Genetic Algorithm Applied to a Computer Chess Engine

Ryan Barrett

December 21, 2015

Introduction

Chess is a game that is built on many complex ideas, both tactical and strategic; the best players understand these complexities and can apply their knowledge to many positions. Because modern computers are faster and more accurate than a human brain at evaluating chess positions, many computer chess players have been coded that convincingly beat even the strongest players in the world.

Computer players are tactically much stronger than their human counterparts, but it is unclear whether computers actually have a better strategic understanding of the game because they play so differently than humans. Oftentimes, a computer player will make a move that appears to be strategically unsound and will seem like a bad move to a human player, but is actually a good tactical or prophylactic move taking into account the distant future. Computer players play drastically differently than human players and grandmasters (the top ~1300 players in the world) have difficulty analyzing computer chess games. The goal of this project was to improve a chess engine's strategic understanding of the game.

Chess students are taught that the pieces have point values: Queen = 9, Rook = 5, Bishop = 3, Knight = 3, and Pawn = 1. The board can be roughly scored by computing the difference between one's point values and one's opponent's point values. Notice that the bishop and the knight have the same point value, although being different pieces that move in dissimilar ways. Grandmasters agree that bishops and knights are not equal under all circumstances: in a closed game (a game where there are many pawns blocking up the center) knights are better because they can jump, but in an open game, bishops are better because they can switch sides of the board more quickly than knights.

Chess engines generally use a point system much sharper than Q=9, R=5, B=3 K=3, P=1; Stockfish uses Q=2521, R=1270, B=836, K=817, and P=198. This experiment tests whether it is strategically advantageous or disadvantageous to trade a bishop for a knight in the opening and midgame. I hypothesize that I will find that bishops are slightly stronger than knights, because the default Stockfish values, which have probably been carefully chosen (perhaps even the result of an evolutionary algorithm), gives bishops a slight edge and because the majority of grandmasters believe that bishops are stronger than knights.

Background

Chess engines such as Stockfish take in a chess position and search for the best move. Stockfish searches for the best move by looking ahead to future possible positions and evaluating each possible position. Chess engines do not however, play full chess matches; they only suggest a move given a position. They are mostly used to study positions and analyze games. A program called Cutechess was used in this project to simulate full chess matches between two chess engines. Cutechess keeps track of the chess position that results from an engine selecting a move and then sending that resulting position to the opponent engine. When the opponent engine selects a move, Cutechess sends the resulting position back to the first engine. This continues until the game ends.

Stockfish defines the piece values in an enumeration in the types.h file. Enumerations are data structures that are designed to be static. To avoid recompiling Stockfish each time a new chess player was evaluated, the enumeration was replaced by an extern that could be written from the main.cpp in Stockfish; environmental variables were declared in the evaluation function that were read by Stockfish at runtime and written to the extern containing the piece values in types.h. Stockfish defines two types of piece values: middle-game and end-game. In this project, only the mid-game values were changed, and the end-game values were left as they were found.

Genesis is a program that assists in implementing genetic algorithms. The user only has to write the fitness function, a template of the genes to be evolved, and specify the parameters for evolution. In this project, the population size was 20, and number of trials was 500. A couple of crossover and mutation rates were tested. The settings were also modified to allow elitism; the fittest chess player from each generation moved onto the next generation. Genesis uses a random seed (a large number) to randomly generate each subsequent generation based on the evolution parameters.

Methodology

Genesis was used to evolve a string of piece values. The string contained five genes, one for each chess piece. The values for Pawn were constrained to 0-400. The values for Knight and Bishop were constrained to 600-1000. The values for Rook were constrained to 1000-1400. The values for Queen were constrained to 2400-2600.

The evaluation function calculated fitness by simulating eight games between a default Stockfish engine with the default piece values and a new Stockfish engine that used the values from the genetic algorithm. The new Stockfish played two games against the default engine with skill level = 19/20, and six games against the default engine with skill level = 20/20. Decreasing the skill level of Stockfish causes it to move more randomly. For example, a Stockfish with Skill level = 19/20 will sometimes choose the second or third best moves given a position. If the piece values used were even reasonable, the engine was almost guaranteed to beat the Stockfish with skill level of 19/20 both times it played; the Stockfish with skill level 19/20 was used to distinguish the chess players that had reasonable piece values from those with very bad ones. The new Stockfish engine earned 2 points for each win, and 1 point for each draw. Thus, the best possible fitness was 16 (8 wins, 0 draws, 0 losses) and the worst possible fitness was 0 (8 losses).

First, the environmental variables were set in the fitness function. One was set for each piece value. For example,

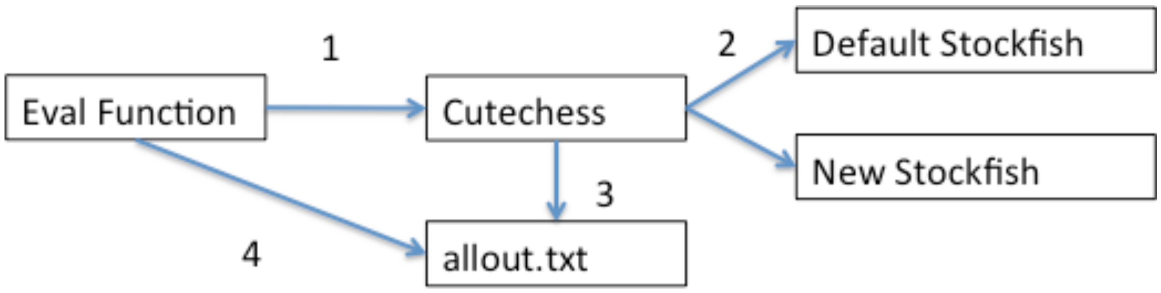
```
setenv("PawnValueMg", "198", 1);
```

would set the pawn value to 198 when the modified version of Stockfish was called. After all of the environmental variables were set, Cutechess was ready to be called.

Cutechess was called from the evaluation function by calling the system.

```
system("./cutechess-cli/cutechess-cli -engine cmd=stockfish -engine cmd=sloppy
-each proto=uci tc=40/1 -rounds 1 >> allout.txt")
```

The time control (tc=40/1) was set to 40 moves per second. In other words, each engine would have 1 second to make its first 40 moves, and another second to make its second 40 moves etc. Thus, both engines played 40 total moves in 2 seconds. The output was directed to a text file, which was called "allout.txt," so that the results could be read by the evaluation function. The Evaluation function evaluated each chess engine as follows.



The evaluation function (1) called Cutechess, then (2) Cutechess called the default Stockfish and the new Stockfish and (3) printed the game results to allout.txt. Then the Eval function (4), which had been waiting, read the game scores from allout.txt and used the values to compute the fitness. This was done eight times for each individual chess player.

The evolution was very slow; it took about 31 seconds to evaluate the fitness of each individual. A population size of 20 was used, and varying rates of crossover and mutation were used in every combination of 0.6 or 0.8 crossover rate with 0.01 or 0.001 mutation rate.

The best piece values, and their averages, from each evolution were subject to further testing. Each engine played a series of 20 matches against the default Stockfish (Skill level = 20/20). Again, each win was worth 2 points and each draw was worth 1 point, so the best score an engine could get was 40. The default engine was also played against itself, serving as a control. Each series of 20 matches were repeated 20 times; the results were averaged, plotted, and tested for statistical significance with respect to the default Stockfish engine.

<u>Results</u>

Crossover rate = 0.6, Mutation rate = 0.001

Engine ID	Generation	Pawn	Knight	Bishop	Rook	Queen	Fitness
1	13	87	722	628	1222	2534	16
2	23	87	722	628	1378	2534	16
3	38	87	722	666	1295	2534	16

Figure 1. There were 3 chess players that achieved a fitness of 16 in the evolution using a crossover rate of 0.6 and a mutation rate of 0.001. Their piece values are shown above.

Crossover rate = 0.6, Mutation rate = 0.01

Engine ID	Generation	Pawn	Knight	Bishop	Rook	Queen	Fitness
4	4	87	740	927	1087	2409	16
5	15	87	743	628	1279	2408	16
6	25	88	759	927	1079	2406	16

Figure 2. There were 3 chess players that achieved a fitness of 16 in the evolution using a crossover rate of 0.6 and a mutation rate of 0.01. Their piece values are shown above.

Crossover = 0.8, Mutation = 0.001

Engine ID	Generation	Pawn	Knight	Bishop	Rook	Queen	Fitness
7	11	210	915	944	1120	2564	16
8	14	87	709	928	1213	2443	16
9	22	210	909	928	1213	2443	16

Figure 3. There were 3 chess players that achieved a fitness of 16 in the evolution using a crossover rate of 0.8 and a mutation rate of 0.001. Their piece values are shown above.

Crossover = 0.8, Mutation = 0.01

Engine ID	Generation	Pawn	Knight	Bishop	Rook	Queen	Fitness
10	4	46	877	994	1298	2438	15
11	5	43	772	648	1140	2534	15
12	14	86	709	648	1140	2543	15
13	21	89	709	926	1059	2415	15
14	28	89	890	648	1077	2537	15
15	28	89	702	926	1077	2537	15

Figure 4. There were no chess players that achieved a fitness of 16 in the evolution using a crossover rate of 0.8 and a mutation rate of 0.01. There were 5 chess players that achieved a fitness of 15. Their piece values are shown above.

Average Values from engines with Fitness = 16

Engine ID	**Avg. Generation**	**Avg. Pawn**	**Avg. Knight**	**Avg. Bishop**	**Avg. Rook**	**Avg. Queen**
Avg_{16}	18.3	114	771	800	1210	2475

Figure 5. The above values are the average piece values that resulted in a fitness of 16.

Average Values of engines with Fitness = 15 from Crossover = 0.8, Mutation = 0.01 evolution

Engine ID	**Avg. Generation**	**Avg. Pawn**	**Avg. Knight**	**Avg. Bishop**	**Avg. Rook**	**Avg. Queen**
Avg_{15}	16.7	74	777	798	1132	2499

Figure 6. The above values are the average piece values that resulted in a fitness of 15 in the evolution using a crossover rate of 0.8 and mutation rate of 0.01.

Results of testing strongest modified engines against the default Stockfish.

Engine ID	**Trial 1**	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Average**
1	38	38	39	36	40	38.2
2	32	35	34	37	38	35.2
3	38	39	38	34	33	36.4
4	37	37	37	38	33	36.4
5	36	38	39	38	40	38.2
6	37	35	35	35	37	35.8
7	40	40	38	38	40	39.2
8	34	36	36	36	36	35.6
9	38	39	39	36	40	38.4
10	31	30	34	36	38	33.8
11	37	36	35	36	35	35.8
12	37	35	36	32	35	35
13	40	35	38	38	36	37.4
14	34	36	37	34	34	35
15	35	35	38	34	34	35.2
Avg_{16}	37	35	40	39	36	37.4
Avg_{15}	33	34	37	35	35	34.8
Default	21	20	20	18	21	20

Figure 6. The figure above shows the results from each trial of 20 consecutive matches between a modified chess engine and the default chess engine. Their averages are shown in the far right column.

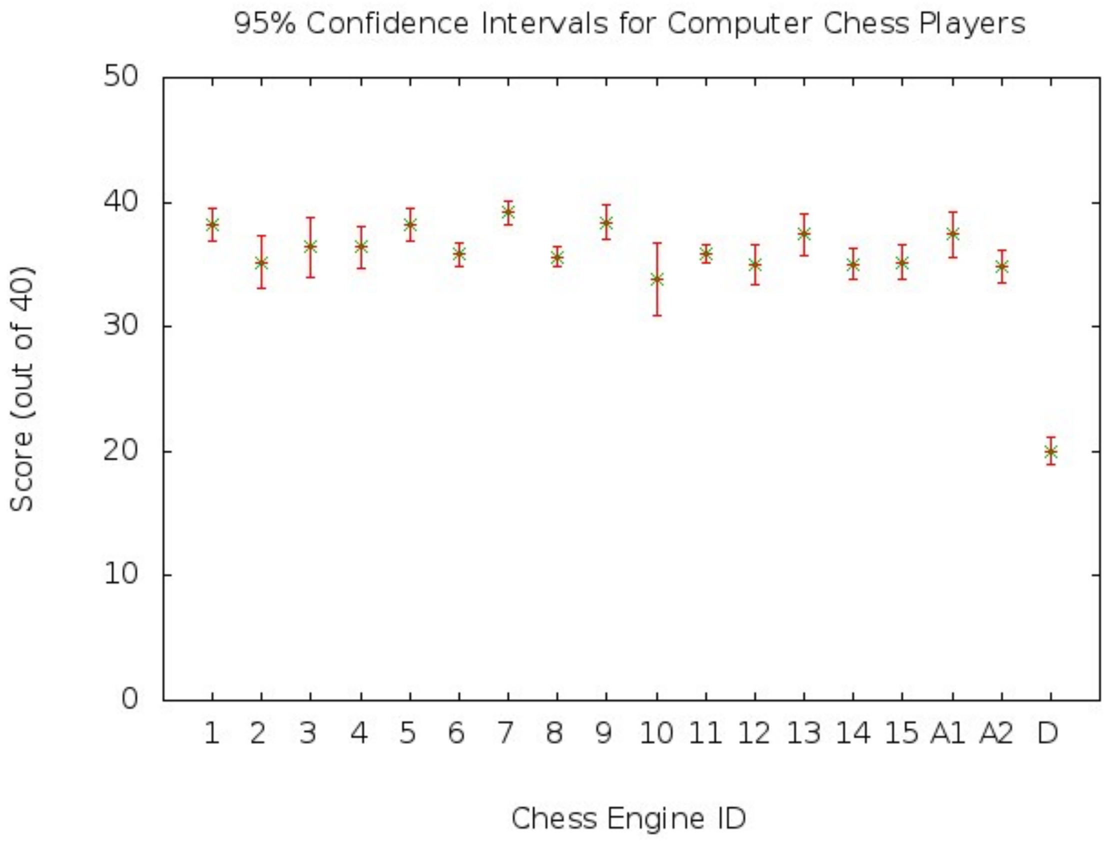


Figure 7. The above image shows the 95% confidence intervals of how the modified engines could be expected to score against the default Stockfish engine when the time control was 40 moves per second. The average values that resulted in fitness = 16 did better than the average values that resulted in fitness = 15 from the evolution using 0.8 crossover rate and 0.01 mutation rate.

Key: A1 = average of piece values that resulted in fitness = 16, A2 = average of values that resulted in fitness = 15 from evolution using 0.8 crossover rate and 0.01 mutation rate, D = Default values.

Discussion

Many combinations of piece values were found that performed better than the default Stockfish piece values when the time control was set to 40 seconds per move and only one processor was available for the move search. Every chess engine that achieved a fitness of 16, every chess engine that achieved a fitness of 15 in the evolution using 0.8 crossover rate and 0.01 mutation rate, and the averages of each of the aforementioned sets of engines beat the Stockfish using the default piece values very consistently. This project however can only conclude that the values that resulted in high fitnesses were stronger than the default Stockfish engine when the time control was 40 moves per second and the number of processor available was 1. To apply these results to a less restrictive time control or to when there are more processors available, other experiments would need to be conducted using a less restrictive time control and/or using multiple processors.

On average, the pawn value that resulted in a fitness of 16 was much lower than the default value used by Stockfish. The average pawn value that resulted in a fitness of 16 was 114; Stockfish uses 198. All of the other average piece values that resulted in a fitness of 16 were lower than the default values used by Stockfish.

Piece	**Avg. of Fitness = 16**	**Default Value**
Pawn	114	198
Knight	771	817
Bishop	800	836
Rook	1210	1270
Queen	2475	2521

The lower piece values can be explained by the strict time control. The time control used was 40 moves per second, which is very fast even for a chess engine. Stockfish was designed to play 40 moves in 40 minutes, not one second. It was also designed to use multiple processors to play at the highest level. The strict time control and its access to only one processor limited how deeply it could search for moves. Both the modified chess engine and the default Stockfish were constrained by the same time constraint and same number of processors; the modified chess engine evolved and adapted to play with a style that was better suited for a move search that couldn't search very far. The modified chess engines that consistently beat the default Stockfish often, though not always, had lower piece values. This indicates that these engines were less worried about material, and more worried about other positional attributes, and would be more likely to sacrifice pieces for a strong attack to give checkmate or simply for a better strategic position.

On average, the bishop was valued slightly higher than a knight, although this was not true of all engines that attained a fitness of 16. This is the result that was expected. Most grandmasters prefer bishops to knights, but will often trade their bishops for knights to gain a different strategic advantage (like doubling their opponents pawns) or tactical advantage (like capturing a pawn). The small difference between the average knight value and average bishop value resulting in a fitness of 16 reflects this idea that the two pieces, although very different, are almost equal, but a bishop will still be traded for a knight if it means gaining a different type of advantage (eg. one that is not material).

As expected, the queen value did not significantly change the strength of the engine. There were strong modified engines with low queen values and high queen values. This was probably because chess players should almost give their queen away for another piece. The only time chess players should give up their queen is in return for their opponent's queen, or to deliver checkmate.

It was unclear which crossover rates and mutation rates were best because only one experiment of each evolution was run. The evolutions using crossover rate = 0.6

mutation rate = 0.001, crossover rate = 0.6 mutation rate = 0.01, and crossover rate = 0.8 mutation rate = 0.001, each found 3 chess players with a fitness of 16. The only evolution that did not find a chess player with a fitness of 16 was the one that used a crossover rate of 0.8 and a mutation rate of 0.01. Because only one experiment for each of the combinations of crossover rate and mutation rate were run, it is still unclear whether or not this was due to random chance because not enough experiments were run. All of the evolutions were clearly learning, though. In the early generations, the fitnesses of all of the individuals were usually low, although sometimes there was an individual that was an anomaly, but in the late generations the fitnesses were always very high; most of the population had fitnesses greater than 10 by the last generation in all four of the evolutions.

Future Works

Stockfish uses many things other than the values of the pieces to evaluate a chess position. For example, it is advantageous for a chess player to place his or her rooks on open files, which are files that don't have pawns on them. It is also advantageous for a chess player to have a pawn that is unobstructed by any of the opponent's pawns so that it can promote more easily; this is called a passed pawn. There are many other attributes not mentioned above that Stockfish uses to quantify a chess position that could be evolved just as easily as the values of the pieces: King safety, doubled pawns, light-square control, dark-square control, bishop pair etc.

The results of this project could be more broadly applied if the same evolution and tests were run with a different, probably less restrictive time control. Stockfish was developed to analyze complex chess positions, and was not designed to play super rapidly, which is required when the time control is 40 moves per second. Because of this, it was not super surprising that the default Stockfish engine was easy to beat when the time control was so strict.

If there were more time to run the evolutions, it would be better to use a population size greater than 20. The populations converged very quickly with such a small population size; this resulted in many individuals sharing many attributes. Notice how many of the chess players that attained a fitness of 16 had a pawn value of exactly 87. It is doubtful that a pawn value of 87 is significantly different from a pawn value of 85 or 90, but the pawn value is much more likely to be 87 because the random seed number generates a fit individual in the first generation that uses 87 as the pawn value. It would also be good to run multiple experiments of each evolution so that the best evolution parameters could be determined.

A population-based incremental learning (PBIL) algorithm could also be a good option to explore. PBIL algorithms evolve populations by shifting the genes away from the worst individual and toward the best individual in a population. A PBIL algorithm should generally converge to the best possible solution by the final generation; this might work better than a genetic algorithm for evolving chess players. The algorithm would be run several times, and the averages of the fittest individuals in the last generation would be observed for further testing.

Acknowledgements

Cutechess: Cutechess.com

Stockfish: StockfishChess.org

Special thanks to Clare Congdon and Stephen Houser for their help and contributions to this project.

Appendix

Environmental Variables

I had never used or even heard of environment variables before this project. The piece values were communicated to Stockfish from the evaluation function using several environmental variables, one for each piece. I learned, with help from Clare and Stephen, how to set and read them both in programs and from the command line.

System calls

I had never implemented a system call from a program, although I had imagined that such a thing existed. The evaluation function calls Cutechess with the following command:

```
system(“./cutechess-cli/cutechess-cli -engine cmd=stockfish -engine cmd=sloppy
-each proto=uci tc=40/1 -rounds 1 >> allout.txt”)
```

The system function takes an array of characters and executes it from the command line. The rest of the program waits until the executed command has completed.

Communicating via Files

The idea of communicating between different programs via files was a new idea to me. It didn’t always work out great (like when the runs crashed on the HPC), but eventually it was a good system. It was easier to implement than combining Genesis, Cutechess, and Stockfish into one program. Although, the final product probably would have been smoother if all of the programs were combined into one.

Using the Network

I learned how to use the high-powered computer (moosehead) and how to submit jobs to it. I also learned how to use Herbert, which was not connected to the network, so I had to learn the basics of Fetch to copy my project to Herbert.

Modifying Files

I also learned how to use the sed command to find and replace text in files. I was using this to modify the types.h class when I was still recompiling each new Stockfish to test. When Stephen showed me how to use environment variables and change the enumeration data structure so that Stockfish could change the piece values at run time, the evolution was able to go much faster, and there was no more need to use the sed command to modify the types.h file because Stockfish was not recompiled with each call to the evaluation function.