Techniques In Chess Programming: A Comprehensive Review

Swoyam Pokharel

October 2025

Abstract

TODO

Contents

Bibliography

Björnsson, Y. and Marsland, T. (2000) "A Review Of Game-Tree Pruning," *Information Sciences*, 122, pp. 23–41. Available at: https://doi.org/10.1016/s0020-0255(99)00097-3.

Knuth, D.E. and Moore, R.W. (1975) "An Analysis of alpha-beta Pruning," *Artificial Intelligence*, 6, pp. 293–326. Available at: https://doi.org/10.1016/0004-3702(75)90019-3.

Shannon, C.E. (1950) "Programming a Computer for Playing Chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41, pp. 256–275. Available at: https://doi.org/10.1080/14786445008521796.

1 · Introduction

The game of chess has served as a proving ground for artificial intelligence reserach for decades now. From Claude Shannon's foundational paper framing chess as a computational problem, to Deepmind's AlphaZero acheiving extremely high strength through sheer self-play; chess has redefined the boundaries of algorithmic reasoning. Today, chess engines have far exceeded human capacity, with top engines like Stockfish and Leela Chess Zero estimated to operate at over 3500 Elo, approximately 800 Elo above the best humans to play chess.

What started as theoretical curiosity, that if solved, would force us to create "mechanized thinking", has now transformed into a vast domain for algorithmic innovation. Shannon recognized early on that exhaustive search was not feasible a typical chess game lasting 40 moves, containing approximately 10^{120} possible position variations; a number that far exceeds the number of atoms in the observable universe (Shannon, 1950, p. 4). This fundamental constraint, paired with the well-defined rules and success criteria, made chess an ideal playground for developing selective search methods, heuristic evaluation, and other fundamental techniques in modern AI.

2 · Foundations of Search

The strength of a chess engine fundamentally depends on its ability to search through the game tree and identify a move that leads to the best position. This section reviews the mathematical and algorithmic foundations that underpin modern chess engines, from classical programs like Stockfish to neural network-based systems like AlphaZero.

2.1 · Minimax and Negamax Framework

The game of chess, like any two-player, zero-sum game, can be represented as a game tree, where nodes represent legal board positions and edges represent legal moves. The foundation of searching for the best move is the determination of the minimax value, defined as the least upper bound on the score for the side to move, representing the true value of a position (Björnsson and Marsland, 2000, p. 3).

2.1.1 · Minimax Formulation

In the traditional minimax framework, two functions, $F(p)$ and $G(p)$, are defined from the perspective of the maximizing player (Max, typically White) and the minimizing player (Min, typically Black), respectively (Knuth and Moore, 1975, p. 4). For a position p with d legal successor positions $p_1, p_2, ..., p_d$, the framework, as described by Knuth and Moore, is defined as follows (Knuth and Moore, 1975).

1. **Maximizing Function**: The function $F(p)$ represents the best value Max can guarantee from position p when it is Max's turn to move. If p is a terminal position ($d = 0$), then $F(p) = f(p)$, where $f(p)$ is an evaluation function defining the outcome (e.g., +1 for a win, 0 for a draw, −1 for a loss). If $d > 0$, then:

$$F(p) = \max(G(p_1), G(p_2), l..., G(p_d))$$

   where $G(p_i)$ is the value of position p_i from Min's perspective.

2. **Minimizing Function**: The function $G(p)$ represents the best value Min can guarantee (in terms of Max's outcome) from position p when it is Min's turn to move. If p is a terminal position ($d = 0$), then $G(p) = g(p) = -f(p)$. If $d > 0$, then:

$$G(p) = \min(F(p_1), F(p_2), l..., F(p_d))$$

where $F(p_i)$ is the value of position p_i from Max's perspective.

3. **Optimal Play Assumption**: Both players play perfectly, with Max choosing the move that maximizes $F(p)$ and Min choosing the move that minimizes $G(p)$. This makes sure that $F(p)$ and $G(p)$ reflect the best possible outcome for each player against a perfectly playing opponent. The zero-sum property guarantees $G(p) = -F(p)$ for all positions p (Knuth and Moore, 1975, p. 3).

2.1.2 · Negamax Simplification:

The name "negamax" comes from "negative maximum" and is a simplification of the minimax algorithm. Unlike minimax, negamax utilizes the zero-sum nature, so, instead of using two functions ($F(p)$ for Max's turn and $G(p)$ for Min's, both from the Max's perspective), negamax uses a single function, $F(p)$ **defined from the perspective of the player to move** to maximize the negative of the opponent's score. This removes the need to oscillate between minimizing and maximizing, making the algorithm easier to implement, and thus is often preffered over minimax (Björnsson and Marsland, 2000, p.5) .

1. **Game Tree:** Similar to minimax, the game is a tree where nodes are positions (p) and the edges are legal moves (d) from position p, that to successor positions ($p_1, p_2, ...p_d$).
2. **Value Function**: The value function $F(p)$ represents the best value that the **player to move** can guarantee from position **p**, assuming both players play optimally.
    - If p is a terminal position ($d = 0$):

      $$F(p) = f(p)$$

      where `f(p)` is an evaluation function that gives the outcome from the perspective of the player to move.
    - If p is non-terminal ($d > 0$):

      $$F(p) = \max(-F(p_1), -F(p_2), ..., -F(p_d))$$

      where $F(p_i)$ is the value of the position p_i from the opponent's perspective, and the negative of that value ($-F(p_i)$), is that value converted to the current player's perspective.

The Negative Sign

The key simplification in negamax is the use of $-F(p_i)$. It takes advantage of the fact that the value of a position to the opponent, is the negative of the value to the current player. For instance,

1. If $F(p_i) = +1$ (opponent wins p_i), then $-F(p_i) = -1$ (loosing for current)
2. If $F(p_i) = -1$ (opponent looses p_i), then $-F(p_i) = +1$ (winning for current)

The current player chooses the move that maximizes $-F(p_i)$.

2.2 · Pruning

A game of chess typically lasts 40 moves, and with a branching factor of 35, at that number there are $\sim 10^{24}$ possible positions reachable form the starting position. As such, it is unfeasable to do an exhaustive search. The time complexity with just negamax is $O(b^{d})$, where $b =$ branching factor $(\sim 35 \text{ in chess}), d = \text{depth}$ (Shannon, 1950, p. 4; Björnsson and Marsland, 2000, p. 4)

2.2.1 · Branch-and-Bound Optimization

Knuth and Moore first present a optimization that improves upon the pure negamax function (say, F) as F_1. F_1 improves F by introducing an upper bound to prune moves that can't be better than the already known options. Knuth and Moore define:

$$F_1(p, \text{bound}) = \begin{cases} F(p) \text{ if } F(p) < \text{bound} \\ \geq \text{bound if } F(p) \geq \text{bound} \end{cases}$$

(Knuth and Moore, 1975, p.5). The intuition behind F_1 is that when evaluating a position p from the current player's perspective with a known bound that represents the best value acheivable till now, F_1 computes and returns the value if it less than the bound, or "$\geq$ bound" if it is equal or greater than the bound; i.e once it determines a move that acheives a value, atleast as good or better than our current best option, it prunes away the branch.

This reduces the number of nodes evaluated from $O(b^{d})$, although the exact reduction depends on other factors such as move ordering and tree structure. This approach bridges the gap between the pure negamax approach F and alpha-beta pruning.

2.2.2 · Alpha Beta Pruning

Alpha-Beta pruning is the most popular and reliable pruning method, that is used to speed up search without the loss of information. (Knuth and Moore, 1975, p.1; Björnsson and Marsland, 2000, p. 11, p.1). Similar to the above procedure F_1, alpha-beta pruning further improves effeciency by maintaining two bounds α and β.

- α: The best score the maximizing player can guarantee
- β: The best score the minimizing player can guarantee

Formally, Knuth and Moore define it as,

$$F_2(p, \alpha, \beta) = \begin{cases} F(p) \text{ if } \alpha < F(p) < \beta \\ \\ \leq \alpha \quad \text{if } F(p) \leq \alpha \\ \\ \geq \beta \quad \text{if } F(p) \geq \beta \end{cases}$$

(Knuth and Moore, 1975, p.6). Pruning happens when $\alpha \geq \beta$, the intuition behind which is that the maximizing player already has an option α that is atleast as good as what the opponent will allow β. Thus, the minimizing player won't allow reaching this position, because **we assume optimial play**, so we prune that branch. (Björnsson and Marsland, 2000, p.4)

Deep Cutoffs

A siginificant advantage of alpha-beta over the single bounded approach is it's ability to do "deep cutoffs". Knuth and Moore demonstrated that $F_2(-\infty, +\infty)$ examines the same number of nodes as $F_1(p, \infty)$ until the fourth look ahead level, but on the fourth and beyond levels, F_2 occasionally make deep cutoffs that F_1 isn't capable of finding. (Knuth and Moore, 1975, p.2, p.7).

Proof Of Optimality

Knuth and Moore further investigated if there were improvements beyond alpha-beta pruning, such as a $F_3(p, \alpha, \beta, \gamma)$ procedure where γ could hold additional information like the second largest value found so far. They concluded that the answer is no, showing that alpha-beta pruning is optimal in the reasonable sense. (Knuth and Moore, 1975, p. 6)

In the best, where the "best" move was examined first at every node, alpha-beta examines $W^{[\frac{D}{2}]} + W^{[\frac{D}{2}]} - 1$ terminal positions. This is a very big improvement over the W^D nodes for exhaustive search. For example, with a branching factor of 35 and a search depth of 6, this reduces the search from ~ 1.8 billion nodes to ~ 86 thousand

However, this performance is critically dependent on move ordering. When a computer plays chess, it rarely searches until the **true terminal** postion, instead, they end at a certain depth and evalute the position using heuristic evaluation functions. As such, to acheive performance closer to the theoritical best case, chess programs emply different move ordering heuristics like examining captures or checks first, or iteratively deepening to prioritize moves that performed well in shallower searches.

2.3 · The Horizon Effect