Proposal: Chess Engine via Minimax and Alpha-Beta

Proposal: Chess Engine via Minimax and Alpha-Beta

Name: Akromjon Akhmadjonov

Course: Advanced Programming in C++

Instructor: Gina Mejia Madrid

Date: 01.13.2026 - 05.01.2026

Abstract

This paper shows the design and application of a simple chess engine using classical artificial intelligence techniques. The main goal of this project is to understand how basic AI search algorithms work in a game environment. Over history, chess is used as the test case because it requires planning, decision making, and evaluation of many possible future moves. The given algorithm must analyze these possibilities and choose the next best move within a limited time.

The chess engine in this project uses a min-max algorithm. The algorithm used in this project first evaluates possible moves for both players and selects the move that leads to win for the current player. The engine tries to maximize the score while the opponent tries to minimize it. This idea was first formally studied in early game search research and is widely used in game-playing AI systems (Knuth & Moore, 1975). However, minimax alone is inefficient because the number of possible positions grows very quickly as the search depth increases.

And also, alpha-beta pruning algorithm is used to improve performance of the engine. This method works like cutting tree branches, as it eliminates the operations that don't affect the result and minimizes total computation. It allows the engine to search deeper without increasing

computation time. Research has shown that alpha-beta pruning significantly improves the efficiency of minimax search while producing the same final result (Marsland, 1986).

The chess engine also includes a legal move generation system to ensure that all moves follow official chess rules. Each position is evaluated using a simple material-based evaluation function, where different chess pieces are assigned fixed values. The engine then selects the move that leads to winning position.

This project is coded in C++ to focus on correctness and simplicity rather than advanced performance. The chess-engine is built to be easy to understand and to be an educational example of how AI can be implemented in games.

The results show that the engine is able to generate legal moves and play complete games correctly. It can make reasonable decisions based on search and evaluation, but its playing strength is limited compared to modern chess engines. More advanced systems use deeper searches, better evaluation functions, and more optimization techniques, but this project focuses only on the main concepts.

Overall, this work shows how minimax and alpha-beta pruning are the foundation of many chess engines and gives a basic understanding of how artificial intelligence is used in games.

Introduction

Chess has been one of the important challenges to test AI. Because it requires strategy, planning, and decision-making. A chess engine needs to evaluate a lot of possible positions and choose the

best one quickly. This makes chess a useful environment for studying how computers can simulate intelligent behavior. In this project, I developed a simple chess engine to understand how basic artificial intelligence techniques are applied in practice.

One of the most important ideas used in game AI's is the minimax algorithm. This algorithm explores possible moves for both players. The engine tries to maximize the score, while the other tries to minimize it. The algorithm builds a search tree of possible future game positions and computes the final positions to select the best move. Early research by Knuth and Moore (1975) formally explained how minimax search works and how game trees can be analyzed in a structured way. Their work also showed that the number of possible moves increases very quickly as the search depth becomes larger.

But minimax alone isn't really efficient enough for practical use in chess, because the number of possible positions gets extremely large very quickly. To deal with this, alpha-beta pruning is used. Basically what it does is cut off parts of the search tree that don't need to be looked at since they can't change the final decision anyway. Marsland (1986) pointed out that alpha-beta pruning can massively reduce the number of nodes evaluated while still giving you the same result as regular minimax. This makes the whole search process faster and lets the engine analyze positions at a much greater depth.

Legal move generation is another important part of a chess engine. It ensures that all moves follow the official rules of chess, including correct movement for each piece and valid capture rules. Without this step, the search process would include illegal positions and produce incorrect results.

The purpose of this project is to combine these basic ideas into a simple chess engine implemented in C++. The focus is on understanding how minimax, alpha-beta pruning, and move generation work together to form a basic artificial intelligence system. The goal is not to build a strong chess engine but to learn how classical search algorithms operate in a real application.

Overall, this project shows how early artificial intelligence techniques are still useful today and form the foundation of modern game-playing systems.

Literature Review

AI research in chess has developed over many years, and a lot of the important ideas used in modern chess engines actually trace back to pretty early work on game tree search. One of the most significant early contributions came from Knuth and Moore (1975) who studied the minimax algorithm and introduced alpha-beta pruning as a way to make the search more efficient. What their research basically showed is that you can cut down the number of nodes explored in a game tree without actually changing the final result. This became the foundation for most later chess engine designs.

Pearl (1980) expanded this work by studying the mathematical properties of minimax search. He explained that the number of possible game states grows very quickly as the search depth increases. This exponential growth makes it impossible to search all possible moves in real time, especially in complex games like chess. Because of this, optimization techniques are necessary to make search practical.

Marsland (1986) provided a detailed review of different pruning methods used in game tree search. He explained how alpha-beta pruning is applied in real systems and showed that its efficiency depends strongly on move ordering. If better moves are checked first, the algorithm can prune more branches and search faster.

Fishburn (1983) introduced some improvements to alpha-beta search by focusing specifically on optimization techniques. He showed that better move ordering can significantly cut down the number of nodes that need to be evaluated, which improves performance without actually affecting the correctness of the result.

Building on this Schaeffer (1989) introduced something called the history heuristic, which improves move ordering by essentially keeping track of moves that worked well in earlier parts of the search. This helps the engine try stronger moves earlier and increases pruning efficiency.

Reinefeld (1983) worked on the Scout search algorithm, which is an alternative approach to minimax search. His research showed that different search strategies can sometimes perform better than standard minimax depending on the position being analyzed.

Heinz (1998, 1999) introduced adaptive null-move pruning, which basically lets the engine skip over parts of the search tree when positions are relatively stable. This was later built on by Donninger (2009) later built on this by adding something called verification search, which made engines faster while still keeping the results reliable. Korf and Chickering (1996) also proposed best-first minimax search around a similar period, the idea being that instead of searching evenly across every branch you focus on the most promising moves first.

In real practical systems, all of these techniques tend to get combined together to build stronger engines overall. A good example of this is Campbell and colleagues (2002) who described Deep Blue, which brought together advanced search methods, sophisticated evaluation functions and seriously powerful hardware to eventually defeat world-class players. Hyatt and Cozzie (2008) also showed how hash tables are used in chess engines to store previously evaluated positions and improve efficiency.

Overall, these studies show that modern chess engines are built on a combination of minimax search, pruning methods, heuristics, and optimization techniques, which form the foundation of this project.

Methodology

The chess engine in this project is developed using C++ and follows a simple and structured design. The main idea is to combine legal move generation with the minimax algorithm and alpha-beta pruning to create a basic artificial intelligence system for playing chess. The system is designed for learning purposes, so the focus is on clarity and correctness rather than advanced performance.

The first part of the system is the board representation. The chess board is stored using a simple array structure where each square contains a value representing a chess piece or an empty space. Each piece is assigned a numerical value so that the program can evaluate the strength of a position. This allows the engine to compare different board states during the search process in a simple way.

The second part is legal move generation. This module is responsible for creating all valid moves according to the rules of chess. Each piece has specific movement rules. For example pawns move forward and capture diagonally, knights do their L-shape, bishops move diagonally, rooks go in straight lines, queens combine both rook and bishop movement and kings can move one square in any direction. The move generator also checks that moves stay within the boundaries of the board and that pieces aren't passing through other pieces when they shouldn't be. This step is important because the search algorithm must only work with legal positions.

The third part is the minimax algorithm. This algorithm works by exploring possible future moves through building out a tree of game states. One player is trying to maximize the score while the other is trying to minimize it, and the algorithm keeps going until it hits a fixed depth, at which point the board gets evaluated. Knuth and Moore (1975) explain that minimax is pretty much the standard method for decision-making in two-player games with perfect information.

To make things more efficient, alpha-beta pruning is applied on top of this. What it does is cut down the number of nodes that actually need to be evaluated by removing branches that couldn't possibly influence the final decision anyway. Marsland (1986) showed that this significantly improves search performance while still producing the same result as plain minimax.

The evaluation function used in this project is kept relatively simple and is based on material value. Each piece is assigned a fixed score and the engine just compares both sides to come up with a final evaluation. It's a straightforward approach but its honestly enough to guide move selection in a basic chess engine.

Overall the methodology is really about combining these simple but important AI techniques together to build a chess engine that actually works, follows the correct rules and makes at least somewhat logical decisions.

Implementation

The chess engine in this project is implemented in C++ using a simple and modular structure. The goal of the implementation is to combine legal move generation, minimax search, and alpha-beta pruning into one working system. The design focuses on clarity and correctness so that the main artificial intelligence concepts can be understood easily.

The first part of the implementation is the board system. The chess board is stored using a two-dimensional array where each element represents a square on the board. Each piece is represented using a value, and empty squares are also marked clearly. This structure allows the program to easily update and read the current state of the game. When a move is made, the board is updated by changing the positions of the pieces in the array.

The second part is the move generation system. This module handles generating all the possible legal moves for any given position. Each piece has its own movement rules and the program checks through these when figuring out what moves are available. Pawns move forward and capture diagonally, knights do their usual L-shape, bishops move diagonally, rooks go in straight lines, queens basically combine the rook and bishop and kings can move one square in any direction.

The move generator also makes sure that moves don't go outside the boundaries of the board and that pieces aren't moving through other pieces when they're not supposed to. This is actually a

pretty important step because the search algorithm can only work properly if it's being fed valid chess positions to begin with, otherwise the whole thing falls apart pretty quickly.

The third part is the search system, which uses the minimax algorithm. The implementation is recursive and explores possible future moves up to a fixed depth. At each level of the search tree the algorithm assumes that one player is trying to maximize the score while the other is trying to minimize it. Once the search hits the final depth the board gets evaluated using a pretty simple scoring function thats based on material value. Knuth and Moore (1975) explain that this method is commonly used in two-player games with perfect information.

To improve performance, alpha-beta pruning is integrated into the minimax function. This technique stops the algorithm from searching branches that cannot affect the final decision. Marsland (1986) shows that alpha-beta pruning reduces the number of nodes evaluated and allows the engine to search deeper within the same amount of time. The implementation itself was kept intentionally simple and easy to follow. It doesn't include any advanced features like opening books or machine learning or anything like that, but what it does do is give a pretty clear picture of how basic chess engines actually work using classical AI techniques.

Conclusion

This project successfully developed a simple chess engine using basic artificial intelligence techniques. The main focus of the project was to understand how the minimax algorithm, alpha-beta pruning, and legal move generation work together to create a decision-making system

for a game like chess. The implementation was done in C++, and was designed to be straightforward and readable rather than highly optimized or competitive in any serious way.

The results showed that the engine can generate legal moves correctly and play through full games without breaking any of the rules of chess. The minimax algorithm lets the system look ahead at possible future game states and make decisions based on what outcomes it expects to see. By essentially simulating both players at once the engine picks moves that should lead to better positions according to its evaluation function. This demonstrates pretty well how classical search algorithms can actually be applied to real problem-solving tasks rather than just existing as theory.

Alpha-beta pruning also played a pretty important role in improving overall performance. Without it the engine would've had to evaluate a much larger number of positions which would've made deeper searches way more expensive in terms of computation time.

References:

1. Aaltonen, J. (2023). Optimization areas of the minimax algorithm. KTH Royal Institute of Technology. https://kth.diva-portal.org/smash/get/diva2:1778372/FULLTEXT01.pdf

2. Buro, M. (1995). Experiments with multi-product and a new high-quality evaluation function for Othello. NEC Research Institute. https://skatgame.net/mburo/ps/improve.pdf

3. Campbell, M., Hoane, A. J., Jr., & Hsu, F.-h. (2002). Deep Blue. Artificial Intelligence, 134(1–2), 57–83. https://doi.org/10.1016/S0004-3702(01)00129-1

4. Donninger, C. (2009). Null move and verification search in computer chess. ICGA Journal, 32(2), 74–78. https://doi.org/10.3233/ICG-2009-32204

5. Edwards, S. (2023). AlphaGambit: Parallelizing MiniMax for Chess. Columbia University. https://www.cs.columbia.edu/~sedwards/classes/2023/4995-fall/reports/AlphaGambit-report.pdf

6. Fishburn, J. P. (1983). Another optimization of alpha-beta search. SIGART Newsletter, 84, 43–46. https://doi.org/10.1145/1056623.1056628

7. Heinz, E. A. (1998). Adaptive null-move pruning. ICCA Journal, 21(2), 123–132. https://doi.org/10.3233/ICG-1998-21204

8. Heinz, E. A. (1999). Scalable search in computer chess. Friedrich Vieweg & Sohn. https://doi.org/10.1007/978-3-322-90178-1

9. Hyatt, R., & Cozzie, A. (2008). Hash tables in computer chess. ICGA Journal, 31(1), 3–15. https://doi.org/10.3233/ICG-2008-31102

10. Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. Artificial Intelligence, 6(4), 293–326. https://doi.org/10.1016/0004-3702(75)90019-3

11. Korf, R. E., & Chickering, D. M. (1996). Best-first minimax search for minimax problems. Artificial Intelligence, 83(1), 1–17. https://doi.org/10.1016/0004-3702(96)00022-4

12. Levinson, R., & Snyder, R. (1991). Adaptive pattern-oriented chess. Proceedings of the 9th National Conference on Artificial Intelligence, 601–605. https://www.aaai.org/Papers/AAAI/1991/AAAI91-096.pdf

13. Madake, J., Deotale, C., Charde, G., & Bhatlawande, S. (2023). CHESS AI: Machine learning and Minimax based Chess Engine. International Research Journal of Modernization in Engineering Technology and Science, 5(5), 184–189. https://doi.org/10.56726/IRJMETS35532

14. Marsland, T. A. (1986). A review of game-tree pruning. ICCA Journal, 9(1), 3–19. https://doi.org/10.3233/ICG-1986-9102

15. Pearl, J. (1980). Asymptotic properties of minimax trees and game-searching procedures. Artificial Intelligence, 14(2), 113–138. https://doi.org/10.1016/0004-3702(80)90004-8

16. Plaat, A., Schaeffer, J., Pijls, W., & de Bruin, A. (1996). Best-first fixed-depth minimax algorithms. Artificial Intelligence, 87(1–2), 255–293. https://doi.org/10.1016/S0004-3702(96)00036-5

17. Reinefeld, A. (1983). An improvement to the Scout tree-search algorithm. ICCA Journal, 6(4), 4–14. https://doi.org/10.3233/ICG-1983-6402

18. Research on different heuristics for minimax algorithm: Insight from game-tree search. (2019). Journal of Computer and Communications, 7(3), 11–21. https://doi.org/10.4236/jcc.2019.73003

19. Sager, I. (2001). The impact of search extensions in computer chess. ICGA Journal, 24(4), 229–240. https://doi.org/10.3233/ICG-2001-24403

20. Schaeffer, J. (1989). The history heuristic and alpha-beta search enhancements in practice. IEEE Transactions on Pattern Analysis and Machine Intelligence, 11(11), 1203–1212. https://doi.org/10.1109/34.41714