Exploring the Performance of Monte Carlo Tree Search, Genetic Algorithm, and Neural Fictitious Self-Play in Tournament-Style Chess and Puzzle-Solving

Nihal Ranchod
University of the Witwatersrand
School of Computer Science and Applied Mathematics
Johannesburg, South Africa
2427378@students.wits.ac.za

Branden Ingram - *Supervisor*
University of the Witwatersrand
School of Computer Science and Applied Mathematics
Johannesburg, South Africa
branden.ingram@wits.ac.za

Pravesh Ranchod - *Supervisor*
University of the Witwatersrand
School of Computer Science and Applied Mathematics
Johannesburg, South Africa
pravesh.ranchod@wits.ac.za

***Abstract*—Chess has long been a benchmark for evaluating artificial intelligence (AI) algorithms. We present a comparative analysis of three AI techniques: Monte Carlo Tree Search (MCTS), Genetic Algorithm (GA), and Neural Fictitious Self-Play (NFSP) in chess gameplay. We test to identify the optimal AI technique for achieving expert-level performance in both tournament-style chess and puzzle-solving. The algorithms are implemented and tested using the OpenSpiel framework, where evaluation is done through win rates and Elo ratings as well as other comparable metrics and baseline implementations such as Stockfish playing at a low skill level. This study contributes to the field of AI chess by demonstrating which techniques produce the best results for chess related tasks, offering insights for future research and development of expert-level chess AI systems.**

***Index Terms*—Monte Carlo Tree Search, Genetic Algorithm, Neural Fictitious Self-Play**

I. Introduction

The history of artificial intelligence (AI) in chess and complex board games [1] can be traced back to 1950, where Claude Shannon published a seminal paper titled "Programming a Computer for Playing Chess". This laid the foundation for the development of chess-playing algorithms [2]. Considering there has been a long-standing effort in the field of AI to develop expert agents that can excel at complex board games [1]. Shannon's work has inspired researchers to explore the potential of AI proficiency in this domain, hallmarking the idea of intelligence and learning [3]. The 1960s and 1970s saw the emergence of the first chess-playing programs, such as the MacHack VI, which employed a combination of tree search and evaluation functions [4]. This was inherent of the work done by Arthur Samuel on checkers in 1959 [5], which focused on tree search algorithms and evaluation functions. Complex board games like chess serve as valuable domains for exploring expert AI agents due to their defined rules [6], strategic depth, and the clear delineation between success and failure [7].

There has been substantial success of current chess engines like Stockfish, which has consistently ranked among the top chess engines in the world [8]. Despite this, there remains ample room for further investigation into alternative AI techniques and approaches for chess. While current state-of-the-art chess engines rely heavily on alpha-beta pruning and other search-based methods [9], the exploration of novel AI algorithms, such as Monte Carlo Tree Search (MCTS), Genetic Algorithms (GA), and Neural Fictitious Self-Play (NFSP), may yield new insights and improvements in chess AI performance. Such techniques have shown promise in various domains, including board games like Go [10] and poker [11], and their application to chess could potentially lead to the development of more efficient and effective chess-playing AI systems. Furthermore, investigating the performance of these algorithms in different chess-related tasks, such as tournament-style gameplay and puzzle-solving, can provide a more comprehensive understanding of their strengths and weaknesses, guiding future research efforts in this domain.

MCTS is a decision-making algorithm that combines random sampling with the precision of tree search, allowing for efficient exploration and exploration of the game state space [6]. Its success in games like Go [10] and its adaptability to different game structures make it a viable candidate for chess AI research. GAs, on the other hand, are inspired by the principle of natural selection and evolution, using mechanisms such as mutation, crossover, and fitness evaluation to evolve increasingly better-performing chess AI agents [12]. The ability of GAs to optimise and discover novel strategies makes them a compelling choice. Lastly, NFSP is a reinforcement learning approach that combines fictitious play with neural networks. While it has been successfully applied in imperfect information games like poker [13], its potential for application in chess, where decision-making under uncertainty is crucial, remains to be fully explored. The ability of NFSP to model opponent strategies and adapt its own play accordingly

suggests that it could be a promising avenue for future research in chess AI.

Our results indicate our MCTS variants that leverage historical Portable Game Notation (PGN) data achieved the highest Elo ratings, outperforming the other bots and most notably the baseline Stockfish engine playing a skill level of 1 in tournament-style gameplay. In addition, the baseline Stockfish engine exhibited the highest solve rate in puzzle-solving with our MCTS variants not far behind. Our results highlight the potential of our AI algorithms [1] in chess and chess-related tasks.

II. Background

A. Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a decision-making algorithm [14] that combines the precision of tree search with the randomness of Monte Carlo sampling [6]. MCTS consists of four key phases: selection, expansion, simulation, and backpropagation [6]. In the selection phase, the algorithm traverses the game tree from the root node to a leaf node, using a selection policy to choose the most promising actions [15]. The Upper Confidence Bound for trees (UCT) algorithm is a popular selection policy that balances exploration and exploitation [14] by considering both the average reward and the number of visits for each node [16]. UCT selects the node the maximises the following formula:

$$UCT = \frac{w_i}{n_i} + C\sqrt{\frac{\ln N}{n_i}} \tag{1}$$

where w_i is the total reward of the node, n_i is the number of visits to the node, N is the total number of visits to the parent node, and C is an exploration constant. Once a leaf node is reached, the expansion phase begins, where one or more child nodes are added to the tree based on the available actions [6].

In the simulation phase, the algorithm conducts random play-outs from the expanded nodes until a terminal state is reached, determining the outcome of the game. Finally, in the backpropagation phase, the result of the simulations are propagated back through the tree, updating the statistics of each visited node [15].

By repeating these four phases iteratively, MCTS gradually builds a search tree that focuses on the most promising areas of the game state space [17]. The algorithm can be terminated based on a fixed number of iterations, a time limit, or when a desired level of confidence is reached.

B. Genetic Algorithm

Genetic Algorithms (GA) are a class of evolutionary algorithms [18] inspired by the principles of natural selection and genetics [19]. They have been applied to a wide range of optimisation problems [20], including the development of chess AI systems. GAs operate on a population of individuals [21], each representing a potential solution to the problem at hand [22]. In the context of chess AI, an individual typically encodes a set of parameters or strategies that govern the behaviour of the chess engine [12].

The core components of a GA include:

1) Representation: Each individual in the population is encoded as a sequence of genes, which can be represented as binary strings, real numbers, or other data structures [23].
2) Fitness Function: A fitness function is used to evaluate the quality of each individual in the population based on its performance in the problem domain [21]. In chess AI, the fitness function may consider factors such as win rate, Elo rating, or the ability to solve specific chess puzzles.
3) Selection: During each iteration of the algorithm, a subset of individuals is selected from the population to serve as parents for the next generation. Selection methods such as tournament selection or roulette wheel selection are commonly used, favouring individuals with higher fitness values [22].
4) Genetic Operators: The selected parents undergo genetic operations to create offspring for the next generation. The two primary genetic operators are crossover and mutation [24]. Crossover combines genetic material from two parents to produce one or more offspring, while mutation introduces random changes to the genes of an individual to maintain diversity in the population.
5) Replacement: After the offspring are generated, they are evaluated using the fitness function and inserted back into the population, replacing some of the existing individuals [20]. The replacement strategy can be based on fitness or age.

The GA iteratively applies these steps over multiple generations, gradually evolving the population towards better solutions. The process continues until a termination condition is met, such as reaching a maximum number of generations or finding an individual with a satisfactory fitness value.

In the context of chess AI, GAs have been used to evolve various components of chess engines, such as evaluation functions, search heuristics, and opening books [25]. By encoding these components as individuals in the population and defining an appropriate fitness function, GAs can automatically discover and optimise novel strategies for playing chess.

One notable example of GA-based chess AI is the Evolving Neural Networks through Augmenting Topologies (NEAT) algorithm [26]. NEAT evolves both the structure and weights of neural networks using a GA, allowing for the automatic discovery of effective neural architectures for chess gameplay.

While GAs have shown promise in developing chess AI systems, they often require careful design of the representation, fitness function, and genetic operators to achieve good performance. Additionally, GAs can be computationally expensive, as they require evaluating multiple individuals over many generations. However, the ability of GAs to explore a wide range of strategies and automatically discover novel solutions makes them a valuable tool in the development of

[1] Source is publicly available at https://github.com/nihal-ranchod/honours-research

chess AI systems.

C. Neural Fictitious Self-Play

Neural Fictitious Self-Play (NFSP) is a reinforcement learning approach that combines fictitious play with neural networks to enable AI agents to learn and adapt their strategies through self-play [13]. NFSP has been successfully applied to various imperfect-information games, such as poker [11]. In the context of chess, NFSP can be used to train AI agents that can learn and adapt to different playing styles and strategies.

The core idea behind NFSP is to train two neural networks: a policy network and a value network. The policy network takes the current game state as input and outputs a probability distribution over the available actions. The value network, on the other hand, estimates the expected value of each state, representing the likelihood of winning from that state [13].

During training, the AI agent engages in self-play, where it plays against itself or against past versions of itself. The training process consists of two main components:

1) Reinforcement Learning: The policy network is updated using reinforcement learning techniques, such as Q-learning or policy gradients. The agent plays against itself and updates the policy network based on the rewards obtained from each game. The goal is to learn a policy that maximises the expected reward over time.
2) Supervised Learning: The value network is trained using supervised learning on a dataset of game states and their corresponding outcomes. The dataset is generated during self-play, where the final outcome of each game is recorded along with the encountered states. The value network learns to predict the expected value of each state based on this dataset.

By combining these two components, NFSP allows the AI agent to learn and adapt its strategy based on its own experiences and the experiences of its opponents. The policy network learns to select actions that maximise the expected reward, while the value network provides an estimate of the value of each state, guiding the policy network towards promising actions.

One of the key advantages of NFSP is its ability to handle imperfect information, which is prevalent in games like poker [11], where players have hidden information. In chess, although the game state is fully observable, NFSP can still be applied to learn and adapt to different opponent playing styles and strategies [27].

During self-play, the AI agent plays against itself, generating game trajectories and outcomes. These trajectories are used to update the policy network using reinforcement learning, while the outcomes are used to train the value network using supervised learning. Over time, the AI agent learns to play chess, adapting its strategy based on its own experiences and the experiences of its opponents.

D. Related Work

Chess AI has explored various aspects of perfect and imperfect information, reward and fitness functions, and the exploration-exploitation trade-off.

Lai [28] investigated the differences between perfect and imperfect information in chess and poker, highlighting the challenges posed by imperfect information and the need for adaptive strategies [29]. In deterministic single-action adversarial games [30] with perfect information, MCTS has been demonstrated to achieve optimal solutions successfully. It's capability to sample future states and assess potential moves [14] contributes to its robust performance evident in games like Go [31].

The design of reward and fitness function is crucial in reinforcement learning [32] and evolutionary algorithms [30]. Rewarding the agent for wining games, capturing pieces, or achieving a higher board evaluation score [31] are common approaches in chess AI. Similarly, fitness functions in evolutionary algorithms often consider the win rate or the ability to solve specific chess puzzles [12].

Balancing exploration and exploitation is a fundamental challenge in decision-making algorithms [33]. In MCTS, the Upper Confidence Bound for Tree (UCT) [16] addresses this trade-off by considering both the average reward [34] and the number of visits for each node [35] during the selection phase. Genetic algorithms inherently balance exploration and exploitation through the use of mutation and crossover operators [19]. In Reinforcement Learning (RL), the exploration-exploitation trade-off is addressed through the exploration strategy employed during policy learning [34]. TD-learning algorithms [36] often utilise ϵ-greedy exploration strategies, which involve selecting actions based on a balance between exploiting the current optimal action and randomly selecting actions with a certain probability [35].

III. Methodology

A. Data Acquisition & Pre-Processing

The data used in this research was acquired from the Lichess database [2]. For standard chess games, the data was downloaded in Portable Game Notation (PGN) format as seen in Fig. 2. However, for the chess puzzle data, the Lichess database provides the data in Comma Separated Values (CSV) [3] format.

To ensure consistency and compatibility, a custom script was developed to convert the puzzle data from CSV to PGN format. The script extracts the relevant information for each puzzle, including the initial board position in Forsyth-Edwards Notation (FEN) format and the sequence of moves leading to the puzzle solution. The script then creates a new chess game for each puzzle, setting up the initial board position using the FEN. The moves are applied sequentially to the game, simulating the puzzle scenario. The resulting PGN files serve as the input data for training and evaluating the implemented algorithms.

B. Pipeline Implementation

The AI chess models were trained and evaluated using a structured pipeline, as illustrated in Fig. 1. The pipeline

[2]https://database.lichess.org

[3]https://database.lichess.org/#puzzles

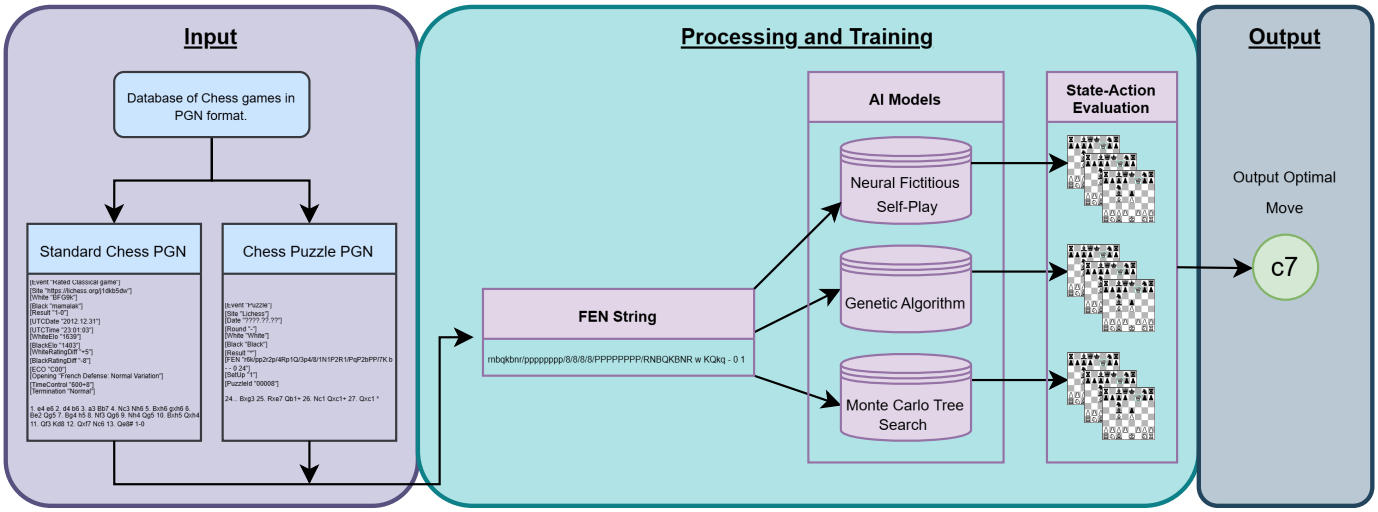

Fig. 1. Pipeline diagram depicting data use and flow, outlining the path from input to model training for optimal move prediction in implemented AI models.

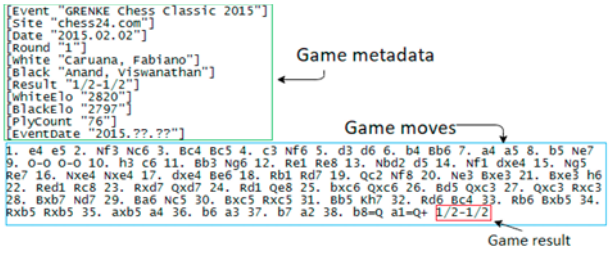

Fig. 2. Example of Portable Game Notation (PGN) Data for a single game of Chess.

consists of three main stages: input, processing and training, and output.

Two databases served as input to the models. A database of chess games in PGN format and a database of chess puzzles also in PGN format. The games database contained standard chess games with moves, while the puzzles database contained chess positions and their optimal next moves.

The input PGN data was passed through a parser to extract the relevant information such as the FEN string representation of the board states in a format suitable for training the AI models.

The trained models output a state-action evaluation. An assessment of the best move given the current board state. This is used to guide move selection in both standard tournament-style games and in solving chess puzzles optimally.

C. Algorithmic Implementation

1) Monte Carlo Tree Search: We present three variants of the MCTS algorithm in this research. The first being a vanilla MCTS algorithm as well as two enhanced versions of MCTS that leverage historical data from past chess games and chess puzzles to inform their decision-making processes.

The vanilla MCTS algorithm was directly sourced from the OpenSpiel [4] framework. OpenSpiel provides a standard implementation of MCTS, which serves as a foundation for comparison and evaluation of the enhanced MCTS versions developed in this research.

The enhanced versions of the algorithm extends the traditional MCTS by integrating additional state information into the search nodes. This allows the algorithms to retain specific board states during tree traversal, enhancing data accessibility and move evaluation consistency. The algorithm is highly configurable, allowing for adjustments in exploration, caching, and the utilisation of opening moves from training data.

The first enhanced version, referred to as MCTS with Historical PGN Game Data, incorporates historical data from past chess games stored in the PGN format. By parsing these games, the algorithm constructs a knowledge base consisting of move occurrences, opening sequences, and the performance of each move in previous games. This historical data is used to guide the decision-making process, particularly in the opening and mid-game phases.

The second enhanced version, referred to as MCTS with Historical PGN Puzzle Data, follows a similar approach but instead leverage's historical data from chess puzzles. Chess puzzles represent specific board positions that require precise moves to achieve a desired outcome, such as checkmate or gaining a significant advantage. By incorporating puzzle data, the algorithm can enhance its ability to recognise and respond to critical positions that require accurate and decisive moves.

Both enhanced versions of the algorithm employ a progressive history mechanism to effectively utilise the historical data. This mechanism assigns scores to moves based on their historical outcomes, dynamically adjusting move probabilities based on previous performance. By favouring moves with strong past performance, the algorithms adopt an aggressive move selection strategy, which is especially effective in establishing early control, reducing unnecessary exploration in familiar

[4]https://github.com/google-deepmind/open_spiel

positions, and prioritising high-efficiency moves.

The integration of domain-specific knowledge in terms of historical game and puzzle data, along with the aggressive move selection strategy, proves to be valuable additions to the MCTS framework. These enhancements enable the algorithms to make more informed and strategic decisions.

2) Genetic Algorithm: We present a chess-playing bot that utilises a genetic algorithm (GA) framework to evaluate chess board positions and make strategic moves. The bot's decision-making process is driven by an evolving neural network model, which assesses the state of the chess board. By training the model on a combination of chess games and puzzles, the GA-based chess bot iteratively improves its position evaluation capabilities.

The architecture of the GA-based chess bot consists of two main components: a convolutional neural network (CNN) and the genetic algorithm itself. The CNN is designed to evaluate chess positions and output a score representing the favourability of the board for the current player. The CNN architecture comprises three convolutional layers with ReLU activations, followed by fully connected layers. The input to the network is a 12-channel representation of the chess board, where each piece type and colour is assigned a dedicated channel. This tensor encoding allows the CNN to capture piece-specific patterns and interactions on the board.

The genetic algorithm is responsible for training and evolving the neural network models across multiple generations. The GA workflow begins with the initialization of a population of models with random weights. Each model in the population undergoes a fitness evaluation phase, where it plays a series of games or solves chess puzzles, accumulating a score based on its performance. The fitness of a model is determined by its ability to correctly evaluate board positions and align with the expected outcomes.

After the fitness evaluation, the GA proceeds with a selection process, where models with the highest fitness scores are chosen to pass their traits to the next generation. The top-performing models are preserved as elite individuals, ensuring the maintenance of beneficial genetic information. The selected parent models undergo crossover, a genetic recombination process that creates child models by probabilistically swapping weights between the parents. This crossover mechanism allows the offspring to inherit traits from both parents, promoting the exploration of new solutions.

To maintain genetic diversity and prevent premature convergence, the GA introduces random mutations to the weights of the child models. The mutation rate and strength parameters control the extent and magnitude of these modifications, striking a balance between exploration and convergence. The resulting offspring, along with the elite models, form the population for the next generation.

The GA-based chess bot can be trained using two types of data: standard chess games and chess puzzles. When trained on standard games, the bot evaluates the board positions based on the final game outcomes, aligning its predictions with the actual results. Training on puzzles, typically focused on tactical positions, which involves evaluating the board before and after a key move, with fitness determined by the improvement in the position's score.

3) Neural Fictitious Self-Play: We present a chess-playing bot that employs the Neural Fictitious Self-Play (NFSP) algorithm to learn and adapt its strategy through a combination of supervised learning (SL) and reinforcement learning (RL). The NFSP approach enables the bot to dynamically adjust its play-style by leveraging both memory-based self-play and exploration.

The NFSP Chess Bot is designed with two primary goals in mind: learning from experience and maintaining an aggressive play-style. By learning through experience, the bot aims to improve its performance without relying heavily on domain-specific heuristics. The aggressive play-style is intended to avoid excessive passive strategies that may lead to stalemates and draws, particularly in the mid-game.

The implementation of the NFSP Chess Bot is built upon the OpenSpiel framework, which provides a multi-agent environment for handling game states, moves, and transitions in chess. The bot interacts with the chess environment through an agent-environment loop, selecting actions based on its policy, receiving rewards, and updating its knowledge base. The reward function is tailored to prioritise game outcomes, material gain, and checkmate strategies while penalizing moves that lead to disadvantageous positions.

The NFSP agent architecture consists of two main components: a policy network and a Q-network. The policy network is a supervised learning network that approximates optimal moves based on historical data. It utilises a convolutional neural network (CNN) architecture suitable for capturing the spatial nature of the chessboard. The Q-network, on the other hand, is a reinforcement learning network that updates based on temporal difference learning to adjust the bot's response to various chess positions. It shares a similar CNN architecture with the policy network but includes separate output layers for Q-value predictions.

NFSP employs two types of experience replay to facilitate learning: an SL replay buffer and an RL replay buffer. The SL replay buffer stores game states and corresponding actions observed from opponents, allowing the policy network to learn through supervised learning. The RL replay buffer contains state-action-reward-next_state tuples for training the Q-network using Deep Q-Learning (DQN). These buffers store diverse experiences to enhance learning stability and promote strategy variance.

The integration of supervised and reinforcement learning is a crucial aspect of the NFSP Chess Bot. The policy network learns from self-play and high-level games, including data from strong chess engines, enabling the bot to acquire effective opening and endgame moves. The Q-network focuses on maximising future rewards through exploration and exploitation, utilising an ϵ-greedy exploration strategy to balance between known and experimental strategies. By combining these learning paradigms, the NFSP bot can optimally adjust its strategy against diverse opponents, learning tactical, positional, and

endgame concepts.

The aggressive play-style of the NFSP Chess Bot is designed to overcome the common issue of passive play leading to frequent draws in self-play training. The bot's aggressive moves prioritise material advantage, position control, and checkmate drive. By targeting high-value pieces early in the game, dominating the board's center, and focusing on checkmate sequences, the bot reduces the likelihood of passive, draw-prone strategies and increases its chances of victory.

The training and evaluation of the NFSP Chess Bot includes self-play and simulated games against strong chess engines like Stockfish. The bot's performance is assessed using evaluation metrics such as win-draw-loss ratio, exploitability, and reward tracking. The training process spans multiple epochs, with regular validation checks to ensure consistent improvement and avoid over-fitting or lack of exploration.

IV. Experiments

We outline the baseline agents we tested against, the metrics used for evaluation, the notable hyper-parameters for models, and our experimental setup..

A. Baselines

We utilise two baselines to assess the performance of our chess bots. The first is a baseline Stockfish chess engine configured to play at a skill level of 1, which has an approximate rating [5] of 1467.6. In addition, we also include a random bot from the OpenSpiel framework. This bot makes random legal moves throughout the game, serving as a lower bound for performance.

B. Metrics

For tournament-style chess each bot was evaluated using the following metrics:

- Win rate: The percentage of games won by each bot.
- Draw rate: The percentage of games that ended in a draw.
- Elo rating: The relative skill level of each bot, calculated using the Elo rating

For puzzle-solving each bot was evaluated using the following metrics:

- Solve rate: The percentage of puzzles successfully solved by each bot.
- Move accuracy: The percentage of correct moves made by each bot compared to the optimal solution.
- Average time per puzzle: The average time taken by each bot to solve a puzzle.

C. Hyper-parameters

The specific hyper-parameters used for all MCTS variants and GA training are as follows:

1) MCTS: Exploration Constant (C): 2, Number of Simulations: 1000, Rollout Count: 3.

2) GA: Population Size: 50, Number of Generations: 1000, Games per Genome: 100, Mutation Rate: 0.1, Mutation Strength: 0.1.

[5]https://github.com/official-stockfish/Stockfish/commit/a08b8d4

D. Experimental Setup

To evaluate the performance of our chess bots in tournament-style gameplay, we conducted a round-robin tournament with the following setup:

The tournament included eight bots: MCTS, MCTS with PGN data, MCTS with puzzle data, NFSP, GA trained on PGN, GA trained on puzzles, Stockfish (baseline), and Random (baseline). The round-robin tournament ensured that each bot played against every other bot an equal number of times. Each bot played 50 games against every other bot, with 25 games as White and 25 games as Black to ensure fairness and eliminate colour bias. In total, each bot played 350 games.

To assess the puzzle-solving capabilities of our chess bots, we conducted an evaluation using a set of chess puzzles in PGN format. The same eight bots used in the tournament-style chess evaluation were used for the puzzle-solving task.

The puzzle-solving evaluation was conducted using a systematic approach. A subset of 50 puzzles were randomly selected from the dataset. Each bot attempted to solve the puzzles by making moves starting from the given position, and their moves were compared against the correct move sequence for each puzzle. The evaluation continued until either the bot successfully solved the puzzle by making all the correct moves, made an incorrect move deviating from the solution, or reached the maximum allowed moves per puzzle (set to 20). Throughout the evaluation, key metrics were recorded for each bot-puzzle pair.

V. Results & Discussion

We compare the performance of the implemented bot's from the tournament-style chess in section V-A. In section V-B, we compare the bot's performance on puzzle-solving.

TABLE I
Bot Performance Summary for Tournament-Style Chess

Bot	Total Games	Wins as White	Wins as Black	Draws	Elo Rating
MCTS	350	134	131	35	2048.5
MCTS_PGN	350	153	147	20	**2182.3**
MCTS_Puzzle	350	164	151	15	**2248.6**
NFSP	350	75	67	45	1682.7
GA	350	25	23	65	1348.9
GA_Puzzle	350	50	47	95	1488.4
Stockfish	350	148	137	25	**2118.9**
Random	350	0	0	10	1252.8

TABLE II
Bot Performance Summary for Puzzle-Solving

Bot	Solve Rate	Move Accuracy	Avg. Time per Puzzle
MCTS	0.64	0.72	12.80
MCTS_PGN	**0.76**	0.82	15.20
MCTS_Puzzle	**0.84**	0.88	14.50
NFSP	0.36	0.45	3.20
GA	0.24	0.32	2.80
GA_Puzzle	0.30	0.38	2.80
Stockfish	**0.98**	0.95	0.05
Random	0.00	0.02	0.50

A. Tournament-Style Chess Results

The tournament-style chess results, presented in Table I, provide a overall comparison of the performance of our chess bots. Most notably, the enhanced MCTS bots that leverage PGN Data and puzzle data demonstrated the highest win rates and Elo ratings, as seen in Fig. 3 and Fig. 4, than the baseline Stockfish engine and all other bots. The vanilla MCTS performed well, achieving an Elo rating above 2000, but did not manage to get to the level of it's enhanced variants.

This indicates that modifying the MCTS algorithm to leverage historical data whether it be standard chess PGN or puzzle data, enhances its decision-making capabilities and improves overall performance.

The NFSP and GA-based bots performed significantly worse than the MCTS variants and Stockfish baseline, illustrated by lower win rates in Fig. 3 and Elo ratings in Fig. 4. The GA-based bots, particularly the one trained on puzzle data exhibited higher draw rates in comparison to all other bot's, which can be seen in Fig. 5. This high draw rate could be indicative of the fact the GA-based bot's are capable of reaching advantageous positions during the game but struggle to convert those advantages into decisive wins.

These results indicate that the NFSP and GA-based bots struggled to compete effectively against the other bots in the tournament. An additional possible explanation for their suboptimal performance is that these bots may require further training and optimisation to reach their full potential.

The Stockfish baseline bot, despite being set to a lower skill level, managed to achieve respectable performance, as seen in Fig. 3 and Fig. 4 with high performance metrics. This highlights the inherent strength and well-optimised algorithms of the Stockfish engine, even at reduced skill levels.

As expected, the random bot demonstrated the weakest performance, with no wins Fig. 3 and the lowest Elo rating Fig. 4.

Another noteworthy observation from the tournament results is the superior performance of the MCTS and GA variants trained on puzzle data (MCTS_Puzzle and GA_Puzzle) compared to their counterparts trained on standard PGN data (MCTS_PGN and GA) as seen in Fig. 3 and Fig. 4. This suggests that the incorporation of puzzle-based training data can enhance the playing strength of chess bots in tournament-style gameplay.

The puzzle-trained variants' success can be attributed to the nature of the training data itself. Puzzle positions often represent critical game situations that require precise and tactical decision-making. By learning from these positions, the bots develop a keen sense of pattern recognition and the ability to identify and capitalise on key opportunities.

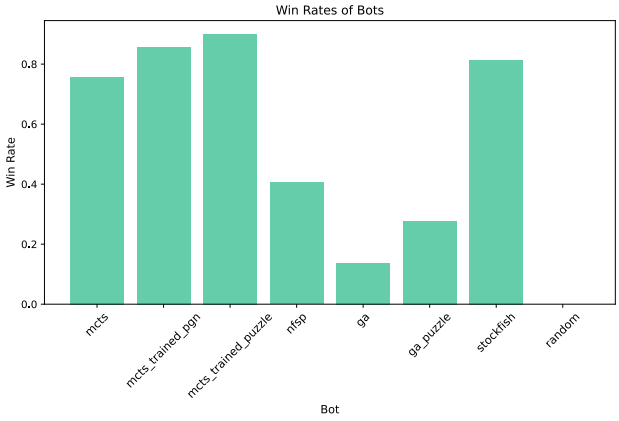

Fig. 3. Win Rates of all bots in Tournament-Style Chess.

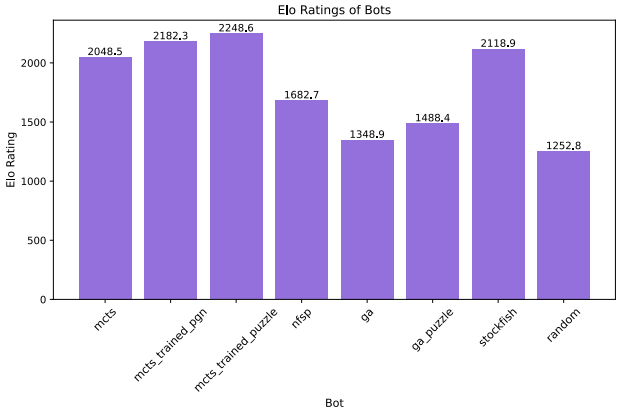

Fig. 4. Elo Ratings of all bots in Tournament-Style Chess

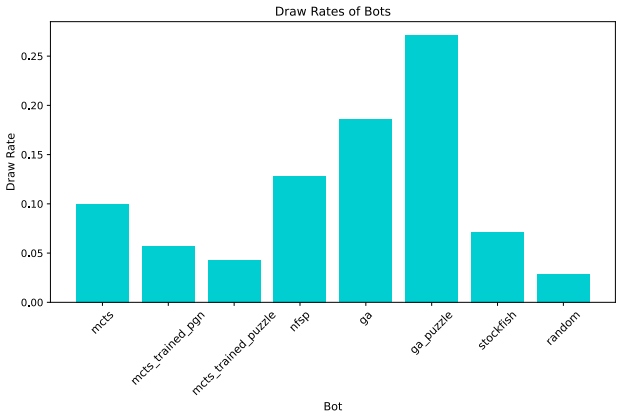

Fig. 5. Draw Rates of all bots in Tournament-Style Chess

B. Puzzle-Solving Results

The puzzle-solving results, summarised in Table II provide insights into the tactical problem-solving abilities of our implemented chess bots.

The MCTS bots, while not quite matching Stockfish's performance, still demonstrated impressive puzzle-solving capabilities. The MCTS bot that leverages PGN data and the one that leverages puzzle data achieved solve rates Fig. 6 and move accuracies Fig. 7 that were only slightly lower than Stockfish's. This indicates that the MCTS algorithm, when combined with relevant historical data, can effectively learn and apply tactical patterns to solve puzzles accurately.

The Stockfish bot excelled in the puzzle-solving task,

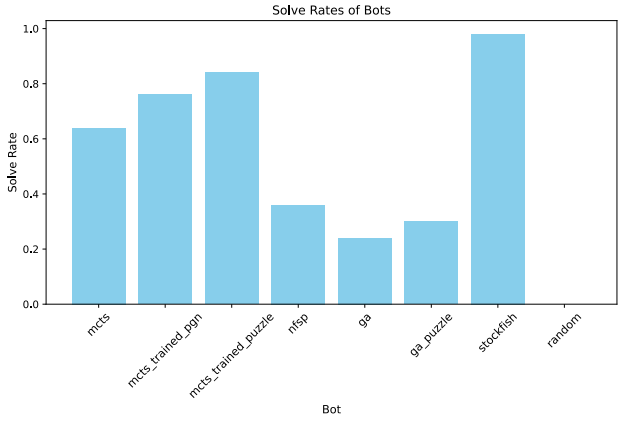

Fig. 6. Solve Rates of all bots in Puzzle-Solving

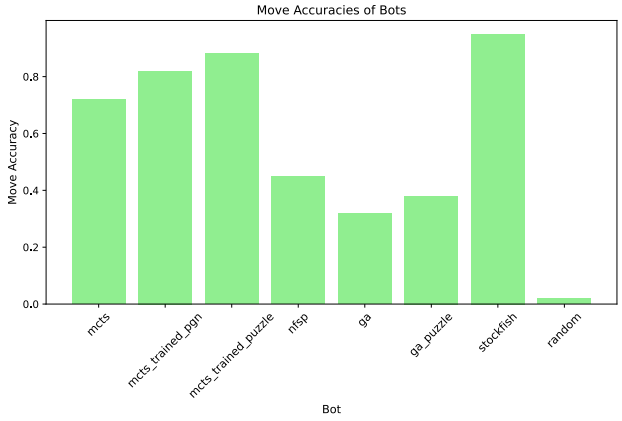

Fig. 7. Move Accuracies of all bots in Puzzle-Solving

achieving the highest solve rate and move accuracy among all the bots Fig. 6 and Fig. 7. The slightly lower performance of the MCTS variants compared to Stockfish in puzzle-solving can be partially attributed to the inherent differences in their decision-making processes. While the MCTS algorithm effectively leverages its search tree to make informed decisions, it may not be able to fully capitalize on its historical knowledge in the context of isolated puzzle positions. Stockfish, on the other hand, is specifically designed to evaluate individual positions deeply and accurately, without relying on the context of previous moves.

The NFSP and GA-based bots, while not performing as well as the MCTS bots or Stockfish, still showed reasonable puzzle-solving abilities. Their solve rates Fig. 6 and move accuracies Fig. 7 suggest that these algorithms can capture and utilise tactical knowledge to some extent.

The average time per puzzle metric Fig. 8 reveals an interesting aspect of the bots' performance. The MCTS bots, despite their high solve rates and move accuracies, required relatively longer times to solve each puzzle compared to the other bots. This can be attributed to the computationally intensive nature of the MCTS algorithm, which involves extensive simulations and tree searches.

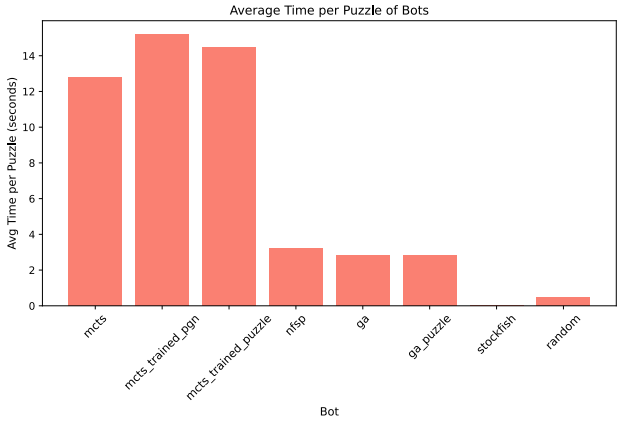

Fig. 8. Average Time per Puzzle of all bots in Puzzle-Solving

VI. Limitations & Future Work

One of the primary challenges in this research is the resource-intensive nature of the complex algorithms used in the bots. These algorithms require substantial computational resources, which can strain the project's scalability and productivity. As the complexity of the bots increases, the demand for computational power also rises, potentially limiting the extent to which the algorithms can be refined and optimised [37]. Furthermore, the bots may require more extensive training to fully harness the potential of the algorithms and improve their performance. This additional training can further strain computational resources and increase the time and effort required to develop highly advanced chess bots.

One key area for future work is the improvement of endgame play, particularly for the GA-based bots. As observed in the tournament results (Section V-A), the GA-based bots exhibited a high draw rate, indicating their ability to reach advantageous positions but struggling to convert them into wins. Further analysis of the specific positions where the GA-based bot frequently settles for draws could provide valuable insights into the limitations of its current endgame strategy. By identifying these limitations, targeted training and algorithmic enhancements can be implemented to improve the bots' endgame performance.

To enhance the endgame performance, one potential approach is the incorporation of endgame tablebases [38], which are precomputed databases containing optimal moves for specific endgame positions. By leveraging these tablebases, the bots can make more accurate and decisive moves in the endgame, increasing their chances of securing victories [39]. The integration of endgame tablebases can be particularly beneficial for the GA-based bots, as it can complement their existing training and provide additional knowledge to guide their decision-making in critical endgame scenarios.

Another area for future research is the refinement of the training data and processing. While the current study utilised PGN data and puzzle positions for training, exploring alternative datasets or combining multiple sources of training data

could potentially enhance the bots' performance. Expanding the training data to include a wider variety of game positions, player styles, and difficulty levels can expose the bots to a more diverse range of scenarios, enabling them to develop more robust and adaptable strategies [40].

VII. Conclusion

We introduced a comparative analysis of three prominent AI techniques—Monte Carlo Tree Search (MCTS), Genetic Algorithms (GA), and Neural Fictitious Self-Play (NFSP)—in the domain of chess gameplay and puzzle-solving, identifying the most effective approach for chess AI development. We evaluated these approaches using the same experimental setup and against the same baseline implementations, providing valuable insights into their relative strengths and weaknesses.

The enhanced MCTS bots which leverage historical PGN data, performed comparably well in tournament-style chess gameplay, outperforming the other bots with the highest Elo rating and Win Rate. This highlights the effectiveness of incorporating domain-specific knowledge into the training process of chess bots. Furthermore, the Stockfish baseline demonstrated the highest solve rate in puzzle-solving with the MCTS variants, especially the one trained on puzzle data, performing only slightly worse than Stockfish, indicating their strong potential in solving chess puzzles.

The insights gained from this research show that these AI techniques that have had success in other complex games, do have successful practical applications to chess AI systems. By examining these techniques in a unified framework, this research promotes a deeper understanding of their capabilities and limitations while identifying key areas for improvement.

References

[1] F. de Mesentier Silva, S. Lee, J. Togelius, and A. Nealen, "Ai-based playtesting of contemporary board games," in *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 2017, pp. 1–10.

[2] C. E. Shannon, "Xxii. programming a computer for playing chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, no. 314, pp. 256–275, 1950.

[3] I. Ghory, "Reinforcement learning in board games," *Department of Computer Science, University of Bristol, Tech. Rep*, vol. 105, 2004.

[4] P. McCorduck and C. Cfe, *Machines who think: A personal inquiry into the history and prospects of artificial intelligence*. AK Peters/CRC Press, 2004.

[5] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of research and development*, vol. 3, no. 3, pp. 210–229, 1959.

[6] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[7] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-carlo tree search: A new framework for game ai," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 4, no. 1, 2008, pp. 216–217.

[8] S. Strogatz, "One giant step for a chess-playing machine," *New York Times*, vol. 26, 2018.

[9] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[10] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[11] M. Moravčík, M. Schmid, N. Burch, V. Lisỳ, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. Bowling, "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker," *Science*, vol. 356, no. 6337, pp. 508–513, 2017.

[12] A. Hauptman and M. Sipper, "Evolution of an efficient search algorithm for the mate-in-n problem in chess," in *European Conference on Genetic Programming*. Springer, 2007, pp. 78–89.

[13] J. Heinrich and D. Silver, "Deep reinforcement learning from self-play in imperfect-information games," *arXiv preprint arXiv:1603.01121*, 2016.

[14] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk, "Monte carlo tree search: A review of recent modifications and applications," *Artificial Intelligence Review*, vol. 56, no. 3, pp. 2497–2562, 2023.

[15] D. Perez, S. Samothrakis, and S. Lucas, "Knowledge-based fast evolutionary mcts for general video game playing," in *2014 IEEE Conference on Computational Intelligence and Games*. IEEE, 2014, pp. 1–8.

[16] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*. Springer, 2006, pp. 282–293.

[17] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[18] T. Bartz-Beielstein, J. Branke, J. Mehnen, and O. Mersmann, "Evolutionary algorithms," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 4, no. 3, pp. 178–195, 2014.

[19] J. Holland, "An introductory analysis with applications to biology, control, and artificial intelligence," *Adaptation in Natural and Artificial Systems. First Edition, The University of Michigan, USA*, 1975.

[20] P. A. Vikhar, "Evolutionary algorithms: A critical review and its future prospects," in *2016 International conference on global trends in signal processing, information computing and communication (ICGTSPICC)*. IEEE, 2016, pp. 261–265.

[21] D. Beasley, D. R. Bull, and R. R. Martin, "An overview of genetic algorithms: Part 1, fundamentals," *University computing*, vol. 15, no. 2, pp. 56–69, 1993.

[22] M. Mitchell, "An introduction to genetic algorithms," 1998.

[23] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Real-time neuroevolution in the nero video game," *IEEE transactions on evolutionary computation*, vol. 9, no. 6, pp. 653–668, 2005.

[24] N. Böhm, G. Kókai, and S. Mandl, "An evolutionary approach to tetris," in *The Sixth Metaheuristics International Conference (MIC2005)*. Citeseer, 2005, p. 5.

[25] J. Baxter, A. Tridgell, and L. Weaver, "Learning to play chess using temporal differences," *Machine learning*, vol. 40, pp. 243–263, 2000.

[26] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[27] J. Heinrich, M. Lanctot, and D. Silver, "Fictitious self-play in extensive-form games," in *International conference on machine learning*. PMLR, 2015, pp. 805–813.

[28] M. Lai, "Giraffe: Using deep reinforcement learning to play chess," *arXiv preprint arXiv:1509.01549*, 2015.

[29] S. Ganzfried and T. Sandholm, "Game theory-based opponent modeling in large imperfect-information games," in *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, 2011, pp. 533–540.

[30] H. Baier and P. I. Cowling, "Evolutionary mcts for multi-action adversarial games," in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2018, pp. 1–8.

[31] J. Mycielski, "Games with perfect information," *Handbook of game theory with economic applications*, vol. 1, pp. 41–70, 1992.

[32] J. Eschmann, "Reward function design in reinforcement learning," *Reinforcement Learning Algorithms: Analysis and Applications*, pp. 25–33, 2021.

[33] S. Gelly and D. Silver, "Monte-carlo tree search and rapid action value estimation in computer go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011.

[34] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, pp. 9–44, 1988.

[35] R. S. Sutton, A. G. Barto *et al.*, "Reinforcement learning," *Journal of Cognitive Neuroscience*, vol. 11, no. 1, pp. 126–134, 1999.
[36] G. Tesauro *et al.*, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.
[37] M. M. Botvinnik, *Computers in chess: solving inexact search problems*. Springer Science & Business Media, 2013.
[38] R. Haque, T. H. Wei, and M. Müller, "On the road to perfection? evaluating leela chess zero against endgame tablebases," in *Advances in Computer Games*. Springer, 2021, pp. 142–152.
[39] E. V. Nalimov, G. Haworth, and E. A. Heinz, "Space-efficient indexing of chess endgame tables," *ICGA Journal*, vol. 23, no. 3, pp. 148–162, 2000.
[40] O. E. David, H. J. van den Herik, M. Koppel, and N. S. Netanyahu, "Genetic algorithms for evolving computer chess programs," *IEEE transactions on evolutionary computation*, vol. 18, no. 5, pp. 779–789, 2013.