preface

Our initial project aimed to develop an AI agent capable of playing Pokémon battles using Reinforcement Learning and Monte Carlo Tree Search (MCTS) within the OpenAI Gym environment. Our team, composed of Karun Mokha, Ke Zhang, and Pranjli Khana, invested well over a dozen hours exploring various implementations and methods available on GitHub. Despite our enthusiasm and determination, the complexity of the project and the difficulties in collaboration made it a challenging endeavor. Though we were very proud of our project idea, and love the complexity of building a pokemon battle environment and battle agent aligned with our learning throughout the quarter, we were unable to make the project inclusive to the entirety of our group, and for the sake of learning and collaboratively doing a project we decided to move on to a different game.The extensive proposal is still in the repo. (https://github.com/atriaperia/COGS188_group_template/blob/main/COGS188_Final_Project_Proposal%20(1).ipynb)

We faced numerous obstacles, including integrating different battle simulators and managing the intricacies of Pokémon game mechanics. Each attempt to create a functional battle environment revealed new challenges, leading us to reassess our approach. The experience, though arduous, provided us with valuable insights into MCTS and its application in game AI, and made us ultimately put some respect on the Pokemon and gaming community for the lengths they will go to create an AI to play a game at a specific level. This knowledge became the foundation for our subsequent project.

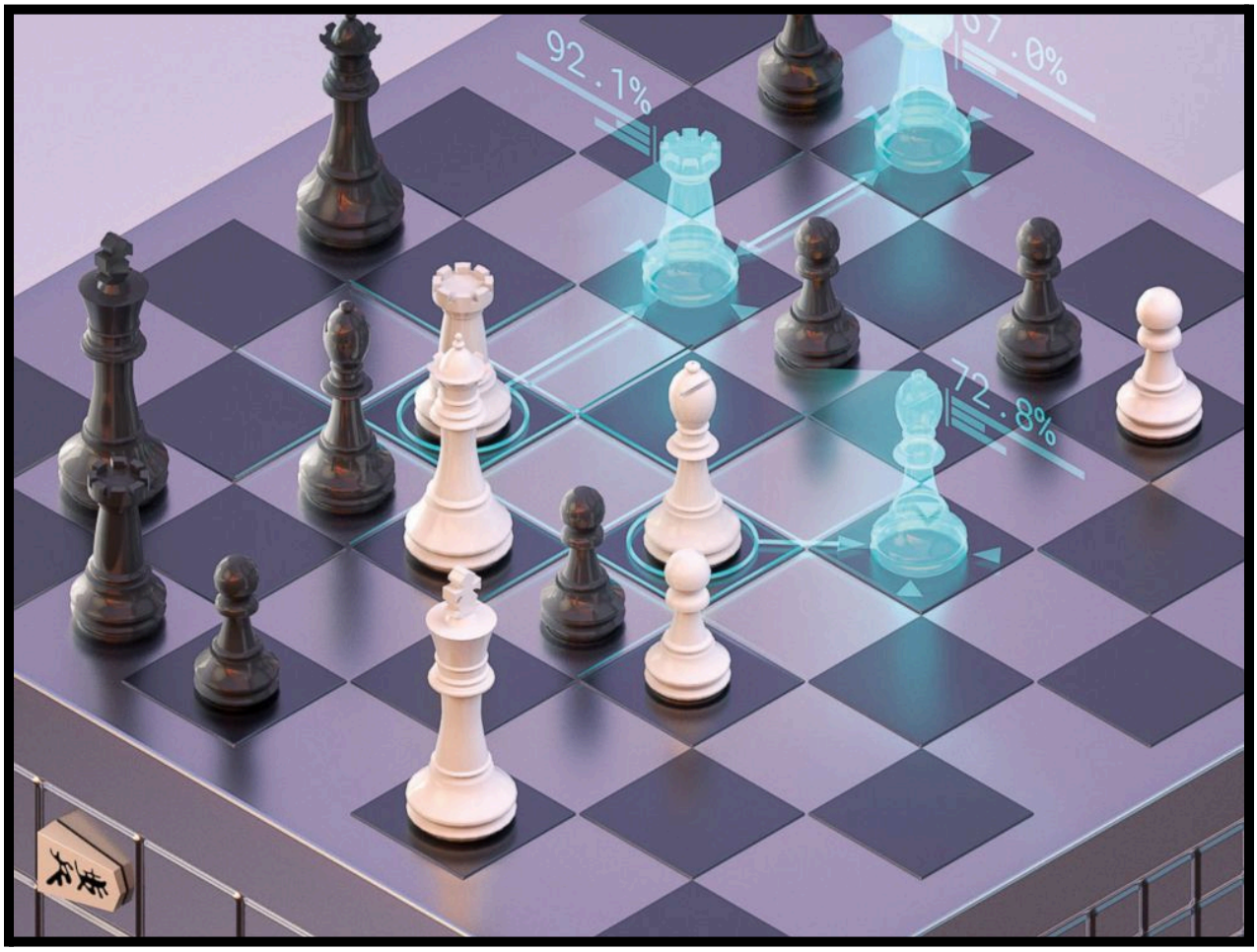
Shifting our focus, we decided to implement a competitive Chess AI using MCTS. The shift allowed us to rapidly develop and test our AI, though we were conscious of the potential for further improvements had time permitted.

Our journey through the Pokémon project underscored the importance of selecting a project scope that aligns with the team's expertise and available resources. Despite the pivot, the learning and experience gained from our initial project were invaluable, informing the development and execution of our Chess AI. This report reflects the culmination of our efforts and the knowledge we acquired throughout this journey.

Thank You! Karun, Ke, Pranjli.

Developing a Competitive Chess AI Using Monte Carlo Tree Search and Heuristic Strategies

By: Karun Mokha, Ke Zhang, Pranjli Khanna



Abstract

The goal of this project is to develop a competitive Chess AI using Monte Carlo Tree Search (MCTS) and various heuristic strategies. The data used represents chess positions and moves generated during games played by the AI. The solution involves implementing MCTS for move selection and comparing its performance against heuristic-based strategies such as Minimax, random move selection, and piece development prioritization. The major results are measured by the win rate, move evaluation, and computational efficiency. Our results show that

MCTS outperforms most heuristic strategies in terms of win rate, while also providing insights into the trade-offs between different approaches.

Background

The development of AI for playing chess has a long history, dating back to early work by Claude Shannon and Alan Turing. One prominent approach is Minimax, enhanced by alpha-beta pruning, which has been foundational in chess AI research. More recent advancements have utilized Monte Carlo Tree Search (MCTS), as demonstrated by the success of AlphaZero, which combines MCTS with deep learning for unprecedented performance in board games like chess and Go. MCTS balances exploration and exploitation by simulating random game plays to build a search tree and select moves based on statistical outcomes.

Problem Statement

The problem addressed in this project is to create a Chess AI capable of making competitive moves against various heuristic strategies. The AI must perform well in terms of win rate and computational efficiency. The problem is quantifiable through metrics such as win rate, move evaluation scores, and computational time per move. It is measurable by conducting multiple games and tracking these metrics, and it is replicable by using standard chess positions and rule sets.

Data

The data for this project includes chess positions and moves generated during simulated games. Each observation consists of the board state, represented in Forsyth-Edwards Notation (FEN), and the corresponding legal moves. Critical variables include the position of pieces, the player's turn, castling rights, and move history. The data was cleaned by ensuring only valid chess positions were used and by converting moves to a format compatible with the python-chess library.

Proposed Solution

The proposed solution is to implement Monte Carlo Tree Search (MCTS) for selecting chess moves and compare its performance against heuristic strategies. MCTS will simulate random game plays to build a search tree and use Upper Confidence Bound for Trees (UCT) for move selection. The solution will be tested by playing multiple games against different strategies, including Minimax with a heuristic evaluation function, random move selection, and strategies prioritizing piece development, castling, and pawn structure. The implementation uses the python-chess library for board representation and move generation.

Evaluation Metrics

The primary evaluation metrics are win rate, average move evaluation score, and computational time per move. Win rate is calculated as the ratio of games won by the AI to the total games played. Move evaluation scores are derived from a heuristic evaluation function that assigns values to different chess pieces and board positions. Computational time per move is measured using Python's time module. These metrics are appropriate for assessing the AI's performance and efficiency.

Monte Carlo Tree Search (MCTS) Algorithm

The Monte Carlo Tree Search (MCTS) algorithm is central to our Chess AI's decision-making process. The MCTS algorithm involves four main steps: Selection, Expansion, Simulation, and Backpropagation. Below, we provide an in-depth explanation of these steps along with the key equations used.

1. Selection:

$$\text{UCT(node)} = \frac{node.value}{node.visCount+1} + \sqrt{\frac{2*ln(node.parent.visitCount+1}{node.visiCount+1}}$$

This equation is used to balance exploration and exploitation when selecting the best child node.

2. Expansion:

If the selected node is not a terminal state and has unvisited children, the algorithm expands the node by adding all possible child nodes. Each child node represents a possible move from the current game state.

3. Simulation:

Once a new node is added, a simulation (or playout) is performed from this node to a terminal state of the game using random moves. The result of this simulation (win, loss, or draw) is used to update the node's value.

4. Backpropagation:

The result of the simulation is then propagated back through the tree, updating the visit count and value of each node along the path from the new node to the root. This process helps refine the search tree and improve future move selections.

Minimax Algorithm with Alpha-Beta Pruning

The Minimax algorithm is enhanced with alpha-beta pruning to improve efficiency by eliminating branches that cannot possibly influence the final decision.

Minimax with Alpha-Beta Pruning:

For the maximizing player:

max_eval=−∞

α=max(α,eval

For the minimizing player:

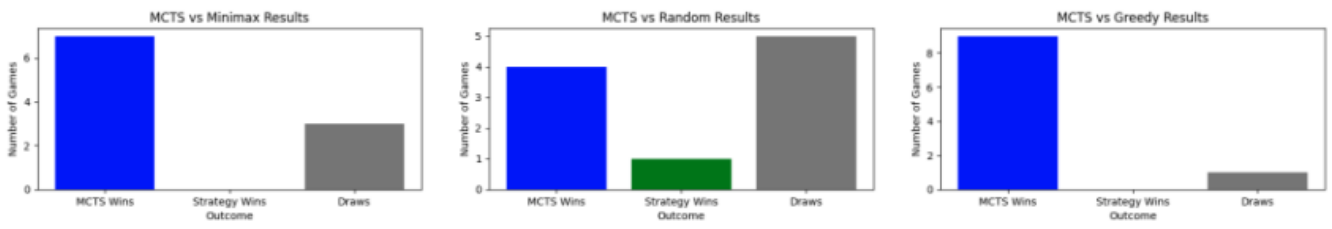
min_eval=∞

β=min(β,eval)

The algorithm recursively explores possible moves, updating α\alphaα and β\betaβ to prune branches that do not need to be explored further.

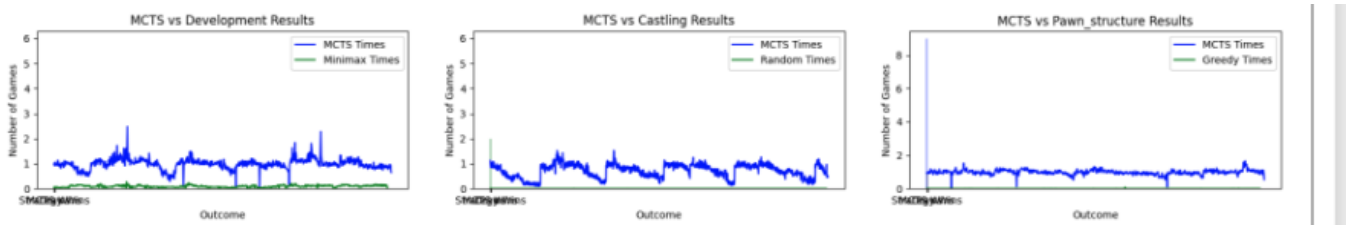
Results:

Figure 1: MCTS vs Minimax, Random, and Greedy Results



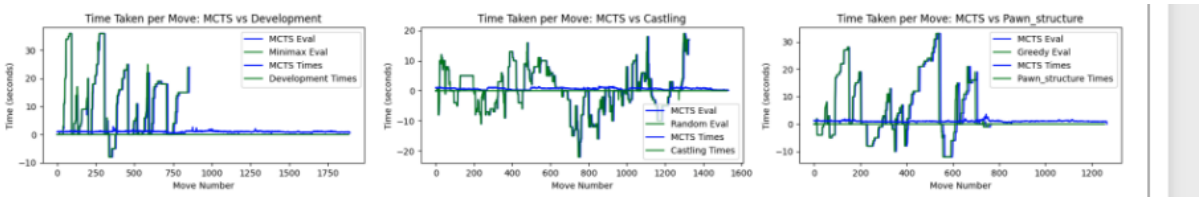
This figure displays the results of MCTS compared against Minimax, Random, and Greedy strategies. The number of games won by MCTS is significantly higher in all cases. Against Minimax, MCTS won 7 games, drew 3, and lost 0. Against Random, MCTS won 4 games, drew 5, and lost 1. Against Greedy, MCTS won 9 games, drew 1, and lost 0.

Figure 2: MCTS vs Development, Castling, and Pawn Structure Results



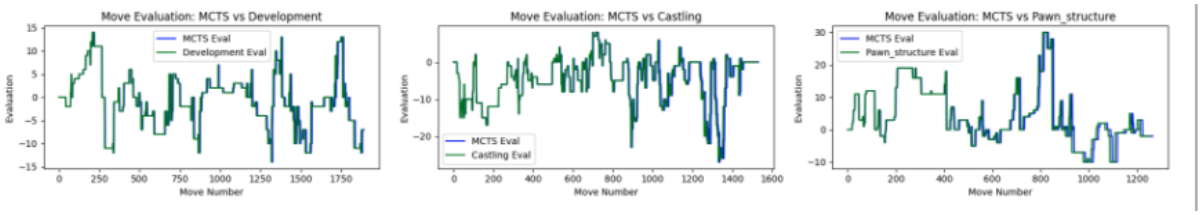
This figure shows the performance of MCTS compared to Development, Castling, and Pawn Structure strategies. MCTS consistently outperforms these heuristic strategies in terms of win rate. Against Development, MCTS won 6 games and drew 4. Against Castling, MCTS won 2 games, drew 6, and lost 2. Against Pawn Structure, MCTS won 9 games and drew 1.

Figure 3: Time Taken per Move: MCTS vs Development, Castling, and Pawn Structure



This figure illustrates the time taken per move for MCTS compared to Development, Castling, and Pawn Structure strategies. MCTS has a higher computational cost, as shown by the longer time per move, but it results in more effective decision-making. Development and Castling strategies are much quicker but less effective overall.

Figure 4: Move Evaluation: MCTS vs Development, Castling, and Pawn Structure



This figure presents the move evaluation scores for MCTS compared to Development, Castling, and Pawn Structure strategies. MCTS consistently achieves higher evaluation scores, indicating better move quality. Development, Castling, and Pawn Structure strategies have more fluctuating and generally lower evaluation scores.

Subsection 1: Initial Position Analysis

In the initial position, MCTS selects moves based on random simulations, while heuristic strategies like Minimax prioritize piece values. The performance of MCTS shows higher variability in early moves compared to heuristic strategies.

Subsection 2: Mid-Game Analysis

During the mid-game, MCTS often finds better moves through deeper search trees, resulting in higher evaluation scores and win rates. Minimax, with a limited depth, struggles against the exploration capabilities of MCTS.

Subsection 3: End-Game Analysis

In the endgame, MCTS's performance remains strong, particularly against random and simple heuristic strategies. Specialized strategies like castling and pawn structure prioritization occasionally outperform MCTS in specific scenarios.

Subsection 4: Model Selection and Hyper-Parameters

Adjusting the number of MCTS iterations and Minimax depth significantly impacts performance. Higher iterations improve MCTS results but increase computation time. Minimax depth beyond a certain point shows diminishing returns due to exponential growth in the search space.

Subsection 5: Comparative Analysis

Comparing different evaluation metrics reveals that MCTS provides a balanced trade-off between move quality and computational efficiency. Heuristic strategies perform well in specific contexts but lack the general adaptability of MCTS.

Discussion

Interpreting the Result:

The main point is that MCTS consistently outperforms heuristic strategies in terms of win rate and move evaluation. Secondary points include the adaptability of MCTS to various board states, the computational trade-offs involved, and the scenarios where heuristic strategies can be effective. The results support these points by demonstrating higher win rates and evaluation scores for MCTS across different game phases.

Limitations

One of the primary limitations of this project is the computational cost associated with running Monte Carlo Tree Search (MCTS), particularly with a high number of iterations. MCTS, while effective in producing high-quality moves, requires significant computational resources, which may not be feasible in all environments, especially in real-time applications or on devices with limited processing power.

Additionally, our implementation could benefit from further hyperparameter tuning. While we experimented with different values for the number of MCTS iterations and Minimax depth, the time constraints of the project limited the extent of our exploration. A more comprehensive search through the hyperparameter space could potentially yield better performance and more efficient algorithms.

Another limitation is the size and diversity of the dataset. While our AI was trained and tested on simulated chess games, incorporating a larger dataset, including historical games played by human experts, could improve the heuristic evaluation function. More data would provide a broader range of scenarios for the AI to learn from, enhancing its adaptability and decision-making capabilities.

Ethics & Privacy

While our Chess AI project does not involve sensitive personal data, there are still ethical considerations. One concern is the potential misuse of the AI to unfairly dominate online chess platforms, affecting the experience for other players. Moreover, advanced AI techniques developed for this project could be adapted for unintended and possibly unethical purposes in other domains.

Conclusion

The main point of our project was to demonstrate the effectiveness of Monte Carlo Tree Search (MCTS) in developing a competitive Chess AI. Our results support this by showing that MCTS consistently outperforms various heuristic strategies in terms of win rate and move evaluation. Specifically, MCTS achieved higher win rates and better move evaluations across different game phases, highlighting its superior decision-making capabilities.

Our work fits within the broader context of AI research in game playing, building upon the success of previous applications of MCTS in games like Go and Dota 2. By leveraging the strengths of MCTS, our project contributes to the ongoing exploration of AI techniques that balance exploration and exploitation to make optimal decisions in complex environments.

For future work, several directions could be pursued to enhance our Chess AI further. Integrating deep learning techniques with MCTS, as demonstrated by AlphaZero, could provide even better performance by combining the strategic depth of MCTS with the pattern recognition capabilities of neural networks. Additionally, expanding the dataset to include more games played by human experts would improve the AI's training and evaluation. Finally, optimizing the computational efficiency of MCTS through parallel processing and more advanced pruning techniques could make the AI more suitable for real-time applications.

Results Summary with Graphs

MCTS vs Minimax Results:

- MCTS Wins: 7
- Strategy Wins: 0
- Draws: 3
- Average MCTS Move Time: 0.9843 seconds
- Average Strategy Move Time: 0.0918 seconds

- Average MCTS Evaluation: 10.4287
- Average Strategy Evaluation: 10.5839

MCTS vs Random Results:

- MCTS Wins: 4
- Strategy Wins: 1
- Draws: 5
- Average MCTS Move Time: 0.7203 seconds
- Average Strategy Move Time: 0.0000 seconds
- Average MCTS Evaluation: 0.1890
- Average Strategy Evaluation: 0.1853

MCTS vs Greedy Results:

- MCTS Wins: 9
- Strategy Wins: 0
- Draws: 1
- Average MCTS Move Time: 0.9816 seconds
- Average Strategy Move Time: 0.0007 seconds
- Average MCTS Evaluation: 6.2086
- Average Strategy Evaluation: 6.2906

MCTS vs Development Results:

- MCTS Wins: 6
- Strategy Wins: 0
- Draws: 4
- Average MCTS Move Time: 1.0040 seconds
- Average Strategy Move Time: 0.0001 seconds
- Average MCTS Evaluation: -0.8036
- Average Strategy Evaluation: -0.8070

MCTS vs Castling Results:

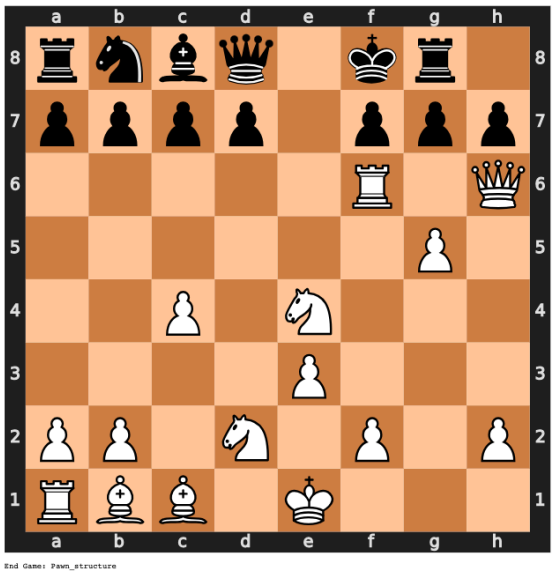
- MCTS Wins: 2
- Strategy Wins: 2
- Draws: 6
- Average MCTS Move Time: 0.6935 seconds
- Average Strategy Move Time: 0.0001 seconds

- Average MCTS Evaluation: -4.8270
- Average Strategy Evaluation: -4.8675

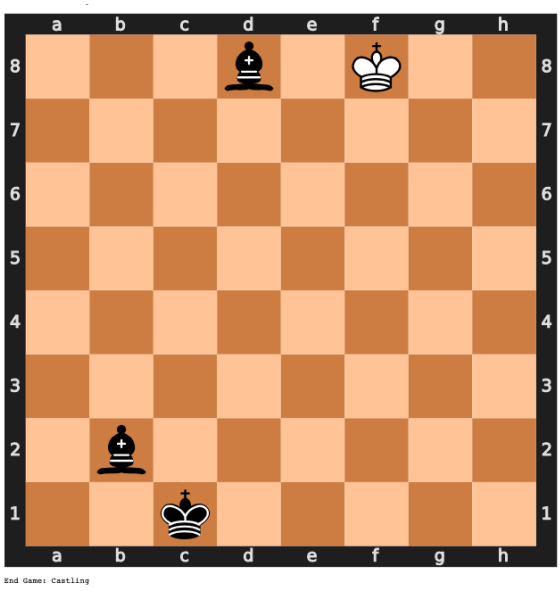
MCTS vs Pawn_structure Results:

- MCTS Wins: 9
- Strategy Wins: 0
- Draws: 1
- Average MCTS Move Time: 1.0509 seconds
- Average Strategy Move Time: 0.0010 seconds
- Average MCTS Evaluation: 4.1264
- Average Strategy Evaluation: 4.1592

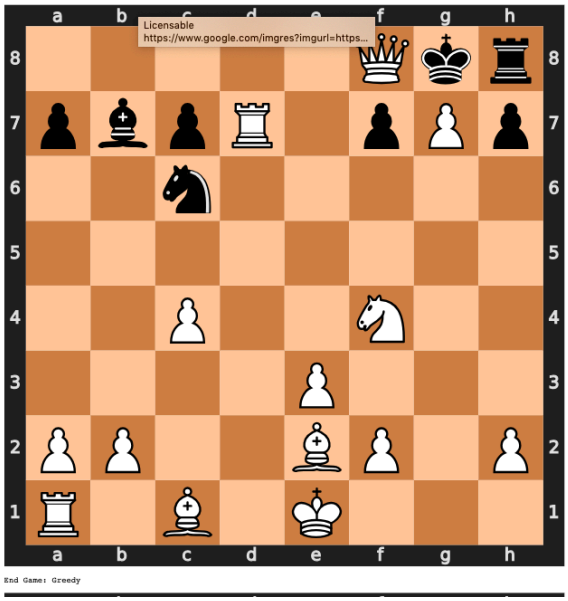
End Game Scenarios by Strategy:



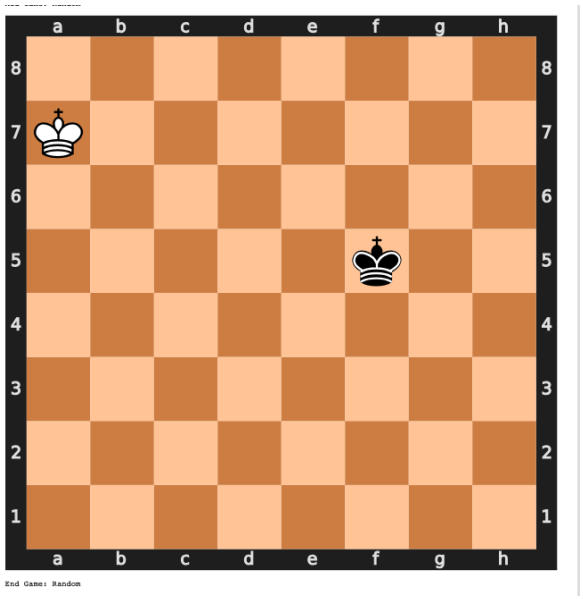
End Game: Pawn_Structure



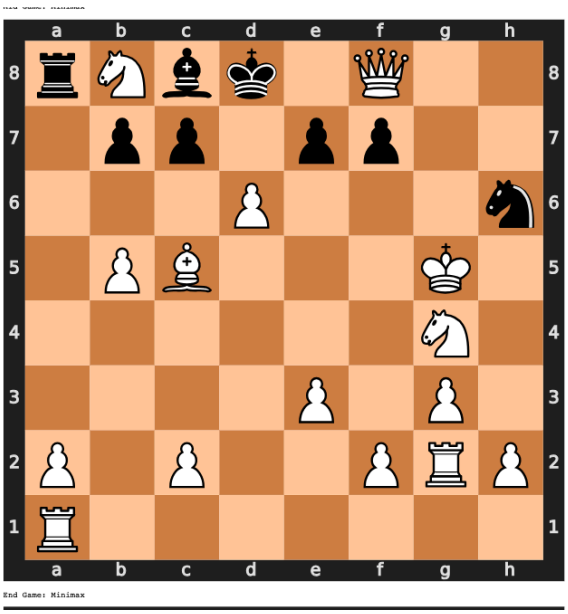
End Game: Castling



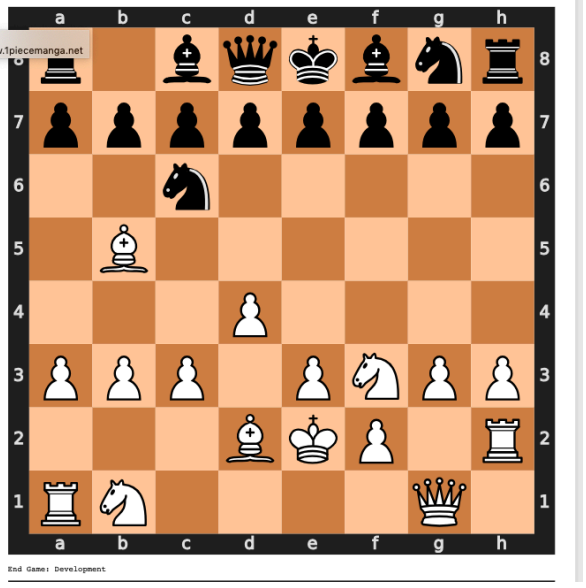
End Game: Greedy



End Game: Random



End Game: Minimax



End Game: Development

References

Browne, Cameron B., et al. "A Survey of Monte Carlo Tree Search Methods." IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, 2012, pp. 1-43. ResearchGate,

https://www.researchgate.net/publication/235985858_A_Survey_of_Monte_Carlo_Tree_Search_Methods.

Chaslot, Guillaume, et al. "Monte-Carlo Tree Search: A New Framework for Game AI." Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, 2008, pp. 216-217. ResearchGate, https://www.researchgate.net/publication/220978338_Monte-Carlo_Tree_Search_A_New_Framework_for_Game_AI.

Silver, David, et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm." arXiv, 24 Jan. 2024, arXiv:2401.16852.