Genetic Chess

Mark Harrison

November 26, 2017

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 release and debug 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_gcc_Makefile.py or python create_clang_Makefile.py to regenerate the Makefile. 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 create both release and debug builds, make release or make debug to only make one sort of build, or make clean to delete all build items (object files and executables).

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

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.

genetic_chess <-player> <-player> Starts a local game played in the terminal with an ASCII art board. The first parameter is the white player, the second is black. The <-player> argument is replaced with one of the following:

- **-human:** a human player entering moves on the command line and seeing the results on a text-based drawing of the board. Moves are specified in standard algebraic notation (SAN) or in coordinates that indicate the starting and ending square.

-genetic: 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 there are two possibilities:

- If the genome file is the output of a gene pool run, then the AI with the smallest ID that was still alive is chosen. As the oldest surviving AI, it is the nearly the most evolved while still having been tested.
- Otherwise, the last genome in the file is loaded (presumably this one is the most evolved). If there is another file containing records of games played during a gene pool run that has the same name as the genome file name with an extra suffix "_games.txt", then the last genome with at least 3 wins will be selected.

-random: an AI player that chooses a random legal move at each turn.

Genetic Chess can communicate with GUI chess programs through the Chess Engine Communication Protocol, including xboard, PyChess, Cute Chess, and others. When using Genetic Chess this way, only specify the arguments for a single player (-genetic or -random). The program will then wait for communication from the GUI.

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.

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 score that the player whose turn it is guaranteed once the game reaches the current state 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.

[1] The little ingrates!

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

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.

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.

Formerly, one result of a mutation was to deactivate a gene entirely. Gene deactivation was thought to be a strategy for exiting local peaks in fitness since a gene could still mutate when deactivated. Since it felt no evolutionary pressure, the deactivated gene was free to random walk somewhere else, perhaps to a more advantageous state that would be discovered upon reactivation. However, it was found that, once deactivated, a gene would only very rarely become reactivated. This was for two reasons. First, a random walk is a very slow way to reach a better genome, especially with no evolutionary pressure. Second, and more importantly, a deactivated gene uses less time when evaluating a board or doing some other calculation. This means that Genetic AIs with more deactivated genes have an advantage in being able to evaluate more positions and see farther ahead in the game. This is a powerful incentive to keep genes deactivated, often

leading to AIs that had most of their genome deactivated aside from the Total Force Gene. This would lead to gene pools where games only ended in draws, stagnating the pool. Now, all genes are always active.

4.1 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.

4.1.1 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 4.2.1) reference this one for their own evaluation purposes.

The number associated with each piece is not scaled to any other piece, but before a piece strength is returned, it is scaled so that that total value of all the pieces at the start of the game (8 pawns, 2 rooks, 2 knights, 2 bishops, and 1 queen) equals 1. This prevents mutations in this gene from changing the priorities of other genes that reference this one. In other words, if a series of mutations doubles the strength of all pieces, the Total Force Gene will not subsequently return double the score for the same board state. Only mutations that change piece strengths with respect to other pieces are effective. The king is not assigned a strength because it is always on the board, so it cannot affect the move chosen.

4.1.2 Look Ahead Gene

This gene determines all aspects of the Genetic AIs time management. There are three 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 probabilistically looking further ahead when not enough time is allocated.

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 $\ln N$, 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 function available in native C++.

$$\sum_{i=n+1}^{\infty} P(i) \times i \approx \int_{n+1}^{\infty} P(t)t dt = \frac{1}{S\sqrt{2\pi}} \int_{n+1}^{\infty} e^{-\frac{1}{2}\left(\frac{\ln t - M}{S}\right)^2} dt$$

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

and

$$\sum_{i=n+1}^{\infty} P(i) \approx \int_{n+1}^{\infty} P(t) dt = \frac{1}{S\sqrt{2\pi}} \int_{n+1}^{\infty} \frac{1}{t} e^{-\frac{1}{2}\left(\frac{\ln t - M}{S}\right)^2} dt$$

$$= \frac{1}{2} \left[1 + \operatorname{erf}\left(\frac{M - \ln n}{S\sqrt{2}}\right)\right].$$

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

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

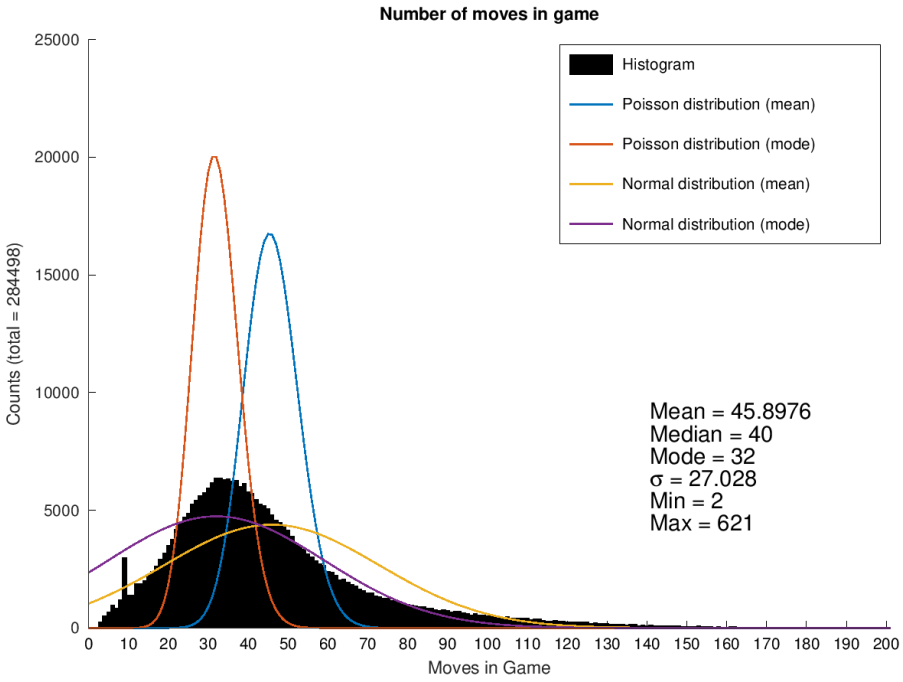

Figure 1: First attempts at determining the distribution of moves in chess games. The black histogram shows the distribution of the number of moves in a game (with one move consisting of both a black and white play). The (mean) and (mode) designations in the legend indicate which statistic was used as 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 (see Section 7.2.5 for details).

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. If a move examination is cut off early for whatever reason (e.g., a game-ending move

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

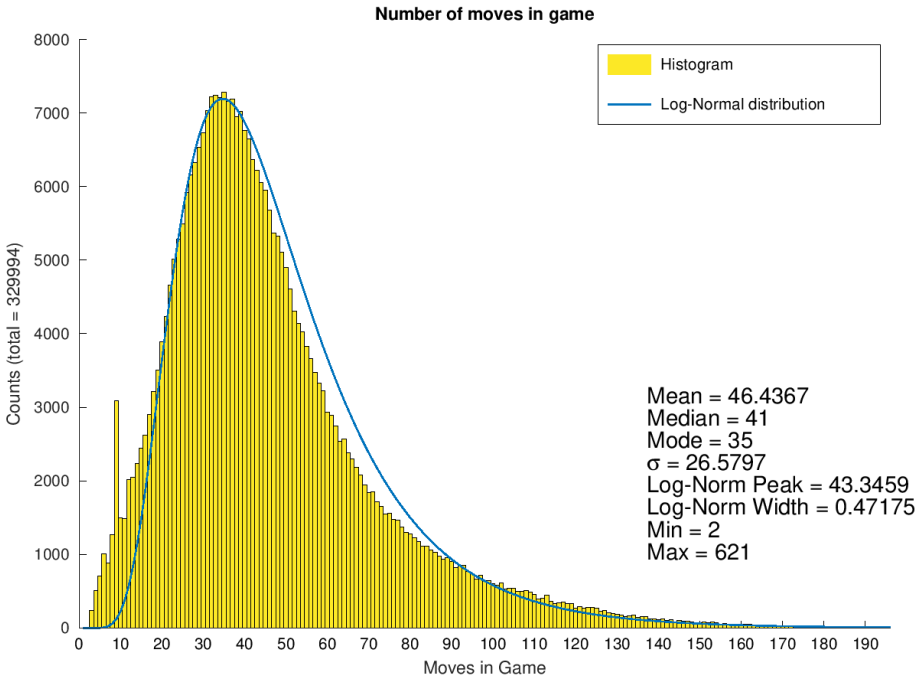

Figure 2: The distribution of the number of moves in a game with a log-normal fit. The Log-Norm parameters listed are M for the Peak parameter and S for the Width parameter. To obtain a better fit, only games longer than 15 moves were considered, as shorter games were primarily by ill-adapted AIs (especially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

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.

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

where T is the time needed to look ahead on this move, N_{legal} is the number of legal moves, and v is the rate of position examinations. This speed is determined by counting the number of positions examined and dividing by the amount of time taken while choosing the previous move. If this time is less than the time allocated for the move, the algorithm looks ahead (the function recurses) to examine the moves the opponent can make in response. Otherwise, if there is

any time at all, the AI looks ahead with a probability given by

$$P(t,T;k) = \left(\frac{t}{T}\right)^{\frac{1-k}{k}}$$

where t is the time available for a move, T is the time needed to look ahead, and k is a genetically determined constant, and the third parameter in the list above. The parameter k, herein referred to as the "speculation constant," varies from 0 to 1, resulting in an exponent that varies from ∞ to 0. When $k = 0.5$, the exponent is equal to one, and the probability of look ahead is proportional to the time remaining. If $k < 0.5$, then the exponent is greater than one, resulting in lower probabilities since $x^p < x$ for $0 < x < 1$ and $p > 1$. If $k > 0.5$, then the exponent is less than one, resulting in higher probabilities. Additionally, transforming $k \to 1 - k$ transforms the exponent into its inverse, giving a nice symmetry to the parameter in that k and $1 - k$ are aggressive-conservative mirror images. We can see the effect of changing k by calculating the average probability of recursion when $0 \leq t \leq T$ and t is assumed to be uniformly distributed in $[0, T]$:

$$\bar{P} = \int_0^T \left(\frac{t}{T}\right)^{\frac{1-k}{k}} \frac{dt}{T} = \int_0^1 u^{\frac{1-k}{k}} du = \frac{1}{\frac{1-k}{k} + 1} = k.$$

Recently, the speculation constant has been divided into two constants: one for board positions containing a capturing move, and all others. Initial results indicate that the capturing speculation constant tends to be much larger (between 0.4 and 0.5) than the non-capturing counterpart (usually less than 0.2). This makes sense since board positions with capturing moves tend to be move volatile in who has the advantage. Capturing a piece is only good if it doesn't result in a greater loss in subsequent moves. This is the problem that is usually solved in other chess engines by a quiescence search, in which all interesting[3] capturing moves are performed on a board before it is scored, thereby only scoring stable, "quiescent" board states [5].

4.1.3 Branch Pruning Gene (deleted)

This gene would prevent the game tree search from examining the moves following a certain move if the board state that immediately resulted from that move was of a sufficiently lower score than the present board state. This gene was supposed to save time by skipping in-depth examinations of moves that were obviously immediately bad. This gene was deleted because it seemed to slow down the evolution of other genes by harshly punishing large swings in board value. This would cause genes that increased in priority to trip this gene, preventing look-ahead, and thus good play.

[3] Meaning, captures that aren't immediately losing.

4.2 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

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

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. In general, the scores are scaled so that a typical board state gets a score of 1 from any gene. 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). The Total Force Gene returns 1 for the pieces at the start of the game, so promoting a pawn can result in a score of more than one.

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

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

4.2.1 Total Force Gene

This gene sums the strength (according to the Piece Strength Gene) of all the player's pieces on the board.

A unique aspect of this gene is that it is the only one whose priority is constrained to be non-negative. This was done to avoid a genome where the Total Force Gene and the Opponent Pieces Targeted Gene have oppositely signed priorities. This would cause mixed pressure on the Piece Strength Gene values, interfering with its evolution. This does not limit the expressiveness of this gene, since the Piece Strength Gene and the Opponent Pieces Targeted Gene are free to take on negative values. Any genome with a negative priority on the Total Force Gene is equivalent to another genome with the signs of the three mentioned genes' priorities negated.

4.2.2 Freedom to Move Gene

This gene counts the number of legal moves available in the current position.

4.2.3 Pawn Advancement Gene

This gene measures the progress of all pawns towards the opposite side of the board. Extra points are awarded to board states that have pawn promotions as past moves in order to prevent promotions from being penalized when such moves remove pawns from the board.

4.2.4 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.

4.2.5 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.

4.2.6 King Confinement Gene

This gene counts the squares the king can reach given unlimited legal moves. Squares that take a larger number of moves to reach are given lower scores. Kings that are hemmed in by squares occupied by friendly pieces and squares attacked by the opponent result in a low score from this gene. This is a measure of how much room the king has to maneuver and escape. It is somewhat in opposition to the King Protection Gene.

4.2.7 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.

4.2.8 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 score can vary based on a genetically determined preference for kingside or queenside castling.

4.3 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.

```
ID: 106768

Name: Piece Strength Gene
B: 9.5685
N: 5.55439
P: -0.285538
```

```
Q: 23.8438
R: 13.3138

Name: Look Ahead Gene
Game Length Uncertainty: 0.394859
Mean Game Length: 33.0766
Speculation Constant: 0.0142434

Name: Total Force Gene
Priority: 400.656

Name: Freedom to Move Gene
Priority: 30.5008

Name: Pawn Advancement Gene
Priority: 138.87

Name: Opponent Pieces Targeted Gene
Priority: 104.614

Name: Sphere of Influence Gene
King Target Factor: 0.260099
Legal Bonus: 0.705431
Priority: 220.958

Name: King Confinement Gene
Priority: -13.7753

Name: King Protection Gene
Priority: -0.469023

Name: Castling Possible Gene
Kingside Preference: 0.390153
Priority: -18.3223

END
```

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. If the game ends with a winner, whether through checkmate or time violation, then the two players mate to produce an offspring by picking each gene randomly from either parent with

equal probability. The offspring is then subject to a a single extra mutation procedure wherein two genes on average are individually mutated. Finally, the offspring replaces the loser in the gene pool. This way, some the genes of losing players are passed on and 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.

If the game ends in a draw, then one of two things happens. With high probability–currently 95%–the players are left as they are and will participate in the next generation. The other possibility is that one of the players is randomly picked to mate with either a randomly generated AI, or a randomly chosen past AI, even long dead ones. The chosen player is then replaced by the offspring. This happens with low probability because it 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 reproduce. However, it is necessary to keep a pool from stagnating due to never-ending drawn games. Bringing in a randomly generated AI injects new genetic information into the pool (though with low probability of high-quality information). Bringing back dead AIs injects good information from the past and should help to keep a gene pool from becoming trapped in a self-reinforcing, pathological playing style that only works against similar players. In effect, both of these strategies are meant to kick a gene pool off of a local peak of genetic fitness.

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), the best player from each pool is transferred to the next pool over. Thus, the best genes are further spread afield so that they can be tested against a wide range of opponents. A useful measure of a pool's strength is how long its best player survives when it enters a new pool.

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 example gene pool configuration file is presented below.

```
# 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

# How many gene pools
gene pool count = 3

# Probability of killing a player after a draw
draw kill probability = 0.05

# Games in between swapping players between pools
pool swap interval = 1000

# Oscillating game time
#
# 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.
minimum game time = 30 # seconds
maximum game time = 120 # seconds
game time increment = 0 # seconds

# 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.
gene pool file = pool.txt
```

5.2 Gene Pool 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, thirdly the new makeup of this gene pool in the aftermath of the games, and finally the IDs of the AIs with the most ever wins and the most number of games survived.

```
Gene pool ID: 0  Gene pool size: 16  New blood introduced: 6 (*)
Games: 6000  White wins: 3402  Black wins: 2511  Draws: 87
Time: 16.65 sec   Gene pool file name: pool.txt

106514 vs 106495: None (Threefold repetition)
106531 vs 106513: White (White mates)
106511 vs 106479: Black (Black mates)
106392 vs 106457: None (Insufficient material) 106392
mates with random / 106392 dies
106516 vs 106497: None (Threefold repetition)
```

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

     ID   Wins  Streak  Draws  Streak
 106457      2       0      3       2
 106479      4       4      0       0 T
 106495      1       0      2       1
 106497      1       0      2       2
 106514      0       0      2       2
 106516      0       0      2       2
 106529      1       1      0       0
 106531      1       1      0       0
 106532      1       1      0       0
 106533      1       1      0       0
 106548      0       0      0       0
 106549      0       0      0       0
 106551      0       0      0       0 *
 106552      0       0      0       0
 106553      0       0      0       0
 106554      0       0      0       0

Most wins:     16 by ID 96392
Longest lived: 27 by ID 96392
```

The `Streak` column indicates the current number of consecutive wins or draws a player has attained in the last few games. In the example above, Player 106479 is on a four-game winning streak since it was born (including victories both in this pool and in pool 2 where it was born). Player 106457 has drawn its last two games after its last win.

The asterisk (`*`) indicates an offspring of the result of a drawn match–in this case, from 106392 vs 106457, with the new AI replacing 106392. The (`T`) indicates that it is the best AI from another gene pool that has been copied to this 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.

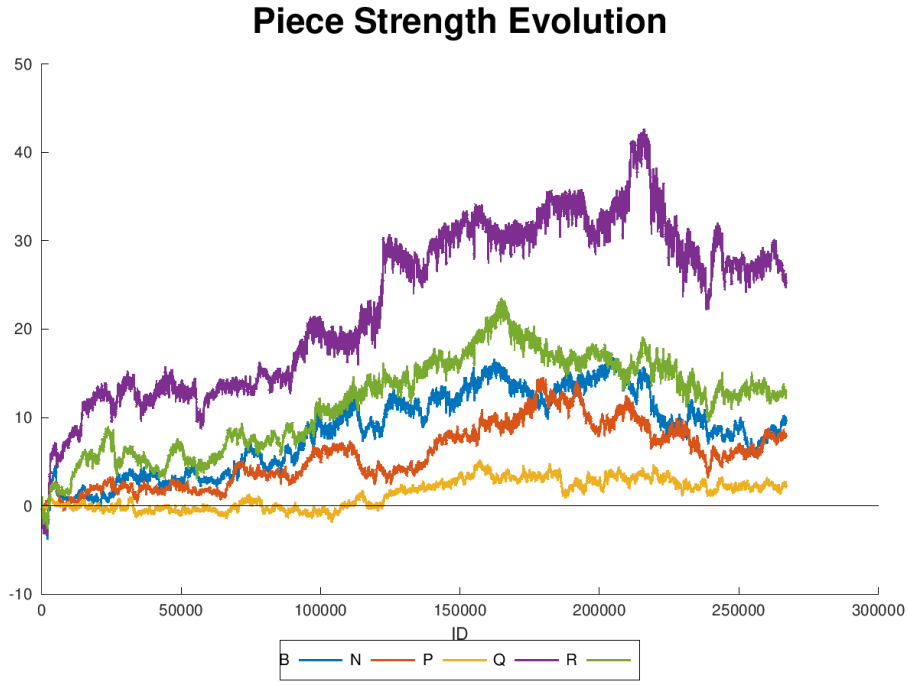

Figure 3: The evolution of the value of pieces to the AIs. The small bump near ID #120,000 in the strength of a pawn that caused the Pawn Advancement Gene crash seen in Figure 5

6.2 White has an advantage.

Of the games ending in checkmate, white wins about 10% more often than black. Figure 4 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.

The Pawn Advancement Gene usually gains higher priority first, probably because it is the simplest gene that makes an immediate difference in the game.

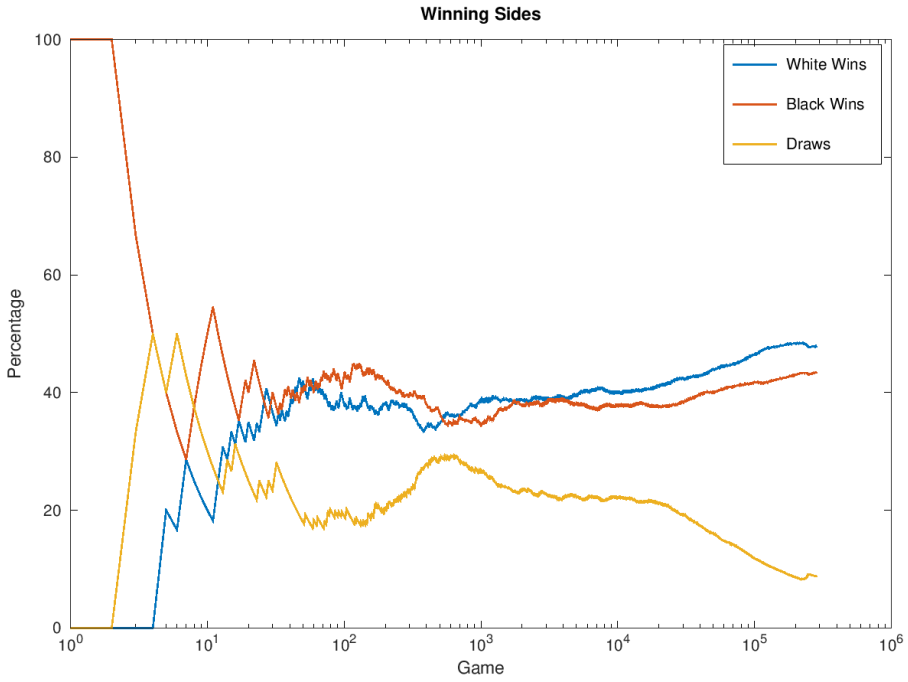

Figure 4: The percentage of games won by white, black, or neither over the course of a gene pool run. The advantage that white has over black seems persistent. Also, as the AIs evolve, the rate of draws decreases, presumably because they evolve a genome that can tell the difference between a good and bad position.

Push the pawns forward both threatens the opponent's pieces with low-risk attacks and increases the chances of promotion.

6.3.1 Late-Breaking News: The Pawn Advancement Gene is a temporary substitute for a valid value of the pawn in the Piece Strength Gene.

Near the birth of the Genetic AI with ID #120,000, the Pawn Advancement Gene suddenly died off, with nearly all AIs in the gene pool deactivating the gene simultaneously (see Figure 5).

The only correlation with another gene near that time is in the Piece Strength Gene (see Figure 3). The change is small, but in the comparison plot shown in Figure 5, one can see that the pawn strength has a small jump in value near ID #120,000. This is the first time that all of the AIs in the gene pools have strictly positive values for the pawn. Prior to this, the value of a pawn was zero on average. When mutations would push the pawn value higher, it would be weighted too much compared to other pieces, leading to losses and

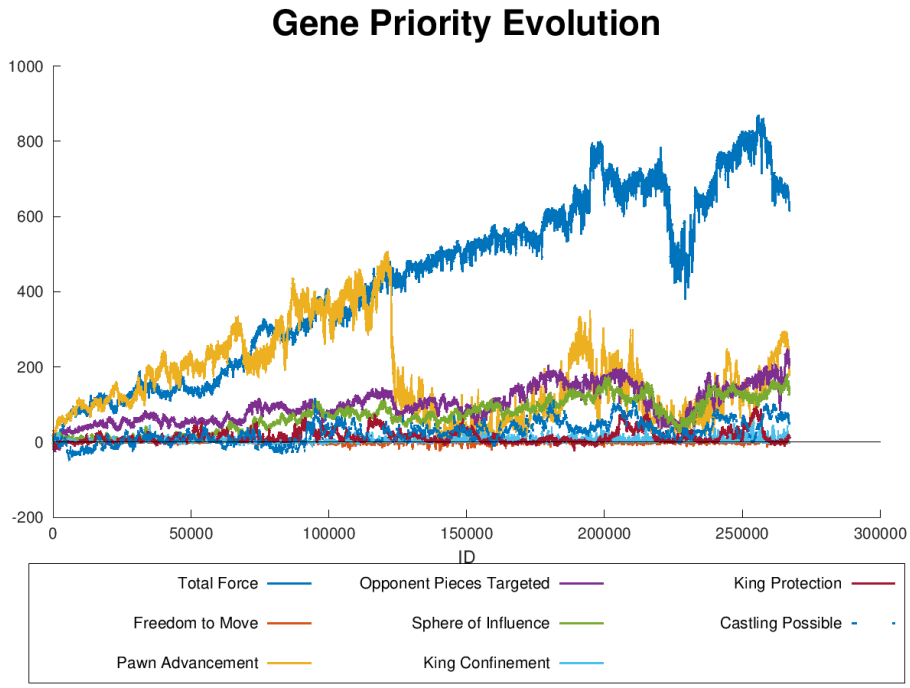

Figure 5: The sudden crash of the Pawn Advancement Gene

a lower value for the remaining population. It was only when the other pieces reached a large enough value that the pawn value could be properly weighted and sustained. Until this happened, the Pawn Advancement Gene evolved a large value in step with the Total Force Gene since pawns are not actually valueless in real games. Until the Piece Strength Gene transition occurred, the Pawn Advanment Gene served as a proxy for the value of a pawn that was more easily tuned in comparison with the Total Force Gene and Piece Strength Gene.

This pattern, in which a gene with a lower proper value or priority needs to wait for properly higher-priority genes to evolve high enough priorities before tuning their own priorities, is common and is discussed more the Section 6.8.

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 gaes.

Piece	Promotions
Bishop	2232
Knight	1648
Rook	7215
Queen	146664

6.5 Threefold repetition is the most common stalemate.

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.

6.7 The Sphere of Influence Gene typically counts legal moves as just as valuable as any other move.

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.

6.7.1 Late-Breaking News: Update after deleting the Branch Pruning Gene

The Sphere of Influence Gene now counts legal moves as 50% more valuable than illegal moves.

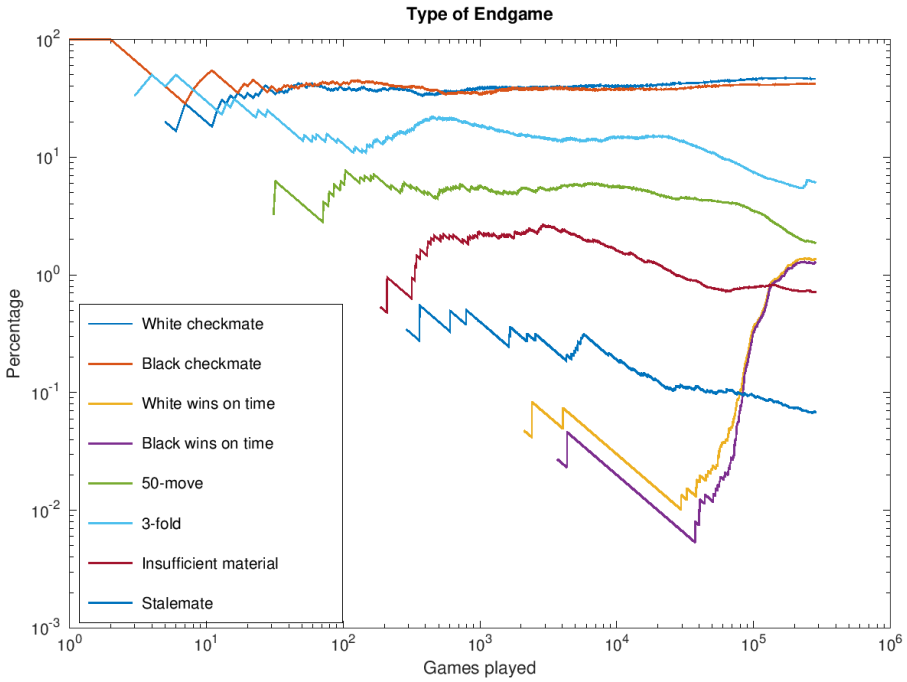

Figure 6: A log-log scale of the percentage of games with various endings. Note that the time forfeits don't get going until about 30,000 games in.

6.8 Genes with properly lower priorities have to wait for higher priority genes to evolve their higher priority before evolving themselves.

Some genes are more important than others. The Total Force Gene dominates because it is nearly always more difficult to win when behind in pieces. So, if another gene evolves too high a priority, it hinders game play and will soon be killed off. This means that, because the rate of evolution is the same for all gene priorities, that genes with a lower proper priority[4] need to wait for those genes with higher proper priority to evolve a large enough value. Before then, it is too likely for a random mutation to push a lower priority too high and get a gene killed off.

The same goes for the Piece Strength Gene with respect to the value of each type of piece. Note in Figure 3 that the values rise and fall with each other.

[4]By proper priority, I mean the as yet unknown relative priority compared with other genes that leads to winning game play. The proper priorities are what this program should be evolving towards.

7 Programmer's Reference

7.1 Overview

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 in the first place is the need for polymorphic storage. Classes representing pieces are derived from a Piece class and need to be stored within Board class instances (namely, in a std::array<Piece*, 64> where the position in the array corresponds to a position on the board). Classes representing moves are either instances of the Move class or derivative classes, and each type of piece stores a list of potentially legal moves.

If a function or method returns a pointer, the caller of that function is not responsible for handling the data pointed to by the pointer. 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.

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. Now, each type and color of piece is 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), I believe this is the simplest implementation. The pieces in this code do not have any modifiable internal state after creation, so a white pawn taken from different boards should be completely indistinguishable.

Inside each derived Piece class, the legal moves for that piece are stored in two locations:

1. A list of std::unique_ptrs to handle resource reclamation,

2. A structure of raw pointers that makes it easy to query the potentially legal moves of that piece starting from a given square ("What are the legal moves of a knight on a3?").

Since there is only one instance of each piece (white rook, black queen, etc.), a move obtained from a piece is valid for every other board with a piece on the same square. Some caution is required, though, since the legality of a move is only check during Board::legal_moves(). See Section 7.2.7 for details.

Finally, and most simply, the genome of a Genetic_AI is stored as a list of std::unique_ptr<Gene>, since each type of gene (Section 4) is derived from the Gene base class.

7.2 Board

7.2.1 ctor()

Constructs a board in the starting state of a standard chess game.

7.2.2 ctor(const std::string& fen)

Construct a board from a string containing a board state in Forsyth-Edwards Notation.

7.2.3 legal_moves()

Returns a std::vector of Moves representing all the legal moves of the current board state. Calling this method only generates the list on the first call after a board state change. Every time afterwards it returns a cached copy of the list, so there is no penalty for calling it method multiple times.

7.2.4 other_moves()

Returns a list of moves that are not currently legal but, if they were, would have the moved piece land on a square inside the board. For example, at the beginning of a standard game, the white rook on a1 has no legal moves, as it is blocked in every direction. But, Ra4 would be in the list of other moves, since a4 is on the board and would be legally reachable if the pawn on a2 was not present. Moves to the left or backwards will not be listed, since these moves land outside the board.

7.2.5 submit_move(const Move& move)

This method causes the state of the Board to be changed by enacting the move represented by the argument. Only copies of a move returned from legal_moves() will be accepted. Otherwise, an assertion will fail in debug builds or undefined behavior will result in release builds. The method returns a Game_Result struct that indicates whether the game has ended and the reason for the ending. Formerly, the end of a game was signaled by throwing an exception derived from Game_Ending_Exception. However, this was found dramatically slow down the game tree search due to the cost of throwing and catching multiple exceptions.

7.2.6 view_piece_on_square(char file, int rank)

Returns a pointer to a const Piece* representing the piece on the square identified by the arguments. An empty square is represented by a nullptr.

7.2.7 Important Notes

The Moves returned by the methods Board::get_move() and Board::legal_moves() are the only moves guaranteed to be valid for a given Board instance, and only for the unmodified originating board or an unmodified copy. Calling Board::submit_move() with a Move from a different source will have three possible results:

1. The move happens to be legal, in which case the game continues.
2. The debugging version of the program will fail an assert.
3. The release version of the program will enter an invalid game state with no warning.

The legality of moves are only checked by the Board methods that generate legal Move instances (the previously mentioned Board::get_move() and Board::legal_moves()), not on submission.

7.3 Piece

7.4 Move

7.5 Clock

7.6 Player

7.7 Gene

7.8 Genome

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] http://chessprogramming.wikispaces.com/Quiescence+Search