Chess Engine – Project in Evolutionary Algorithms

by Guy Perlberg

and Lev Poliak

Table of Content

Introduction

Chess is a two-player strategy game played on a gameboard, which consists of 64 squares arranged in an 8x8 grid. Each player starts with 16 pieces:

one king – can move in any direction one square

one queen – can move in any direction any amount of squares

two rooks – can move vertically and horizontally

two knights – can choose to move in “L” shape, two moves diagonally/horizontally and one move with the complement move.

two bishops – can move diagonally

and eight pawns – can only move forwards, and capture pieces that are diagonally in front of the pawn (one step ‘above’ the pawn and one step to either side).

The goal of the game is to checkmate the opponent's king, which means the king is in a position to be captured (“in check") and there is no way to move the king out of capture (mate). The game is won by the player who checkmates the opponent's king, lost if the player's own king is checkmated, or drawn if either there is a streak of pieces that move back and forth from both sides for 3 straight moves, or if the only pieces remain at the board are kings.

There are three stages in chess:

1. The opening is the first part of the game, where the pieces are developed and positioned on the board. It typically lasts for the first 10-20 moves and the goal is to control the center of the board and develop your pieces to their best squares.
2. The middlegame is the most complex and dynamic stage of the game, where the pieces are mobilized for attack and defense. It typically lasts for the next 20-40 moves and the goal is to create threats and opportunities to gain an advantage over your opponent.

3. The endgame is the final stage of the game, where there are only a few pieces left on the board. It typically lasts for the remaining moves of the game and the goal is to checkmate the opponent's king or force a resignation.

A standard way of describing a chess board is with a FEN – short for 'Forsyth–Edwards Notation'. For example, take the FEN of the starting board:

rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1

The white-colored pieces are annotated with a capital letter as opposed to black's pieces in small letters.

- The first argument is the distribution of all the pieces on the board, starting from the last square – h8 – there resides the black rook, and ends on a1 where the white rook stands. '/' appears when the row is lowered, note that there are 7 occurrences of '/'. An empty square will be announced by a number, and in this example, from the third row all the way to the sixth, there are no pieces at all.

- The second argument is the first letter of the color that is given the turn, in this case it's white.

- The third argument relates to 'castling' – when the king and a rook from one side have a path to each other and none made a move from the start, the king can be moved all the way one step near the rook, and the rook will switch spots with the near square of the king in the opposite direction. Only one castling is allowed for each color and the letters stands for 'king side' – castling with the rook closer to the king, or 'queen side'.

- The fourth argument is whether there are any 'en passant' squares - a square over which a pawn has just passed while moving two squares; it is given in algebraic notation. If there is no en passant target square, this field uses the character "-".

- The fifth argument is a counter of the number of moves since the last capture or pawn advance, in the case it is over 75 a draw is called.

- The sixth argument is the count of full moves since the start of the game, including the current move to be played.

A chess engine is a program designed to play the game of chess. It is typically integrated into a chess GUI, which allows a user to play against the engine, analyse positions, and access a variety of chess-related information. The engine uses a chess-specific algorithm to analyse the position on the board and make decisions in order to find the best move to play. The strength of a chess engine is often measured by its ELO rating, which is a rating system used in chess to indicate a player's relative skill level. The most powerful chess engines today swiftly defeat even the strongest human players.

One of the chess engines to note is ‘Stockfish’ - a free and open-source chess engine. It is considered one of the strongest chess engines in the world alongside the recent trending ‘Mittens’, and is often used by professional chess players to analyse their games, prepare for tournaments. It is also used by chess-playing websites such as Chess.com.

Evolutionary programming is a type of artificial intelligence algorithm that is inspired by the process of natural evolution. It can be used to find solutions to both mundane and complex problems. In evolutionary programming, a population of solutions is randomly generated, and then evolved over time by applying a selection process and genetic operators (e.g. mutation, crossover) to generate new (and hopefully better) solutions.

The use of evolutionary programming to solve chess has been researched by several groups, with many publishing promising results on the use of genetic programming on chess engines. However, the underlying truth is that the top-ranking engines today altogether avoid the use of use genetic programming.

The Problem

One of the main challenges in trying to solve a given chess board is the high computational cost. The number of possible moves in a chess game is astronomical the more you want to “look into the future”, making it difficult to evaluate a sufficient solution in a reasonable amount of time.

On one hand, evolutionary algorithms can generate a large diverse set of chess-playing individuals. This diversity allows the evolutionary algorithm to explore a wide range of solutions, which increases the chances of finding a good strategy to a given board. However, this approach could be very time consuming due to the number of games being processed by the evolutionary program.

On the other hand, “traditional” engines hold a significant advantage in terms of computational efficiency. The search algorithm used by the 'Stockfish' engine for example, is highly optimized, which allows it to explore the chess game tree more deeply in a shorter amount of time than an evolutionary algorithm.
Additionally, the engines use endgame table-bases, which are precomputed databases of the optimal play in endgame positions, to find the best move more accurately (that way chess is actually “solved” with under 8 pieces remaining on the board). However, the engines perform at their peak level when they are calculating very deep into the game, with a ‘depth’ level – how many moves ahead the engine is calculating – that is exponentially high (e.g. at the starting board, there are 324 different ways to play the first 2 moves), and once again will lead to a time consuming process.

Both approaches have their ups and downs, and we would like to opt for a third approach – combining the use of both evolutionary programming and an existing engine.

The Solution

Chess engines mainly are based on other algorithms such as Alpha-Beta - algorithms that dramatically reduces the number of possible moves to calculate in a game engine with a given depth level. There are less chess engines that deploy genetic programming in their calculations.

In this project we want to raise and hopefully answer the following questions:

1. Could the use of genetic programming help us find a good outcome of a given chess board (an outcome that can be achieved in an actual game between two 'competent' players - with the ELO of around 1800)?
2. Could the use of genetic programming theoretically boost the performance of a 'simpler' chess engine that doesn't have a high depth?

We will build a python program, that with a given chess board, will calculate the best strategy it can find, with up to 180 moves ahead. It will basically act as a prototype of a chess engine that strives to produce the optimal strategy to play with a given FEN.

This program will use the help of evolutionary programming and an existing chess engine.

The genetic programming will be handled using EC-KitY – a python-based module that serves as a comfortable basis for creating evolutionary programs. EC-KitY was created and is being updated by Moshe Sipper, Achiya Elyasaf, Itai Tzruia, and Tomer Halperin

The chess engine we will use in our program and comparison is 'Stockfish'.

The evolutionary programming will do the following instructions:

1. Population and Individuals – We will use one subpopulation consisting of 200 individuals – each will hold a chess board related to a given FEN, the board will consist of kings, and pawns that may only promote to a queen.
2. Fitness – based on the color the algorithm will play on, the higher the score the better, a positive score indicates our algorithm has an advantage in the current board, and a negative one indicates disadvantage. A 'checkmate' score in favor of the algorithm is the best possible score capped at 100,000, followed by 'Mate In i' in favor of the algorithm's color with a fitness score of 100,000 – i .
3. Evaluation -

    3.1 Before an evaluation (apart from the first one configured in the Algorithm when the population is initialized), the algorithm will "hypothetically" play with an engine 6 moves ahead in all of the boards with the use of a cloned board for each.

    3.2 Evaluation of the individuals will be scored based on the engine inserted into the program, after each iteration of 3.1
4. Selection – In each generation, the algorithm will select the elites, the 1% of the population with the highest fitness score, it will clone them randomly and proceed onto the next iteration of the algorithm.
5. Mutation – each individual will hold a short array of strategies for the algorithm's 3 moves – {RightDefense, LeftDefense, RightAttack, LeftAttack} – this list will be used to increase the weight of the moves that correlate to the strategy given for the round – the direction is for the square the moves lead into, and the defense/attack is for whether the move is going upwards or downwards in the board (in relation to the color's view of the board). In each breeding of the population, each individual has a given chance to change its strategy array completely or stick with the old array.

6. Termination – An early termination will happen if either the best individual has managed to end the game In a mate, or all games ended in a draw. If no board was “solved” in favor of the algorithm in 30 generations, termination will occur.

The Program

The program will include all the modules that accompanied by EC-KitY, and also the standard python-chess module. We will also use 'Stockfish', specifically the latest build (as of December 2022) - v15.1 with the support of 'AVX2' architecture set, running on linux-based builds.

Adding to that are the files written specifically for this project - the files that inherit, apply, and reconfigure files from EC-KitY (BoardCreator, BoardEvaluator, BoardEvolution, BoardIndividual, ChessTerminationChecker, ElitismAndRepopulatingSelection, MutateStrategyFactor), and other files that calculate and integrate the use of all modules into the main running program (MovementCalculation and Communicator). More on every file below:

main.py

The main program, will initialize and call for the algorithm to run, may receive as an argument the FEN of a chess board or use the default chess board, that includes only 2 kings and 8 pawns, and pawns may only promote to a queen.

The algorithm we built will always play the first move regardless of its color (can be configured to let the engines play the first move with a variable).

It will also be responsible for closing all working threads/engines that are not automatically closed by the evolutionary algorithm implementation of ec-kity.

Communicator

The object that handles the communication between the algorithm and a number of chess-engines (stockfish by default) with the use of BoardEvolution's defined threadpool to create a "Thread per Engine" communication object.
This object will define two functions:

- send_move_request(ind, threadpool): Will receive from the algorithm the individual and the algorithm's threadpool. it defines within its scope two functions that will each call the other until either the needed amount of moves were played, or the temporary board's game has ended. The functions are:

  1. algorithm_calculation() – that will use MovementCalculation in order to calculate a move for algorithm's side and use it accordingly, and when the given amount of moves was made - will also stop the recursive call. This function will only be handled by one thread initialized by the communicator object.

  2.engine_communicate() – Will receive a move result object from one of the engines and make the move on the board. This function will always be handled by the threadpool of the algorithm.

  Based on whether the algorithm plays first or not, will send to the appropriate threadpool one of the functions with the scope defined.

- send_evaluation_request(ind) – Based on self.depth moves played in the temporary board, will evaluate the score of the temporary board, and only then proceed to do the first move of the algorithm and the engine on the individual's board (in the case that these two moves happened, else the game ended and the appropriate score will be returned accordingly. This function will be handled by the threadpool the evolutionary algorithm holds.

BoardIndividual

Every individual from the population will have its accompanied board stored, a clone of that board, alongside the necessary variables the ec-kity modules needs – a fitness class that when the individual is evaluated, the fitness variable will get a hold of the fitness of the board of the individual – in this case, how the board is evaluated regarding the color that the algorithm play.

The individual will also store an array of 3 randomly chosen enum values called MovementFactor, necessary for the mutation method and used by MovementCalculation's calculations.

BoardEvaluator

A small class written inside main.py, communicates the evaluation calls on each individual by directing to *send_evaluation_request* of the communicator class.

ChessTerminationChecker

The class that handles when to perform an early termination of the program, in this case, we configure the algorithm to terminate when an individual's board's game is over, either by a stalemate or mate in favor of the algorithm.

BoardEvolution

The object that runs the evolutionary algorithm, inherited from SimpleEvolution from ec-kity, will add a necessary step – for each individual of the population it will call for *send_move_request*, and wait until all the individuals' temporary boards were played the needed amount of moves.

ElitismAndRepopulatingSelection

Will handle the picking of the most suited individuals, and randomly repopulate with them until the new population reaches the amount of the previous one.

MutateStrategyFactor

Each individual has a change to completely change its array of move strategies, this class is responsible for deciding on the change for each individual it receive - basically randomly.

BoardCreator

Handles the creation of the population of individuals based on the FEN it is given from the main program.

MovementCalculation

An object that is given to all individuals, in each turn of the algorithm the communicator object will call *calculate_next_move* which will create a list of legal moves for current board with weights for each move (will define the probability of the move to be picked).

Each function will test a few general cases and modify the weight of the move, the use of all the defined functions will create an edited weight list, as the weight of a given move is altered by the number of cases resolves or fails in. For example, a pawn move that has free passage to queening will receive a significant boost in its weight in the given round, as in most cases the promotion to a queen is critical in winning or losing the game, more so when there are only kings and pawns in play.

Noted functions:

king_in_danger :

Checks whether the algorithm's king is in danger of mate, calls *find_king_escape* accordingly, which in turn calls *king_run* and *piece_move_for_escape.*

king_run:

Finds the move with the most advantageous direction for the king to escape and increases its weight.

piece_move_for_escape:

Finds the pieces that can move and also help the king with escaping in a danger of check, and increases the appropriate moves' weight.

advances_of_pawn:

Decides whether a given pawn move has a clear path to promotion and how close it is to fulfilling it, also if there are any neighboring pawns that can protect him and increases the weight in that case. Moreover, will decide on decreasing weight if the path is blocked with many pieces or the other color's king can capture it before the pawn can promote.

build_pawn_structure:

Check which pawn can move forward, in order to control more advanced squares while maintaining a defendable structure and change the weight accordingly.

give_check:

Check if a move that gives check to the opponent fits some requirements(e.g. does not risk losing a piece in the opponent's turn).

pawn_protector:

Check if given pawn move is the linchpin of another piece's defense.

good_trade:

Check if given piece can capture an opposing piece that is worth as much or more.

Experiment

Using several FENs, and FEN of the standard board including only the kings and pawns, and for each color, we will run the program numerous times and compare the results of both in the context of every run.

For a picked run, we will use ‘stockfish’ to analyze the moves and decide which the algorithm made it regarded as good moves (close to its optimal move by the evaluation comparison).

We chose the FENs as they demonstrate specific situations we based our engine on. Moreover, the standard board FEN was picked as it gives no advantage by stockfish with an evaluation depth of 48 and the standard board can lead to a vast number of strategies to promote a pawn to a queen and attack the opponents king. In this experiment we can expect some aggressive plays with the king, since while they are risky, king moves may open the way for a pawn to promote. Due to the “openness” of the standard FEN we will run more tests on it in comparison to other FENs.

We will track which first moves were picked the most for both sides, the average outcome of the match, the best strategy we found as well as the average time the program ran.

Findings

Standard board FEN:

Results for White:

14 games were won by our algorithm, 2 by the engines, and 13 “undecided” games occurred.

In the algorithm game that occurred on the 12th run, 65 moves were played, in which the critical move was our promotion to a queen at move 53.

On that run, the starting moves were: e2e4 e7e5 e1e2 f7f6 e2d3 e8f7 (Two king moves at the start!)

Amongst our wins, the algorithm played these moves:

e2e4 5 times, d2d4 3 times, f2f3 2 times, f2f4 2 times, c2c3 once, and e1d1 once (king move at the first round).

In our losses, e2e3 was played 2 times and f2f4 was played once.

It is interesting to note that regardless of wins or losses, the engine only played e7e5 and d7d5, 20 and 10 times respectively, these are standard aggressive moves by the black’s side.

The average running time of the program was 36 seconds, with 114 moves made on the board on average.

This run was tested on pc running Ubuntu (Dual Boot) with AMD Ryzen 5800HS processor.

Results for Black:

*Before testing the algorithm with black's board, we decided to use the FEN of the first move already played by white – e2e4 – as it was the dominant move to pick both by our algorithm and especially by stockfish with depth of 33.

14 games were won by our algorithm, 2 by the engines, and 13 "undecided" games occurred, interestingly coinciding with the results for white.

In the algorithm game that occurred on the 20th run, 60 (59 plus e2e4) moves were played, in which the critical move was our promotion to a queen at move 51.

On that run, the starting moves were: f7f6 c2c3 e8f7 d2d4 d7d6 e1d2 (one king move at the second move)

Amongst our wins, the algorithm played these first moves:

e7e5 - 11 , e7e6 - 3 , d7d6 - 4 , h7h5 - 1 , c7c6 - 4 , f7f6 - 6 , g7g6 – 1

The engine's first (second) moves were:

d2d3 - 3 , e1e2 - 11 , d2d4 - 6 , f2f4 - 2 , c2c3 – 8

In the algorithm's losses, e7e5 was the first move made.

As we deduced, the king was a key factor in this sort of board and the engine understood that as well, making e1e2 it's move in more than a third out of all the games played.

The average running time of the program was 34 seconds, with 160 moves made on the board on average.

This run was tested on pc running Ubuntu (Dual Boot) with AMD Ryzen 5800HS processor.

Other FENs:

We also compared a few 'unequal' FENs, where the positions of black and white were different and could theoretically be reached in normal games.

We received the following results:

- When white had a slightly better position (0.4 score advantage according to stockfish analyze), our program won 5 wins out of 10 runs, the other 5 were draws. This may indicate that in endgame positions where there is already some development of pieces the program can capitalize on that advantage and have a chance at winning the game.
- When our program had a notably worse position (-3.6), our program managed to win and draw once out of 10 games. This shows that even in losing position the program may win against stockfish with depth of 9 when it blunders the advantage (we should note that when playing the opposite color in this position our program won every time).

Lastly, we decided to further analyze a given move order we received - where the program won despite having a disadvantage of (-3.6) - and compare both the program and the engine's moves (with depth of 9) to the stockfish engine analysis (with depth of 33). Out of the 25 moves it made, 18 moves were of the same worth to the stockfish analysis!

*All results can be found in the folder "FEN_results" at the github page.

Conclusion

While we weren't surprised that an aggressive king play could win a game with the rules we defined (at the second and third moves however we were), our program managed to do well over our expectations, as the ELO of stockfish with a depth of 9 is around 2165 (and depth of 12 is above 2300!), and while we added around 600 lines of code in MovementCalculation, it by itself does not hold a candle to stockfish's use of Alpha-Beta without the aid of genetic programming, without even considering the slower runtime of running the algorithm several times, which also plays a huge factor while building a chess engine.

Even though deducing strategies from a given FEN took us well over 1000 seconds, we can see how the extra step of genetic programming can help a simple program go toe to toe with a serious chess engine, in which many contributors help build. With more weight related functions for the unused pieces and an adequate board evaluator of our own replacing the aid of stockfish, we could have a full chess engine altogether (We can also implement genetic programming on the evaluation process).

It is important to note that when given the standard FEN, the algorithm needs to choose 1 out of over 1000 possible combinations of 3 moves, some of them being critical to the later outcome, and while we used the evaluation of the engine to guide us to find the elite individuals, the odds of us always picking the top moves are very much not in our favor as opposed to the engine, yet the results still speak for themselves, much like chess.

Bibliography

(1) https://stockfishchess.org/about/

(2) https://www.chess.com/learn-how-to-play-chess

(3) https://www.chess.com/terms/chess-engine

(4) https://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/chessmain.html giving a brief review for algorithm used in chess engines.

(5) https://web.ist.utl.pt/diogo.ferreira/papers/ferreira13impact.pdf, pages 76-77 for stockfish's ELO analysis

(6) https://lichess.org/editor, convenient website for creating chess boards both with GUI and importing/exporting with FEN. Can be used to analyze boards with a build in engine support.

(7) https://github.com/ec-kity/ec-kity/ repo of ec-kity used in this project.

(8) https://tb7.chessok.com/ - A website on a table base of any chess board with 7 or less pieces