Techniques In Chess Programming: A Comprehensive Review

Name: Swoyam Pokharel
Student Number: 2431342
Supervisor: Prakriti Regmi
Reader: Siman Giri

Submitted On: November 27, 2025

Abstract

This literature review explores the core techniques that have shaped modern chess programming. It begins by covering the mathematical foundations of game tree search, analyzing minimax, negamax, and alpha-beta pruning that form the backbone of traditional engines. After which, it explores the implementation approaches for board representation, different move generation techniques, and memory optimization through transposition tables and Zobrist hashing.

Furthermore, this paper covers the transition from classical evaluation methods to the recent shift toward neural networks. It examines search optimizations like iterative deepening, quiescence search, null move pruning, and move ordering heuristics, that enable engines to search deeper while maintaining efficiency. Finally, the paper evaluates the classical alpha-beta paradigm (Stockfish) with neural network-guided Monte Carlo Tree Search approaches (AlphaZero, Leela Chess Zero), analyzing their trade-offs and the current convergence toward hybrid systems. In essence, This paper establishes the current state of chess programming and identifies the techniques that have proven essential for competitive engine development.

Contents

1 · Introduction 4
2 · Foundations of Search 4
2.1 · Minimax and Negamax Framework 4
2.2 · Pruning 5
2.3 · The Horizon Effect 6
3 · Entity Representation & Move Generation 7
3.1 · Approaches To Board Representation 7
3.2 · Move Generation 7
4 · Foundations Of Evaluation 9
4.1 · Hand Crafted Evaluation (HCE) 9
5 · Search Enhancements & Optimizations 10
5.1 · Memory-Aided Search 10
5.2 · Iterative Deepening 11
5.3 · Advanced Alpha-Beta Variations 12
5.4 · Move Ordering Heuristics 12
5.5 · Selective Search Extensions 12
5.6 · Pruning Techniques 13
5.7 · Parallel Search 14
6 · Evaluation Optimizations & Enhancements 14
6.1 · Monte Carlo Tree Search and Neural Network Engines 14
6.2 · NNUE (Efficiently Updatable Neural Networks) 15
6.3 · The State Of Neural Network Based Engines 17
7 · System Analysis & Architecture Patterns 17
7.1 · The Classical Paradigm: Alpha-Beta + HCE 17
7.2 · Neural Network Based: MCTs + Deep-Learning 18
7.3 · Hybrid Approach 19
Bibliography 22

1 · Introduction

The game of chess has served as a proving ground for artificial intelligence research for decades now. From Claude Shannon's foundational paper framing chess as a computational problem, to Deepmind's AlphaZero achieving 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 has now transformed into a vast domain for algorithmic innovation. Shannon recognized early on that exhaustive search was not feasible, as a typical chess game lasting 40 moves, contains 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 clear win 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.

2.1 · Minimax and Negamax Framework

The game of chess 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 calculation of the minimax value, defined as the least upper bound on the score for the side to move, that represents the true value of a position (Björnsson and Marsland, 2000, p. 3). This framework remains the foundational principle for search tree traversal in classical game AI, with universal consensus across modern implementations (Björnsson and Marsland, 2000, p.3); (Rasmussen, 2004, p.24-p.26); (Brange, 2021, p.18).

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 (typically White) and the minimizing player (typically Black), respectively (Knuth and Moore, 1975, p. 4). For a position p with d legal successor positions such that, $p_1, p_2, ..., p_d$, represents all the valid reachable positions, the framework is:

- **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 $F(p) = f(p)$. If $d > 0$, then:

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

- **Minimizing Function:** The function $G(p)$ represents the best value Min can guarantee 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:

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

The fundamental assumption is that both players play perfectly. The zero-sum property guarantees $G(p) = -F(p)$ for all positions p (Knuth and Moore, 1975, p. 3).

2.1.2 · Negamax Simplification

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 (Björnsson and Marsland, 2000, p.5).

- **Value Function:** The value function $F(p)$ represents the best value that the player to move can guarantee from position p, with the assumption that both players play optimally.
- If p is a terminal position ($d = 0$): $F(p) = f(p)$
- If p is non-terminal ($d > 0$): $F(p) = \max(-F(p_1), -F(p_2), ..., -F(p_d))$

The negative sign takes advantage of the fact that chess is a zero-sum game. Hence, the current player chooses the move that maximizes $-F(p_i)$, the equivalent of minimizing $F(p_i)$.

2.2 · Pruning

A game of chess typically lasts ≈ 40 moves, and has a branching factor of 35. Even for a game lasting around 40 moves per player (80 plies), this yields a search space on the order of $10^{40} - 10^{100}$. This makes exhaustive search unfeasible, as the time complexity of basic negamax search is $O\left(b^d\right)$, where $b =$ branching factor, $d =$ depth (Shannon, 1950, p. 4); (Björnsson and Marsland, 2000, p. 4). This computational constraint has driven the development of pruning techniques.

2.2.1 · Branch-and-Bound Optimization

Knuth and Moore present an optimization F_1 that improves upon the pure negamax function F by introducing an upper bound to prune moves that can't be better than already known options (Knuth and Moore, 1975, p.5):

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

Once it determines that a position achieves a value at least as good as the bound, the exact value is irrelevant since the opponent will avoid this line of play, allowing the branch to be pruned away.

2.2.2 · Alpha Beta Pruning

Alpha-Beta pruning is the most popular and reliable pruning method, recognized universally as the most vital algorithmic optimization for achieving practical search depth in traditional chess engines (Björnsson and Marsland, 2000, p.1, p.11); (Rasmussen, 2004, p.28); (Brange, 2021, p.22); (Vrzina, 2023, p.19). It maintains 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 (Knuth and Moore, 1975, p.6):

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

Pruning happens when $\alpha \geq \beta$. A significant advantage is its ability to do "deep cutoffs" beyond what single-bounded approaches can find (Knuth and Moore, 1975, p.2, p.7). Knuth and Moore proved that alpha-beta is optimal in a reasonable sense (Knuth and Moore, 1975, p. 6), establishing it as the definitive pruning algorithm for game-tree search.

In the best case, alpha-beta examines $W^{\lceil \frac{D}{2} \rceil} + W^{\lceil \frac{D}{2} \rceil} - 1$ terminal positions versus W^D nodes for exhaustive search. However, this performance critically depends on move ordering.

AlphaZero's success with Monte Carlo Tree Search guided by deep neural networks demonstrated that selective search based on learned policy estimates can outperform traditional alpha-beta approaches, despite examining $\approx 1000\times$ fewer positions (Silver et al., 2017, p.3-p.5), representing a paradigm shift from guaranteed pruning through mathematical bounds to probabilistic selection through learned heuristics.

2.3 · The Horizon Effect

Shannon was amongst the earliest to recognize the "horizon effect" (Shannon, 1950, p.6). This effect describes a program's tendency to "hide" inevitable material loss by making delaying moves until said loss is beyond its maximum depth. This problem emerges from a lack of computing power that forces programs to limit the depth of the search (Brange, 2021, p.14).

Shannon acknowledged the importance of evaluating only "relatively quiescent" positions (Shannon, 1950, p.6), defining a quiescent position as one that can be assessed accurately without needing further deepening (Björnsson and Marsland, 2000, p.7).

2.3.1 · Quiescence Search

Quiescence search is the principal approach to solving the horizon effect problem, by ensuring that a position is stable before evaluation. Sources universally agree that Quiescence Search (QS) is critical for handling tactical volatility (Björnsson and Marsland, 2000, p.7); (Rasmussen, 2004, p.41); (Bijl, Tiet and Bal, 2021, p.11). QS continues evaluation of all forcing moves until a "quiet" position is reached. With that said, a consistent theoretical framework defining "true quiescence" remains undeveloped (Björnsson and Marsland, 2000, p.8).

When Tesseract added quiescence search,

- **Execution time:** 709.48s → 266.21s (62.5% faster),
- **Evaluation score:** 7,314 → ,520 (+1,206 points),
- **Effective branching factor:** 5.99 → 4.23 (-1.76)

(Vrzina, 2023, p.20, p.31, p.50). This enforces the consensus that quiescence search remains valuable for competitive chess engines.

3 · Entity Representation & Move Generation

Storing the board state efficiently is one of the most fundamental considerations for any chess engine (Brange, 2021, p.14); (Columbia et al., 2023, p.13). Multiple sources agree that BitBoards have emerged as the dominant method for representing board state in competitive chess engines (Fiekas, 2018, p.5); (Bijl, Tiet and Bal, 2021, p.5); (Herranz and Qiu, 2025, p.30).

3.1 · Approaches To Board Representation

3.1.1 · Array Based Representations

The two-dimensional array approach is intuitive but comes with performance costs. Indexing requires calculating memory location and performing boundary checks, making it inefficient (Bijl, Tiet and Bal, 2021, p.4); (Vrzina, 2023, p.6). In testing, this approach came in at 39.189 Mn/s in Perft and 6.327 Mn/s in search speed (Bijl, Tiet and Bal, 2021, p.19).

The 0x88 variant pads the array to 16x8 with sentinel values, reducing out-of-bounds checks to a single comparison (Bijl, Tiet and Bal, 2021, p.4); (Vrzina, 2023, p.6-p.7). In performance tests, 0x88 was nearly equal to BitBoards in Perft speed at 46.496 Mn/s (Bijl, Tiet and Bal, 2021, p.19).

3.1.2 · BitBoards

BitBoards are piece-centric representations that utilize the fact that an unsigned 64-bit integer has the same number of bits as squares on a chess board (Rasmussen, 2004, p.47-p.50); (Bijl, Tiet and Bal, 2021, p.5); (Columbia et al., 2023, p.16-p.26). This representation allows logical operations to be executed in parallel using single CPU instructions (Fiekas, 2018, p.5); (Bijl, Tiet and Bal, 2021, p.5); (Herranz and Qiu, 2025, p.30).

3.1.3 · Hybrid Approaches

Modern engines incorporate both BitBoards and a mailbox-style approach. BitBoards are used for filtering and move generation, while the mailbox is used for fast data access (Bijl, Tiet and Bal, 2021, p.5); (Vrzina, 2023, p.6-p.7).

3.1.4 · Why BitBoards Dominate

Bijl & Tiet's findings challenge the conventional narrative: traditional array based representations like 0x88 boards can achieve comparable speeds to BitBoards in isolated move generation tasks (Bijl, Tiet and Bal, 2021, p.20). However, BitBoards are preferred because they accelerate evaluation, which heavily relies on fast bitwise operations (Bijl, Tiet and Bal, 2021, p.20); (Vrzina, 2023, p.10). In complete engine tests, move generation accounted for only about 10% of processing time, with evaluation forming the primary bottleneck.

3.2 · Move Generation

3.2.1 · Pseudo-Legal vs Legal Move Generation

A pseudo-legal move follows the rules of how pieces typically move but does not account for whether the king is in check. A legal move accounts for the king being in check (Columbia et al., 2023, p.65).

Although pseudo-legal move generation requires checking for legality during search (Columbia et al., 2023, p.11), it tends to be preferred. During search with pruning, if a cutoff occurs, the engine avoids wasting time on moves that would have been pruned regardless (Rasmussen, 2004, p.56); (Bijl, Tiet and Bal, 2021, p.20). Modern engines, including Stockfish, prefer the pseudo-legal approach.

3.2.2 · Generating Moves For Non-Sliding Pieces

For non-sliding pieces (kings and knights), the standard approach uses pre-computed lookup tables storing a BitBoard representing the attacks from each square. To filter out capturing friendly pieces, it's simply `knight_attacks[28] & !friendly` (Brange, 2021, p.27); (Bijl, Tiet and Bal, 2021, p.6); (Vrzina, 2023, p.7).

3.2.3 · Move Generation for Sliding Pieces

For sliding pieces, simple lookup tables do not suffice because their movement depends on the blocker configuration.

Magic BitBoards

Magic Bitboards are an advanced optimization that convert the move generation problem into a lookup operation using multiplicative hashing. During runtime, the index is calculated as (Bijl, Tiet and Bal, 2021, p.7-p.8); (Vrzina, 2023, p.10):

```
index = (blockers * magic_number) >> shift_amount
```

Magic BitBoards provide constant-time move generation and have become the de facto standard (Bijl, Tiet and Bal, 2021, p.7); (Herranz and Qiu, 2025, p.48-p.51).

PEXT Boards

They are a part of the BMI2 instruction set introduced in 2013, and act as an alternative. PEXT directly computes the necessary index in a single CPU cycle (Fiekas, 2018, p.10); (Bijl, Tiet and Bal, 2021, p.8); (Vrzina, 2023, p.10); (Herranz and Qiu, 2025, p.51). At runtime:

```
index = PEXT(blockers, ray_mask)
```

Based on 100 runs of Stockfish's benchmark suite, PEXT BitBoards provide only a 2.3% speedup over Magic BitBoards (Fiekas, 2018, p.10), corroborated by Bijl & Tiet (Bijl, Tiet and Bal, 2021, p.20).

3.2.4 · Move Representation

Modern engines use a 16 bit representation for moves (Shannon, 1950, p.10); (Bijl, Tiet and Bal, 2021, p. 8-9); (Vrzina, 2023, p.12):

```
0000 000000 000000
____ ______ ______
prom to     from
```

This encoding allocates 6 bits for from/to squares and 4 bits for promotion piece type. When Tesseract adopted this encoding over a naive struct implementation, move generation speed increased by nearly 50% (Vrzina, 2023, p.12).

3.2.5 · Perft

Perft (performance test) is a fundamental debugging and validating tool that recursively generates the entire game tree for a specific position up to a given depth and counts all resulting nodes (Columbia et al., 2023, p.67); (Vrzina, 2023, p.16); (Herranz and Qiu, 2025, p.41). Due to the branching factor of chess, just 9 plies deep from the starting position yields over 2.4 trillion leaf nodes.

4 · Foundations Of Evaluation

Shannon first introduced the concept of an approximate evaluation function $f(P)$ to guide chess engines, recognizing that searching the entire game tree is unfeasible (Shannon, 1950, p. 4-6); (Herranz and Qiu, 2025, p.18). Sources consistently agree that evaluation must address both material balance and positional factors (Shannon, 1950, p.17); (Björnsson and Marsland, 2000, p.3); (Bijl, Tiet and Bal, 2021, p.12); (Herranz and Qiu, 2025, p.33).

Shannon suggested that $f(P)$ should include material advantage, pawn formation, piece mobility, and king safety (Shannon, 1950, p.5, p.17). These classical components, proposed in 1950, formed the baseline that evolved into increasingly specialized heuristics (Silver et al., 2017, p.10).

4.1 · Hand Crafted Evaluation (HCE)

Engines have historically used Hand Crafted Evaluation functions that account for different features (Shannon, 1950, p. 5); (Silver et al., 2017, p.10); (Świechowski et al., 2022, p.2).

4.1.1 · Materialistic Approach

Material score, with weighted piece values, forms the quantitative baseline (Shannon, 1950, p.17); (Björnsson and Marsland, 2000, p.2); (Herranz and Qiu, 2025, p.34). The technique subtracts the total material scores of the two sides, generally represented in centipawns:

$$\text{Pawn} = 100$$
$$\text{Knight} = 320$$
$$\text{Bishop} = 330$$
$$\text{Rook} = 550$$
$$\text{Queen} = 950$$

(Björnsson and Marsland, 2000, p.3); (Herranz and Qiu, 2025, p.34). Although these values are the de-facto standard, Bijl & Tiet also note a study from S. Droste and J. Furnkranz for assigning values to pieces using reinforcement learning that yielded the following (Bijl, Tiet and Bal, 2021, p.12):

$$\begin{aligned}\text{Pawn} &= 100\\ \text{Knight} &= 270\\ \text{Bishop} &= 290\\ \text{Rook} &= 430\\ \text{Queen} &= 890\end{aligned}$$

4.1.2 · Positional and Strategic Heuristics

Positional elements (mobility, pawn structure, king safety) are necessary for strong play (Shannon, 1950, p.17); (Björnsson and Marsland, 2000, p.2); (Herranz and Qiu, 2025, p.34).

Piece Square Tables (PSTs)

These are piece-specific, precomputed tables that assign a bonus or penalty for a piece depending on its square (Brange, 2021, p.31); (Vrzina, 2023, p.33); (Herranz and Qiu, 2025, p.35). Tesseract's implementation caused evaluation to go from 5640 to 8255, the single biggest evaluation impact amongst heuristics (Vrzina, 2023, p.38-p.39).

Tapered Evaluation

This is a technique to adjust heuristic scores dynamically based on game phase (midgame versus endgame) to reflect the shifting value of pieces (Bijl, Tiet and Bal, 2021, p.13); (Herranz and Qiu, 2025, p.35). Engines typically employ 2 different sets of PSTs and interpolate between them (Vrzina, 2023, p.33); (Herranz and Qiu, 2025, p.35).

4.1.3 · Parameter Tuning

Bijl & Tiet's sequential tuning resulted in an average win rate increase of 15% (Bijl, Tiet and Bal, 2021, p.20). Their study revealed that optimal piece values depend on search depth, with Knight Material Score decreasing with increasing depth, but bishop pair increasing. This depth-dependent variation challenges the notion of fixed material scores.

5 · Search Enhancements & Optimizations

5.1 · Memory-Aided Search

5.1.1 · Transposition Tables

In chess, the same positions can be reached in different sequences of moves (transpositions). Transposition tables are data structures, typically hash tables that store the evaluation of previously reached positions for re-use (Björnsson and Marsland, 2000, p.13); (Bijl, Tiet and Bal, 2021, p. 10); (Herranz and Qiu, 2025, p.45). Sources universally recognize TPT as essential for exact forward pruning and move ordering (Björnsson and Marsland, 2000, p.13); (Rasmussen, 2004, p.34); (Bijl, Tiet and Bal, 2021, p.10); (Vrzina, 2023, p.20).

Zobrist Hashing

This is universally accepted as the standard algorithm for computing position hash keys (Zobrist, 1970); (Björnsson and Marsland, 2000, p.14); (Bijl, Tiet and Bal, 2021, p.9); (Vrzina, 2023, p.18). It involves calculating the hash by XOR-ing together pregenerated 64 bit numbers

corresponding to every piece type on every square. Although collision probability exists (0.000003% with 1 billion moves stored) (Zobrist, 1970, p.10), the chance is small enough for practical purposes. The key advantage is incremental updating via 2-4 XOR operations (Zobrist, 1970, p.5, p.10); (Björnsson and Marsland, 2000, p.14); (Rasmussen, 2004, p.36); (Bijl, Tiet and Bal, 2021, p.9); (Vrzina, 2023, p.18).

Each table entry stores the Zobrist Hash, Evaluation, Depth, Best Move, Age, and Node Type (EXACT, LOWERBOUND, UPPERBOUND) (Björnsson and Marsland, 2000, p.14); (Rasmussen, 2004, p.100); (Brange, 2021, p.36); (Herranz and Qiu, 2025, p.48).

Since Transposition Tables are often fixed in size (Zobrist, 1970, p.2); (Björnsson and Marsland, 2000, p.16), replacement schemes like Depth Preferred Replacement preserve the most computationally expensive searches.

5.1.2 · Syzygy Tablebases

Chess endgames with seven or fewer pieces have been completely solved through exhaustive retrograde analysis (Rasmussen, 2004, p.11). Engines can leverage tablebases like Syzygy to achieve perfect endgame play (Bijl, Tiet and Bal, 2021, p.21). However, storage requirements are substantial: 3-5 piece endgames require 939 MiB, 6-piece endgames 149.2 GB, and full 7-piece tablebase 16.7 TiB.

5.1.3 · Refutation Tables

A refutation table is a lightweight data structure that stores effective refutations and main continuations, often referred to as space-efficient alternative to transposition tables. For devices with no memory constraint, this technique is still used as an additional aid (Björnsson and Marsland, 2000, p.16).

5.2 · Iterative Deepening

All traditional engines employ Iterative Deepening as a standard procedure to manage search time and enhance performance (Björnsson and Marsland, 2000, p.19); (Brange, 2021, p.38); (Bijl, Tiet and Bal, 2021, p.11); (Vrzina, 2023, p.21). Instead of immediately searching to depth D, the search goes 1-ply, then 2-ply, and so on until reaching D. Although this may seem counter-intuitive, engines use information from shallow searches to prioritize moves in deeper searches, which prunes many branches (Bijl, Tiet and Bal, 2021, p.10); (Brange, 2021, p.38); (Herranz and Qiu, 2025, p.32).

Benefits include time management (interrupted searches return previous depth result), move ordering (promising moves from previous searches are tried first), and aspiration windows (narrow search windows based on previous scores increase cutoffs) (Björnsson and Marsland, 2000, p.17); (Rasmussen, 2004, p.39); (Rasmussen, 2004, p.33); (Vrzina, 2023, p.21). In KLAS engine analysis, Iterative Deepening with PV-Ordering decreased average search time by 28.7% (Brange, 2021, p.47).

5.3 · Advanced Alpha-Beta Variations

5.3.1 · Principle Variation Search (PVS) / Negascout

PVS optimizes alpha-beta by assuming the first move is likely best. It searches the first move with a full window $[\alpha, \beta]$, then searches subsequent moves with a minimal window (typically $[\alpha, \alpha + 1]$) to quickly verify they score no better. If a minimal window search fails, PVS re-searches with the full window (Björnsson and Marsland, 2000, p.9); (Rasmussen, 2004, p.40); (Vrzina, 2023, p.22).

5.3.2 · MTD(f)

MTD(f) performs multiple minimal window searches that converge on the minimax value. It starts with an initial guess and repeatedly searches with minimal windows, adjusting bounds based on results. This approach performs less work per search but requires searching multiple times. A strong transposition table is essential to avoid redundant re-computation. In practice, MTD(f) can outperform PVS when combined with effective hashing (Plaat, 1997), though PVS remains more widely adopted due to simpler implementation.

5.4 · Move Ordering Heuristics

Move ordering is critical for pruning effectiveness, as it establishes the threshold against which other positions are evaluated (Rasmussen, 2004, p.31); (Herranz and Qiu, 2025, p.22).

TT Move

This ordering uses the transposition table's best moves from previous searches (Björnsson and Marsland, 2000, p.13); (Rasmussen, 2004, p.37).

MVV-LVA (Most Valuable Victim - Least Valuable Aggressor)

This ordering prioritizes positive material trades. In KLAS engine assessment, MVV-LVA resulted in the single biggest performance impact, decreasing execution time by 68.5% (Silver et al., 2017, p.11); (Brange, 2021, p.34); (Herranz and Qiu, 2025, p.42); (Brange, 2021, p.45).

Killer moves

This ordering prioritizes non-capture moves that caused beta cutoffs at the same depth in sibling positions (Björnsson and Marsland, 2000, p. 12); (Rasmussen, 2004, p.38); (Herranz and Qiu, 2025, p. 42-p.43).

History Heuristic

This ordering tracks how often a move causes a beta cutoff across the entire search tree, and prioritizes those moves (Björnsson and Marsland, 2000, p.12); (Rasmussen, 2004, p.39).

5.5 · Selective Search Extensions

These mechanisms strategically increase search depth of certain moves, beyond the fixed depth (Björnsson and Marsland, 2000, p.3). Shannon categorizes this as a type B strategy (Shannon, 1950, p.13).

Check Extensions
Extends analysis when the opponent is in check, as checks limit opponent responses and might lead to checkmate (Rasmussen, 2004, p.42).

Pawn Pushes
Extend when pawns near promotion (7th/2nd rank) create significant threats (Björnsson and Marsland, 2000, p.8); (Rasmussen, 2004, p.43).

Singular Extensions
Extend when the best move is clearly forced, identified by reduced-depth search excluding the best candidate (Rasmussen, 2004, p.10); (Bijl, Tiet and Bal, 2021, p.13).

One Reply Extensions
Extend when only one legal move exists, since there are no alternatives to consider (Rasmussen, 2004, p.42).

5.6 · Pruning Techniques

5.6.1 · Null Move Pruning (NMP)

Sources recognize NMP as a highly effective speculative heuristic (Rasmussen, 2004, p.43); (Silver et al., 2017, p.10); (Bijl, Tiet and Bal, 2021, p.12); (Vrzina, 2023, p.25). The technique allows the side to move to "pass" (make a null move), giving the opponent two consecutive moves, and searches the resulting position with reduced depth. If this deliberately weakened position still produces a score $\geq \beta$, the subtree can be pruned (Rasmussen, 2004, p.43); (Silver et al., 2017, p.10).

However, this relies on the assumption that zugzwang positions are rare. Since zugzwang occurs more frequently in endgames, engines typically disable null move pruning in such positions or when in check (Bijl, Tiet and Bal, 2021, p.12).

5.6.2 · Late Move Reduction (LMR)

LMR exploits strong move ordering to reduce search effort on moves unlikely to be best. Rather than searching all moves to full depth D, LMR searches later moves to reduced depth, typically $D - R$ where R increases with move number. To avoid missing tactical opportunities, LMR prevents reduction of captures, promotions, checks, check evasions, and killer moves. If a reduced-depth search returns a score within $[\alpha, \beta]$, the engine re-searches at full depth (Bijl, Tiet and Bal, 2021, p.12); (Vrzina, 2023, p.26); (Herranz and Qiu, 2025, p.55).

LMR effectiveness is heavily dependent on move ordering quality. It is implemented in Stockfish (Bijl, Tiet and Bal, 2021, p.12), but several developers found it resulted in worse performance due to aggressive pruning causing blunders (Vrzina, 2023, p.27); (Herranz and Qiu, 2025, p.63). Tesseract demonstrated significant gains, reducing average search time from 83.87 to 64.13 milliseconds, but with score decrease from 8584 to 8124 (Vrzina, 2023, p.26). A research gap remains in creating robust implementations of aggressive pruning like LMR that do not suffer search instability.

5.6.3 · Futility Pruning

Futility pruning eliminates moves unlikely to raise the score above α when the search is near the horizon. If a position's static evaluation plus a generous margin still falls below α, and only a few plies remain, then quiet moves are unlikely to dramatically improve the position and can be safely pruned (Björnsson and Marsland, 2000, p.11); (Rasmussen, 2004, p.41).

5.7 · Parallel Search

As modern CPUs evolved to include multiple cores, parallelizing the search became the natural next step. However, alpha-beta search is inherently sequential. When work is distributed across threads, pruning information is not immediately available across threads, causing diminishing returns: only 9.2x speedup was observed on 22 processors (Rasmussen, 2004, p. 3, p.78).

5.7.1 · Young Brothers Wait Concept (YBWC)

YBWC searches the first child node sequentially, then distributes remaining nodes among threads. During the sequential phase, helper threads remain idle (Rasmussen, 2004, p.62); (Herranz and Qiu, 2025, p.55). This aligns with alpha-beta node types but proves suboptimal for Type 3 (ALL) nodes. The AlphaDeepChess project observed performance degradation rather than improvement due to synchronization overhead (Herranz and Qiu, 2025, p.62).

5.7.2 · Lazy SMP

Lazy Symmetric MultiProcessing spawns independent threads that each perform a complete search autonomously, sharing information only through the transposition table (Brange, 2021, p.39); (Vrzina, 2023, p.27). This "lazy" technique allows redundant searches but proves remarkably effective. Implementations employ randomized move ordering at the root node, ensuring each thread's search diverges early (Brange, 2021, p.39); (Vrzina, 2023, p.27).

Lazy SMP achieved a 33.1% reduction in average execution time on a four-core system in KLAS (Brange, 2021, p.58) and a 40% speedup in Tesseract (Vrzina, 2023, p.27). Despite its wasteful nature, Lazy SMP remains the dominant multithreaded search method, outcompeting more theoretically sound alternatives through sheer simplicity.

6 · Evaluation Optimizations & Enhancements

HCE functions face a fundamental limitation: they rely on human domain expertise and are bounded by the strategies humans can explicitly model (Shannon, 1950, p.5); (Silver et al., 2017, p.2). This gap paved the way for the shift toward neural network based evaluation (Silver et al., 2017, p.12); (Nasu, 2018, p.1); (Nasu, 2018).

6.1 · Monte Carlo Tree Search and Neural Network Engines

Engines like AlphaZero and Leela Chess Zero employ Monte Carlo Tree Search (MCTs), which grows its tree asymmetrically, concentrating computational effort on the most promising variations (Silver et al., 2017, p.3). This represents a paradigm shift from guaranteed pruning through mathematical bounds to probabilistic selection through learned heuristics.

6.1.1 · MCTs Algorithm

MCTs operates through four iterative phases: Selection (traverse tree by balancing exploration/ exploitation using UCB1), Expansion (add unvisited position as new node), Simulation (evaluate new position), and Backpropagation (update value estimates along path to root).

6.1.2 · AlphaZero's Neural-Guided MCTs

In AlphaZero, MCTs is guided by a deep neural network $f_\theta(s)$ that outputs Policy ($\boldsymbol{p}$, probability distribution over moves) and Value (v, expected game outcome from −1 to +1) (Silver et al., 2017, p.2). AlphaZero has no handcrafted chess knowledge beyond basic rules and learns entirely through self-play reinforcement learning. The network trains by minimizing a loss function l combining

- **Value Loss**: $(z - v)^2$, minimizing the mean-squared error between the predicted outcome v and the actual game outcome z (where $z = +1$ for win, 0 for draw, -1 for loss)
- **Policy Loss**: $-\boldsymbol{\pi}^T \log \boldsymbol{p}$, maximizing similarity between the network's policy $\boldsymbol{p}$ and the search probabilities $\boldsymbol{\pi}$ generated by MCTs

6.1.3 · Comparison with Traditional Engines

AlphaZero examines approximately 80,000 positions per second while Stockfish evaluates roughly 70 million. However, AlphaZero compensates through superior selectivity, using its neural network to prioritize the most promising lines and achieving superior results despite searching $\approx 1000\times$ fewer positions (Silver et al., 2017, p.4). While older MCTs implementations proved weaker than alpha-beta (Silver et al., 2017, p.12), coupling MCTs with deep neural networks achieved superiority. In head-to-head competition using 64 threads and 1GB hash, AlphaZero convincingly defeated all opponents, losing zero games to Stockfish (Silver et al., 2017, p.5).

6.2 · NNUE (Efficiently Updatable Neural Networks)

The Efficiently Updatable Neural Network (NNUE) represents a major shift in how chess engines approach evaluation functions. Originally proposed by Yu Nasu (Nasu, 2018) for computer shogi, NNUE introduces a hybrid paradigm merging pattern recognition strength of neural networks with the speed of traditional handcrafted evaluators. Unlike deep models like AlphaZero, NNUE is designed explicitly for CPU execution rather than GPU acceleration. It uses a fully connected, shallow neural network optimized for rapid, low-precision inference, enabling incremental evaluation updates after each move (Nasu, 2018).

A distinctive component is its difference based mechanism, where the system maintains an accumulator. When a piece moves, only relevant features (encoded through HalfKP relationships) are updated. This allows near instantaneous position evaluation. Quantization of weights into integer domains (8–16 bits) allows leveraging SIMD instructions for better speed on standard CPUs.

NNUE's introduction into major engines such as Stockfish and Komodo Dragon resulted in strength gains exceeding 100 Elo (Nasu, 2018); (Stockfish-Team, 2025).

6.2.1 · Architecture

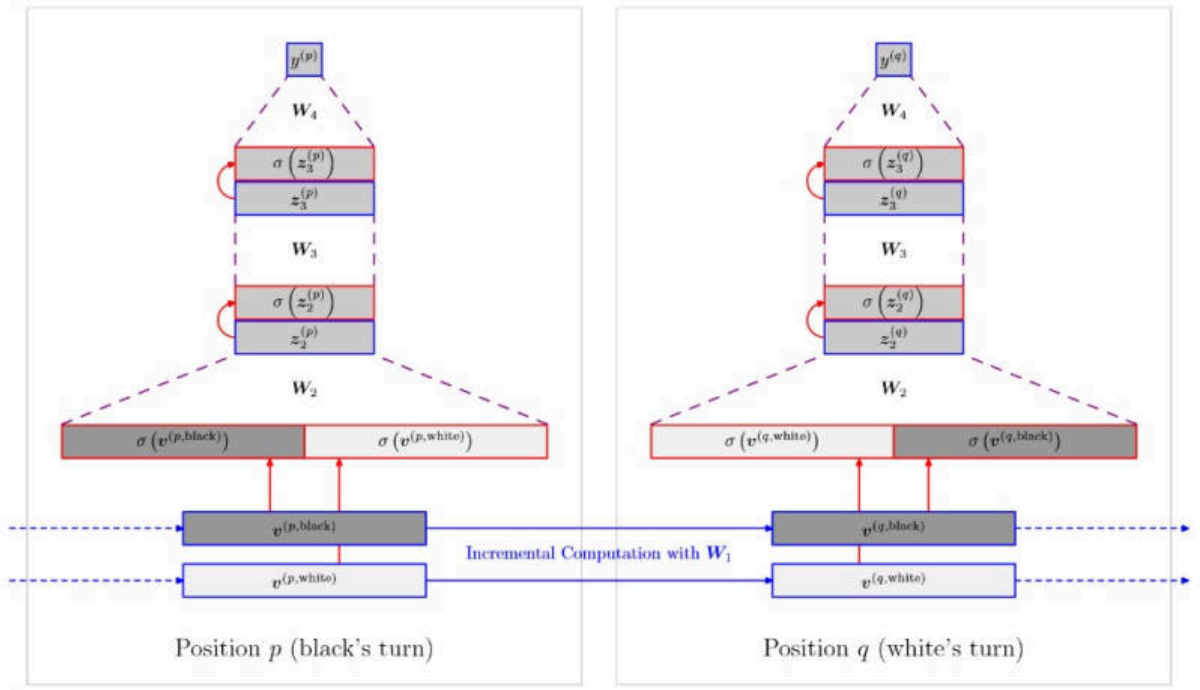

Figure 1: NNUE Stockfish Architecture

The NNUE architecture consists of four layers. The input layer uses a sparse binary representation called HalfKP (or HalfKAv2 in modern versions), where features represent piece presence on specific squares relative to each king's position. The network processes two "halves" simultaneously.

In the original HalfKP, each half receives 41,024 binary inputs (64 king positions $\times$ 641 inputs per position), connecting to a 256 neuron hidden layer per half, resulting in over 10 million weights. The modern HalfKAv2 uses 45,056 inputs per side mapped to a 512 neuron feature transformer per side.

The cornerstone of NNUE's efficiency is the accumulator mechanism. Rather than re-calculating the entire feature transformer output for each position during search, the engine maintains an accumulator. When a piece moves, only weights corresponding to the moved piece need updating: Subtract weights for old square, Add weights for new square. This transforms what would be an O(n) operation into an O(1) operation.

After the feature transformer, the network passes through three smaller fully-connected layers: 512 inputs $\rightarrow$ 32 outputs, 32 inputs $\rightarrow$ 32 outputs, 32 inputs $\rightarrow$ 1 output (evaluation score). These layers use ClippedReLU activation functions clipping values to [0, 127].

All network weights and intermediate values use quantized integer arithmetic. The feature transformer uses 16 bit integers, while subsequent layers use 8 bit integers. This quantization enables efficient SIMD operations using AVX2.

A quantitative gap exists in optimizing NNUE: developing superior feature sets and refining quantization and layer structure to achieve greater accuracy without sacrificing speed (Stockfish-Team, 2025). Additionally, sophisticated methods for automatically tuning complex sets of handcrafted heuristics are still needed (Bijl, Tiet and Bal, 2021, p.17).

6.3 · The State Of Neural Network Based Engines

Neural network evaluation architecture changes drastically based on whether it targets traditional CPU-bound search or selective MCTs systems. NNUE is optimized for CPU based low-latency with alpha-beta (Nasu, 2018); (Stockfish-Team, 2025), while AlphaZero's CNNs are better suited for GPU/TPU acceleration in MCTs (Silver et al., 2017). The integration of NNUE into Stockfish demonstrates a hybrid approach rather than complete replacement. Given that engines must maintain compatibility with consumer hardware, NNUE is often preferred despite MCTs potentially offering stronger evaluation (Stockfish-Team, 2025).

A core research gap remains in quantifying performance comparisons between highly optimized alpha beta engines with neural evaluations versus pure neural network-guided MCTs systems, especially under varying time controls and hardware constraints.

7 · System Analysis & Architecture Patterns

This section examines Stockfish and LC0 to examine how they translate theoretical techniques into working systems.

7.1 · The Classical Paradigm: Alpha-Beta + HCE

The classical approach dominated from the 1960s until around 2017, fundamentally relying on exhaustive search to a fixed depth, accelerated by alpha-beta pruning, and guided by hand-crafted evaluation functions based on human chess principles.

Stockfish, first released in 2008 as a fork of Glaurung, became the world's strongest chess engine through years of relentless optimization of the classical alpha-beta paradigm. Pre-NNUE Stockfish represented the pinnacle of hand-crafted evaluation and highly optimized search functions.

Stockfish maintains both BitBoard and mailbox representations together. It maintained 12 different BitBoards for efficient move generation and evaluation, and a 64-element array to provide O(1) piece lookups. Although this has approximately 2x representation overhead and synchronization cost, it enables both components to operate at peak efficiency.

Stockfish implements both Magic BitBoards and PEXT Boards, selecting between them at compile-time based on CPU capabilities. PEXT provides simpler code and 2.3% speedup (Fiekas, 2018, p.10) but requires Haswell+ CPUs. By maintaining both implementations, Stockfish achieves maximum performance on modern hardware while remaining functional on older systems.

Stockfish originally used YBWC for parallel search, but later switched to Lazy SMP noting that it scales better for high number of threads. Although no official explanation, the theoretical advantage of coordinated search was likely negated by mutex contention, thread creation/destruction overhead, complexity in managing thread pools, and helper threads idling.

Stockfish implements a sophisticated multi-tiered move ordering system: TT Move (highest priority), Winning Captures (MVV-LVA), Killer Moves (two per ply), Counter Moves, History Heuristic, and Losing Captures (searched last).

Pre-NNUE Stockfish had estimated Elo: 3450-3480, approximately 800 Elo above the best human players, and was dominant in computer chess championships. Despite its dominance, Stockfish's understanding was ultimately constrained by the extent to which humans could model their intuition. Complex positional factors like long term piece coordination were difficult to model in explicit heuristics. This limitation paved the way for the next generation of engines powered by neural networks.

7.2 · Neural Network Based: MCTs + Deep-Learning

December 2017 marked a paradigm shift. AlphaZero, having learned chess through self-play over four hours, played 100 games against Stockfish. The results were remarkable: AlphaZero won 28 games, drew 72, and lost none.

Neural network-based engines sacrifice speed, examining 80,000 positions per second compared to Stockfish's 70 million (Silver et al., 2017, p.4), in favor of evaluation accuracy and selectivity. Leela Chess Zero emerged as the open-source implementation of this paradigm.

7.2.1 · Case Study: Leela Chess Zero (LC0)

Leela Chess Zero, launched in 2018 as an open-source reimplementation of AlphaZero's approach, inverts the classical approach: instead of fast, shallow evaluation of millions of positions, it performs slow, deep evaluation of thousands of carefully selected positions.

LC0 and AlphaZero employ a deep residual convolutional neural network (ResNet), typically with 15-40 residual blocks. This neural network takes the board state as input (encoded in multiple planes) and outputs:

- **Policy Head:** A probability distribution over all legal moves
- **Value Head:** A win/draw/loss evaluation of the position

The tradeoffs include training costs (requires millions of self-play games and GPUs), inference speed (evaluates hundreds of times fewer positions), evaluation quality (learns subtle patterns beyond human modeling), and hardware dependency (requires GPU for competitive performance). The network's depth allows it to find patterns in pawn structures, piece coordination, and king safety through entirely learned features.

LC0 implements neural network-guided MCTs using the PUCT algorithm. Instead of random or exhaustive play, each node expansion queries the neural network for policy and value, selects moves based on $Q(s, a) + U(s, a)$ balancing exploitation and exploration, and back propagates evaluation up the tree. This approach allows MCTs to explore promising lines deeply rather than every line uniformly.

LC0 implements virtual loss: when a thread selects a node for exploration, it temporarily decreases that node's value as if it had lost. This discourages other threads from immediately exploring the same line. Once neural network evaluation returns, virtual loss is removed and replaced with actual evaluation. Virtual loss enables near-linear scaling up to 8-16 threads with a single GPU.

LC0 trains exclusively through self-play reinforcement learning:

1. Generate games using the current network with added exploration noise
2. Store positions, move probabilities, and game outcomes

3. Train the network to predict both the move probabilities (policy target) and game result (value target)
4. Deploy the improved network and repeat

The distributed nature of LC0′s training, with volunteers worldwide contributing self-play games, demonstrates both the power and challenge of this approach.

LC0 has:

- Estimated Elo: 3500-3550 (with large networks on strong GPUs),
- Approximately 50-100 Elo stronger than pre-NNUE Stockfish.
- Playing style is characterized by deep positional understanding and long-term planning
- Particularly strong in complex middle-games and closed positions.

Despite its strength, LC0 has limitations:

- Computational Requirements (requires high-end GPUs),
- Tactical Blindness (probabilistic MCTs occasionally misses forcing sequences that alpha-beta would find instantly),
- Opening Book Dependency (early networks struggle in sharp opening lines)
- Time Management (gradual convergence performs relatively worse in fast time controls)
- Explainability (neural network decision-making is opaque). These limitations highlight the complementary nature: neural MCTs excels at strategic understanding and pattern recognition, while alpha-beta excels at tactical calculation and computational efficiency. This realization led to the next evolution: a hybrid approach.

7.3 · Hybrid Approach

August 2020 represents another paradigm shift, as Stockfish officially integrated Efficiently Updatable Neural Networks (NNUE). This represented a combination of both traditional and neural network approaches, combining the computational efficiency of alpha-beta search with the pattern recognition capability of neural networks. This hybrid allowed Stockfish to see a dramatic strength gain of 100 Elo; the biggest jump in a long time.

Originally developed by Yu Nasu in 2018 for the Shogi engine YaneuraOu. Unlike LC0 and AlphaZero’s dependence on deep neural networks, NNUE uses a shallow but highly optimized architecture for CPU inference, and learns evaluation through supervised learning from billions of positions. This results in an engine that searches with alpha-beta but evaluates with neural networks.

The critical innovation of NNUE is incrementally updated accumulators. The NNUE network typically consists of:

- Input Layer (40,960 features),
- Hidden Layer (256-1024 neurons with ClippedReLU),
- Output Layers (two small layers producing evaluation score).

Instead of recomputing the entire network for each position, NNUE maintains an “accumulator” that tracks the hidden layer’s state. When a move is made, only the features affected by said move are updated, reducing evaluation cost.

The tradeoffs include:

- Evaluation speed (10-100x slower than hand-crafted evaluation, but 100-1000x faster than GPU-based deep networks),
- Search depth (reduced by 1-2 plies compared to classical Stockfish, but better position understanding),
- Memory footprint (network weights 20-50MB fit in CPU cache),
- Training requirements (requires hundreds of billions of training positions, but training is one-time and can be done offline).

Stockfish implemented aggressive quantization and CPU-specific optimizations:

- 8-bit Integer Quantization (network weights and activations use 8-bit integers instead of 32-bit floats),
- SIMD Vectorization (AVX2/AVX-512 instructions process 16-32 neurons simultaneously),
- Weight Permutation (network weights reordered to optimize cache access),
- Sparse Input Optimization (computation skips zero inputs).

Quantization introduces small evaluation errors (~5-10 Elo), but the speed gain is worth it. Combined optimizations achieve 3-5x speedup over naive implementation. Modern Stockfish evaluates 1-5 million positions per second with NNUE on high-end CPUs, compared to 50-70 million with classical evaluation. The 10-20x slowdown is acceptable because NNUE's superior evaluation quality means fewer positions need to be searched for the same playing strength.

Current Performance:

- Estimated Elo: 3600-3650,
- Improvement: 150-200 Elo stronger than pre-NNUE Stockfish,
- Comparison to LC0: Roughly equal strength with different stylistic characteristics,
- Dominance: Won TCEC Season 19 (November 2020) and has remained dominant in classical time controls.

The NNUE integration succeeded where previous neural network attempts failed because it respected the constraints of CPU-based alpha-beta search: Computational Efficiency (shallow architecture and incremental updates keep evaluation fast enough for deep search, 100-1000x faster than LC0′s networks while only 10-20x slower than classical evaluation), Training Efficiency (supervised learning from engine positions converges in days rather than months), Incremental Adoption (Stockfish could integrate NNUE while preserving existing search infrastructure), and Hardware Accessibility (CPU-only execution runs on commodity hardware, democratizing access to top-level chess engines).

Despite its success, NNUE-based engines have limitations:

- Evaluation Speed (still 10-20x slower than classical evaluation, limiting search depth in very fast time controls),
- Architecture Constraints (shallow network approaching strength ceiling, research continues on architectures balancing depth and efficiency),
- Training Data Quality (NNUE's strength depends on training data quality, generating high-quality training data remains active research area),
- Interpretability (like all neural networks, NNUE's decisions are opaque)

- Hardware Dependency (peak performance requires modern CPUs with AVX2/AVX-512 support).

The hybrid approach has become the dominant paradigm in chess programming as of 2025. Top engines are within 50-100 Elo of each other, with stylistic differences rather than clear strength hierarchies. The success of NNUE demonstrates that the future likely lies not in purely classical or purely neural approaches, but in carefully designed hybrids that leverage the strengths of both paradigms. Alpha-beta provides computational efficiency, while neural networks provide positional understanding and pattern recognition. Together, they have pushed chess engine strength to levels that would have seemed impossible just a decade ago.

Bibliography

Bijl, P., Tiet, A. and Bal, H.E. (2021) Exploring Modern Chess Engine Architectures. Available at: https://www.cs.vu.nl/~wanf/theses/bijl-tiet-bscthesis.pdf.

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.

Brange, H. (2021) Evaluating Heuristic and Algorithmic Improvements for Alpha-Beta Search in a Chess Engine. Available at: https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=9069249&fileOId=9069251.

Columbia, S. et al. (2023) Chess Move Generation Using Bitboards. Available at: https://libres.uncg.edu/ir/asu/f/Columbia_Sophie_Spring%202023_thesis.pdf.

Fiekas, N. (2018) Finding Hash Functions for Bitboard Based Move Generation. Available at: https://backscattering.de/magics2.pdf.

Herranz, J.G. and Qiu, Y.W. (2025) AlphaDeepChess: motor de ajedrez basado en podas alpha-beta = AlphaDeepChess: chess engine based on alpha-beta pruning. Trabajo de Fin de Grado. Available at: https://docta.ucm.es/rest/api/core/bitstreams/4e289e34-0b84-4c1b-9d19-bc0911cfb48b/content.

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.

Nasu, Y. (2018) NNUE: Efficiently Updatable Neural-Network-based Evaluation Functions for Computer Shogi. Translated by D. Klein. Available at: https://github.com/asdfjkl/nnue/blob/main/nnue_en.pdf.

Plaat, A. (1997) A Minimax Algorithm faster than NegaScout. technical report. Available at: https://arxiv.org/pdf/1404.1511.

Rasmussen, D. (2004) Parallel Chess Searching and Bitboards. Available at: https://www.cs.cmu.edu/afs/cs/academic/class/15418-s12/www/competition/www.contrib.andrew.cmu.edu/~jvirdo/rasmussen-2004.pdf.

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.

Silver, D. et al. (2017) Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. Available at: https://arxiv.org/pdf/1712.01815.

Stockfish-Team (2025) NNUE: Efficiently Updatable Neural Network — Stockfish Docs. Available at: https://official-stockfish.github.io/docs/nnue-pytorch-wiki/docs/nnue.html.

Vrzina, S. (2023) Piece By Piece Building a Strong Chess Engine.. Available at: https://www.cs.vu.nl/~wanf/theses/vrzina-bscthesis.pdf.

Zobrist, A.L. (1970) "A New Hashing Method With Application For Game Playing," The University Of Wisconsin [Preprint]. Available at: https://research.cs.wisc.edu/techreports/1970/TR88.pdf.

Świechowski, M. et al. (2022) “Monte Carlo Tree Search: A Review of Recent Modifications and Applications,” arXiv preprint arXiv:2103.04931 [Preprint]. Available at: https://arxiv.org/pdf/2103.04931.