Genetic Chess

Mark Harrison

January 14, 2020

Abstract

This work is a program for evolving chess-playing AIs. By pitting a population of AIs against each other in chess matches, killing off the losers, and breeding the winners, it is hoped that one specimen will be able to stand up against a more traditionally developed engine (if only on the easiest difficulty setting). Though it is written in C++, it is the hope of the author that the style and architecture are comprehensible.

Contents

1 Building

There is nothing OS-specific about the C++ code, so the code can be compiled with any C++ compiler via

```
<compiler> -I include -D NDEBUG <all *.cpp files>
```

where -I indicates the option to specify the base directory of the header files and -D NDEBUG instructs the compiler to skip tests and assertions. For testing and debugging, use

```
<compiler> -I include -D DEBUG <all *.cpp files>
```

instead.

1.1 Linux

Run make to create various executables in a bin/ subfolder that will be created if it does not already exist. If, in the course of working on this project, new files are created or the #include files are changed within a file, run python create_Makefile.py [compiler] to regenerate the Makefile. The choices for the compiler are gcc and clang. These

python script assumes all *.cpp files in the current directory and all subfolders are part of the project and need compiling. Then, run make to build any of the following: build for an executable with debugging symbols and no optimization, profile for an optimized build that generates profiling information while running, and release for a fully optimized executable.

1.2 Windows

The .sln and .vcxproj files are project files for Visual Studio. These will load all source files for one-click compiling.

2 Running

In the options below, words surrounded by angle brackets <like this> are meant to be replaced by a value for the option. If the word is also surrounded by straight brackets [<like this>], then it can be omitted.

These first options are standalone operations. If multiple options are used, all besides the first are ignored.

genetic_chess -help Print the help text that describes how to run the program.

genetic_chess -test Run tests of the program for chess rule conformance and for Genetic AIs working properly.

genetic_chess -perft: Test game logic and speed by counting legal moves from a list of board positions.

genetic_chess -speed: Measure the speed of various components of the chess engine.

genetic_chess -genepool <file name> This will start up a gene pool with Genetic_AIs playing against each other—mating, killing, mutating, all that good Darwinian stuff. The required file name parameter will cause the program to load a gene pool and other settings from a configuration file. A record of every genome and game played will be written to text files.

The following options can be used together to start a single chess game.

genetic_chess <player> [<player>] Starts a local game played in the terminal with an ASCII art board. If two players are specified, the first parameter is the white player, and the second is black. If only one player is specified, the program waits for GUI input on stdin. The <player> argument is replaced with one of the following:

- **-genetic [<file name> [<number>]]:** a Genetic AI player. If a file name follows, load the genes from that file. If there are several genomes in a file, the file name can be followed by a number to load the genome with that ID. If no number is specified, then the last genome in the file is loaded (presumably this one is the most evolved)..
- **-random:** an AI player that chooses a random legal move at each turn.

Other game options that can be added:

-time <number>: The starting about of time on the game clocks in seconds.

-repeat_moves <number>: The number of moves a player must make before the clocks are reset to their starting time.

-increment <number>: The amount of time to add to the clocks after every move.

-pondering: Allow chess engines/AIs to think while it is not their turn to move.

-board <FEN> A description of a board setup in Forsyth-Edwards Notation on which to start the game.

Genetic Chess can communicate with GUI chess programs (xboard, PyChess, Cute Chess, and lichess.org among others) through the Chess Engine Communication Protocol (CECP) or the Universal Chess Interface (UCI). When using Genetic Chess this way, only specify the arguments for a single player. The program will then wait for communication from the GUI through stdin.

3 Non-Evolutionary Aspects

These sections describe the aspects of the chess AI that are not genetically modifiable, usually because of at least one of the following reasons:

- There is no sense in which the aspect of play is improvable. Any modification would be detrimental to the chess playing;
- It would take far too much time to evolve that aspect of play from a random starting configuration, or it would interfere too much with the evolution of other aspects;
- I cannot conceive of how to represent the state space of that particular play strategy so that it may be genetically encoded.

On the last point, it has been suggested to me that, instead of the specific genes listed in Section 4, the genes should encode more abstract and generic strategies and heuristics for evaluating a board state. While this would probably better mimic biological evolution (wherein adenine and thymine are rather neutral as to their teleology), I have no idea how to program such an abstract representation and how to translate the action of such genes into chess moves. So, what results from all this programming is a glorified tuning algorithm for parameters in predefined genes with hard-coded meanings.

On the other hand, so is every other genetic algorithm (see [1], [2], and others). Plus, these genes can evolve to have near-zero influence on game decisions, so these AIs are perfectly capable of telling me exactly what they think of my painstakingly crafted genes.[1]

3.1 Endgame Scoring

Winning gives a positive infinite score. Losing gives a negative infinite score. Draw gives zero.

Why not evolve these numbers? While the priorities of various genes can be varied to yield different playing styles, the only reasonable score to assign to a win is one that is larger than any other score. It can only be a disadvantage to prefer anything to a winning move. While this would result in upward evolutionary pressure on the score assigned to winning, it would stall the evolution of all other genes while the score assigned to winning was pushed high enough to always be preferred.

[1] The little ingrates!

The specific values were chosen to make the scoring symmetrical between the two players, in that the score for one side is the negative of the score as seen from the other side (assuming the same player does the scoring). What is good for one player is bad for the other player by the same amount.

3.2 Mini-maxing

The principle behind the minimax algorithm is that the quality of a move is measured by the quality of moves it allows the opponent to make. Of course, the quality of those moves is measured by the quality of the moves that follow. Ideally, the only required board evaluation scores would be 1 for a win, 0 for a draw, and -1 for a loss, and all possible sequences of moves would be examined to find the guaranteed outcome. Unfortunately, the number of positions to examine in a typical game is far too large to examine in a few minutes, so the search has to be cut off at some point and a heuristic evaluation employed to estimate the probability of winning from that stopping point. In this program, the decision of when to stop and the heuristic evaluation is genetically determined and evolved over many generations.

3.3 Alpha-Beta Pruning

Alpha-beta pruning is based upon keeping a record of two game state evaluations:

Alpha: is the highest guaranteed score that the player whose turn it is can reach once the game reaches the current state (i.e., the root of the current game tree) and cannot be avoided by the opponent making different moves later in the game. That is, the player whose turn it is at this point in the game tree can force the game into a state with at least this score.

Beta: is the lowest score to which the opponent can limit the current player by making moves that avoid higher scores. If the current player finds a move with a higher score than Beta, then the opponent would make a different move earlier in the game that makes this game state impossible to reach. This earlier move is called a "refuting" move. Once a move is found that is refuted by an earlier opponent move, the examination of that set of moves is abandoned, as the opponent will not allow the game to reach that state.

Each time a move examination reaches a greater depth in the game tree, these values switch roles to represent the view of the board from the opponent's perspective.

An extra optimization implemented in this program is one where the search is cut off if Alpha represents a win at a shallower depth in another branch of the tree branch. If a checkmate can be forced in fewer moves by making different earlier moves, there's no point in looking for a win in the current move sequence.

3.4 Principal Variation Recall

If the best move is chosen based upon the probable resulting future board state based on a sequence of moves (a variation) found through minimaxing with alpha-beta pruning, and if the opponent makes the next predicted move in that variation, then the moves leading to that board state are examined first during the next move. This high-scoring board state should lead to early cutoffs from alpha-beta pruning, especially if that board state was a game-ending state. If that board state is actually avoidable by

the opponent, then the shallower depth of that state during the next turn should lead to a faster refutation, leaving time to examine alternate variations.

3.5 Move Ordering

Besides Principal Variation Recall, those moves that capture an opponent's piece are examined before other moves. Capturing moves usually result in the largest changes in position evaluations (see Section 6.3), so they should quickly establish alpha and beta values that will lead to quick cutoffs.

4 The Genome

The genome is the repository for genetic information in the Genetic AIs and controls all aspects of game play not mentioned in the previous section. All are subject to mutation, which can change the behavior and influence of a given gene.

4.1 A Note on Designer Genes

Evolution is an effective means of optimizing when every mutation makes a difference in the fitness of an organism. In an abstract sense, the fitness function of an organism should be

1. a continuous function of the parameters of its genome and
2. without flat sections.

The first of these qualities is due to the fact that large changes in behavior are rarely beneficial. A small change to the genome should not result in a large change in behavior. This is why the deactivating-gene mutation was discarded. It is better to let a gene fall in importance by gradually reducing its priority. The second point means that gene parameters should not be allowed to take on values where mutations no longer have any affect on behavior. This leads to random walk types of gene movements until the gene parameters once again reached a range of values that changed the behavior of the AI. This random walk period only represents wasted time. To this end, certain gene parameters are restricted to, for example, non-negative values.

4.2 Regulatory Genes

A regulator gene refers to a gene that does not participate in evaluating the state of a game board. These genes either control other aspects of the Genetic AIs or are queried by other genes for information.

4.2.1 Piece Strength Gene

This gene specifies the importance or strength of each different type of chess piece. Other genes like the Total Force Gene (Section 4.3.1) reference this one for their own evaluation purposes.

Previously, the king was not assigned a strength because it is always on the board, so it cannot affect the move chosen. This is not quite true. The score returned by the Total Force Gene cannot be affected by the king's strength because there is always one black and one white king on the board, canceling their contribution to a board

position's score (see Sections 4.3 and 4.3.1). However, the Opponent Pieces Targeted Gene does have a use for that value, since putting an opponent in check is an important component of the game. Some surprising results of adding a strength value for the king are discussed in Section 6.1.1.

The values of the piece strengths are modified in a similar fashion and for similar reasons to the board-scoring gene priorities as described in the summary section of Section 4.3.

4.2.2 Look Ahead Gene

This gene determines all aspects of the Genetic AIs time management through a set of genetically determined components to this gene that determine how the AI spends its time:

1. the average number of moves per game,
2. the uncertainty in the above average.
3. a constant related to over-allocating search time while counting on alpha-beta pruning to not use too much

When the AI starts the algorithm to choose a move, it allocates a fraction of the time left on the clock so that all subsequent moves get equal time. That is, if there are 20 moves estimated to be left in the game, then 1/20 of the remaining time is used for this move. The number of moves left is estimated by assuming the number of moves in a game is modeled by a log-normal distribution [3][4] with a mean and spread given by the first two parameters listed above. Originally, it was assumed that the number of moves in a game was well-modeled by a Poisson distribution. This proved to be false (see Figure 1), and the log-normal distribution—wherein the logarithm of the variable is normally distributed—is now used.

The number of moves left in the game is estimated by

$$N(n) = \frac{\sum_{i=n+1}^{\infty} P(i) \times i}{\sum_{i=n+1}^{\infty} P(i)} - n$$

where $N(n)$ is the estimated number of moves left in the game given that n moves have already been made, and $P(i)$ is the *a priori* probability distribution of the number of moves by one player in a single game. The numerator is a truncated calculation of the average number of moves in a game, and the denominator is a truncated probability distribution to renormalize (since the probability of a game lasting 10 moves is zero if 11 moves have been played).

The log-normal probability distribution is given by

$$P(x) = \frac{1}{xS\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{\ln x - M}{S}\right)^2},$$

where S and M are the standard deviation and mean of the logarithm of the number of moves in a game, respectively. Even though this is a continuous probability distribution, it is a good fit for the length of chess games as can be seen in the plot of the number of moves in a long gene pool run can be seen in Figure 2. Instead of using the summation formula above to estimate the number of moves remaining in the game, the result can be approximated with integrals[2] to yield a closed-form expression using

[2]Courtesy of Wolfram Alpha: http://www.wolframalpha.com/

functions available in native C++.

$$\begin{aligned}\sum_{i=n+1}^{\infty} P(i) \times i &\approx \int_{n+1}^{\infty} P(t)tdt \\ &= \frac{1}{S\sqrt{2\pi}} \int_{n+1}^{\infty} e^{-\frac{1}{2}\left(\frac{\ln t - M}{S}\right)^2} dt \\ &= \frac{1}{2} e^{M+S^2/2} \left[1 + \operatorname{erf}\left(\frac{M + S^2 - \ln n}{S\sqrt{2}}\right)\right]\end{aligned}$$

and

$$\begin{aligned}\sum_{i=n+1}^{\infty} P(i) &\approx \int_{n+1}^{\infty} P(t)dt \\ &= \frac{1}{S\sqrt{2\pi}} \int_{n+1}^{\infty} \frac{1}{t} e^{-\frac{1}{2}\left(\frac{\ln t - M}{S}\right)^2} dt \\ &= \frac{1}{2} \left[1 + \operatorname{erf}\left(\frac{M - \ln n}{S\sqrt{2}}\right)\right].\end{aligned}$$

Here, erf is the error function given by

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

This results in the number of moves left in a game ($N(n)$) after n moves is

$$N(n) = \left(e^{M+S^2/2}\right) \frac{1 + \operatorname{erf}\left(\frac{M+S^2-\ln n}{S\sqrt{2}}\right)}{1 + \operatorname{erf}\left(\frac{M-\ln n}{S\sqrt{2}}\right)} - n.$$

If the number of moves left in the game is greater than the number of moves that will result in a clock reset, then the latter will determine how much time will be used. In a game with 40/5 time control, where five minutes are added to the clock every 40 moves, every 40th move can use all the remaining time since an extra five minutes will be added after the move. Whatever the final number of moves left is used, the time on the clock is divided by that number to determine how much time to take to choose a move.

When choosing a move from the current board, the amount of time to consider a move is equal to the amount of time left for this board position divided by the number of legal moves left to consider. This naturally limits the depth of search while allowing deeper searches for positions with fewer legal moves, which are often a series of forced moves due to check and so deserve deeper inspection. If a move examination is cut off early for whatever reason (e.g., a game-ending move is found or through alpha-beta pruning), then the remaining time is available for as yet unexamined moves.

An effective chess player needs to look at the consequences of a move to decide if a move is good. In order to decide how far to look ahead (and if there is time remaining to examine this move), at each step, the AI divides the number of legal moves in the board position after the move under consideration by the number of positions it can examine per second.

$$T = k \frac{N_{\text{legal}}}{v}$$

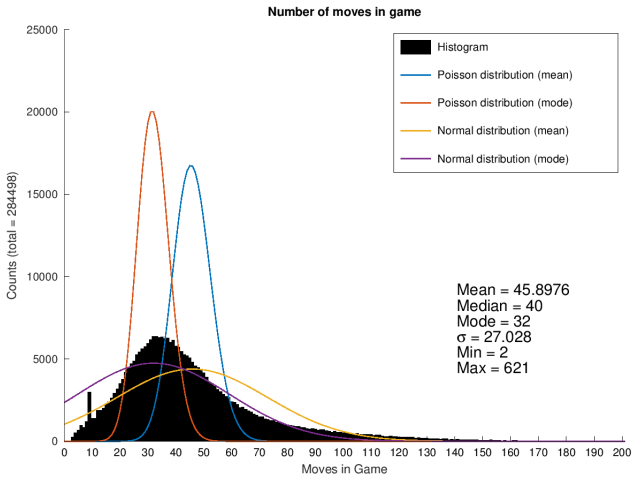

Figure 1: First attempts at determining the distribution of moves in chess games. The black histogram shows the distribution of the number of moves in a game (with one move consisting of both a black and white play). The (mean) and (mode) designations in the legend indicate which statistic was used as the mean parameter of the fitted Poisson and normal distributions (neither of which are a good fit). The spike at 9 moves was created just after a change to the program that resulted in much faster changes in board state.

where T is the time needed to look ahead on this move, N_{legal} is the number of legal moves on the board after the move is made, v is the rate of position examinations, and k is a genetically determined speculation constant that adjusts the amount of time to use. The speed v is determined by counting the number of positions evaluated (leaf nodes in the game tree) and dividing by the amount of time spent evaluating during the previous move. If the time T is less than the time allocated for the move, the algorithm looks ahead (the search function recurses) to examine the moves the opponent can make in response. Otherwise, it evaluates the current position and moves on to the next legal move.

4.2.3 Mutation Rate Gene

This gene controls how many randomly chosen genes get mutated every time there's a call to Genome::mutate(). The unique aspect of this gene is that it is the only one that does not change the behavior of the AI during its lifespan. Mutations only occur upon birth, so this gene only controls how the offspring of the containing AI are modified. There is a risk that a moderately successful AI will shutdown mutation to preserve a currently winning genome. Thus, the minimum number of mutations is one.

4.3 Board-Scoring Genes

These genes are used to give a score to a board state. The higher the score, the more desirable the moves that lead to this board. The score is calculated by

$$Score = \sum_{g} Priority(g) \times Score(g, B)$$

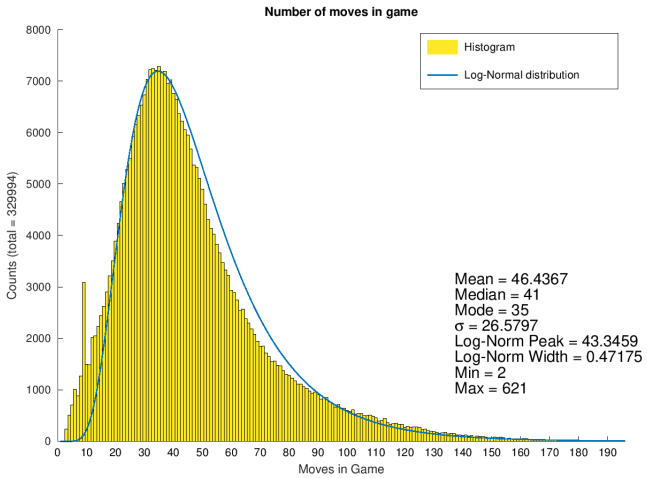

Figure 2: The distribution of the number of moves in a game with a log-normal fit. The Log-Norm parameters listed are M for the Peak parameter and S for the Width parameter. To obtain a better fit, only games longer than 15 moves were considered, as shorter games were primarily by ill-adapted AIs (especially the spike at 9-move games), especially at the beginning of the gene pool run.

where g represents each gene, $Priority(g)$ is a genetically determined scalar multiplicative factor that determines how much the gene's score of the board influences the final score, and $Score(g, B)$ is the result of the scoring procedure of that gene on a given board B. During mutation, the size of the priority modification is equal to the square root of the current priority. This is done for these reasons:

- Mutations to high priority genes can still have an effect on gameplay,
- Gene priorities that have a large proper value (namely, the Total Force Gene (see Sec. 6.3) can achieve that value quickly,
- If a gene has a proper value closer to zero, then it can settle on a proper value with respect to other low-priority genes using smaller mutations.

The square root of the current priority value was chosen as a compromise between a constant mutation size (which would lead to linear growth at best) and a mutation size proportional to the priority value (which would lead to exponential grown and probably wild swings during mutation at all priority values). A minimum priority mutations size of 10.0 is set so that the priority does not get trapped near zero.

Since it is not only important to find position that are advantageous to the player, but also disadvantageous to the opponent, the final heuristic score for a board position is given by

$$Heuristic\ Score = Score(Player) - Score(Opponent).$$

In general, the scoring function outputs of each gene are scaled so that a typical board state gets a score near one. For example, the Freedom to Move Gene divides the number of legal moves by the number of legal moves at the start of the game (20 for standard chess). This way, the priorities of genes can be easily compared as evolution progresses.

4.3.1 Total Force Gene

This gene sums the strength–according to the Piece Strength Gene (see Section 4.2.1)–of all the player's pieces on the board. The score returned is divided by the total value of the starting set of pieces (8 pawns, 2 rooks, 2 knights, 2 bishops, and 1 queen) so that the result is between 0 and 1.

Since there is always a black king and a white king on the board, the king values cancel in this gene's scoring method. So, the king value is only determined by the Opponent Pieces Targeted Gene (Section 4.3.5). This may be interpreted as the value of putting the other player in check.

As noted in Section 4.3 above, the priority of this gene is restricted to be non-negative.

4.3.2 Freedom to Move Gene

This gene counts the number of squares under attack by a side that are also not occupied by friendly pieces. This represents how many moves are available without considering whether the king is in check or if any pieces are pinned. This count is divided by the number of squares under attack at the beginning of a game to bring the maximum score near one.

4.3.3 Pawn Advancement Gene

This gene measures the progress of all pawns towards the opposite side of the board.

$$S = \frac{1}{8} \sum_{p} \left| \frac{rank(p) - home(p)}{5} \right|$$

Here, S is the total score, the sum is over all pawns p of the scoring side, $rank(p)$ is the rank position of the current pawn, and $home(p)$ is the home rank of the pawn (2 for White and 7 for Black). The division by 5 (the farthest a pawn can advance before promotion) and the division by 8 (the maximum number of pawns) normalize the score to be between 0 and 1.

4.3.4 Passed Pawn Gene

The gene counts the number of pawns that cannot be stopped or captured by an opponent's pawns. That is, a pawn is a passed pawn if there are no pawns ahead of it in its file or any directly adjacent files. Partial points are awarded for pawns that have fewer pawns ahead than usual (1/3 for each free column ahead of mid-board pawns, 1/2 for for pawns on the edge).

4.3.5 Opponent Pieces Targeted Gene

This gene sums the total strength (as determined by the Piece Strength Gene) of the opponent's pieces currently under attack. The returned result is divided by the total score of pieces at the start of the game, similarly to the Total Force Gene, although the value of the king is included in this divisor. Despite relying on values from the Piece Strength Gene, it is not restricted to non-negative priorities like the Total Force Gene because the Total Force Gene provides enough evolutionary pressure on the Piece Strength Gene to force it to a consistent sign.

Interestingly, since both kings are always on the board, they always cancel each others effect in the Total Force Gene, leaving this gene alone to drive their strength value. The result of this is discussed in the results section (Section 6.1.1).

4.3.6 Sphere of Influence Gene

This gene counts the number of squares attacked by all pieces. Bonus points are awarded if the square can be attacked with a legal move. That is, if a piece cannot reach a square in one move (perhaps because such a move is blocked by another piece), then that square is still counted as falling under the influence of the side owning that piece. Further bonus points are awarded based on how close the attacked square is to the king. Because of its subcomponents, this gene's priority is restricted to non-negative values.

The effective formula for the score of this gene can be expressed as

$$score = \sum_{s} legal(s) \left(1 + \frac{KTF}{1 + distance(s, king)}\right)$$

where the sum is over all squares s on the board. The function $legal(s)$ returns 0 if s is not attacked at all, or one of two genetically determined factors depending on if the the attack is legal (ignoring checks or pins) or not. An illegal move is one that is blocked by another piece. The term KTF is the king target factor and is a genetically determined weight that adjusts the importance of attacking squares close to the opposing king. The function $distance(s_1, s_2)$ gives the distance between two squares in king moves and is equal to the maximum of the difference in file or rank between the squares. In the formula above, *king* represents the square the opposing king occupies.

4.3.7 King Confinement Gene

This gene tracks the manner in which the king is prevented from moving. In general, a king is safe if it is surrounded by friendly pieces or, failing that, it is free to move around the board when there are opposing pieces near. In scoring a board, this gene tracks the squares the king can reach given unlimited consecutive legal moves up to a genetically determined maximum distance.

For a more detailed picture, each square potentially reachable by the king is classified into three types given one of three genetically determined scores for each:

- Free: the king is free to move on to these squares,
- Friendly block: the king is blocked by pieces of the same color, and
- Opponent block: the king is blocked due to the square being attacked by an opponent piece.

Note that opponent pieces by themselves do not block the king since they can be captured. Once all reachable squares have been found within a genetically determined maximum distance (measured in king moves), the score is calculated by the following:

$$S = \frac{W_{friendly} \sum_{s \in R} Friend(s) + W_{opponent} \sum_{s \in R} Opponent(s)}{(|W_{friendly}| + |W_{opponent}|)\left(1 + \sum_{s \in R} Free(s)\right)}$$

where S is the resulting score, $\sum_{s \in R}$ is a sum over every square s on the board that is reachable ($\in R$) from the current king location, W_{friend} is the genetically determined

weight of squares blocked by friendly pieces, $W_{opponent}$ is the genetically determined weight of squares blocked by opponent attacks, $Friend(s)$ returns 1 if the square s is blocked by a friendly piece and 0 otherwise, $Opponent(s)$ returns 1 if the square s is attacked by an opposing piece and 0 otherwise, and $Free(s)$ returns 1 if the square s is reachable by the king and 0 otherwise. One is added to the denominator to prevent division by zero in the case where the king is in check and unable to move. The further division by the sum of the absolute values of the weights normalizes the score so that the result is approximately in the range $-1 \leq S \leq 1$.

Usually, the weight W_{friend} will be positive and $W_{opponent}$ will be negative. So, a king that is hemmed in by friendly pieces will receive a positive score while a king hemmed in by opposing attacks receives a negative score. In both cases, a greater number of reachable free squares brings the score closer to zero. A situation where the king is far from friendly protection and opponent's attacks is thus neutral.

The priority of this gene is non-negative so as to hasten the evolution of the friend/opponent weights to their proper signs.

4.3.8 King Protection Gene

This gene counts the squares that have access to the king by any valid piece movement and are unguarded by that king's other pieces. In other words, it measures how exposed the king is to hypothetical attacks. A higher score means a less exposed king.

4.3.9 Castling Possible Gene

This gene returns a positive score to indicate that castling is possible or has already happened. A higher score indicates that castling is closer to being a legal move due to intervening pieces being moved away, the king and rook remaining unmoved, and opposing pieces not attacking. The score can vary based on a genetically determined preference for kingside or queenside castling. Because of these preference values, the priority is non-negative.

4.3.10 Pawn Islands Gene

A pawn island is a group of pawns on neighboring files that are isolated from other pawns by empty files, and thus cannot help to defend pawns on other islands. This gene calculates the average number of pawns per island with the assumption that a greater number of islands with fewer pawns each is unwanted.

4.3.11 Stacked Pawns Gene

This gene counts the pawns that have a pawn of the same color ahead of them in the same file. The motion of these pawns is limited by the pawn in front. The score returned is the negative of the count since stacked pawns are generally considered to be disadvantageous.

4.3.12 Checkmate Material Gene

This gene returns a score of one if the scoring side has enough pieces to checkmate a lone king. Otherwise, it returns zero. This should help avoid piece trade-offs resulting in one side having a material advantage but unable to checkmate—king and knight vs. king and pawn, for example.

4.3.13 Null Gene

This gene does nothing. It has no effect on gameplay. The only purpose of this gene is to have a priority that mutates. In a way, this gene represents the null hypothesis: that evolution does nothing to tune the behavior of a chess-playing program. It will be interesting to see how the plots of this gene compare with others so that there is a contrast between genes with alleged evolutionary pressure and a gene with none.

4.4 Genome File Format

An example genome file is shown below. Each genome starts with the `ID:` line and ends with `END`. Each gene starts with `Name:` and ends with a blank line. Each component of a gene is a single line and specified by a name followed by a colon and the numerical value. Spacing within the line does not matter.

```
ID: 78551
Name: Piece Strength Gene
B: 106.646
K: 4.9872
N: 101.383
P: 3.57441
Q: 290.247
R: 142.083

Name: Look Ahead Gene
Game Length Uncertainty: 0.165787
Mean Game Length: 49.6397
Speculation: 1.82359

Name: Mutation Rate Gene
Mutation Rate: 3.52766

Name: Total Force Gene
Priority: 6508.77

Name: Freedom to Move Gene
Priority: 224.753

Name: Pawn Advancement Gene
Priority: 1133.36

Name: Passed Pawn Gene
Priority: 1517.21

Name: Opponent Pieces Targeted Gene
Priority: 232.564

Name: Sphere of Influence Gene
Illegal Square Score: 4.61644
King Target Factor: -1.0678
Legal Square Score: 8.33143
```

```
Priority: 1987.48

Name: King Confinement Gene
Friendly Block Score: -89.6045
Opponent Block Score: -9.85331
Priority: -192.247

Name: King Protection Gene
Priority: -119.064

Name: Castling Possible Gene
Kingside Preference: 2.78117
Priority: 179.767
Queenside Preference: 0.641198

Name: Stacked Pawns Gene
Priority: -318.024

Name: Pawn Islands Gene
Priority: -41.3929

Name: Checkmate Material Gene
Priority: 251.059

Name: Null Gene
Priority: -143.872

END
```

In genome files that are the output of gene pool runs (see Section 5) there will also be lines in the following format:

```
Still Alive: 2 : 759 849 872 896 899 901 902 903 # ... etc.
```

This line lists the ID numbers (after the second colon) of the AIs that are still alive within gene pool #2 (the number after the first colon).

5 The Gene Pool: On the Care and Feeding of Chess AIs

In each generation, the players in a gene pool are randomly matched up with each other to play a single game of chess. After the game, the two players mate to produce one offspring by picking each gene randomly from either parent with equal probability (recombination [7]). The offspring is then subject to a single extra mutation procedure wherein a number genes are individually mutated (that number being controlled by a dedicated gene, see Section 4.2.3). Finally, the offspring replaces the loser in the gene pool. If the game ended in a draw, the player to be replaced by the offspring is picked by coin flip. In the literature, this setup is classified under Evolutionary

Strategies [8], more specifically $(N/2+N)$-ES [9], where N is the population size after the chess games (half the initial population), $/2$ refers to two parents producing one offspring through recombination, $+$ means that the offspring compete with the parents (elitism), and the final N refers to how many offspring are generated to keep a constant population of $2N$, half of which are killed off every generation.

In this procedure, some the genes of losing players are passed on and are only slowly weeded out since a single game does not actually provide much information about the fitness of any gene with respect to game play. The removal of individuals destroys genetic information. By "information," I mean filtered genomes. If a gene (including small variations, given the continuous nature of these genotypes) is present is a large percentage of the species, then that gene must have been present in a great number of AIs that were victorious in their games through many generations and many different opponents. Genes found in losing organisms slowly fade away as their host organisms fail to survive long enough reproduce multiple times. On average, a gene from a losing AI will be present in 0.5 AIs (50% chance of being passed on) in the next generation; a gene from a winning AI will be present in 1.5 AIs (1.0 from the parent and another 0.5 from the 50% chance of being passed on) in the next generation.

One final means of preventing gene pool stagnation and preserving genetic diversity is the use of multiple gene pools. Each gene pool evolves separately for a long time, allowing each to genetically diverge. Then, every once in a long while (the time being user-specified), all of the AIs are shuffled between all of the gene pools. Thus, the best genes are further spread afield so that they can be tested against a wide range of opponents and combined with the best genes from other pools to create stronger offspring.

On Linux, the gene pool can be paused by pressing Ctrl-z. After the key press, the currently running games will finish and the results recorded before pausing. Pressing Ctrl-c exits the program immediately without waiting for current games to finish. To exit the program after the current round of games finish, pause first with Ctrl-z, then exit with Ctrl-c.

On Windows, pressing Ctrl-c exits the program after the current round of games is finished and recorded. Pressing Ctrl-c twice ends the program immediately without waiting for current games to finish.

5.1 Gene Pool Configuration File

A gene pool is configured with a text file that is reference in the program starting arguments (see Section 2). An explanatory example gene pool configuration file is presented below. All parameters are required to be in the file.

```
# Gene Pool Configuration (# indicate comments)

# The number of processors used will be the minimum
# of this number and half the gene pool population.
maximum simultaneous games = 8

# How many players in each pool
gene pool population = 16

# There are two possible values for reproduction type:
#  1. "sexual" : in which the winner and the loser create the offspring
```

```
#  2. "cloning" : in which the offspring is a mutated clone of the winner
reproduction type = sexual

# How many gene pools
gene pool count = 3

# How many mutations a newly created Genetic AI should undergo
# at the start of a gene pool run
initial mutations = 100

# Complete rounds (in which all pools play a set of games) in between
# swapping players between pools
pool swap interval = 1000

# Oscillating game time
#
# If "oscillating time" is "yes", the time for each game starts
# at the minimum, then goes up by the increment after each round
# of games. When it reaches the maximum, the increment is reversed
# and the time for each game goes down until it reaches the minimum.
# Then, the cycle starts again. If the "oscillating time" is "no",
# then the game time will stop and remain at the maximum time until
# the end of the run.
minimum game time = 5 # seconds
maximum game time = 30 # seconds
game time increment = 0.01 # seconds
oscillating time = yes

# The name of the file where the genomes will be recorded.
# Games will be recorded in a file with "_games.txt" appended
# to the name. If the file already exists from a previous run,
# then the gene pool procedue will pick up from the last
# stopping point.
gene pool file = pool.txt
```

5.2 Gene Pool Terminal Output

An example of typical output during a gene pool run is shown below. First, some general information about the pool is shown, then the results of the random matchups, and finally the new makeup of this gene pool in the aftermath of the games.

```
Gene pool ID: 0  Gene pool size: 16  Rounds since pool swaps: 2/1000
Games: 6000  White wins: 3402  Black wins: 2511  Draws: 87
Time: 16.65 sec   Gene pool file name: pool.txt

106514 vs 106495: Black (Time forfeiture)
106531 vs 106513: White (White mates)
106511 vs 106479: Black (Black mates)
106392 vs 106457: None (Insufficient material) --> 106392 dies
106516 vs 106497: None (Threefold repetition) --> 106516 dies
```

```
106532 vs 106530: White (White mates)
106529 vs 106433: White (White mates)
106533 vs 106534: White (White mates)

     ID   Wins  Draws
 106457      2      3
 106479      4      0 T
 106495      1      2
 106497      1      2
 106529      1      0
 106531      1      0
 106532      1      0
 106533      1      0
 106549      0      0
 106550      0      0
 106551      0      0
 106552      0      0
 106553      0      0
 106554      0      0
 106555      0      0
 106556      0      0
```

The (T) indicates that the AI is a copy that has been transferred from another pool.

6 Some Consistent Results (in rough order of discovery)

Here are a few results that are reliably reproduced in multiple simulations. In these runs, there were three gene pools with 16 players each (so the number of games equaled the number of processors on my computer, i.e., 8). Each game is played with 30 seconds per side for the entire game with no increment.

6.1 Piece values are rated in near-standard order.

In descending order of valuation by a Genetic_AI: Queen, Rook, Bishop and Knight nearly equally, and Pawn. As time goes on,there is a lot of variation, especially in the relative order of the Rook, Knight, and Bishop. But, the standard order is preserved for the most part, as can be seen in Figure 3.

6.1.1 Piece Strengths with the king

After assigning the king a strength entry in the Piece Strength Gene (see Section 4.2.1), the other pieces retained their relative values, but the king evolved a very negative value. See Figure 4.

My current guess is that the king entry is affected by the Opponent Pieces Targeted Gene and the Look-Ahead Gene—specifically the Capturing Speculation Constant. A large value for the Capturing Speculation Constant means that the board position that is actually evaluated has a smaller chance of having pieces threatened with capture. It is even less likely that the opponent's king is in check since this usually results in

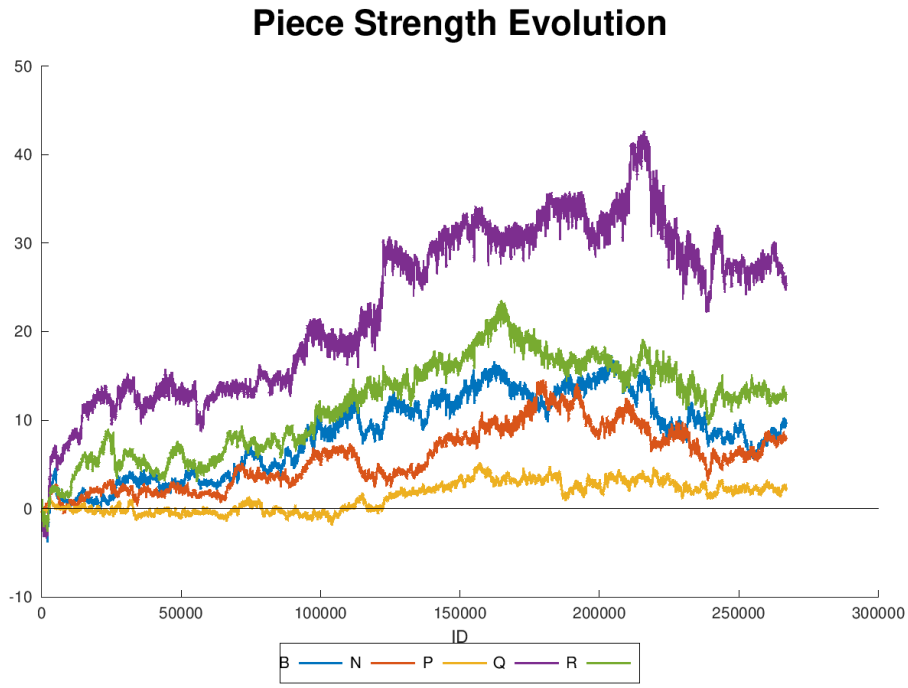

Figure 3: The evolution of the value of pieces to the AIs.

a reduced number of moves available, making recursion to the next move more likely. Thus, an evaluated board position with only the king in check is see as rather worthless unless offset by an attack on another piece that will probably result in capture. The king cannot be captured, so attacking it had better serve some other purpose. A similar evolution is seen in the Sphere of Influence Gene in that the King Target Factor is often negative.

This conclusion is corroborated by the King Target Bonus in the Opponent Pieces Targeted Gene (see Section 4.3.5), which always has a negative or near-zero value that causes a preference for piece arrangements that attack more of the board away from the king.

6.2 White has an advantage.

Of the games ending in checkmate, white wins about 10% more often than black. Figure 5 shows that the advantage is persistent through almost the whole of a long gene pool run. Wins by time are shared by black and white equally (see Figure 6).

6.3 The Total Force Gene and the Pawn Advancement Gene typically dominate.

That the Total Force Gene dominates is fairly predictable. Highly skilled human players will usually resign after the loss of a minor (bishop or knight) piece without compensation.

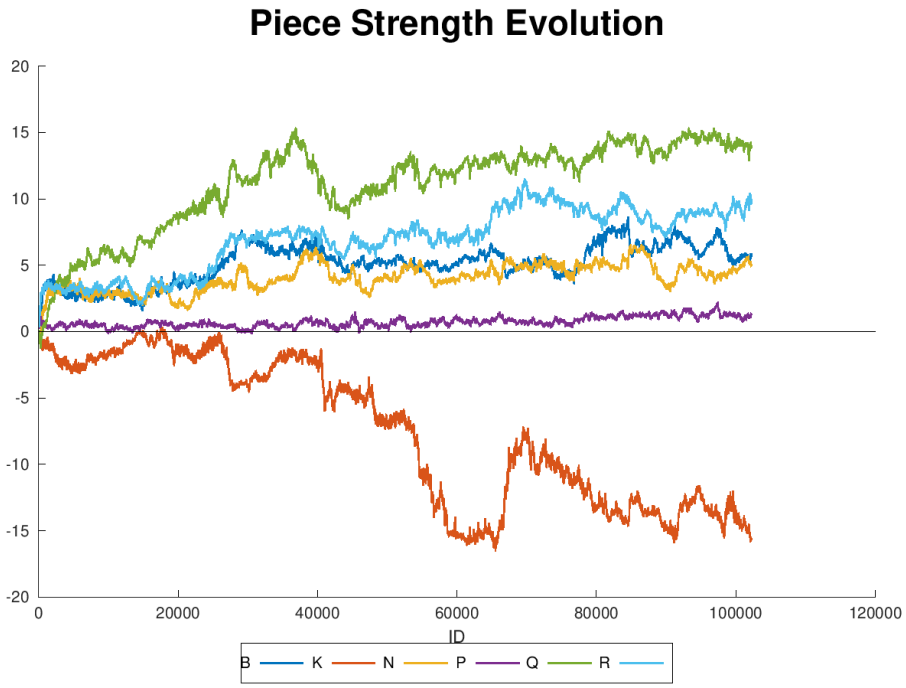

Figure 4: The evolution of piece strengths after adding an entry for the king.

The Pawn Advancement Gene usually gains higher priority first, probably because it is the simplest gene that makes an immediate difference in the game. Push the pawns forward both threatens the opponent's pieces with low-risk attacks and increases the chances of promotion.

6.4 The Queen is the most popular piece for promotion.

Even when the Piece Strength Gene has not been tuned at all, the queen is the overwhelming favorite, followed by the rook, then bishop, and finally the knight. In human games, only the queen and knight are chosen since they have different move patterns. If you need at least a rook or bishop, you might as well take a queen since that piece provides both. Only the knight provides a viable alternative (usually to avoid a stalemate if the queen was chosen).

As an example, the following is a count of all promotions in a gene pool run after more than 300,000 games.

Piece	Promotions
Bishop	2232
Knight	1648
Rook	7215
Queen	146664

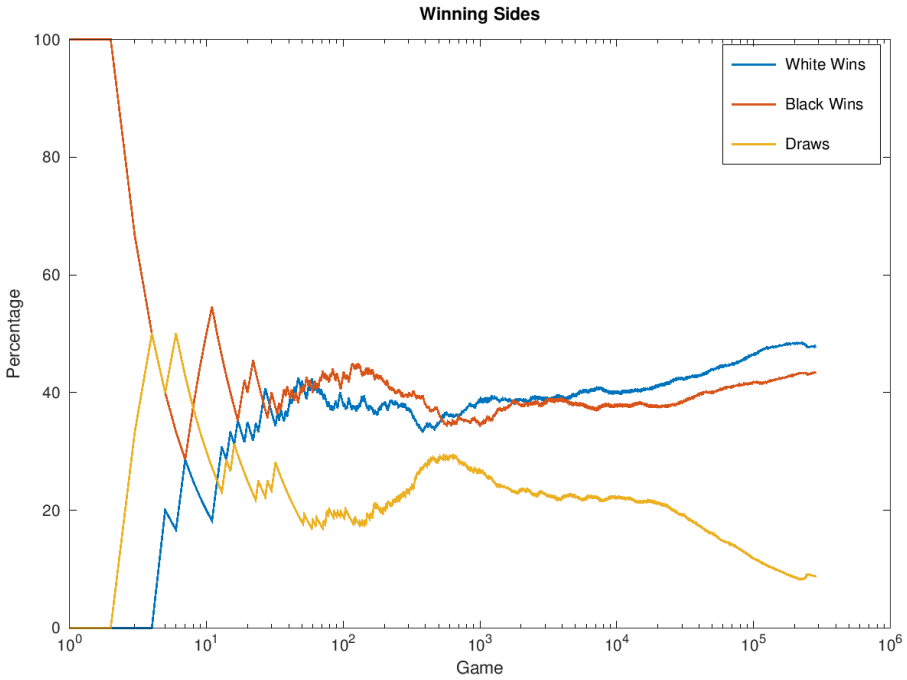

Figure 5: The percentage of games won by white, black, or neither over the course of a gene pool run. The advantage that white has over black seems persistent. Also, as the AIs evolve, the rate of draws decreases, presumably because they evolve a genome that can tell the difference between a good and bad position.

6.5 Threefold repetition is the most common draw.

Most games with human players end in a draw when neither side can force an advantage. This happens when one side can block a crucial move (e.g., a pawn promotion) and the other side cannot remove this block. Since the blocking player does not have a reason to move, he can just repeat moves to maintain the block. This would lead to threefold repetition if most players did not verbally draw the game beforehand. Since these Genetic AI players don't offer or accept draws, they play out all the repetitions, resulting in what is seen in Figure 6.

6.6 The Look Ahead Gene is a late bloomer.

The plot in Figure 6 shows the counts of how games end. It seems that the Look Ahead Gene does not experience significant evolutionary pressure until the board-scoring genes have been tuned to a semi-decent state. My hypothesis is that if the board-scoring function is not able to tell a good position from bad, then looking ahead only increases the risk of losing by time forfeiture with no benefit. However, when the board-scoring genes are in a decent state, the increase in look ahead is very quick. This leads to a rapid increase in the rate of deaths by running out of time, but apparently this is worth the risk as those who are more conservative with their time lose to those who see farther ahead.

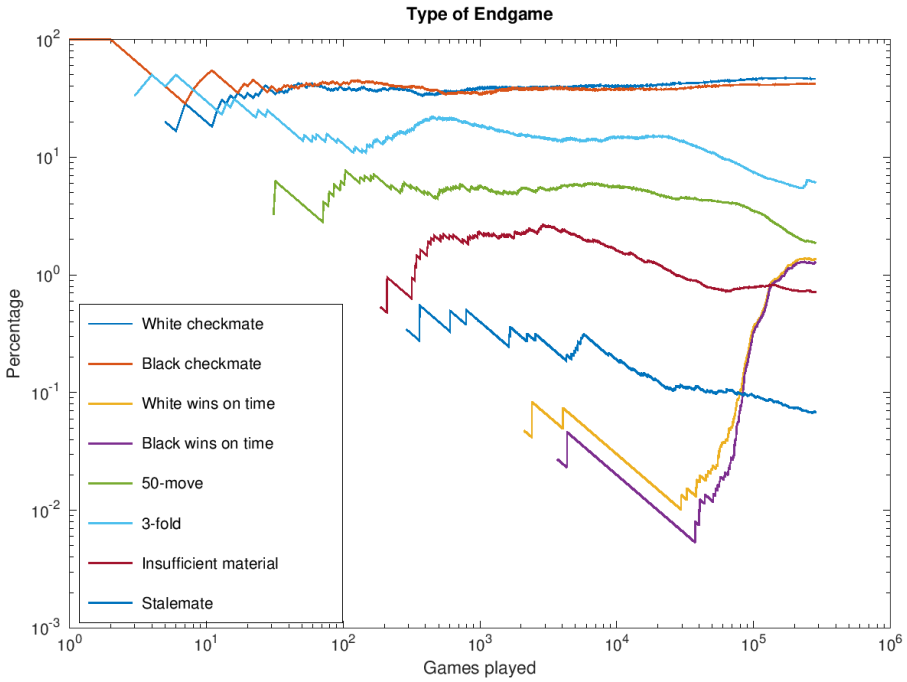

Figure 6: A log-log scale of the percentage of games with various endings. Note that the time forfeits don't get going until about 30,000 games in.

6.7 The Sphere of Influence Gene typically counts legal moves as having lower value than other moves.

This was unexpected. I thought that the legal moves would count more since they present a greater threat to the opponent. You cannot capture your opponent's Queen if your own King is in check. Perhaps Genetic_AIs find this gene more useful as a forward-looking view of the game. Or, the Opponent Pieces Targeted Gene relieves this gene of having to consider legal moves.

7 Programming Quirks

Since this software is still in heavy development, the following sections are likely to change. As such, here is a summary of aspects of the code that might seem weird upon first—and possibly every other—reading. Specifically, if this is modern C++, why are there pointers everywhere?

Though the operators new and delete appear nowhere in the code and smart pointers are used, raw pointers appear everywhere in the code. The driving reason for using pointers is polymorphic storage. Classes representing moves are derived from a Move so that they can override various legality and side-effect methods. Storing different types of moves requires storing Move*s. The same goes for Gene storage in the Genome class.

Originally, instances of the Board class contained an array of std::shared_ptrs so that copying a board (for when computer players needed to think ahead) did not present difficulties in managing the piece resources (namely, when to call delete). While easy to code, the bookkeeping required by shared pointer copying slowed down the speed of the AIs traversal of the game tree. Then, each type and color of piece was stored as a static instance in the Board class and pointers to these instances are used as markers for board positions. While this does mean that Piece classes are treated like singletons (or, rather, doubletons, since there are black and white instances), this was simpler and faster. The pieces in this code do not have any modifiable internal state after creation, so any two white pawns—even ones taken from different boards—should be completely indistinguishable. In the most recent version, every piece is simply an unsigned integer where individual bits represent the color and type of a piece. This is the simplest and fastest way to represent an entity with no state other than its identity. The integer representation is used to access arrays of pointers to Moves.

If a function or method returns a pointer, the caller of that function is not responsible for deleting the data pointed to by the pointer (and, in fact, should not). Most public methods and functions take reference arguments to prevent problems with null pointers. Where functions do take pointer arguments (such as Board::print_game_record() for the player instances), that argument is optional parameter that can take a nullptr to indicate no data.

References

[1] G.S. Hornby, A. Globus, D.S. Linden, J.D. Lohn, "Automated Antenna Design with Evolutionary Algorithms." AIAA

[2] W.H. Miner, Jr., P.M. Valanju, S.P. Hirshman, A. Brooks, N. Pomphrey, "Use of a genetic algorithm for compact stellarator coil design." IAEA Nuclear Fusion, Vol. 41, No. 9. 1185–1195

[3] https://en.wikipedia.org/wiki/Log-normal_distribution

[4] https://chess.stackexchange.com/a/4899/5819

[5] https://www.chessprogramming.org/Quiescence_Search

[6] http://www.thechessdrum.net/PGN_Reference.txt Section 16.1

[7] https://en.wikipedia.org/wiki/Genetic_recombination

[8] https://en.wikipedia.org/wiki/Evolution_strategy

[9] http://ls11-www.cs.tu-dortmund.de/~beyer/EA-glossary/node41.html