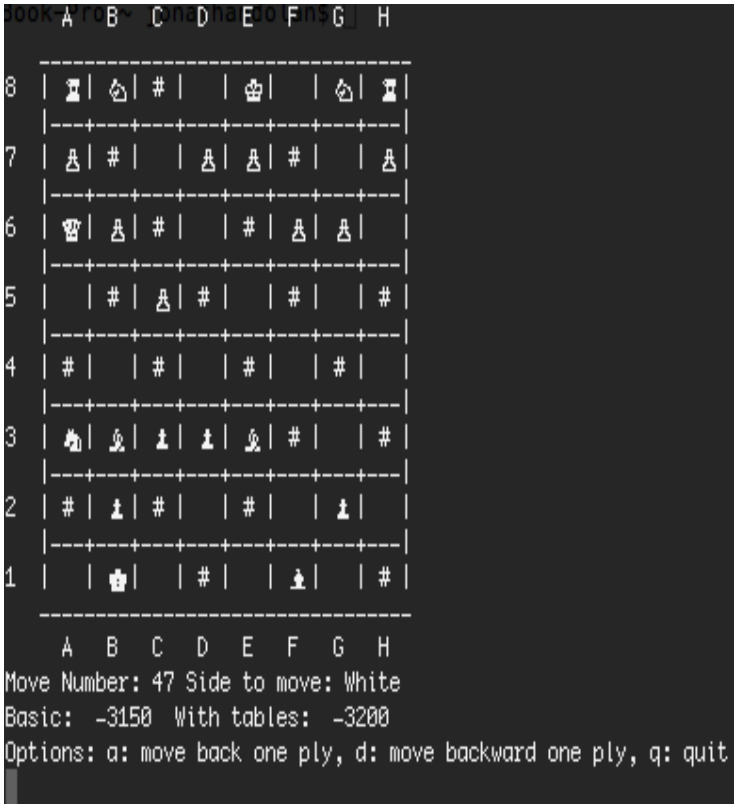
Thus far, a very solid walking skeleton has been implemented. KACCE plays chess correctly, and follows all the rules, but it does not play well. The standard way of ranking a chess players strength is called the Elo rating system. KACCE has an Elo score of about 1000, while other chess engines can score well above 3000.

Board representation, move generation, searching using the MiniMax algorithm with alpha/beta pruning, and gameplay have all been implemented and are working correctly.

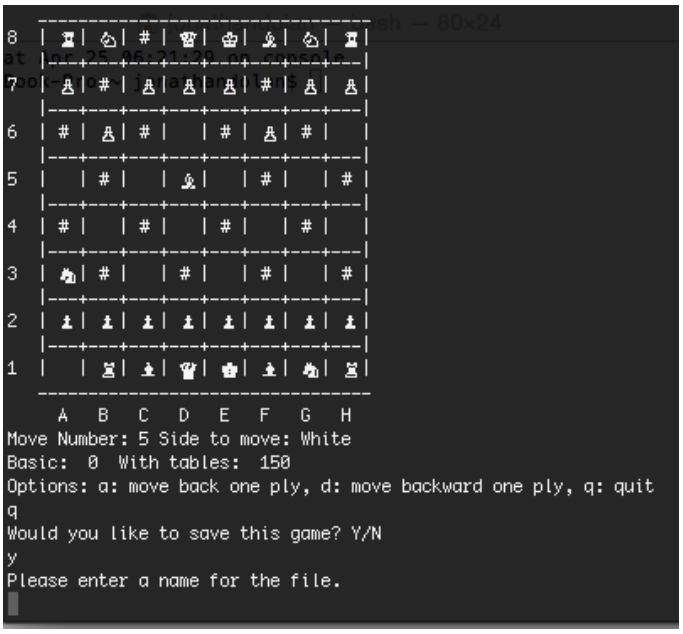
At this point, KACCE can play against another instance of itself with a different evaluation function, play against another chess engine using the UCI protocol, save a previously played game, and view saved games.

Screenshots

Gameplay screen
Displays board with unicode chess pieces



Viewing a previously saved game
Shows the evaluation of the position from two evaluation functions, and options.



When viewing a game, you have the option to save the game

```
1: rnbqkbnr/pppppppp/8/8/8/N7/PPPPPPPP/R1BQKBNR b KQkq - 1 1
2: rnbqkbnr/p1pppppp/1p6/8/8/N7/PPPPPPPP/R1BQKBNR w KQkq - 0 2
3: rnbqkbnr/p1pppppp/1p6/8/8/N7/PPPPPPPP/1RBQKBNR b Kkq - 1 2
4: rn1qkbnr/pbpppppp/1p6/8/8/N7/PPPPPPPP/1RBQKBNR w Kkq - 2 3
5: rn1qkbnr/pbpppppp/1p6/8/8/N7/PPPPPPPP/R1BQKBNR b Kkq - 3 3
6: rn1qkbnr/pbppp1pp/1p3p2/8/8/N7/PPPPPPPP/R1BQKBNR w Kkq - 0 4
7: rn1qkbnr/pbppp1pp/1p3p2/8/8/N7/PPPPPPPP/1RBQKBNR b Kkq - 1 4
8: rn1qkbnr/p1ppp1pp/1p3p2/3b4/8/N7/PPPPPPPP/1RBQKBNR w Kkq - 2 5
9: rn1qkbnr/p1ppp1pp/1p3p2/3b4/8/N7/PPPPPPPP/R1BQKBNR b Kkq - 3 5
10: rn1qkbnr/p2pp1pp/1p3p2/2pb4/8/N7/PPPPPPPP/R1BQKBNR w Kkq c6 0 6
11: rn1qkbnr/p2pp1pp/1p3p2/2pb4/8/N7/PPPPPPPP/1RBQKBNR b Kkq - 1 6
12: rn1qkbnr/p2pp1pp/1p3p2/2p5/8/N7/bPPPPPPP/1RBQKBNR w Kkq - 0 7
13: rn1qkbnr/p2pp1pp/1p3p2/2p5/8/N7/bPPPPPPP/R1BQKBNR b Kkq - 1 7
14: rn1qkbnr/p2pp1pp/1p3p2/2pb4/8/N7/1PPPPPPP/R1BQKBNR w Kkq - 2 8
```

A sample of the game history files
Uses FEN notation

Further development

I do plan to come back and continue development on KACCE. There is a solid walking skeleton, and a good platform for continued development and refinement. I would like to use what I have done so far to continue to test the original idea that most chess engines have a weakness that can be exploited.

I will have to finish the overhaul of the board representation and move generation components and make them faster and more efficient. Then I will be able to make more advanced evaluation functions in order to make KACCE stronger.

Conclusion

This project had many difficulties, all of which were unexpected. I thought the main challenges were going to be from an algorithmic standpoint of how to make a computer beat another computer, but most of the challenges were with copyright and using another programmer's library.

This project was also highly rewarding, and I learned a lot from it. I learned not to rely completely on someone else's work unless you have thoroughly tested it. I also learned more about functional programming, as this was the first project that I had been involved with that used it. It opened up a new paradigm of programming for me, and I will definitely use it in the future.