Analysis and Comparison of Chess Algorithms

Vesela Trajkoska and Gjorgji Dimeski
Faculty of Computer Science and Engineering
University "Ss. Cyril and Methodius"
Skopje, Macedonia

vesela.trajkoska@students.finki.ukim.mk
gjorgji.dimeski@students.finki.ukim.mk

***Abstract*—In this paper we analyze the results of three different algorithms programmed for playing chess – genetic algorithm, Monte Carlo, and Minimax. The algorithms are implemented in Python through 5 players that play chess against the Stockfish engine, each for 10 games, after which their Elo rating, game evaluation, game status, and time per move are compared. The results show that the algorithms cannot compare to an extensively trained and optimized chess engine such as Stockfish, and only 2 games of 50 total were won by the Minimax algorithm. There were no draws. The genetic algorithm is very fast, with less than a second needed for each move, while the other two are much slower, with times sometimes reaching over a minute. The Minimax algorithm's speed decreases over time, while the Monte Carlo algorithms' speeds increase over time.**

***Index Terms*—chess, genetic algorithm, Minimax, Monte Carlo, Stockfish, comparison**

I. Introduction

One of the greatest achievements for artificial intelligence is the defeat of the world chess champion by a computer program. The increasing popularity of chess and the eternal admiration of the game due to the number of possible moves and combinations continuously evoke the curiosity of many researchers and are the reasons for the continuous creation of new chess-playing algorithms. At the same time, it is important to have a clear picture of the performance of the most famous and most used such algorithms, in order to direct further research in the right direction. We would not want to invest in an algorithm that has a generally poorer performance compared to another. Additionally, it is good to have a concise way of presenting those results to the general public, so that laymen unfamiliar with computer science, including chess enthusiasts, have a clearer picture of the progress of AI in this specific field.

This research paper focuses on three popular algorithms – genetic, Minimax, and Monte Carlo – and their implementations in Python intended for playing chess. After conducting experiments where these algorithms play chess against the current most popular chess engine – Stockfish, their performance is analysed and compared.

II. Related work

There is very little research where experiments aim to directly compare different chess algorithms. We found one paper that analyzed and compared chess engines, which is closely related to our research questions. There is an abundance of literature available regarding human chess games, which is relevant to our paper in the context of deciding how to analyse chess games. Additionally, research revolving around chess ranking systems is applicable when deciding which comparison method to use. In the following subsections, we provide a brief overview of three related papers.

A. Analysis of human chess games

In [1], the researchers analyse 4.78 million chess games between humans, by simulating them using Stockfish. The research found that there was a strong correlation between Elo ranking and winning chances. The mean ply per game was 80, which means an average of 40 moves were taken by each of the players. Interestingly, in games of players with higher Elo rankings, the ply per game was lower since they usually get a head start earlier in the games. The average ply per game also changed throughout the years. The research provides additional analysis on the effect of the color played and the opening moves, which had a different effect before and after openings theory was introduced.

B. Analysis and comparison of chess ranking systems

In [2], the authors conduct an experiment in order to compare the Elo, Chessmetrics, German Evaluation Number (DWZ), and Glicko-2 ranking systems. The goal was to explore whether one of them would be able to overcome the shortcomings of Elo. The results showed that the Glicko-2 rating system is the most appropriate for ranking evolutionary algorithms.

C. Performance comparison between selected chess engines

In [3], the author compared different chess engines to find the strongest and weakest, including their data and CPU usage. The top three strongest algorithms were Stockfish at first place, Rybka, and Bikjump, although all of them required a higher amount of computational power and resources. In this paper, the same methodology is used as in our research, with the equal goal of discovering the best algorithm. The difference is that it included official chess engines which use varieties and combinations of algorithms, while we compare raw algorithmic implementations.

III. Definitions

A. Genetic algorithm

Genetic algorithms are a branch of evolutionary algorithms which are inspired by and created according to the rules of evolution, using the Darwinian principle of natural selection. They provide a procedure that leads to the optimized or best possible solution of a problem. The algorithm goes through all of the phases that occur within reproduction and natural selection: inheritance, selection, crossover (recombination), and mutation. The goal is to reach the fittest offspring from each generation [4].

B. Minimax algorithm

The Minimax algorithm, most popular in game theory, is an adversarial search algorithm, where the movement of an opponent is traced and the algorithm's decision making is influenced by it. The goal is to find the best achievable utility against a rational (optimal) adversary on each turn. The algorithm tries to minimize the possible loss in a maximum loss scenario, while maximizing the minimum gain. This algorithm is very useful in zero-sum games, where the player's gain is equivalent to the opponent's loss, such as chess [5].

C. Monte Carlo algorithm

Monte Carlo is a broad class of randomized algorithms which use repeated random sampling to obtain numerical results. The underlying concept is to use randomness to solve problems that might be deterministic in principle. This means that running the algorithm repeatedly will eventually produce correct results, as long as the probability of a correct answer is greater than zero and there is a method to determine the correctness of the answer [6].

D. Elo rating

The Elo rating system is a method of calculating the skill level of players in zero-sum games. The rating is represented by a number that rises or falls depending on whether the side is winning or losing. How much the rating shifts is determined by the rating difference of the two sides, the greater the difference – the greater the increase and decrease of the rating in both sides [7].

IV. Methodology

For the experiments within our research, we use freely available implementations of the aforementioned algorithms found on GitHub.

A. Genetic algorithm

The genetic algorithm is implemented by Victor Sim [8]. It generates the best agent using crossover, mutation, and fitness evaluation. We generated two agents – one trained for 6 generations with a population size of 8, and another trained for 15 generations with a population size of 10. During each generation, all of the agents play chess among themselves and a fitness function constantly reevaluates their fitness. During the games, legal moves on each turn are traversed using a Monte Carlo search tree and each of them is evaluated by a neural network with 5 layers for 5 epochs. The move with the highest evaluation score is played [9].

After the agent is trained, the same method that was used to evaluate each position in the training phase is again used to generate moves throughout the games. This method takes the current position of the board and the evaluation function as inputs, and returns the best calculated move.

B. Minimax algorithm

The Minimax algorithm is implemented by Ishaan Gupta [10]. At its core, this algorithm is a tree search algorithm with every node of the tree representing a specific move, or rather, board state, possible at that point in the game. The depth of the tree represents the number of moves the algorithm has analysed ahead of the current state. As the algorithm needs to minimize the loss in a maximum loss scenario while maximizing the gain in a minimal gain scenario, it must use an evaluation function to determine the loss or gain in a given state. The implementation uses predefined weights for each of the pieces with the King having the greatest weight – infinity. Conversely, the pawns have the least weight out of all pieces.

Analyzing every state is a costly operation that takes a lot of computing time, and every extra level analyzed adds to that complexity. The time needed for the algorithm to analyze all of the board states grows exponentially with the depth of the tree and increasing the depth for greater analysis becomes cumbersome. Therefore, an optimization is applied to the Minimax algorithm called alpha-beta pruning, which stops the evaluation of nodes that have already been proven to be worse than the current best. The alpha value is the maximum evaluation the algorithm can guarantee while the beta value is the minimum evaluation. With alpha-beta pruning, the Minimax algorithm can substantially increase its performance while not sacrificing its efficacy.

For our experiments we used a Minimax algorithm optimised with alpha-beta pruning with a maximum search depth of 3.

C. Monte Carlo algorithm

We used Ishaan Gupta's implementation of the Monte Carlo algorithm as well for our research [10]. The Monte Carlo algorithm conducts multiple iterations of three phases known as rollout, rollback, and expand, in order to choose the best evaluated move. In the rollout phase the algorithm plays out a game from a particular board position, or tree node in our case, simulating random games. After the rollout phase is completed, the algorithm performs a rollback, where all of the moves done in the rollout phase are undone, returning to the original position. This is done so that the algorithm can try different continuations of the position and see how they compare to the continuation used in the rollout phase. During the expand phase, the algorithm analyses the continuations used during the rollout phase and the most promising ones are selected for further exploration [11].

We experimented with two different iteration counts for the Monte Carlo algorithm, one with 10 and another with 30 iterations.

D. Setup for the experiment

Since games could last exceedingly long, we limited the maximum number of turns to 30. If the game reached the 30th turn inconclusively, we evaluated the board position and assigned the winner according to the evaluation. For the results, we adjusted the evaluations to reflect the algorithms' performance with negative evaluations correlating to bad performance and positive evaluations to good performance. If the game was a loss for our algorithms, we assigned a negative max evaluation. Conversely, if it was a win we assigned a positive max evaluation.

Each algorithm's starting Elo rating was set to 400.0 at the beginning of the experiment. We used the Stockfish engine [**?**] as an adversary for the algorithms, set its Elo rating to 400.0 as well and kept it constant in order to match the starting rating of the algorithms. We evaluated the algorithms across 10 games.

The experiments were ran on a computer with an AMD Ryzen 7 4800H processor and 16GB of RAM clocked at 3200MHz. The complete source code for the experiments is uploaded to our GitHub repository [**?**].

V. Results

The results for each of the algorithms during the 10 games where measured through: Elo rating, the game evaluation (positive value – better board evaluation for the algorithm, negative value – worse board evaluation for the algorithm or 0 – equal board evaluation), and game status (1 – win, -1 – loss or 0 – draw).

After the 10 games, the genetic and Monte Carlo algorithms had a continuous decrease in their Elo ratings (due to losing all of their games), while the Minimax algorithm had a slight increase from the 8th and 10th games. The Elo ratings of the genetic and Monte Carlo algorithms were identical with the Minimax algorithm's up to the 7th game, and then decreased to 361.95, 357.46, and 353.10 in the 8th, 9th, and 10th game respectively.

From the aspect of game evaluation, both genetic agents performed similarly, even though the second one was trained across more than double the amount of generations. Their game evaluation was constantly -9999, except when the second agent reached a game evaluation of -4 in the second game, which means that it almost evened out the board position by the end of the turn limit. Similarly, both Monte Carlo algorithms have almost identical results, with the only difference being that the first player managed to reach a game evaluation of -3 in the last game, again meaning that it evened out the board position by the end of the turn limit. The rest of the evaluations were -9999 as well.

From Table I we can see that Minimax was the only algorithm which managed to "win" against Stockfish, by having a better evaluation after the turn limit in two games (not by checkmate – this is an important distinction to make, Minimax did not checkmate its opponent but rather had a better board position after the turn limit passed).

TABLE I
MINIMAX ALGORITHM

Game Index	Game statistics		
	Elo	*Game evaluation*	*Game status*
1	395.00	-132	-1
2	390.07	-1317	-1
3	385.21	-1	-1
4	380.42	-274	-1
5	375.70	-2300	-1
6	371.05	-9999	-1
7	366.47	-3	-1
8	371.95	338	1
9	367.35	-388	-1
10	372.82	1277	1

A. Speed visualization

In "Fig. 1", the average time per move is visualized for the Minimax and Monte Carlo algorithms. The genetic algorithm always had a time per move between 0 and 1 (within milliseconds), but closer to 0, which made it incomparable with the other two algorithms.

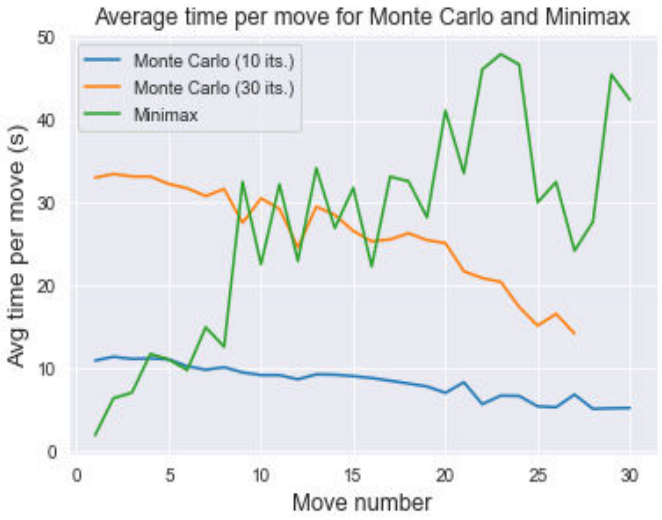

Fig. 1. Comparison of average time per move

Here we see opposite behavior – the Minimax algorithm starts with a very low time per move, which increases through the games, while Monte Carlo starts with a higher time per move, which decreases throughout the games. Still, Monte Carlo's lowest times vary around 5 seconds for 10 iterations and around 15 seconds for 30 iterations, which can be considered fast or comparable to a human player, but extremely slow compared to the genetic agent.

VI. Discussion

A. Game results

In general, all of our algorithms performed poorly even against the Stockfish engine set to lowest performance.

The genetic agent generated from a population of size 8 across 6 generations lost all of the games against the engine before reaching the maximum turn. The second agent managed to nearly even out the board state before the maximum turn had been reached at the third game, but lost the rest of the games. It is expected for the second more extensively trained agent to perform somewhat better. Still, the board got a negative evaluation resulting in a loss for the agent.

The situation is similar for the Monte Carlo algorithms. Interestingly, the Monte Carlo algorithm set to 10 iterations evened out the board state at the last game while the one set to 30 iterations got checkmated at every game. Considering that Monte Carlo is based on random sampling, these results aren't out of the ordinary.

The Minimax algorithm performed best. With a maximum search depth of 3, it managed to last through all of the 30 turns in all but one game – all while almost evening out two games and winning 2 by evaluation. It is also the only algorithm that managed to win against the engine, even if not by checkmate but by board evaluation. This does not mean that the specific implementation of the Minimax algorithm is much better than the other two in the general case, it simply confirms that Minimax is better suited for zero-sum games.

B. Speed

The genetic agent had outstanding results for required time per move – nearing 0 seconds – making it incomparable with the other two algorithms which sometimes needed entire minutes to calculate their next move. However, the agent needs time to be generated and trained, while the other algorithms need no previous setup. The required training time depends on the size of the population and number of generations. Since higher values for these parameters yield a better trained agent, the required training time for a large-scale chess engine based on a genetic algorithm would need to be very long for the engine to reach acceptable results.

If we compare Minimax and Monte Carlo separately, as we previously stated, they had opposite behaviors – Minimax started with a low time per move which gradually increased, while Monte Carlo started with a high time per move which gradually decreased.

In Minimax's case, this is due to the increase of possible moves as the game progresses. At the opening phase of the game, there is a small amount of possible moves and outcomes, which increase as the game transitions. However, this increase in possible moves will eventually stagnate as more pieces are removed from the board and the possibilities are reduced, and will decrease as the games nears its end.

Monte Carlo's iterations linearly affect the time needed to make a move. Hence, for 30 iterations the algorithm needs triple the time compared to 10 iterations. As the game progresses, the amount of pieces on the board decreases, which leaves less simulations for Monte Carlo to execute. This interpretation could explain the decreasing time.

VII. Conclusion

The Minimax algorithm proved to be the best at playing chess. Further increasing the depth of the search tree would yield even better results, but doing so would require high processing power and memory. Extra optimizations on top of alpha-beta pruning could prove to be helpful in decreasing the search time.

On the other hand, even though the genetic agent had worse game performance, it made moves almost instantly on the downside of requiring extra time for training before being usable. If the requirement is faster decisions, genetic agents can prove better.

Lastly, the Monte Carlo algorithm turned out not to be the most suitable algorithm for playing chess, which aligns with other research papers in this field.

As our research focused on the general performance of the algorithms across multiple games, future work could be done on performance analysis on specific aspects of the game. This could include the three phases of chess and even specific board positions and openings. Furthermore, research on Minimax optimization besides alpha-beta pruning could prove beneficial, along with enhanced board evaluation heuristics.

References

[1] M. Acher and F. Esnault, "Large-scale Analysis of Chess Games with Chess Engines: A Preliminary Report," *arXiv preprint arXiv:1607.04186*, Apr. 2016. [Online]. Available: https://arxiv.org/pdf/1607.04186.pdf [Accessed: 28 Apr. 2016].

[2] N. Veček, M. Črepinšek, M. Mernik, and D. Hrnčič, "A Comparison between Different Chess Rating Systems for Ranking Evolutionary Algorithms," in *2014 Federated Conference on Computer Science and Information Systems, FedCSIS 2014*, 2014, pp. 511-518. doi: 10.15439/2014F33.

[3] M. Sojka, "Performance comparison between selected chess engines," *Journal of Computer Sciences Institute*, vol. 24, pp. 228-235, 2022. [Online]. Available: https://doi.org/10.35784/jcsi.2975.

[4] S. Mirjalili, "Genetic Algorithm," in *Evolutionary Algorithms and Neural Networks*, Studies in Computational Intelligence, vol. 780, Cham: Springer, 2019, pp. 47-69. doi: 10.1007/978-3-319-93025-1_4.

[5] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2003.

[6] D. P. Kroese, T. J. Brereton, T. Taimre, and Z. I. Botev, "Why the Monte Carlo method is so important today," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 6, no. 6, 2014. doi: 10.1002/wics.1321.

[7] "Elo rating system," *Wikipedia*, Jan. 28, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Elo_rating_system.

[8] V. Sim, "Genetic Algorithm for Chess," 2021. [Online]. Available: https://github.com/victorsimrbt/chess_mc_ga.

[9] V. Sim, "Building a Chess AI that Learns from Experience," *Towards Data Science*, 2021. [Online]. Available: https://towardsdatascience.com/building-a-chess-ai-that-learns-from-experience-5cff953b6784.

[10] I. Gupta, "Chess Bot AI Algorithms," 2020. [Online]. Available: https://github.com/Ish2K/Chess-Bot-AI-Algorithms.

[11] I. Gupta, "Monte Carlo Tree Search Application on Chess," *Medium*, 2020. [Online]. Available: https://medium.com/@ishaan.g