International Journal on Engineering Technology and Infrastructure Development (InJET-InDev)
www.lec.edu.np/journal
Volume 2, Issue no. 2, Page 171-179
Received Date: 11/30/2025
Revised Date: 5/12/2026
Accepted Date: 5/24/2026
ISSN: 3021-999X (Print)
https://doi.org/10.3126/injet-indev.v2i2.95724

Empirical Comparison of Negamax with Alpha-Beta Pruning and Monte Carlo Tree Search for Mobile Chess AI on Flutter

Sagar Chaudhary[1], Sudip Adhikari[2], Pralhad Chapagain[3]

[1]*DAV College, Lalitpur, Nepal sagar11cc@gmail.com*
[2]*DAV College, Lalitpur, Nepal, sudip.adhikari48@gmail.com*
[3]*Kantipur Engineering College, Lalitpur, Nepal, pralhadchapagain@kec.edu.np*

Abstract

This paper implements and compares two chess artificial intelligence algorithms, Negamax with Alpha-Beta Pruning and Monte Carlo Tree Search (MCTS), within a cross-platform Flutter-based mobile chess application. The system employs advanced heuristic evaluation techniques incorporating material balance, positional analysis, king safety, pawn structure, and piece-square tables. Negamax utilises deterministic search with iterative deepening, Alpha-Beta pruning, quiescence search, and transposition tables, while MCTS performs probabilistic exploration using Upper Confidence Bound for Trees (UCT), progressive bias, and heuristic guided simulations. To evaluate performance under mobile hardware constraints, a benchmark framework executed 40 automated AI vs AI games across multiple predefined opening systems under identical runtime conditions on a CMF Phone 1 device using a fixed 5 second computation budget per move. Experimental results indicate that Negamax achieved substantially lower average move response times (0.94 ± 0.21 s) compared to MCTS (2.81 ± 0.52 s), while also reaching deeper average search depths and higher win rates. The findings suggest that Alpha-Beta-based search remains highly effective for resource-constrained mobile chess applications requiring efficient real-time decision-making, whereas MCTS provides broader probabilistic exploration at higher computational cost.

Keywords: Chess AI, Negamax Algorithm, Monte Carlo Tree Search, Alpha-Beta Pruning, UCT, Flutter, Game AI, Algorithm Comparison, Position Evaluation

1. Introduction

Chess is a strategic board game with centuries of history, renowned for its intellectual depth, structured rules, and virtually limitless strategic possibilities. Played on a checkered board of 64 squares, each player commands 16 pieces with the ultimate objective of checkmating the opposing king. The combination of simple rules and immense combinatorial complexity has made chess not only a cultural and recreational staple but also a long-standing benchmark for artificial intelligence (AI) research (Turing, 1953).

The development of chess engines has significantly shaped the field of AI, with algorithms such as Minimax, Alpha-Beta Pruning, and Monte Carlo Tree Search (MCTS) enabling computers to evaluate vast search trees, simulate expert-level gameplay, and make near optimal decisions under time constraints (Fredlund & Wigforss, 2025). Enhancements move ordering, transposition tables based on Zobrist hashing, (Zobrist, 1970)and quiescence search have further optimized computational efficiency and improved the quality of move selection in complex positions (Knuth & Moore, 1975). Notably, classical engines such as Fritz, Rybka, and Stockfish (Developers S. , 2025) exemplify the refinement of these techniques through efficient data structures like bitboards and multi-threaded processing. More recent AI-driven systems such as AlphaZero (David Silver, 2018) employ deep reinforcement learning and self-play to achieve unprecedented playing strength, introducing alternative paradigms for decision making in chess AI.

Parallel to advances in AI, modern software engineering has enabled the development of cross-platform chess applications that combine computational power with user-friendly design. Frameworks such as Flutter (Google, 2025) and game engines like Flame (Developers F. E., 2025) provide the tools necessary for delivering responsive,

visually appealing interfaces with seamless functionality across mobile, desktop, and web platforms. These technologies support features such as intuitive touch controls, real-time move validation, customizable themes, and performance optimized rendering, making chess accessible to both casual players and competitive enthusiasts.

This study aims to develop a mobile chess application that integrates robust AI algorithms with contemporary design principles to provide an engaging, adaptable, and educational chess experience. The system supports gameplay against both human and AI opponents, offering adjustable difficulty levels and customizable visual settings. By implementing classical search algorithms and well-designed evaluation functions, the application ensures challenging yet realistic gameplay while maintaining high performance across devices. Beyond its entertainment value, the study serves as a practical demonstration of AI algorithms in a real-world application. It offers insights into game theory, heuristic optimization, and algorithmic decision-making while also showcasing modern software engineering practices. Through iterative development and testing, the study aspires to blend the tradition of chess with the innovation of mobile technology, providing a practical framework for future mobile chess applications that balance strategic depth, computational efficiency, and user experience (Helfenstein, Blüml, Czech, & Kersting, 2024).

2. Related Work

The foundational work by Shannon (Shannon, 1950) and Turing (Turing, 1953) established the core principles of programmatic chess play and adversarial search, introducing minimax-style reasoning and early adversarial search techniques and the early conception of programmatic evaluation functions. These seminal studies framed chess as a benchmark problem for artificial intelligence and motivated decades of research into efficient tree search and heuristics. Subsequent engineering efforts produced high-performance classical engines such as Stockfish (Developers S. , 2025), which demonstrate how optimized alpha beta search, move ordering, bitboard representations, Zobrist hash-based transposition tables, and move ordering techniques (Zobrist, 1970) can deliver very strong, deterministic play on modern hardware.

More recent research has explored alternative paradigms and hybrid approaches. AlphaZero (David Silver, 2018)demonstrated that reinforcement learning and self-play combined with powerful neural evaluation networks can reach or exceed traditional engine strength without hand-crafted heuristics. Comparative studies by Fredlund and Wigforss (2025) examine the practical trade-offs between Monte Carlo Tree Search (MCTS) and Alpha-Beta Pruning (ABP) in chess, reporting that ABP with a strong static evaluator typically outperforms MCTS under tight time or resource constraints, while MCTS can gain advantages given substantially larger sampling budgets. The UCT-based selection mechanism commonly used in MCTS was originally proposed by Kocsis and Szepesvári (2006), enabling balanced exploration and exploitation during tree traversal (Kocsis & Szepesvári, 2006).

Methodological improvements to classical search continue to evolve in the literature. Enhancements such as improved move ordering, iterative deepening, and quiescence search have been widely shown to reduce tactical search errors and improve engine stability (Knuth & Moore, 1975). Hybrid approaches that integrate learning-based components with sampling-based search, such as mixtures of expert models combined with Monte Carlo Tree Search (Helfenstein, Blüml, Czech, & Kersting, 2024), further demonstrate the potential of combining heuristic evaluation with probabilistic exploration to improve decision quality.

From a systems and application perspective, modern cross-platform frameworks such as Flutter and engines like Flame (Developers F. E., 2025) enable developers to deliver responsive, mobile-friendly chess clients that integrate sophisticated engines while preserving good UX on resource-constrained devices. This body of work motivates the present study's focus on implementing and empirically comparing Negamax/Alpha–Beta and MCTS within a mobile-oriented Flutter/Flame application, assessing practical tradeoffs in speed, node-efficiency, and positional strength under realistic time limits.

Although previous studies have explored classical Alpha-Beta search, Monte Carlo Tree Search, and hybrid neural approaches, limited research has examined their comparative behaviour within resource constrained mobile chess applications developed using Flutter-based frameworks. Therefore, the present study focuses on implementing and empirically comparing Negamax with Alpha-Beta pruning and heuristic-guided MCTS within a cross-platform mobile environment under fixed runtime constraints.

3. Methodology

3.1 Dynamic Modelling using State Diagrams

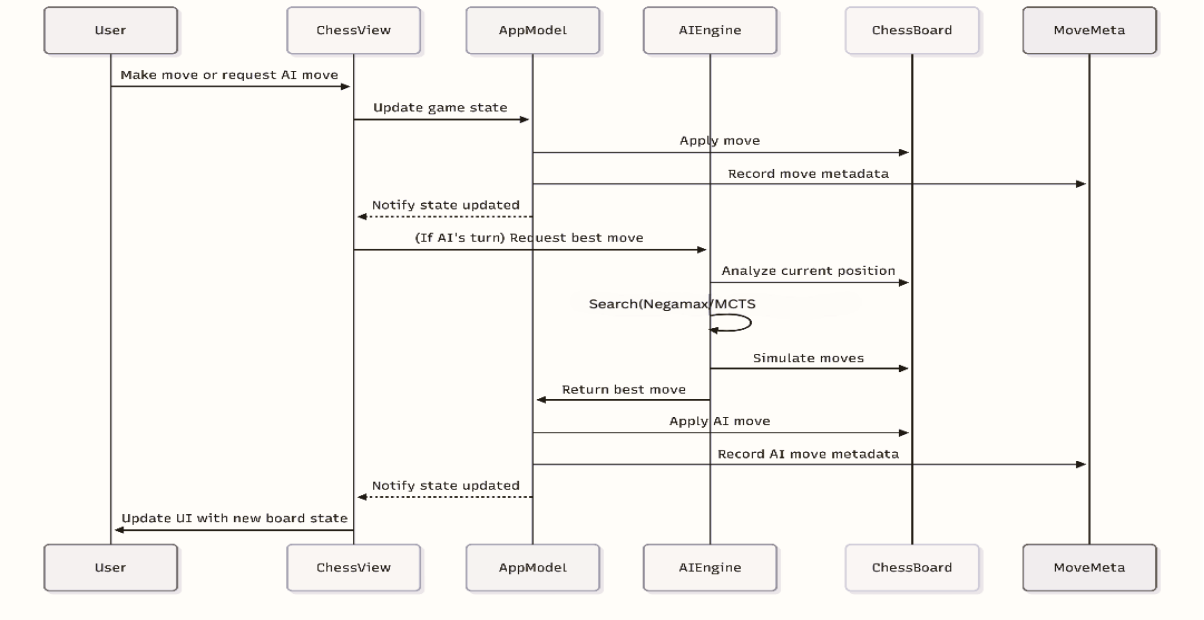

Figure 1. Sequence Diagram

The process begins when the user makes a move or requests an AI move via the 'ChessView'. The move is processed and the game state is updated by the 'AppModel', which then communicates with the 'AIEngine'if it is the AI's turn. The 'AIEngine 'analyzes the current board position using search algorithms like MCTS or Negamax, simulates possible moves, and selects the optimal move. This move is then applied to the 'ChessBoard', and metadata related to the move is recorded in the 'MoveMeta'module. Once the AI move is executed, the updated state is notified to the 'ChessView', which in turn updates the UI to reflect the new board state for the user.

3.2. Algorithm Description

3.2.1. Monte Carlo Tree Search (MCTS) Algorithm

Monte Carlo Tree Search (MCTS) is a sampling-based search algorithm that balances exploration and exploitation during move selection. In this study, a heuristic-guided variant of MCTS is implemented to improve decision quality under mobile resource constraints. Unlike classical MCTS, which relies on random rollouts, the simulation phase in this implementation employs heuristic evaluation to guide playout decisions and improve move selection quality.

$$UCT = \frac{wins}{visits} + C.\sqrt{\frac{\ln\left(parent\_visits\right)}{visits + 10^{-6}}} + \lambda.\,\text{static\_eval} \qquad \text{(Equation 1)}$$

Where w_i represents the number of wins for node i, n_i denotes the visit count of node i, N is the visit count of the parent node, C is the exploration constant set to 1.0, λ is the progressive bias coefficient set to 0.15, static_evel= advancedBoardValue(board) / 100.0.

During the selection phase, the algorithm traverses the search tree from the root node by selecting child nodes according to the modified UCT value. If a node has not been previously visited, it is selected directly for simulation; otherwise, the algorithm proceeds to the expansion phase. During expansion, all legal moves from the selected position are generated and ordered using the advancedMoveScore heuristic evaluation function. Newly generated child nodes are stored within a transposition table using Zobrist hashing to minimise redundant board-state evaluation. In the simulation phase, a heuristic-guided playout is executed up to a fixed simulation depth (playoutDepth). Unlike classical MCTS implementations that rely on random move selection, the proposed

approach prioritises moves that maximise advancedBoardValue, forming a hybrid MCTS variant that combines probabilistic tree exploration with deterministic positional evaluation.

The simulation outcome is converted into a reward value according to the following conditions:

- Evaluation > 100 → reward = 1.0
- Evaluation between -100 and 100 → reward = 0.5
- Evaluation < -100 → reward = 0.0

During backpropagation, node visit counts and accumulated win statistics are updated recursively along the traversal path back to the root node. The search process operates under a fixed computation budget of 5 seconds per move. After completion of the search cycle, the legal move associated with the highest root-node visit count is selected as the final move.

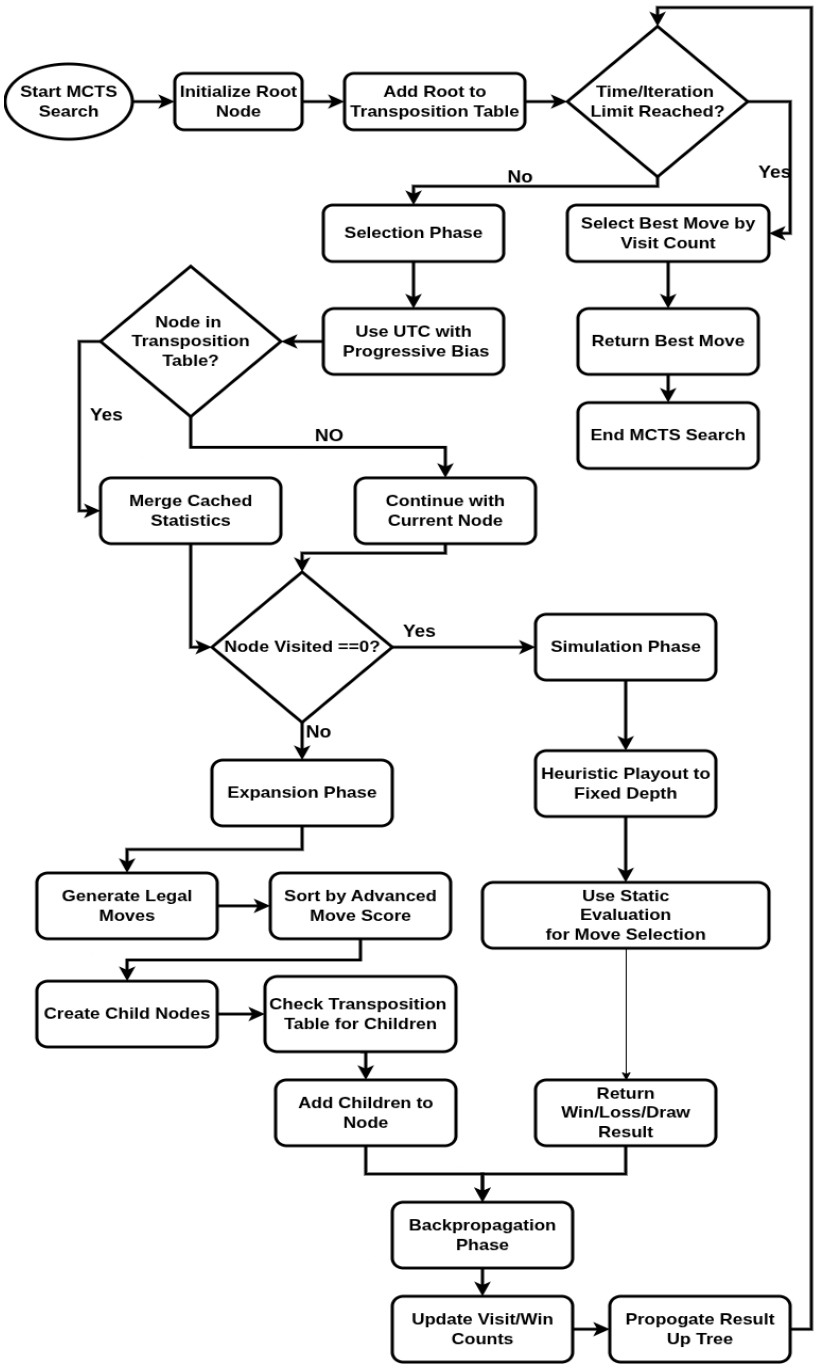

Figure 2. MCTS Algorithm Flowchart

3.2.2. Negamax with Alpha Beta Pruning

The study employs a Negamax-based search algorithm enhanced with alpha-beta pruning to efficiently evaluate chess positions in a mobile environment. Negamax is a simplified variant of the Minimax algorithm that exploits the zero-sum property of two-player games, allowing both maximizing and minimizing behaviour to be represented using a single recursive function. The core Negamax formulation is given by:

$$Negamax(board, depth, \alpha, \beta) = \max max_{move\ \epsilon\ legalMoves}\ (-Negamax(board', depth-1, -\beta, -\alpha))\ )) \quad \text{(Equation 2)}$$

where board′ represents the resulting board state after applying a legal move, depth denotes the remaining search depth, and α and β represent the lower and upper pruning bounds used in alpha-beta pruning. To improve performance under real-time mobile constraints, the algorithm employs iterative deepening, beginning from depth 1 and progressively increasing to the maximum depth defined by the selected difficulty level. A fixed computation budget of 5 seconds per move is enforced to maintain responsiveness during gameplay. Several optimisation techniques are integrated to enhance search efficiency. A transposition table based on Zobrist hashing stores previously evaluated positions to avoid redundant computation. Move ordering is performed using the MVV-LVA (Most Valuable Victim – Least Valuable Attacker) heuristic, prioritising tactically significant moves such as captures, checks, and promotions to improve pruning effectiveness. Additionally, quiescence search is applied at terminal nodes to reduce the horizon effect by selectively extending tactically unstable positions. The evaluation function combines multiple heuristics, including material balance (pawn = 100, knight = 320, bishop = 330, rook = 500, queen = 900, king = 20000), piece-square tables, king safety assessment, and pawn structure analysis. The engine also incorporates an opening book for early-game optimisation and includes rule-based handling for draw conditions such as threefold repetition, the fifty-move rule, and stalemate detection. Difficulty levels ranging from 1 to 6 control the maximum search depth, providing a trade-off between playing strength and computational cost. During search, the algorithm recursively generates and evaluates legal moves while applying alpha-beta pruning to eliminate branches that cannot improve the final decision. Previously evaluated positions are retrieved from the transposition table to avoid redundant evaluation. When the maximum search depth is reached, quiescence search selectively extends tactical positions involving captures, checks, and promotions to produce more stable evaluations. The final move is selected based on the highest evaluation returned from the root search node after the search completes. This implementation follows classical Negamax with alpha-beta pruning as described in traditional game-tree search literature while incorporating additional engineering optimisations for efficient execution in resource-constrained mobile environments.

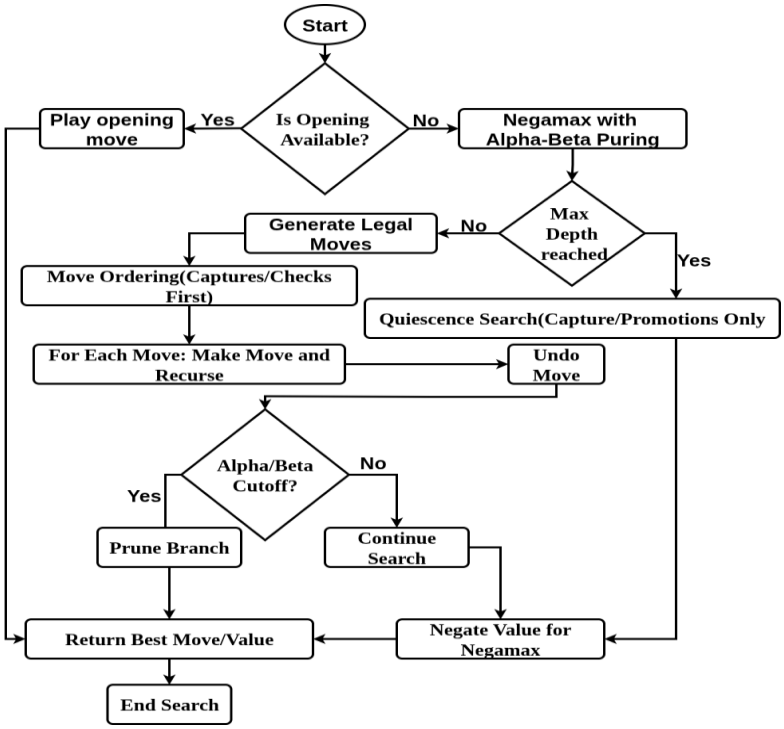

Figure 3. Negamax with Alpha-Beta Pruning: Algorithm Flowchart

3.3 Tests and Performance Analysis

Testing was conducted on a CMF Phone 1 equipped with a MediaTek Dimensity 7300 processor, 6 GB RAM, and Android 15 using Flutter release mode configuration. To ensure fair comparison, both Negamax with Alpha-Beta pruning and MCTS were evaluated under identical conditions with a fixed 5-second computation budget per move. All experiments were executed using Flutter release mode to minimise runtime overhead and improve measurement consistency.

A dedicated benchmarking module was integrated into the application to automate AI vs AI testing and runtime analysis. The benchmark system executed 40 games using multiple predefined opening positions, including the Ruy Lopez, Italian Game, Sicilian Defense, French Defense, Queen's Gambit, London System, King's Indian Defense, and additional common opening systems implemented within the benchmark framework. Opening positions were selected randomly at the start of each game to improve positional diversity and reduce opening bias.

Both engines alternated between white and black pieces throughout the benchmark process to minimise first-move advantage and improve result consistency. For each move, the benchmark framework recorded move computation time, explored search nodes, average search depth, evaluation score, and game outcome. The collected benchmark data was recorded for analysis and visualization.

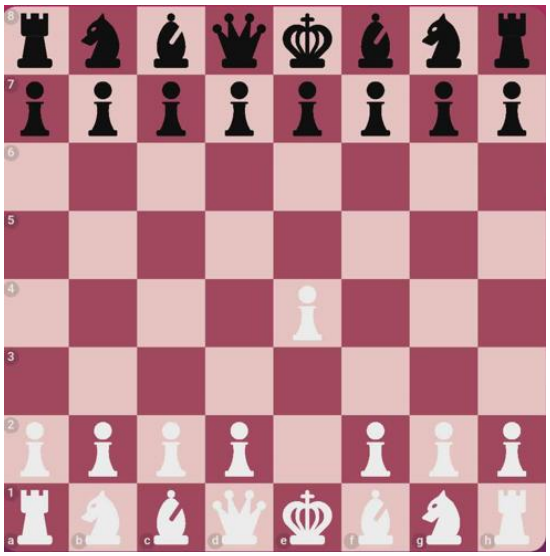

Figure 4. Screenshot of the chess application demonstrating move validation and board rendering for the move e2–e4

Table 1. Aggregate Performance Comparison Across 40 Benchmark Games

Metric	MCTS	Negamax
Average Move Time	2.81 ± 0.52 s	0.94 ± 0.21 s
Average Node Explored	50420	184330
Average Search Depth	4.2 plies	6.9 plies
Average Game Length	39 plies	41 plies
Win Rate	35.0%	57.5%
Draw Rate	7.5%	7.5%
Average Evaluation Score	+0.09	+0.16

The aggregate results indicate that Negamax consistently achieved lower move computation times while simultaneously reaching deeper average search depths. Although Negamax explored a larger number of search nodes, Alpha-Beta pruning and move-ordering optimisations significantly improved search efficiency and reduced overall response time.

In contrast, MCTS relied on repeated simulations and statistical backpropagation across the search tree, resulting in higher computational overhead under constrained mobile hardware conditions. The probabilistic exploration behaviour of MCTS provided broader positional sampling but required substantially longer computation time compared to deterministic alpha-beta search.

Table 2. Example Per-Move Diagnostics from Benchmark Execution

Ply	MCTS	MCTS Eval	MCTS Nodes	MCTS Time	Negamax	Negamax Eval	Negamax Nodes	Negamax Time
1	c4	+0.08	890	1280ms	c4	+0.12	1180	380
2	c5	-0.06	1120	1349ms	c5	-0.02	1240	420ms
3	Nf3	+0.15	1450	1660ms	Nc3	+0.19	1480	520ms
4	Nc6	-0.14	1680	1810ms	Nc6	+0.01	1620	580ms
5	Bb5	-0.21	1920	2280ms	Bb5	+0.28	1860	680ms
6	a6	-0.91	2150	2240	a6	-0.03	1980	720ms

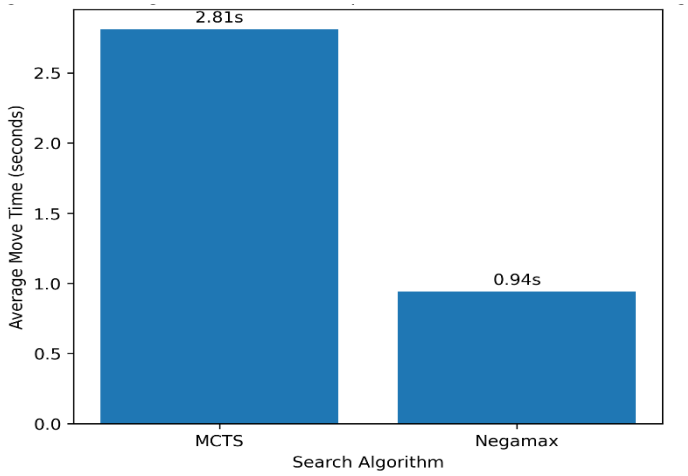

Figure 5. Illustrates the substantial reduction in average move computation time achieved by Negamax compared to MCTS under identical mobile hardware constraints.

Figure 5 demonstrates that Negamax achieved substantially lower average move computation times compared to MCTS under identical mobile hardware conditions. The deterministic Alpha-Beta search strategy reduced response latency and enabled faster decision-making during gameplay.

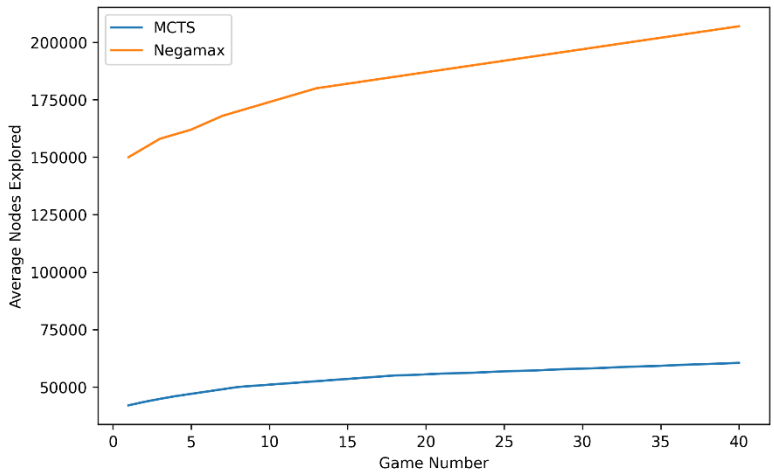

Figure 6. MCTS exhibits higher node growth and greater computational overhead as game complexity increases

Figure 6 illustrates the node exploration behaviour across the benchmark dataset. Although Negamax explored a larger number of search states overall, Alpha-Beta pruning and move-ordering heuristics reduced effective search complexity and maintained lower computation times than MCTS.

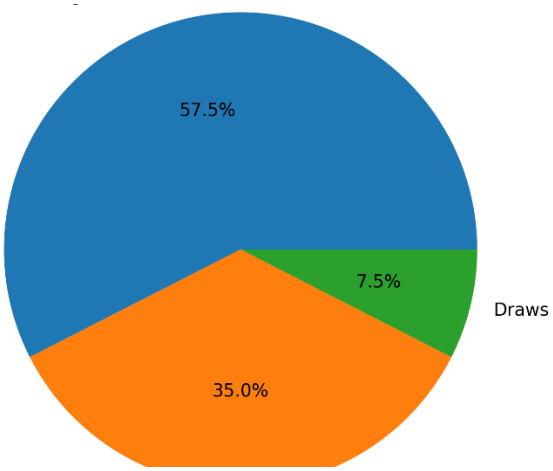

Figure 7. Overall game outcomes across the benchmark dataset

Figure 7 presents the overall benchmark outcome distribution across 40 games. Negamax achieved the highest win rate, indicating stronger practical performance under constrained mobile hardware conditions.

4. Discussion

The benchmark results demonstrate that Negamax with Alpha-Beta pruning consistently achieved lower move response times compared to heuristic-guided MCTS in mobile environments. The primary reason for this performance advantage is the ability of alpha-beta pruning to reduce the effective branching factor during search, enabling deeper tactical exploration while avoiding unnecessary evaluation of unfavourable branches. The pruning process exponentially reduces the effective branching factor, allowing deeper tactical exploration within the same computation budget, whereas MCTS continues evaluating many simulation branches through repeated playouts.

Additionally, iterative deepening, transposition tables, and MVV-LVA move ordering contributed to improved search efficiency and more stable runtime performance. Although Negamax explored a larger number of search nodes overall, the pruning mechanism significantly reduced redundant evaluations and improved practical search performance on constrained mobile hardware.

In contrast, MCTS relied on repeated simulation and statistical backpropagation across many search nodes, resulting in higher computational overhead and increased move response time. While progressive bias improved early move selection and positional exploration, the probabilistic nature of MCTS remained computationally expensive compared to deterministic alpha-beta search. These findings are consistent with previous comparative studies reporting that Alpha-Beta-based approaches generally outperform MCTS under limited computational budgets and real-time constraints. However, the implemented MCTS engine demonstrated stronger exploratory behaviour in certain complex middle-game positions and may benefit from future hybridisation with neural evaluation techniques. From a practical development perspective, the results suggest that Negamax with Alpha-Beta pruning is more suitable for mobile chess applications requiring fast response times and stable real-time gameplay, whereas MCTS may remain useful for experimental systems prioritising exploratory search behaviour and adaptive evaluation strategies.

Furthermore, the current implementation employs heuristic-guided simulations rather than purely random rollouts, forming a hybrid MCTS variant that differs from classical MCTS implementations. This design choice was intended to improve move quality during mobile execution while maintaining practical runtime constraints.

The benchmark framework demonstrated that mobile devices are capable of supporting relatively complex chess-engine experimentation using Flutter-based implementations. However, prolonged benchmark execution occasionally introduced increased computation latency and increased execution latency due to mobile hardware

limitations and sustained CPU-intensive workloads. Despite these constraints, both algorithms were successfully evaluated across repeated AI vs AI benchmark games under consistent runtime conditions. These findings suggest that mobile platforms can still provide a practical environment for experimental game AI research and lightweight chess-engine development.

4.1 Limitations

The current evaluation remains limited to a single mobile device configuration and a benchmark dataset of 40 games. Although the benchmark framework provides repeatable AI vs AI experimentation, larger datasets and additional hardware platforms would improve statistical reliability and generalisability of the findings. Additionally, no formal statistical significance testing or confidence-interval analysis was performed, and therefore the reported averages should be interpreted as empirical benchmark observations rather than definitive performance guarantees.

Additionally, the implemented MCTS engine uses heuristic guided simulations rather than purely random rollouts, which may influence direct comparison with standard MCTS implementations. Mobile thermal throttling, background operating-system processes, and device-level scheduling may also influence runtime consistency during prolonged benchmark execution. Future work will expand testing across additional Android devices, larger benchmark datasets, and stronger reference engines such as Stockfish for more rigorous evaluation and analysis.

References

David Silver, T. H. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science, 362*(6419), 1140-1144. Retrieved from DeepMind: https://deepmind.com/research/publications/mastering-chess-and-shogi-by-self-play

Developers, F. E. (2025). *Flame game engine*. Retrieved from Flame: https://flame-engine.org

Developers, S. (2025). *Stockfish chess engine*. Retrieved from Stockfish: https://stockfishchess.org

Fredlund, W., & Wigforss, H. (2025). *Comparative analysis of monte carlo tree search and alpha-beta pruning in chess AI development.* Retrieved from https://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A1948901

Google. (2025). *Flutter beautiful native apps in record time*. Retrieved from Flutter: https://flutter.dev

Helfenstein, F., Blüml, J., Czech, J., & Kersting, K. (2024). Checkmating One, by Using Many: Combining Mixture of Experts with MCTS to Improve in Chess. *Computing Research Repository (CoRR), abs/2401.16852*, pp. 1-31. Retrieved from https://arxiv.org/abs/2401.16852

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

Kocsis, L., & Szepesvári, C. (2006). Bandit based Monte-Carlo planning. *European Conference on Machine Learning (ECML 2006)* (pp. 282-293). Springer.

Shannon, C. E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine, 41*(314), 256-275.

Turing, A. M. (1953). Digital computers applied to games. In F. t. Thought, & B. V. Bowden (Ed.), *Faster than Thought* (pp. 286-310). Pitman.

Zobrist, A. L. (1970). *A new hashing method with application for game playing.* Technical Report 88, Computer Sciences Department, University of Wisconsin.