Benchmarking Minimax and MCTS Chess Engines Against Stockfish

Alex Tang **Eric Ge** **Guru Balamurugan** **Kyler Chen**

Abstract

This report presents a performance benchmark of two custom chess engines, Minimax (with Alpha-Beta pruning) and Monte Carlo Tree Search (MCTS), against the Stockfish engine at ELO strengths 1320, 1600, 2000, and 2400. The project architecture features a Python backend with FastAPI serving engine logic and a React frontend. The Minimax engine uses a heuristic evaluating material, piece-square tables, and mobility. The MCTS engine uses a fixed simulation count and time limit. Benchmarking revealed significant performance challenges for both engines. Minimax achieved a 20% win rate only against the 1320 ELO Stockfish, failing against higher ratings. MCTS lost all games across all ELO levels. These results underscore the gap between the current implementations and established engines, highlighting limitations in search depth, simulation budget, and heuristic sophistication.

1 Introduction

Developing strong chess-playing AI remains a classic challenge. This project involved creating two engines: one using Minimax with Alpha-Beta pruning [Knuth and Moore, 1975], a standard game tree search algorithm, and another using Monte Carlo Tree Search (MCTS) [Browne et al., 2012], a probabilistic approach widely used in games like Go. To gauge their practical strength, we benchmarked them against the powerful open-source engine Stockfish [Stockfish developers, 2024], configured to play at various ELO levels. This provides a quantitative measure of performance against calibrated opposition.

2 Software Architecture and Experimental Design

2.1 Software Architecture

The project is structured into a backend engine and API, a frontend user interface, and benchmarking scripts.

- **Backend (Python):** Built using the FastAPI framework, the backend provides a REST API (‘api.py‘) to manage game state and interact with the chess engines. It utilizes the ‘python-chess‘ library [python-chess developers, 2024] for board representation, move generation, and game state evaluation.
- **Game Logic (‘src/game‘):** Encapsulates the core chess rules and game progression, leveraging ‘python-chess‘.
- **Engines (‘src/engine‘):** Contains the implementations of the chess engines. Both ‘MinimaxEngine‘ and ‘MCTSEngine‘ inherit from an abstract ‘BaseEngine‘ class, ensuring a consistent interface (‘get_best_move‘, ‘get_evaluation‘).

- **Frontend (React):** A web-based interface built with React and Vite allows users to play against the selected engine via the backend API. It displays the board (‘react-chessboard‘) and handles user input. (Note: The frontend was not used during the automated benchmark.)
- **Benchmarking (‘benchmark.py‘):** A separate Python script designed specifically for performance evaluation. It directly instantiates the engine classes and uses ‘python-chess‘’s engine communication capabilities (UCI protocol) to interact with Stockfish, orchestrating the matches and collecting results.

This modular architecture separates concerns, allowing engine logic, API handling, user interaction, and benchmarking to be developed and tested independently.

2.2 Experimental Design

The benchmark was designed to evaluate the two custom engines against Stockfish under controlled conditions.

- **Engines Under Test:**
  - *Minimax*: Depth-3 search with Alpha-Beta pruning. Heuristic based on material (Sunfish values), piece-square tables (Sunfish PSTs), and a simple mobility bonus (+1 per net legal move advantage).
  - *MCTS*: 500 simulations/move, 10-ply max rollout depth, 1 sec/move time limit. Random rollouts with material evaluation.
- **Benchmark Opponent:** Stockfish engine, strength-limited via UCI options to ELO 1320, 1600, 2000, and 2400. Stockfish search depth also limited to 3 plies for comparison.
- **Match Setup:** For each (engine, ELO) pair, 5 games were played using ‘benchmark.py‘, alternating colors between games.
- **Metrics:** The primary metric was the engine’s win rate against Stockfish. Wins, draws, losses, and a 95% confidence interval (CI) for the win rate were calculated.

3 Results

Table 1 summarizes the benchmark outcomes. The Minimax engine showed limited success only at the lowest ELO level, while the MCTS engine performed poorly across all levels.

Table 1: Benchmarking Results: Engine Performance vs. Stockfish ELO ($n = 5$ games per setting)

Engine	Stockfish ELO	Wins	Draws	Losses	Win Rate	95% CI
Minimax	1320	1	2	2	0.20	0.351
Minimax	1600	0	0	5	0.00	0.000
Minimax	2000	0	0	5	0.00	0.000
Minimax	2400	0	0	5	0.00	0.000
MCTS	1320	0	0	5	0.00	0.000
MCTS	1600	0	0	5	0.00	0.000
MCTS	2000	0	0	5	0.00	0.000
MCTS	2400	0	0	5	0.00	0.000

Figure 1 visualizes these win rates. The Minimax engine’s 20% win rate against ELO 1320 drops to 0% against stronger opponents. The MCTS engine consistently shows a 0% win rate. Note the large CI at 1320 ELO due to the small sample size.

4 Discussion

The benchmark results clearly show that both engines are significantly outperformed by Stockfish, even at reduced strength levels.

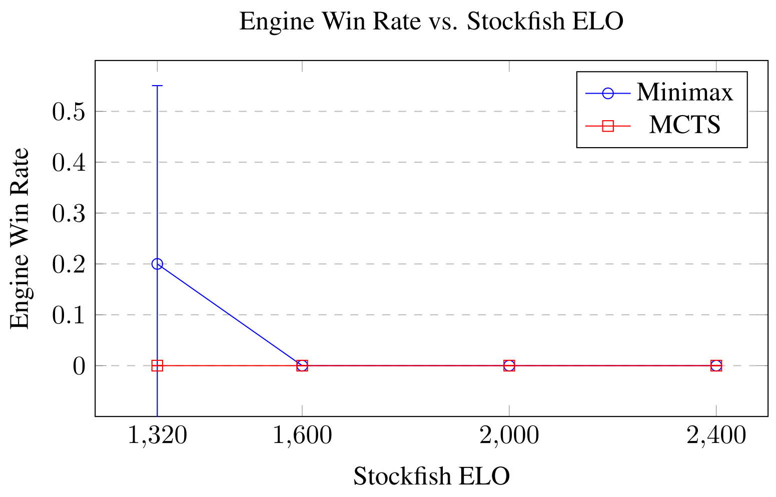

Figure 1: Win rate of Minimax and MCTS engines against Stockfish at different ELO levels (n=5). Error bars represent 95% CI.

The Minimax engine's depth-3 search is its primary limitation. Seeing only 1.5 moves ahead is insufficient for strategic play or avoiding deeper tactics. Its heuristic, while incorporating material, PSTs, and basic mobility, is relatively simple compared to state-of-the-art engines. It lacks sophisticated features like detailed pawn structure analysis, king safety evaluation, or dynamic adjustments based on game phase (beyond what's implicit in PSTs). This combination of shallow search and a basic heuristic explains its inability to compete beyond the lowest ELO tested.

The MCTS engine's complete failure suggests its parameters (500 simulations, 1 sec/move) were inadequate for chess. MCTS often requires vastly more simulations than used here to be effective, especially with a simple random rollout policy and a material-only evaluation. Chess requires precise tactical calculation, which is challenging for MCTS with a limited budget compared to Alpha-Beta's exhaustive search (within its depth).

Overall, the performance gap highlights the need for deeper search (Minimax), significantly more simulations or smarter rollouts (MCTS), and more sophisticated, chess-specific knowledge encoded in the evaluation functions or search guidance.

5 Conclusion and Future Work

Our Minimax and MCTS chess engines currently lack the playing strength to compete effectively against Stockfish, even at low ELO settings. Key limitations include shallow search depth (Minimax), insufficient simulations and basic rollouts (MCTS), and relatively simple heuristic evaluations in both.

Future improvements should prioritize:

- **Minimax:** Increasing search depth (using iterative deepening, quiescence search), refining the heuristic (adding king safety, pawn structure, etc.).
- **MCTS:** Dramatically increasing simulation count/time, implementing heuristic rollout policies instead of random moves, potentially integrating neural network evaluations (like AlphaZero).
- **Both:** Incorporating opening books and potentially endgame tablebases.
- **Benchmarking:** Using a larger number of games for more statistically reliable results.

Addressing these areas is essential for developing more competitive engines. Exploring reinforcement learning to tune heuristics remains a potential future direction once baseline engine performance is improved.

References

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*, 4(1):1–43, 2012.

K. Hoki and T. Kaneko. Computer Shogi: Creating Sophisticated Playing Programs. *IEICE Transactions on Information and Systems*, E96-D(11):2332-2338, 2013.

D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4): 293–326, 1975.

python-chess developers. python-chess library. `https://python-chess.readthedocs.io/`, 2024. Accessed: April 29, 2025.

Stockfish developers. Stockfish engine. `https://stockfishchess.org/`, 2024. Accessed: April 29, 2025.