Techniques In Chess Programming: A Comprehensive Review

Swoyam Pokharel

October 2025

Abstract

TODO

Contents

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, 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.

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). This framework, first formalized in the 1950s, 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, as described by Knuth and Moore, 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 $F(p) = f(p)$, 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.

- **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.

The fundamental assumption is that both players play perfectly, with Max choosing the move that maximizes $F(p)$ and Min choosing the move that minimizes $G(p)$. This ensures 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 $F(p)$ and $G(p)$, negamax uses a single function, $F(p)$, but one that is **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 as such is often preferred over minimax (Björnsson and Marsland, 2000, p.5). Similar to minimax, the game can be defined as a tree where nodes are positions (p) and the edges are legal moves (d) from position p that lead to successor positions ($p_1, p_2, ...p_d$).

- **Value Function**: The value function $F(p)$ represents the best value that the **player to move** can guarantee from position p, with the assumptoin that 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 the negative sign, $-F(p_i)$. It takes advantage of the fact that chess is a zero-sum game, i.e the advantage of a position to the opponent is the negative of said advantage to the current player. For instance,

- If $F(p_i) = +1$ (winning position for opponent), then $-F(p_i) = -1$ (losing position for current player)
- If $F(p_i) = -1$ (losing position for opponent), then $-F(p_i) = +1$ (winning position for current player)

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 with a branching factor of 35, at that number there are $\sim 10^{24}$ possible positions reachable just from the starting position. This, makes it unfeasible to do an exhaustive search, as the time complexity with just negamax is $O(b^d)$, where $b = \text{branching factor}, d = \text{depth}$ (Shannon, 1950, p. 4; Björnsson and Marsland, 2000, p. 4). This computational constraint has driven the development of pruning techniques that maintain search accuracy while dramatically reducing the nodes examined.

2.2.1 · Branch-and-Bound Optimization

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

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

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 achievable till now, F_1 computes and returns the value if it is less than the bound, or "$\geq$ bound" if it is equal or greater than the bound; simply put, once it determines that a position achieves a value atleast 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.

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, used to speed up search without loss of information. Recognized universally as the most vital algorithmic optimization for achieving practical search depth in traditional chess engines, it offers exponential complexity reduction in the best case (Björnsson and Marsland, 2000, p.1, p.11; Rasmussen, 2004, p.28; Brange, 2021, p.22; Vrzina, 2023, p.19). Similar to the above procedure F_1, alpha-beta pruning further improves efficiency 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,

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

(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 at least as good as what the opponent will allow β.

Thus, the minimizing player will not allow reaching this position, because we assume optimal play, so we prune that branch (Björnsson and Marsland, 2000, p.4).

Deep Cutoffs

A significant advantage of alpha-beta over the single bounded approach is its 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 makes deep cutoffs that F_1 is not capable of finding (Knuth and Moore, 1975, p.2, p.7). This capability represents a fundamental improvement in pruning efficiency that has made alpha-beta the foundation for all subsequent search optimizations (Knuth and Moore, 1975, p.6).

Proof Of Optimality

Knuth and Moore further investigated whether improvements beyond alpha-beta pruning existed, 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 a reasonable sense (Knuth and Moore, 1975, p. 6). This theoretical optimality established alpha-beta as the definitive pruning algorithm for game-tree search, cementing its dominance for decades in traditional chess engines.

In the best case, where the "best" move was examined first at every node, alpha-beta examines $W^{\lceil \frac{D}{2} \rceil} + W^{\lceil \frac{D}{2} \rceil} - 1$ 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** position; instead, they end at a certain depth and evaluate the position using heuristic evaluation functions. As such, to achieve performance closer to the theoretical best case, chess programs employ different move ordering heuristics like examining captures or checks first, or iteratively deepening to prioritize moves that performed well in shallower searches.

Challenging Alpha-Beta's Dominance

While Knuth and Moore's proof established alpha-beta's theoretical optimality within the framework of brute-force minimax search, recent developments challenge whether this remains optimal. AlphaZero's success with Monte Carlo Tree Search (MCTS) guided by deep neural networks demonstrated that selective search based on learned policy estimates can outperform traditional alpha-beta approaches, despite examining $\sim 1000\times$ fewer positions (Silver et al., 2017, p.3-p.5). This represents a paradigm shift, going from guaranteed pruning through mathematical bounds $[\alpha - \beta]$ to probabilistic selection through learned heuristics.

Traditional alpha-beta engines work using brute-force with guaranteed pruning, while neural network guided MCTS engines seek efficiency through highly accurate selectivity learned from self-play (Silver et al., 2017, p.3-p.5). While earlier MCTS implementations proved weaker than alpha-beta (Silver et al., 2017, p.12), coupling MCTS with deep neural networks achieved superiority, challenging the widespread belief that alpha-beta was inherently better suited for chess.

However, the subsequent integration of NNUE / ƎUИИ into Stockfish suggests a hybrid approach rather than a full replacement; with modern engines combining alpha-beta's efficiency with neural evaluation (Stockfish-Team, 2025). A core research gap remains in definitively quantifying the

performance comparison between these evolved approaches under varying time controls and hardware constraints.

2.3 · The Horizon Effect

Beyond pruning efficiency, search algorithms must also address some tactical limitations. Shannon was amongst the earliest to recognize what is known today as the "horizon effect" (Shannon, 1950, p.6). This effect describes a program's tendency to "hide" the inevitable material loss by making delaying moves until said loss is far enough out of its maximum depth (the horizon). This problem emerges from a lack of computing power that forces programs to limit the depth of the search and make the "best move" based on incomplete information (Brange, 2021, p.14).

The core issue is that positions evaluated at the edge of the search depth may appear favorable, but extending the search by even a few additional moves would reveal better alternatives (Bijl, Tiet and Bal, 2021, p.10-11). While alpha-beta pruning significantly enhances search efficiency and enables deeper analysis, it still remains vulnerable to the horizon effect since the problem persists at whatever depth the search terminates at.

Shannon acknowledged the importance of evaluating only those positions that are "relatively quiescent" (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). This matters because positions at the horizon frequently occur amidst tactical sequences like captures, checks, or other forcing moves, creating a situation that a static evaluation will not be able to capture accurately (Björnsson and Marsland, 2000, p.19).

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, extending search beyond the depth limit (horizon) to ensure evaluation occurs only in "quiet" positions (Björnsson and Marsland, 2000, p.7; Rasmussen, 2004, p.41; Bijl, Tiet and Bal, 2021, p.11). QS is a type of search extension that continues evaluation of all the forcing moves until a "quiet" position is reached. Rather than terminating the search at a fixed depth regardless of the position's characteristics, quiescence search adapts, extending analysis in tactical positions.

With that said, a consistent theoretical framework defining "true quiescence" remains undeveloped, forcing reliance on heuristic thresholds for when QS should stop (Björnsson and Marsland, 2000, p.8). This represents an ongoing research gap about what exactly is a "quiet" position means, with it's definition varying between implementations. As of now, no mathematical definition has emerged to replace the intuitive heuristics currently used.

Performance Impact

QS often comes with big performance impacts. When Tesseract added quiescence search to an engine already equipped with transposition tables, iterative deepening, and MVV-LVA move ordering, the results were:

- **Execution time**: Reduced from 709.48s to 266.21s (62.5% faster)
- **Evaluation score**: Increased from 7,314 to 8,520 (+1,206 points)
- **Effective branching factor**: Decreased from 5.99 to 4.23 (-1.76)

The time reduction, despite searching additional moves, occurs because accurate leaf node evaluations produce more effective pruning throughout the tree. The substantial score improvement demonstrates how severely the horizon effect degrades tactical play when unaddressed (Vrzina, 2023, p.20, p.31, p.50). This enforces the consensus that quiescence search, remains a valuable techniques for competitive chess engines, despite the true definition of a “quiescent” position being theoretically ambiguous.

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). In particular, the representation of the board has a significant impact on move generation performance. Multiple sources agree that BitBoards have emerged as the dominant method for representing board state in competitive chess engines, though reaching this verdict has revealed some nuances about the relationship between theoretical and practical performance (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

These are intuitive approaches to representing a chess board, with representations that mirror the physical board. The evolution of board representation from the 1950s onwards reflects a progression from intuitive array-based systems to more specialized BitBoard approaches (Bijl, Tiet and Bal, 2021, p.4-p.5).

The Two-Dimensional Array

This representation is intuitive because it directly corresponds to the physical board layout. In implementation, an 8×8 two-dimensional array `chess_board[8][8]` stores piece objects at indices matching their board positions

But, its intuitiveness, this comes with performance costs. Most notably, indexing the array requires calculating the memory location `8 * rank + file` and performing multiple boundary checks, making it very inefficient (Bijl, Tiet and Bal, 2021, p.4; Vrzina, 2023, p.6). In Bijl & Tiet's testing, they found that this 2D array approach was the slowest, coming in at `39.189 Mn/s` in Perft and `6.327 Mn/s` in search speed (Bijl, Tiet and Bal, 2021, p.19).

Mailbox

This representation mimics a physical board, generally using a single-dimensional array of 64 elements instead of a two two-dimensional one, where each index can either contain a piece or be empty. While simple, this representation is inefficient for move generation as it requires loops and conditional checks for things like off-board movement (Columbia et al., 2023, p.15). Thus, in practice, a more common approach is the `0x88`.

0x88

This is a variant of the mailbox approach that pads the array, resulting in a 16x8 array with sentinel values. This padding helps eliminate out-of-bounds checks, reducing them to a single sentinel value comparison (Bijl, Tiet and Bal, 2021, p.4; Vrzina, 2023, p.6-p.7). In performance tests, the 0x88 based approach was nearly equal to BitBoards in Perft speed, coming in at `46.496 Mn/s` (Bijl, Tiet and Bal, 2021, p.19). This finding challenges the conventional wisdom that bitboard's primary advantage lies solely in move generation speed.

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. First applied to chess in 1970 (Bijl, Tiet and Bal, 2021, p. 5), this insight uses each bit as a corresponding representation of a square on the

chess board (Rasmussen, 2004, p.47-p.50; Bijl, Tiet and Bal, 2021, p.5; Columbia et al., 2023, p.16-p.26). BitBoards represent board information using 64-bit integers, allowing logical operations (such as unions, intersections, and shifts) 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). Since most CPUs today use 64-bit instructions, this representation proves to be a very efficient approach for chess engines, making common operations such as filtering trivial.

This representation generally uses different BitBoards for each piece type and each color. Thus, the entire board is the logical sum (bitwise `OR`) of all these separate boards:

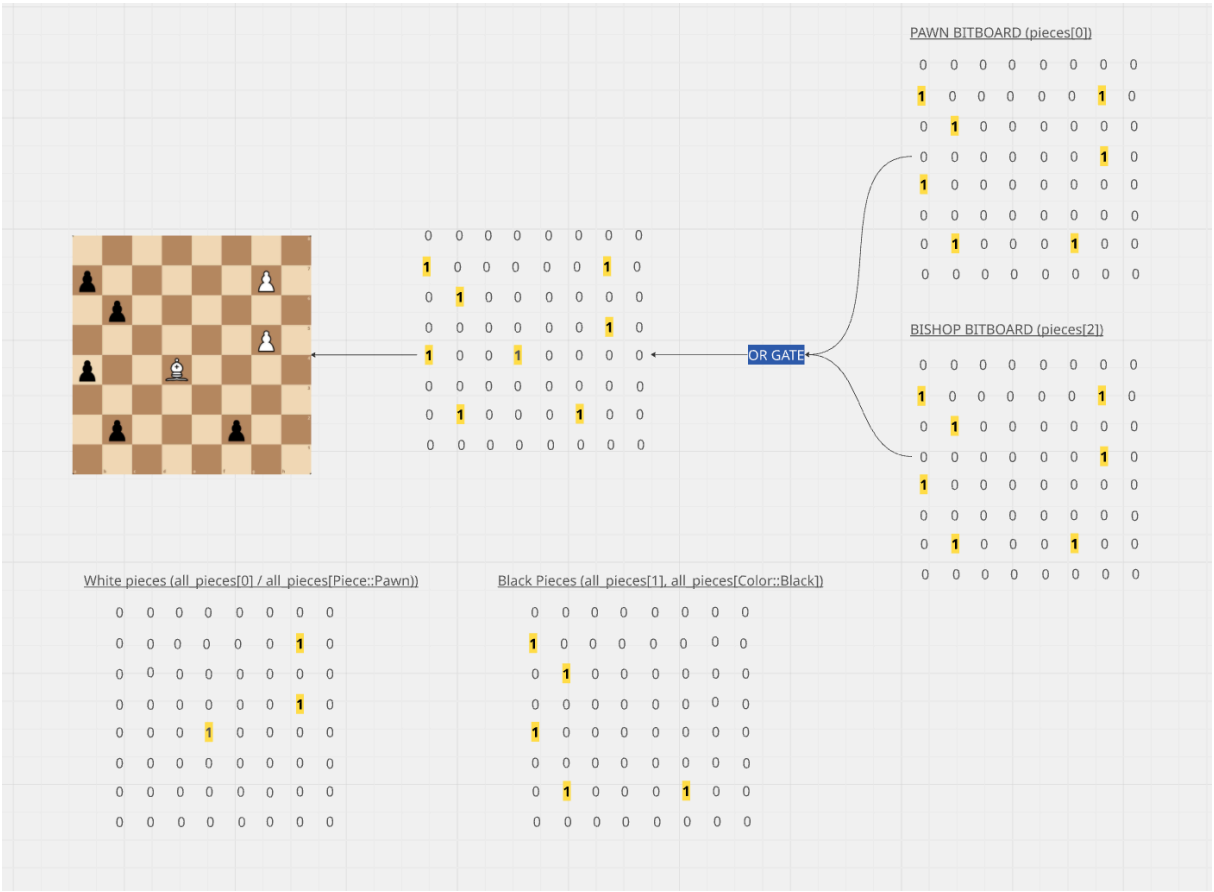

Figure 1: Example Position Represented Using BitBoards

It is worth noting that BitBoards aren't limited to representing piece occupancies; they can also represent attack patterns, which is the core idea behind pre-computed lookup tables for fast, constant-time move generation, with techniques such as PEXT or Magic Bitboards.

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. This comes at a slightly larger memory cost and the overhead of having to incrementally update multiple data structures per move (Bijl, Tiet and Bal, 2021, p.5; Vrzina, 2023, p.6-p.7), but the speed benefit's it offers are generally considered to be worth this overhead.

3.1.4 · Beyond Move Generation: Why BitBoards Dominate

While sources universally agree that BitBoards are the preferred representation for competitive engines (Bijl, Tiet and Bal, 2021, p.5; Herranz and Qiu, 2025, p.30), the reasons behind this preference reveal an important nuance.

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 (Perft tests) (Bijl, Tiet and Bal, 2021, p.20). This is surprising given that move generation, particularly through techniques like PEXT or Magic Bitboards, appeared to be the primary justification for adopting Bitboards.

However, BitBoards are preferred for full-purpose engines not primarily because of move generation, but because they accelerate other aspects such as evaluation, which heavily relies on fast bitwise operations where array based methods struggle (Bijl, Tiet and Bal, 2021, p.20; Vrzina, 2023, p.10). This distinction is important as the common assumption that board representation performance matters primarily for move generation is incorrect. Both architectures perform adequately in move generation, but evaluation benefits significantly from bitboard representations (Bijl, Tiet and Bal, 2021, p.20).

In Bijl & Tiet's complete engine tests, move generation accounted for only about 10% of processing time, with evaluation forming the primary bottleneck. Thus, BitBoards remain the optimal choice not solely due to move generation speed, but because of their performance advantages across the entire engine pipeline.

3.2 · Move Generation

Move generation is a fundamental aspect of any chess engine, as no engine should make illegal moves. Thus, given any position, generating all legal moves from that position quickly and accurately is critical. There are two different approaches to move generation.

3.2.1 · Pseudo-Legal vs Legal Move Generation

Engines approach generating legal moves differently. Some engines produce legal moves directly, while others first produce pseudo-legal moves and defer legality checks until later.

Pseudo-Legal Move Generation

A pseudo-legal move is one that follows the rules of how pieces typically move but does not account for whether the king is in check. If an engine takes this route, it is forced to check for legality afterward, generally by making that move on a copy of the board and verifying that it doesn't leave the king in check.

Legal Move Generation

A legal move is a subset of pseudo-legal moves that accounts for the king being in check. This approach is more complex than pseudo-legal move generation as it needs to account for pinned pieces, checking pieces, and typically requires producing a checkmask that later filters the moves (Columbia et al., 2023, p.65).

Pseudo-Legal Vs Legal Move Generation

Although pseudo-legal move generation adds to the running time of the move generation algorithm because of the need to check for legality during search (Columbia et al., 2023, p.11), it tends to be preferred. During the search phase with pruning heuristics such as alpha-beta, if a cutoff occurs, the engine avoids wasting time generating or verifying the legality of moves that would have been pruned regardless (Rasmussen, 2004, p.56; Bijl, Tiet and Bal, 2021, p.20). As such, modern engines, including the highest-rated engine Stockfish, prefer the pseudo-legal move generation approach.

3.2.2 · Generating Moves For Non-Sliding Pieces

To generate moves for non-sliding pieces (kings and knights), the standard approach is to use a pre-computed lookup table. The idea is to have a 64-element array that stores a BitBoard representing the attacks of a non-sliding piece from each square. For example, consider the following position:

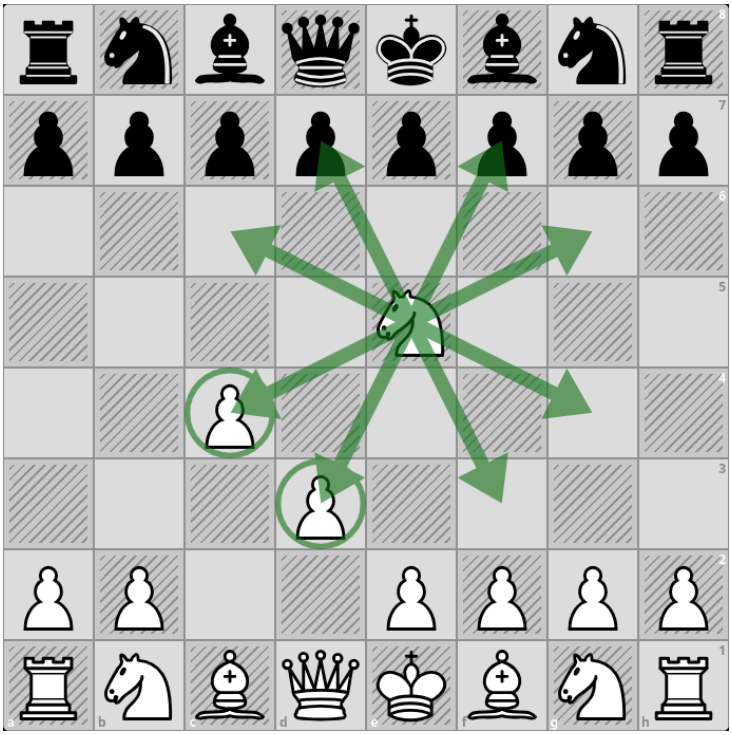

Figure 2: Attacks for a knight on e4

In this position, the arrows highlight all the legal moves the white knight on "e4" can make, and the circles highlight all the illegal moves (i.e., capturing a friendly pawn). Our attacks array `knight_attacks[64]` is defined such that:

```
// ("e4" = 28th bit on a BitBoard, assuming a1 = 0)

knight_attack[0] = ...
knight_attack[1] = ...
knight_attack[2] = ...
...
knight_attacks[28] = 44272527353856
...
knight_attack[63] = ...
```

In a BitBoard, where `"a1"` is the LSB, the square `"e4"` corresponds to the 28^{th} bit. As such, `knight_attacks[28]` represents the attack pattern of a knight at the `"e4"` square:

```
//
// where the number 44272527353856,
// displayed in the form of a BitBoard, is:
//
```

```
8 . . . . . . . .
7 . . . . . . . .
6 . . . X . X . .
5 . . X . . . X .
4 . . . . . . . .
3 . . X . . . X .
2 . . . X . X . .
1 . . . . . . . .
  a b c d e f g h

// note that the X's here represent 1's and the dots represent 0's
```

Now, to filter out capturing friendly pawns, it's simply `knight_attacks[28] & !friendly`, where `friendly` is another BitBoard representing all the friendly pieces for the current side to move (Brange, 2021, p.27; Bijl, Tiet and Bal, 2021, p.6; Vrzina, 2023, p.7). The same principle applies to kings, only the array's elements differ.

3.2.3 · Move Generation for Sliding Pieces

For sliding pieces like bishops, rooks, and queens, simple lookup tables do not suffice because their movement depends on the blocker configuration. The simplest approach is to iterate over the squares until the end of the board is reached, but this is inefficient. The evolution of techniques for handling sliding pieces represents a key advancement in chess programming: initial runtime calculations evolved to static table lookups facilitated first by complex hash techniques like Magic BitBoards, and then by the introduction of hardware-accelerated PEXT instructions (Fiekas, 2018, p.10). Thus, there are two main approaches to tackle this problem:

Magic BitBoards

Magic BitBoards are an advanced optimization used in chess engines to efficiently generate pseudo-legal moves for sliding pieces. They convert the move generation problem into a lookup operation; essentially a hashing technique that uses the blocker configuration as a key to index the correct pseudo-legal attack bitboard. Before 2013, Magic Bitboards were considered the fastest practical solution (Fiekas, 2018, p.26; Bijl, Tiet and Bal, 2021, p.8). This technique consists of three key components:

1. **Precomputing Phase**: At initialization, the engine first enumerates all possible blocker configurations for each square and piece type using the carry-rippler or similar technique. Afterward, the engine calculates and stores the resulting pseudo-legal moves. This creates a large but manageable lookup table mapping blocker configurations to attack BitBoards (Bijl, Tiet and Bal, 2021, p.7-p.8; Vrzina, 2023, p.10).
2. **The Magic Number**: Magic BitBoards use multiplicative hashing with carefully chosen constants (magic numbers) that act as perfect hash functions, transforming the blocker configuration bitboard into unique indices. These magic numbers are found through brute force, generally done once during development, and then used as static values afterward.

During runtime, the index is calculated as:

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

```
// blockers      : pieces along the ray of the sliding piece
// magic_number  : precomputed constant unique to each square
// shift_amount  : typically (64 - number_of_potential_blockers)
// and used as: sliding_piece_<bishop or rook or queen>[index]
```

Magic BitBoards provide constant-time move generation for sliding pieces and have become the de facto standard for modern engines (Bijl, Tiet and Bal, 2021, p.7; Herranz and Qiu, 2025, p.48-p.51). While other variants like Black magics and Fixed Shift Magics exist, they tackle the same fundamental problem (Fiekas, 2018, p.30).

PEXT Boards

The PEXT instruction, part of the BMI2 instruction set introduced in 2013 with Intel Haswell processors, acts as an alternative to magic BitBoards. PEXT directly computes the necessary index in a single CPU cycle, eliminating the need for magic numbers (Fiekas, 2018, p.10; Bijl, Tiet and Bal, 2021, p.8; Vrzina, 2023, p.10; Herranz and Qiu, 2025, p.51). The PEXT instruction performs parallel bit extraction:

```
source: 0b10110101
mask:   0b11001100
result: 0b1011 // (bits at positions where mask=1, packed together)
```

For sliding pieces, PEXT eliminates the need for finding and storing magic numbers. At runtime, the index is simply:

```
index = PEXT(blockers, ray_mask)
// and used as: sliding_piece_<bishop or rook or queen>[index]
```

Magic vs. PEXT

The distinction between Magic BitBoards and PEXT Bitboards represents a hardware-driven evolution rather than a purely algorithmic advancement. PEXT's theoritical superiority depends entirely on hardware support (machines running pre-Haswell for Intel and some pre-Excavator/pre-Zen for AMD dont support BMI2), making Magic Bitboards the essential fallback for platforms lacking this instruction (Fiekas, 2018, p.10; Vrzina, 2023, p.10).

However, empirical testing reveals a surprising result. 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's findings (Bijl, Tiet and Bal, 2021, p.20). The choice between Magic and PEXT thus becomes less about raw performance and more about implementation tradeoffs: PEXT offers simpler code without magic number generation, while Magic provides broader hardware compatibility.

Performance Implications Across Representations

Bijl & Tiet's study yielded the following results:

Type	Perft speed (MN/s)	Search speed (MN/s)
2D array based	39.189	6.327
0x88 based	46.496	7.216
Magic BitBoards	48.772	10.992
PEXT BitBoards	48.740	11.038

Table 1: Bijl & Tiet's findings comparing PERFT and Search Speed across representations

In summary, their findings challenge the general consensus that the main advantage of BitBoards is move generation speed (Rasmussen, 2004, p.49; Vrzina, 2023, p.6, p.10; Columbia et al., 2023, p.4). This study shows that mailbox approaches like `0x88` can keep up in Perft tests. However, in terms of evaluation, Bitboards are significantly faster, yielding more nodes searched during actual gameplay. (Bijl, Tiet and Bal, 2021, p.19).

3.2.4 · Move Representation

When an engine employs PEXT or Magic BitBoards, what it ends up getting as pseudo-legal / legal moves is a raw bitboard. This will not be sufficient as special moves, such as en-passant, castling, captures, promotions etc., require updating multiple different Bitboards. As such, to contextualize the `make_move` function, we need to pack the raw moves into an efficient structure.

The fundamental information that this structure has to capture is the `from` and the `to` square. Following the tradition of using unsigned integers to represent the entities, representing moves requires us to have an integer that is at least 12 bits long at minimum, with each 6-bits representing the `from` and the `to` square.However, modern engines like Stockfish use a 16 bit representation for the 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 the `from` square (0-63)
- 6 bits for the `to` square (0-63)
- 4 bits for the promotion piece type

This encoding offers significant advantages, the most notable ones being:

- Compact Storage: This representation fits into a single CPU register, enabling efficient passing and manipulation
- Speed: With direct bit manipulation, parsing the `from`, `to` and `promo` values are significantly faster compared to struct fields.
- Cache Efficiency: Smaller size means that more moves can fit into the CPU cache lines

Although engines mostly stick to a 16 bit representation, some split the bits differently. Another approach is the (6-6-2-2) encoding scheme:

- 6 bit: source
- 6 bit: destination
- 2 bit: move type
- 2 bit: promotion piece

This variant explicitly tags special moves, which simplifies the move execution logic and can help later during move ordering, but comes at a cost of limiting the promotion encoding.

Performance

When Tesseract adopted this 16-bit encoding scheme over a naive struct/class implementation, move generation speed increased by nearly 50% (Vrzina, 2023, p.12). This demonstrates how critical efficient move representation is to overall engine performance.

3.2.5 · Perft

Correctness and Validation

Perft, short for performance test (also referred to as move path enumeration), is a fundamental debugging and validating tool in chess engine development. It operates by recursively generating the entire game tree for a specific position up to a given depth and counting all of the resulting nodes (Columbia et al., 2023, p.67; Vrzina, 2023, p.16; Herranz and Qiu, 2025, p.41). A developer can compare the nodes that their engine calculates with reputable engines or visit websites that share a consensus such as chess programming wiki.

Performance Indicator

Because of the branching factor of chess, just 9 plies deep from the starting position yields over 2.4 trillion leaf nodes in the game tree. Due to this computationally heavy nature, Perft can also act as a measure of performance in an engine as evident in Tesseract's benchmarks and Bijl & Tiet's study (Bijl, Tiet and Bal, 2021, p.19; Vrzina, 2023, p.17).

4 · Foundations Of Evaluation

Shannon first introduced the concept of an approximate evaluation function $f(P)$ to guide chess engines in selecting the best move, as he recognized that searching the entire game tree ($10^{\{120\}}$) is unfeasible (Shannon, 1950, p. 4-6; Herranz and Qiu, 2025, p.18). This foundational insight established evaluation as a critical component of chess programming, with sources consistently agreeing 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).

Originally, Shannon described this evaluating function $f(P)$ as one based on a combination of various established chess concepts and general chess principles that approximates the long-term advantages of a position. He also noted that $f(P)$ would produce a continuous quality range that reflects the "quality" of a move, as no move in chess is completely wrong or right. Most notably, 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 and complex heuristics through the 1980s and 1990s (Silver et al., 2017, p.10).

This section, taking basis from Shannon's work, covers the techniques concerned with evaluating a position that chess engines have implemented over the years.

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). Historically, evaluation was an exercise in highly complex hand-crafted functions, relying on human chess knowledge to identify and weight features (Shannon, 1950, p.5; Silver et al., 2017, p.2). Although the exact combinations of these heuristics differed from engine to engine, the key factors Shannon suggested formed the foundation for all subsequent HCE development.

4.1.1 · Materialistic Approach

Material advantage is generally a stronger indicator compared to other positional factors and is also perhaps the simplest form of evaluation. The intuition is simple: "if you are up pieces, then you are probably winning." Material score, with weighted piece values, forms the quantitative baseline for evaluation (Shannon, 1950, p.17; Björnsson and Marsland, 2000, p.2; Herranz and Qiu, 2025, p.34). This technique simply subtracts the total material scores of the two sides, and these values are generally represented in centipawns:

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

(Björnsson and Marsland, 2000, p.3; Herranz and Qiu, 2025, p.34). Although these values are the de facto standard, Bijl & Tiet 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

Of course, in a game of chess, material is not everything; other factors such as king safety, mobility, and pawn structure determine whether a position is good or bad. Positional elements (Piece Square Tables/PSTs, 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). As such, to encapsulate these factors, chess engines have employed various techniques:

Piece Square Tables (PSTs)

Piece Square Tables are piece-specific, precomputed tables that assign a bonus or a penalty for a piece depending on its square. They are used to represent the fact that a piece's effectiveness is dependent on its position. For instance, take the following position into consideration:

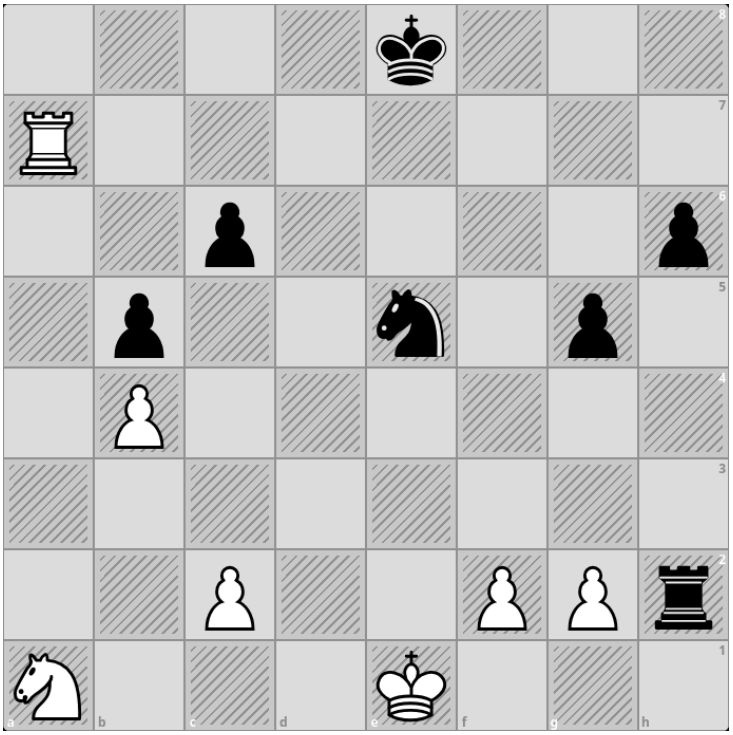

Figure 3: A Position With Equal Material Count

Although both sides have the same material count, the white knight is arguably better than black's as it is towards the center and covers more squares (Brange, 2021, p.31; Vrzina, 2023, p.33; Herranz and Qiu, 2025, p.35).

Tesseract's performance analysis shows that the implementation of PSTs caused the evaluation to go from 5640 to 8255, the single biggest evaluation impact amongst other heuristics. He also concludes that the most important heuristics for the evaluation function were the material and positional scores (Vrzina, 2023, p.38-p.39). This empirical validation demonstrates PSTs' critical role in bridging the gap between pure material counting and nuanced positional understanding.

Pawn Structure

Another positional aspect is the pawn structure; isolated, doubled, or backwards pawns are weak. As such, engines penalize the evaluation of such positions (Bijl, Tiet and Bal, 2021, p.15; Vrzina, 2023, p.36).

Mobility

Mobility can be defined as the number of legal moves available to a piece (Rasmussen, 2004, p.57; Bijl, Tiet and Bal, 2021, p.14). This is typically calculated by using popcount on our final attack BitBoard. A higher mobility score yields a better evaluation compared to a lower one.

King Safety

The King is the most important piece, as such its safety matters very much. Engines often approximate this by accounting for the proximity of enemy pieces and that of the friendly pieces. The bonus or penalty is then applied as needed (Vrzina, 2023, p.34; Herranz and Qiu, 2025, p.53).

Tapered Evaluation

To account for the fact that a piece's value and its position are also dependent on the stage of the game, this technique is used. The technique of tapered evaluation is agreed upon as necessary to adjust heuristic scores dynamically based on the game phase (midgame versus endgame) to reflect the shifting value of pieces and positional constraints (Bijl, Tiet and Bal, 2021, p.13; Herranz and Qiu, 2025, p.35). This is done to capture the fact that, say, a pawn in the early game is worth less compared to that in the endgame, or the fact that a king towards the center of the board is a huge problem in the early game but is actually wanted in the endgame. As such, engines typically employ 2 different sets of PSTs and interpolate between them depending on the stage of the game (Vrzina, 2023, p.33; Herranz and Qiu, 2025, p.35).

4.1.3 · Parameter Tuning

Tuning these PSTs and values is a way to increase the efficiency of these techniques. Bijl & Tiet's sequential tuning resulted in an average win rate increase of 15%. Their study also revealed that search depth was an important factor that determined the value of a piece. They found that the optimal Knight Material Score decreased with increasing depth, but the bishop pair increased. Their study also shows that stacked rooks were ranked high across all iterations of tuning (Bijl, Tiet and Bal, 2021, p.20). This depth-dependent variation in piece values challenges the notion of fixed material scores and suggests that optimal evaluation is context-sensitive. The findings imply that S. Droste and J. Furnkranz's reinforcement learning-derived values might've been more accurate than traditional centipawn assignments (Bijl, Tiet and Bal, 2021, p.12), though this remains an open question requiring further empirical validation.

5 · Search Enhancements & Optimizations

This section expands upon the fundamentals and cover more advanced optimization techniques that modern engines implement. Building upon the foundational search algorithms, these enhancements represent the accumulated refinements developed over decades of chess programming, transforming theoretical frameworks into practical, high-performance systems.

5.1 · Memory-Aided Search

5.1.1 · Transposition Tables

In any chess game, the same positions can be reached in different sequences of moves. For instance, take the following move sequences into consideration:

$$\text{Starting position} \rightarrow 1.\ e4\ e5\ 2.\ Nf3\ Nc6$$
$$\text{Starting position} \rightarrow 1.\ Nf3\ Nc6\ 2.\ e4\ e5$$

Although the order in which the moves were made are different, the final position reached is inherently the same. These sequences are called transpositions. When an engine explores the game tree, it encounters the same position in multiple branches. Without a transposition table, the engine would, for each of these branches, re-calculate the evaluation for the same position over and over again. Transposition tables are data structures, typically hash tables that store the evaluation of a position that has already been reached, for it to be re-used later (Björnsson and Marsland, 2000, p.13; Bijl, Tiet and Bal, 2021, p. 10; Herranz and Qiu, 2025, p.45). Sources universally recognize Transposition Tables (TPT) as essential aids to pruning, enabling exact forward pruning (avoiding redundant searches for previously solved positions) and providing crucial information for 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

Zobrist Hashing is the most popular way to generate the hash for game positions. It is universally accepted as the standard algorithm for computing position hash keys for transposition tables (Zobrist, 1970; Björnsson and Marsland, 2000, p.14; Bijl, Tiet and Bal, 2021, p.9; Vrzina, 2023, p.18). It is an incremental hashing technique that involves calculating the hash by `XOR`-ing together pregenerated 64 bit numbers corresponding to every piece type on every square, together with other game states like castling rights, en passant square, and the side to move. Although Zobrist Hashing is not perfect, as it yields a chance to collide (0.000003% with 1 billion moves stored) (Zobrist, 1970, p.10), the chance is small enough to be effectively zero for practical purposes. The key advantage of this technique is its incremental nature, allowing the hash to be updated in just 2-4 `XOR` operations, rather than recalculating from scratch. Zobrist keys are efficiently updated incrementally via 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).

Transposition Table Entry

Each entry in the table stores multiple things to maximize its effectiveness (Björnsson and Marsland, 2000, p.14; Rasmussen, 2004, p.100; Brange, 2021, p.36; Herranz and Qiu, 2025, p.48):

- The Zobrist Hash: The full 64-bit hash of the position. This is used to verify that the entry in the table is correct and detect index collisions

- Evaluation: The result yielded by the evaluation function
- Depth: The depth to which the search was calculated. This value is generally used to determine if the entry should be overridden with a more extensive search.
- Best Move: The best move found during the search, this is the foundation for move ordering in future searches.
- Age: This is used to identify stale entries from previous searches.
- Node Type: Due to alpha-beta pruning, not all searches result in exact scores, the node type represents these cases
  - `EXACT`: The search completed fully without cutoffs, the exact evaluation score for the position is searched. This occurs when the score falls between the search window ($\alpha <$ `score` $< \beta$)
  - `LOWERBOUND`: A beta cutoff occurred, meaning that the score is at least as good as the stored value, but it could also be better. This happens when a good move was found, (`score` $>= \beta$), causing the search to end early. Thus, this stored score can only be used if it's greater than or equal to the current β value
  - `UPPERBOUND`: An alpha cutoff occurred, meaning that none of moves scored better than the current best value (`score` $<= \alpha$). Thus, this stored score can only be used if it's less than or equal to the current α value.

Replacement Schemes

Since Transposition Tables are often fixed in size due to resource limitations (Zobrist, 1970, p.2; Björnsson and Marsland, 2000, p.16), entries in the table need to be overwritten. The most common replacement strategies are:

- Always Replace: The simplest strategy is to unconditionally overwrite any existing entry with a new one. While simple to implement, it has significant drawbacks. This strategy is prone to shallow searches replacing the deeper ones, losing valuable information. As such, this strategy is rarely seen in chess engines.
- Depth Preferred Replacement: This technique acknowledges that deeper searches are more valuable than the shallower ones, as such an entry is replaced only if the new entry is greater in depth than the currently stored one. This preserves the most computationally expensive searches, while still allowing updates where it is better.

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, such as Ronald de Man's Syzygy tablebases, to achieve perfect endgame play (Bijl, Tiet and Bal, 2021, p.21). These tablebases work by analyzing positions backwards from known outcomes (checkmate, stalemate, or drawn positions) to determine the optimal move and outcome for every possible configuration.

However, the storage requirements are substantial. The complete Syzygy tablebases scale dramatically with piece count: 3-5 piece endgames require 939 MiB, 6-piece endgames expand to 149.2 GB, and the full 7-piece tablebase consumes 16.7 TiB of storage source. This massive data requirement echoes Shannon's original proposal for a "dictionary" storing optimal moves for all positions (Shannon, 1950, p.4); an idea he dismissed as impractical due to size constraints. While Shannon's vision of

solving the entire game remains infeasible (10^{120} positions), modern engines have realized a practical subset: perfect play for the simplified positions that matter most, once sufficient material has been traded off the board.

5.1.3 · Refutation Tables

In chess, a refutation is a move that punishes the opponent's last move, proving that it was a mistake. For instance,

```
Black plays: Nf6 (developing the knight)
White responds: e5 (kicks the knight, "refutes" the idea)
```

and if this refutation worked well, the engine remembers to try the same move next time. A refutation table is a lightweight data structure that stores these effective refutations and main continuations. It is much simpler than the transposition table employing arrays instead of hashes, and are often referred to as space-efficient alternatives to transposition tables. This table is often preferred for low end devices with memory constraints. For devices with no memory constraint, this technique is still used as an additional aid for the search (Björnsson and Marsland, 2000, p.16).

5.2 · Iterative Deepening

Iterative Deepening, also known as "iterated aspiration search" or "progressive deepening"; a term first coined by de Groot (Groot, 1965), is an optimization technique that chess engines employ, especially those that implement alpha-beta pruning. All traditional engines employ Iterative Deepening as a standard procedure to manage search time and enhance performance by improving move ordering and hash table utility across increasing depths (Björnsson and Marsland, 2000, p.19; Brange, 2021, p.38; Bijl, Tiet and Bal, 2021, p.11; Vrzina, 2023, p.21). The idea behind iterative deepening is that when a search is requested to D plies, the search will first go 1-ply, then 2-ply, and so on until it reaches D. Although this may seem counter-intuitive, since it means we're repeating the same search over and over again in each iteration,which is true to some extent,engines use the information gained from these shallow searches to prioritize the best moves in deeper searches, which prunes a lot of branches right off the bat. If caches like transposition tables are also implemented, it's possible that iterative deepening searches faster than an immediate search to the same depth (Bijl, Tiet and Bal, 2021, p.10; Brange, 2021, p.38; Herranz and Qiu, 2025, p.32).

5.2.1 · Benefits of Iterative Deepening

Time Management

Iterative Deepening is perhaps the de facto standard for time management, as it ensures that if a search is interrupted (e.g., due to a time limit), we have the result from the previously completed depth. As such, the result from the previous shallower depth search can be used rather than the deeper but incomplete search (Björnsson and Marsland, 2000, p.17; Rasmussen, 2004, p.39).

Move Ordering

Iterative Deepening helps move ordering significantly. Generally, the promising moves from previous shallower searches are searched first, and as such, the likelihood of finding a good move goes up, causing more pruning. The overall efficiency of iterative deepening comes from the fact that it can use the information from the previous search to get the Principal Variation, and then use that information to reorder moves in the current deeper search.

Aspiration Windows
The search score from a previous position provides a strong approximation for the expected value of the current search. This can be utilized to set a tight aspiration window for the new search, thus leading to more cut-offs (Rasmussen, 2004, p.33; Vrzina, 2023, p.21).

In an empirical analysis of the KLAS engine, Brange mentions that the use of Iterative Deepening along with PV-Ordering caused the average search time to decrease by 28.7% on average (Brange, 2021, p.47).

5.3 · Advanced Alpha-Beta Variations

5.3.1 · Principle Variation Search (PVS) / Negascout
PVS or Negascout is an optimization of alpha-beta that exploits move ordering. Assuming the first move is likely best, PVS searches it with a full window $[\alpha, \beta]$ to determine its exact value. Subsequent moves are searched with a minimal window (typically $[\alpha, \alpha + 1]$) to quickly verify they score no better than the first move. These narrow window searches are significantly faster because they produce more cutoffs. If a minimal window search fails, indicating a move may actually be superior, PVS searches it again with the full window to find its true value. In this case, the algorithm takes the cost of a re-search, but with good move ordering this situation is rare enough that the approach remains beneficial overall (Björnsson and Marsland, 2000, p.9; Rasmussen, 2004, p.40; Vrzina, 2023, p.22).

5.3.2 · MTD(f)
MTD(f), short for Memory-enhanced Test Driver with node f, takes a different approach from PVS by performing multiple minimal window searches that converge on the minimax value. Instead of searching once with a full window, it starts with an initial guess (typically from a previous iteration or transposition table) and repeatedly searches with minimal windows around that guess. If the search yields a value $\geq \beta$, the true value is at least β, so the algorithm searches again with a higher window. If the search yields $< \beta$, the value is below β, prompting a search with a lower window. This process continues until the bounds converge on the exact minimax value.

This approach performs less work per individual search since minimal windows produce more cutoffs, but it requires searching multiple times. As such, a strong transposition table is essential to avoid redundantly re-computing positions across multiple passes. In practice, MTD(f) can outperform PVS when combined with effective hashing (Plaat, 1997). Despite its promise, PVS remains more widely adopted due to its simpler implementation and less strict dependency on transposition tables.

5.4 · Move Ordering Heuristics

Move ordering is critical for pruning effectiveness, as it establishes the threshold against which other positions are evaluated and thus subsequent inferior branches can be quickly ignored (Rasmussen, 2004, p.31; Herranz and Qiu, 2025, p.22). Move ordering heuristics (Killer/History) were developed specifically to maximize alpha-beta's pruning capability (Björnsson and Marsland, 2000, p.12). Several heuristics exist to improve move ordering:

5.4.1 · Transposition Table Move (TT Move)
The intuition behind the TT Move ordering is that the transposition table stores previously searched positions along with their best moves. So, when an engine encounters the same position, the table

tells it what move was best last time. Depending on the depth, it's fair to assume that the same move is still probably very good since it's not just a heuristic guess from the evaluation function but a proven score from the search itself. This is the key idea behind prioritizing TT moves. Thus, transposition tables help both avoid re-computation and improve move ordering (Björnsson and Marsland, 2000, p.13; Rasmussen, 2004, p.37).

5.4.2 · MVV-LVA

The Most Valuable Victim - Least Valuable Aggressor (MVV-LVA) is a simple yet reasonably effective heuristic for ordering captures. It prioritizes positive material trades; for example, ordering a pawn capturing a queen ahead of a queen capturing a pawn. The idea is simple, winning material is good, and doing so without risking your valuable pieces is even better. This heuristic is fast to compute and works well because captures that win material often cause beta cutoffs (Silver et al., 2017, p.11; Brange, 2021, p.34; Herranz and Qiu, 2025, p.42). In an assessment of the KLAS engine, MVV-LVA ordering resulted in the single biggest performance impact, decreasing execution time by 68.5% (Brange, 2021, p.45), which is evidence of its effectiveness.

5.4.3 · Killer Heuristics

Killer moves are aliases for non-capture moves that caused beta cutoffs at the same depth in sibling positions. The key insight is that if a move was strong enough to cause a cutoff in a position at this depth, that move is likely to do the same at other positions at the same depth too, and as such searching this move early is probably beneficial. Typically, the two most frequently occurring "killers" at each level of the search tree are tracked, and a quiet move that matches a tracked "killer" is given a bonus score to prioritize it amongst other quiet ones (Björnsson and Marsland, 2000, p. 12; Rasmussen, 2004, p.38; Herranz and Qiu, 2025, p. 42-p.43).

5.4.4 · History Heuristic

The History Heuristic tracks how often a move causes a beta cutoff across the entire search tree. Generally, this is done by maintaining a table indexed by `[from_square][to_square]`, incrementing the score each time that move causes a cutoff. Unlike killers, history is **global across all depths and positions** and thus captures broader patterns about which moves tend to perform well. The history heuristic is often applied to sort the remainder of the non-capture moves after other ordering schemes like killer moves have been applied (Björnsson and Marsland, 2000, p.12; Rasmussen, 2004, p.39).

5.5 · Selective Search Extensions

These are mechanisms used in game-tree searching to strategically increase the search depth of certain moves, beyond the fixed depth (Björnsson and Marsland, 2000, p.3). The primary purpose of selective search extensions is to shape the game-tree so that "interesting" positions are explored more thoroughly and uninteresting ones aren't. Shannon categorizes this as a **type B** strategy (Shannon, 1950, p.13). However, these extensions need to be controlled as the tree can explode in size if done too frequently or extensively (Rasmussen, 2004, p.43).

5.5.1 · Check Extensions

Checks are the most forceful type of move, as they limit the responses from the opponent. The rationale behind check extension is that, if an opponent is in check, it is reasonable to assume that it might lead to a checkmate, as such extending this might be beneficial. And since the opponent's responses are limited, it's not too computationally expensive. As such, check extensions are the most

common type of extension heuristic. Check extensions differ from quiescence search in that they occur during the main alpha-beta search before the depth limit is reached, while quiescence search happens after the normal search depth is exhausted and continues until the position is tactically quiet (Rasmussen, 2004, p.42).

5.5.2 · Pawn Pushes

In this mechanic, the search is extended if the pawn is near promotion, typically when a pawn is moved to the 7th (for white) or 2nd (for black) rank. Passed pawns advancing to these ranks create significant threats that can drastically alter the evaluation, making deeper analysis necessary to assess promotion threats and defensive resources accurately. This should optimally be added to quiescence search itself if possible (Björnsson and Marsland, 2000, p.8; Rasmussen, 2004, p.43).

5.5.3 · Singular Extensions

This extension focuses on situations where the best move is very clear or forced. The engine performs a reduced-depth search excluding the best move candidate; if all alternative moves fail significantly below the current best move's value, the best move is considered "singular" and the search is extended. This identifies tactically or strategically forced moves that warrant deeper analysis (Rasmussen, 2004, p.10; Bijl, Tiet and Bal, 2021, p.13).

5.5.4 · One Reply Extensions

When a position has only one legal move (or one non-losing move), the search is extended since the response is forced. Since there are no alternative moves to consider, extending the search incurs minimal computational cost while ensuring forced sequences are analyzed completely. This helps resolve tactical lines where the opponent has no meaningful choice (Rasmussen, 2004, p.42).

5.6 · Pruning Techniques

5.6.1 · Null Move Pruning (NMP)

Null move pruning exploits the observation that, in most positions, making any legal move is preferable to passing a turn. Sources recognize Null Move Pruning (NMP) as a highly effective speculative heuristic that can provide significant speedup (e.g., cutting 2 plies), provided constraints are applied to avoid illegal states (check) or unreliable results (zugzwang endgames) (Rasmussen, 2004, p.43; Silver et al., 2017, p.10; Bijl, Tiet and Bal, 2021, p.12; Vrzina, 2023, p.25). The technique operates by allowing the side to move to "pass" (make a null move), giving the opponent two consecutive moves, and searching the resulting position with reduced depth. If this deliberately weakened position still produces a score $\geq \beta$, the engine can safely assume that the current position is so strong that at least one real move will exceed β, allowing the subtree to be pruned (Rasmussen, 2004, p.43; Silver et al., 2017, p.10).

The search after the null move is typically performed with a reduced depth (commonly $D - R - 1$, where R is the reduction factor, usually 2 or 3) and a narrow window around β to quickly verify the position's strength. However, this technique relies on the fundamental assumption that zugzwang positions, where passing would be preferable to any legal move, are rare. Since zugzwang occurs more frequently in endgames with few pieces, engines typically disable null move pruning in such positions or when in check, as the null move assumption breaks down (Bijl, Tiet and Bal, 2021, p.12).

5.6.2 · Late Move Reduction (LMR)

Late move reduction exploits strong move ordering to reduce search effort on moves that are unlikely to be best. In a well-ordered move list, the most promising moves appear first, while later moves are statistically less likely to improve upon the current best line. Rather than searching all moves to the full depth D, LMR searches later moves to a reduced depth, typically $D - R$ where R increases with move number and decreases with depth. Unlike Principal Variation Search (PVS), which operates with narrow search windows at full depth, LMR fundamentally alters the search depth itself. To avoid missing tactical opportunities, LMR includes safeguards that prevent reduction of tactically critical moves such as captures, promotions, checks, check evasions, and killer moves. If a reduced-depth search returns a score within the $[\alpha, \beta]$ window,indicating the move may be better than expected,the engine re-searches the move at full depth. While this re-search incurs additional cost, effective move ordering ensures such cases are rare enough that the overall trade-off remains positive (Bijl, Tiet and Bal, 2021, p.12; Vrzina, 2023, p.26; Herranz and Qiu, 2025, p.55).

5.6.3 · The Sensitivity of LMR Implementation

The effectiveness of LMR is heavily dependent on move ordering quality. LMR is implemented in top engines like Stockfish, indicating its value when refined (Bijl, Tiet and Bal, 2021, p.12). However, several developers found implementing LMR reliably resulted in worse performance or Elo drops due to its very aggressive pruning settings causing blunders (Vrzina, 2023, p.27; Herranz and Qiu, 2025, p.63).

AlphaDeepChess reported no improvement from implementing LMR due to insufficient move ordering strength (Herranz and Qiu, 2025, p.65). In contrast, the Tesseract engine demonstrated significant performance gains, reducing average search time from 83.87 to 64.13 milliseconds, but with a corresponding decrease in score from 8584 to 8124 (Vrzina, 2023, p.26).

LMR is a sensitive technique whose net benefit is heavily dependent on the quality of auxiliary systems, especially accurate move ordering heuristics that prevent good moves from being misclassified as “late”. Weak supporting systems cause LMR to fail (Vrzina, 2023, p.27; Herranz and Qiu, 2025, p.63). This trade-off illustrates the delicate nature of LMR and other aggressive pruning techniques. A research gap remains in creating accessible and robust implementations of aggressive pruning like LMR that do not suffer search instability or blunders when used outside of the highly optimized environments of top commercial engines (Vrzina, 2023, p.26).

5.6.4 · Futility Pruning

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

This optimization is particularly effective when applied with quiescence search, as it helps limit the explosive branching factor of the quiescence tree. Futility pruning typically applies only at nodes one or two plies from the horizon and to quiet moves, as tactical moves (captures, promotions, checks) can cause non-linear evaluation changes that go against the futility assumption.

5.7 · Parallel Search

As modern CPUs have evolved to include multiple cores, parallelizing the search has become the natural next step. The intuition is simple: if one core is fast, multiple cores should be faster. However, the reality is more nuanced. Alpha-beta search is inherently sequential; the results from searching one move provide critical information for pruning subsequent moves. When the work is distributed across threads to search different subtrees simultaneously, this pruning information is not immediately available across threads. Each thread ends up searching more nodes than would be examined in a sequential search, because they lack real-time access to each other's cutoff discoveries. This is why parallelization yields diminishing returns: a speedup of only 9.2x was observed on 22 processors, far short of the theoretical 22x (Rasmussen, 2004, p. 3, p.78).

To prevent threads from redundantly searching the same positions, shared data structures like the transposition table are employed. However, concurrent access to these global structures introduces its own costs; synchronization overhead from mutex locks or atomic operations can become significant. The tree size growth from parallelization overhead appears to be roughly linear (Rasmussen, 2004, p. 78), meaning that the combined effect of sub-linear speedup and linear growth in nodes searched results in only modest time reductions when using many processors. At some point, adding more processors no longer translates to a meaningfully faster search. This section examines the two most common approaches to parallelizing chess search: Young Brothers Wait Concept (YBWC) and Lazy SMP.

5.7.1 · Young Brothers Wait Concept (YBWC)

The Young Brothers Wait Concept (YBWC) represents an early, theoretically principled approach to parallelizing alpha-beta search. The algorithm is straightforward: search the first child node sequentially with the main thread, then distribute the remaining "young brother" nodes among multiple threads for parallel evaluation. During the sequential phase, helper threads remain idle, waiting for the principal variation search to complete before they can begin their work (Rasmussen, 2004, p.62; Herranz and Qiu, 2025, p.55).

This design aligns with the structure of alpha-beta node types. In Type 1 (PV) nodes, where all children must be searched, YBWC's sequential first approach establishes tight alpha and beta bounds before parallelizing the remaining children (Rasmussen, 2004, p. 78). Similarly, for Type 2 (CUT) nodes with good move ordering, a cutoff typically occurs after searching the first child, meaning the young brothers never need to be searched at all,making the wait concept perfectly efficient (Rasmussen, 2004, p. 62). However, YBWC proves suboptimal for Type 3 (ALL) nodes, where all children must be searched regardless. Here, forcing the first child to be searched sequentially wastes potential parallelism, as all children could have been evaluated simultaneously from the start.

Despite its theoretical soundness, YBWC has struggled in practice. The AlphaDeepChess project implemented YBWC for its multithreaded search but observed performance degradation rather than improvement. The decline was attributed to synchronization overhead, the costs of thread creation and destruction, and the implementation's inability to effectively leverage a shared transposition table for concurrent access (Herranz and Qiu, 2025, p.62). These practical challenges have led modern engines, most notably Stockfish, to switch away from YBWC to Lazy SMP.

5.7.2 · Lazy SMP

Lazy Symmetric MultiProcessing (Lazy SMP) takes a very different approach to parallelization. Rather than carefully coordinating threads, it 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; allowing threads to redundantly search similar positions instead of enforcing perfect work distribution, sounds counterintuitive, yet proves remarkably effective in practice. To prevent threads from exploring identical lines simultaneously, implementations employ randomized move ordering at the root node, ensuring each thread's search diverges early (Brange, 2021, p.39; Vrzina, 2023, p.27).

In practice, Lazy SMP achieved a 33.1% reduction in average execution time on a four-core system in the KLAS engine (Brange, 2021, p.58) and a 40% speedup in Tesseract (Vrzina, 2023, p.27). However, these gains come with tradeoffs; memory usage increases substantially, and garbage collection overhead can become significant, as noted in the KLAS implementation (Brange, 2021, p.58). Nevertheless, despite these costs and its inherently wasteful nature, Lazy SMP remains the dominant multithreaded search method in modern chess engines, outcompeting more theoretically sophisticated alternatives through sheer simplicity and effectiveness.

A research gap still remains in determining the optimal way to integrate modern parallel processing techniques into the core alpha-beta algorithm to maximize parallel scaling benefits.

6 · Evaluation Optimizations & Enhancements

Despite their historical dominance and continued utility, HCE functions face a fundamental limitation: they rely on human domain expertise and are bounded by the strategies and heuristics humans can explicitly model (Shannon, 1950, p.5; Silver et al., 2017, p.2). This means that even the best HCE is theoretically limited to the level of the best human player's explicit understanding. This limitation becomes particularly apparent when modeling complex, non-linear positional relationships where intuition guides understanding, but precise formalization remains elusive (Silver et al., 2017, p.12; Nasu, 2018, p.1). This gap has paved the way for the shift toward neural network-based evaluation, which leverages machine learning to capture patterns that evade explicit human definition, resulting in demonstrably stronger evaluations. These days, the standard approach to counteract HCE's limitations is NNUE, which we explore in the following sections.

6.1 · Monte Carlo Tree Search and Neural Network Engines

Moving away from traditional alpha-beta architectures, engines like AlphaZero and Leela Chess Zero employ Monte Carlo Tree Search (MCTS), a fundamentally different approach to finding the best move. Unlike alpha-beta search which aims for exhaustive coverage within a depth limit, MCTS grows its tree asymmetrically, concentrating computational effort on the most promising variations (Silver et al., 2017, p.3). This represents the paradigm shift identified in the search algorithms literature: traditional high performance chess engines like Stockfish relied heavily on refined, hand-crafted heuristics guiding highly efficient alpha-beta search, while AlphaZero's algorithm entirely replaced alpha-beta search with a general purpose MCTS guided by a deep neural network, demonstrating that selective search based on learned policy value estimates can surpass the brute-force efficiency of alpha-beta search (Silver et al., 2017, p.3-p.5).

6.1.1 · MCTS Algorithm

MCTS operates through four iterative phases that build the search tree incrementally:

1. **Selection**: Starting from the root position, traverse the tree by selecting moves that balance exploring new possibilities with exploiting known strong lines, guided by the UCB1 (Upper Confidence Bound) formula.
2. **Expansion**: When reaching an unvisited position, add it to the tree as a new node.
3. **Simulation**: Evaluate the new position to estimate its value (traditionally via random playouts, but in AlphaZero using neural network evaluation).
4. **Backpropagation**: Update the value estimates and visit counts for all positions along the path from the new node back to the root.

This process repeats thousands of times per move, gradually building confidence about which moves are strongest.

6.1.2 · AlphaZero's Neural-Guided MCTS

In AlphaZero, MCTS is guided by a deep neural network $f_\theta(s)$ that takes the board position s as input and outputs two critical values (Silver et al., 2017, p.2):

1. **Policy ($\boldsymbol{p}$)**: A probability distribution indicating which moves are most promising.
2. **Value ($\boldsymbol{v}$)**: An estimate of the expected game outcome from this position (ranging from −1 for a loss to +1 for a win).

AlphaZero has no handcrafted chess knowledge beyond the basic rules and learns entirely through self-play reinforcement learning. The network trains by minimizing a loss function l via gradient descent, where l combines two components (Silver et al., 2017, p.3):

- **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

Notably, AlphaZero optimizes for expected outcome (accounting for draws as 0), whereas its predecessor AlphaGo Zero treated draws as losses, optimizing only for win probability (Silver et al., 2017, p.3).

6.1.3 · Comparison with Traditional Engines

This neural-guided MCTS approach differs fundamentally from traditional alpha-beta engines:

Aspect	AlphaZero / MCTS	Stockfish / Alpha-Beta
Primary Algorithm	Monte Carlo Tree Search	Alpha-Beta Pruning
Evaluation	Deep Neural Network	HCE / NNUE
Knowledge Source	Learned from self-play	Handcrafted + tuning
Search Strategy	Selective, focused on promising lines	Broad, exhaustive within depth
Evaluation Speed	80,000 positions/second	70,000,000 positions/second
Search Depth	Deeper in critical lines	Uniform depth with extensions
Hardware	GPU-optimized	CPU-optimized
Training Cost	Massive (thousands of TPU-hours)	Incremental tuning
Interpretability	Black box	Transparent heuristics

Table 2: Comparison of AlphaZero and Traditional Engine Approaches

The most striking difference is evaluation speed: AlphaZero examines approximately 80,000 positions per second while Stockfish evaluates roughly 70 million. However, AlphaZero compensates for this speed disadvantage through superior selectivity, using its neural network to prioritize the most promising lines and achieving superior results despite searching $\sim 1000 \times$ 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, challenging the widespread belief that alpha-beta was inherently better suited for these domains. In head-to-head competition, using 64 threads and a hash size of 1GB, AlphaZero convincingly defeated all opponents, losing zero games to Stockfish (Silver et al., 2017, p.5).

6.2 · NNUE (ƎUИИ Efficiently Updatable Neural Networks)

The Efficiently Updatable Neural Network (NNUE) represents a major shift in how chess engines particularly, approach evaluation functions. Originally proposed by Yu Nasu (Nasu, 2018) for computer shogi, NNUE introduces a hybrid paradigm that merges the pattern recognition strength of neural networks with the speed and deterministic precision of handcrafted linear evaluators. The shift towards neural networks began in specialized fields like Shogi (Sankoma-Kankei models) (Nasu, 2018), accelerating quickly with the successes of AlphaZero (2017) (Silver et al., 2017). The later creation of the NNUE architecture (2018-2019) enabled top alpha-beta engines like Stockfish to successfully migrate to neural network evaluation without sacrificing the speed of alpha-beta search, representing the current state of the art (Stockfish-Team, 2025). The architecture was first integrated into Stockfish in 2019, and a big performance leap was observed.

Unlike deep convolutional or reinforcement-learning models such as 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, which enables the network to update evaluations incrementally after each move, rather than recalculating from scratch. (Nasu, 2018)

A distinctive component of NNUE is its difference, based mechanism, where the system maintains an internal accumulator. When a piece moves, only the relevant features, encoded through HalfKP relationships, are updated. This allows near instantaneous position evaluation while preserving the expressive nonlinearity of a neural network. Furthermore, quantization of weights and activations into integer domains (often 8–16 bits) allows the network to leverage SIMD instructions for massive speed-ups on standard CPUs.

NNUE's introduction bridged the gap between hand engineered heuristics (HCE) and learned evaluation. It's implementation into major engines such as Stockfish and Komodo Dragon resulted in strength gains exceeding 100 Elo, demonstrating that lightweight neural architectures can coexist with traditional search algorithms without the computational demands of deep learning frameworks. (Nasu, 2018; Stockfish-Team, 2025).

6.2.1 · Architecture

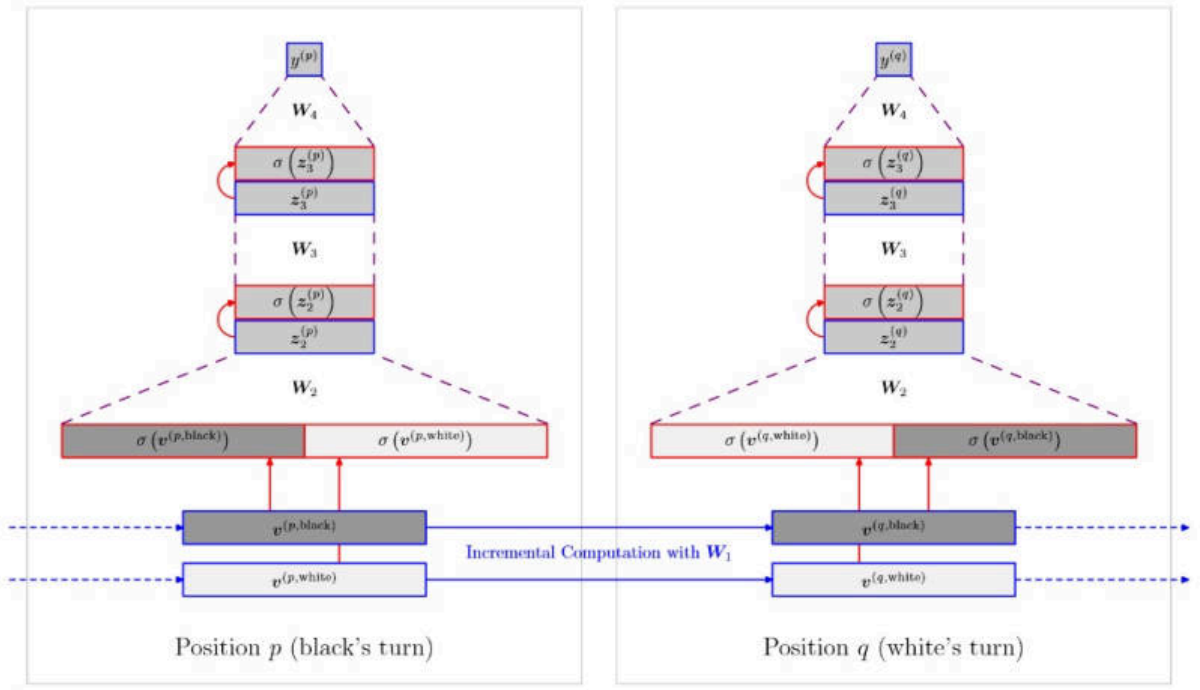

Figure 4: NNUE Stockfish Architecture

The NNUE architecture consists of four layers designed for efficient evaluation of positions. Unlike traditional deep neural networks used in other chess engines like Leela Chess Zero, NNUE's design prioritizes incremental updates during alpha-beta search, making it computationally efficient enough to maintain high search speeds.

6.2.2 · Input Layer and Feature Transformer

The input layer uses a sparse binary representation called HalfKP (or HalfKAv2 in modern versions), where features represent the presence of specific pieces on specific squares relative to each king's position. The network processes two "halves" simultaneously, one for each king, with each half containing information about all pieces on the board relative to that king's position.

In the original HalfKP architecture, each half receives 41,024 binary inputs (64 king positions × 641 inputs per position), where each input indicates whether a particular piece occupies a particular square or not. These inputs connect to a 256 neuron hidden layer per half, resulting in over 10 million weights in this feature transformer. This massive overparameterization allows the network to learn complex position dependent patterns.

The modern HalfKAv2 architecture improves upon this by using 45,056 inputs per side (11 piece types × 64 squares × 64 king positions) mapped to a 512 neuron feature transformer per side. This version eliminates redundancy by considering that the king's own square doesn't need to be encoded as a separate feature.

6.2.3 · The Accumulator

The cornerstone of enabling 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 the weights corresponding to the moved piece need updating:

- Subtract the weights for the piece's old square

- Add the weights for the piece's new square

This transforms what would be an O(n) operation (processing all active features) into an O(1) operation (updating only changed features). During alpha-beta search, where the engine evaluates millions of positions per second, this incremental update provides massive computational savings. For example,

```
move(piece from A to B):
  accumulator -= weights[piece][A]  // remove old position
  accumulator += weights[piece][B]  // add new position
```

6.2.4 · Hidden Layers

After the feature transformer, the network passes through three smaller fully-connected layers:

- First hidden layer: 512 inputs → 32 outputs
- Second hidden layer: 32 inputs → 32 outputs
- Output layer: 32 inputs → 1 output (evaluation score)

These layers use ClippedReLU activation functions, which clip values to a $[0, 127]$ range. The smaller size of these layers means they contribute minimal computational cost compared to the feature transformer.

6.2.5 · Quantization for Speed

All network weights and intermediate values use quantized integer arithmetic rather than floating point calculations. The feature transformer uses 16 bit integers, while subsequent layers use 8 bit integers. This quantization enables efficient SIMD operations using CPU instructions like AVX2, processing multiple values simultaneously.

6.2.6 · NNUE's Architectural Constraints and Research Gaps

NNUE's architecture is fundamentally defined to provide fast evaluation within the tight performance loop of a high speed, single-threaded alpha-beta search engine (Nasu, 2018; Stockfish-Team, 2025). A quantitative gap exists in optimizing NNUE: developing superior feature sets beyond HalfKP / HalfKAv2 and refining the quantization and layer structure to achieve greater accuracy without sacrificing the necessary speed (Stockfish-Team, 2025). Additionally, sophisticated methods for automatically tuning complex sets of handcrafted heuristics (like advanced Texel tuning approaches) are still needed to close the gap between man made and machine learned evaluations further (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 low-latency CPU inference within alpha-beta's tight performance loop, aiming for exhaustive search (Nasu, 2018; Stockfish-Team, 2025), while AlphaZero's CNNs are better suited for GPU/TPU acceleration and batch processing in MCTS (Silver et al., 2017).

The integration of NNUE into Stockfish demonstrates a hybrid approach rather than complete replacement of alpha-beta search. Given that engines must maintain compatibility with consumer hardware, NNUE is often preferred despite MCTS potentially offering stronger evaluation (Stockfish-Team, 2025).

Be that as it may, a core research gap remains in quantifying performance comparisons between highly optimized alpha beta engines with neural evaluations (like Stockfish/NNUE) versus pure neural network-guided MCTS systems, especially under varying time controls and hardware constraints.

7 · System Analysis & Architecture Patterns

This section examines two chess engines, namely Stockfish and LC0 to examine how they translate the theoretical technique into working systems. These 2 engines were chosen because they effectively encapsualte all major techniques in chess programming:

7.1 · The Classical Paradigm: Alpha-Beta + HCE

The classical approach dominated the field of chess programming from the 1960s until around 2017. This paradigm was fundamentally relied on exhaustive search to a fixed depth, and accelerated by alpha-beta pruning. It was guided by hand-crafted evaluation functions that were founded on the human chess principles. Engines like pre-2020 Stockfish were a testimant to this approach, acheiving very strong playing strength through years of incrementally refined heuristics and search optimizations.

Stockfish, first released in 2008 as a fork of Glaurung, became the world's strongest chess engine through years of tedious, relentless optimization of the classical alpha-beta paradigm. Before the integration of NNUE in 2020, Stockfish represented the pinnacle of hand-crafted evalaution and highlighy optimized search functions

The pre-NNUE Stockfish architecture consisted of several tightly coupled components, designed for maximum effeciency. The section below explores the architectural choices made int he development of Stockfish:

Bitboard + Mailbox Hybrid Representation

The Problem

Bitboards are proven to be excellent at bulk operations and enable computing attacks with the help of magic bitboards or PEXT boards, but they are inefficient for random access queries of piece and type occupancy on the board.

The Solution

Thus, stockfish maintains both the bitboard and mailbox representations together to counter bitboard's limitation. Stockfish maintained 12 different bitboards, one for each piece type and color for effecient move generation and evaluation, and a 64-element array to provide `O(1)` piece lookups.

Tradeoffs

- **Memory Cost:** Approximately 2x representation overhead (12 bitboards + 64-byte mailbox + auxiliary data). The board can be represented with just 8 bitboards without the mailbox.
- **Synchronization Cost:** Every move must update both representations
- **Speed Gain:** Move generation benefits from bitboard operations (no need to filter piece type with white_pieces & rooks or black_pieces & rooks) while evaluation benefits from mailbox lookups

Move generation uses bitboards with Magic BitBoards for sliding pieces, achieving constant-time lookup. Evaluation functions query the mailbox for piece-square table indexing. The hybrid approach enables both components to operate at peak efficiency rather than compromising one for the other.

Move Generation

The Problem

Stockfish uses precomputed bitboards for non-sliding pieces, but generating moves for sliding pieces (bishops, rooks, queens) requires determining which squares are attacked based on the blocker configuration. This is computationally intensive and occurs millions of times per second.

The Solution

Stockfish implements both, Magic Bitboards and PEXT Boards to acheive constant-time generation of moves for sliding-pieces. It selects between them at compile-time based on the CPU's capabilities.

Tradeoffs

- **PEXT Benefits:** Simpler code (no magic number generation), slightly faster (2.3% speedup) (Fiekas, 2018, p.10)
- **PEXT Costs:** Requires Haswell+ (Intel 2013) or Zen+ (AMD 2018) CPUs; some AMD processors have slow PEXT implementation
- **Magic Bitboards Benefits:** Universal compatibility, predictable performance
- **Magic Bitboards Costs:** Complex initialization (finding magic numbers), more code complexity

By maintaining both implementations, Stockfish achieves maximum performance on modern hardware while remaining functional on older systems. The compile-time detection ensures zero runtime overhead from abstraction.

Lazy SMP for Parallelization

Stockfish originally used Young Brothers Wait Concept (YBWC) for parallel search, but later switched to Lazy SMP noting that it scales better than YBWC for high number of threads.

Trade-offs

- **Redundant Work:** Threads inevitably search some identical positions
- **Implementation Simplicity:** No complex synchronization logic, no thread coordination overhead
- **Scalability:** Near-linear speedup up to 8-16 cores, diminishing returns beyond

Why YBWC "failed"

Although no official explaination, the theoretical advantage of coordinated search was likely negated by:

- Mutex contention on shared data structures
- Thread creation/destruction overhead
- Complexity in managing thread pools and work queues
- Helper threads idling while waiting for principal variation

Move Ordering Heuristics

The problem

It is established that Alpha-beta pruning's effectiveness depends critically on move ordering. Searching good moves first enables earlier cutoffs, potentially reducing the effective branching factor from 35 to under 6 in well-ordered trees.

The Solution

Stockfish implements a sophisticated multi-tiered move ordering system:

1. **Transposition Table Move** (highest priority): If the position has been searched before, try that move first
2. **Winning Captures** (MVV-LVA): Captures that win material, ordered by victim value minus aggressor value
3. **Killer Moves** (two per ply): Non-captures that caused cutoffs in sibling positions
4. **Counter Moves**: Moves that historically refuted the opponent's last move
5. **History Heuristic**: Global statistics on which moves tend to cause cutoffs
6. **Losing Captures**: Captures that lose material, searched last

Trade-offs:

- **Complexity:** Maintaining multiple overlapping heuristics increases code complexity
- **Memory:** History and killer tables consume additional RAM
- **Computation:** Move ordering itself takes time; as such it must be faster than searching misordered moves to make it a positive trade.

Strength:

- Estimated Elo: 3450-3480 (depending on hardware and time controls)
- Approximately 800 Elo above the best human players
- Dominant in computer chess championships (TCEC, CCC)

Limitations

Despite it's dominance, Stockfish had one major fundamental constraint,it's understanding of a chess position, is ultimately constrainted by the extent to which humans could model their intuition.

The handcrafted evaluation function, despite decades of refinement, was still fundamentally bounded by human understanding. Complex positional factors like long term piece coordination and king safety nuances in unusual positions were difficult to model in explicit heuristics. This limitation paved the way for the next generation of chess engines, ones powered by neural networks.

7.2 · Neural Network Based: MCTS + Deep-Learning

December 2017 marked a paradigm shift in the world of chess programming. AlphaZero, having learned the rules of chess through self-play over four hours, played 100 games against Stockfish, the strongest engine at that time. The results were remarkable: AlphaZero won 28 games, drew 72, and lost none.

Neural network-based engines like AlphaZero employ Monte Carlo Tree Search which is guided by deep neural networks that are trained through self-play. This approach sacrifices 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. This architecture is fundamentally dependent on GPU acceleration for neural network inference and represents a completely different philosophy than exhaustive search: search the most promising positions accurately rather than all positions efficiently.

Leela Chess Zero emerged as the open-source implementation of this paradigm, enabling the broader chess programming community to experiment with neural MCTS approaches without the computational resources of DeepMind.

7.2.1 · Case Study: Leela Chess Zero (LC0)

Leela Chess Zero, launched in 2018 as an open-source reimplementation of AlphaZero's approach, represents the neural network paradigm in chess programming. Unlike Stockfish's hand-crafted evaluation, LC0 learns position evaluation entirely through self-play, combining Monte Carlo Tree Search with deep neural networks trained via reinforcement learning.

The LC0 architecture fundamentally inverts the classical approach: instead of fast, shallow evaluation of millions of positions, it performs slow, deep evaluation of thousands of carefully selected positions. This section examines the architectural decisions that enable this paradigm shift.

Neural Network Architecture

The Problem

Traditional hand-crafted evaluation functions have a difficult time trying to capture the non-linear patterns that come with chess positions. Engines like Stockfish were seeing diminishing returns from their refinements because there is simply so much nuance that we can explicitly model in hand-crafted evaluation functions. This limitation made it so that traditional approaches struggled with long-term positional understanding and subtle tactical nuances that go beyond simple material or mobility metrics.

The Solution

LC0 and AlphaZero employ a deep residual convolutional neural network (ResNet) architecture, typically with 15-40 residual blocks. This neural network takes the board state as input (encoded in multiple planes representing piece positions, castling rights, en passant, etc.) and outputs two values:

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

The Tradeoffs

- **Training Costs:** This shoots up compared to traditional approaches. As these models require millions of self-play games and explicitly require GPUs to do so, these models aren't really viable for the average user.
- **Inference Speed:** These models by their nature sacrifice the number of positions evaluated in favor of accuracy and selectivity, often evaluating hundreds of times fewer positions than their traditional counterpart.
- **Evaluation Quality:** It balances out the low number of positions analyzed by learning subtle patterns over its self-play phase, which allows it to select the most promising lines and see qualities beyond what humans have explicitly modeled in their hand-crafted heuristics.
- **Hardware Dependency:** Requires GPU for competitive performance; CPU-only inference is prohibitively slow.

The network's depth allows it to find patterns in pawn structures, piece coordination, and king safety through entirely learned features rather than programmed rules.

Monte Carlo Tree Search (MCTS)

The Problem

Traditional alpha-beta search assumes that each chess position can be accurately evaluated in isolation. However, positions in chess are often too complex for immediate evaluation, and the true value only emerges after exploring possible continuations. Although traditional architectures have also

acknowledged this and attempt to counter this using techniques like quiescence search, MCTS takes a fundamentally different approach.

The Solution

LC0 implements a neural network-guided MCTS using the PUCT (Predictor + Upper Confidence Bounds for Trees) algorithm. Instead of random or exhaustive playouts, each node expansion consists of:

1. Querying the neural network for policy (move probabilities) and value (position evaluation)
2. Selecting moves to explore based on: Q(s,a) + U(s,a), where U balances exploitation and exploration
3. Backpropagating the neural network's evaluation up the tree

Tradeoffs

- **Selectivity vs. Coverage:** MCTS explores promising variations deeply rather than all variations uniformly
- **Uncertainty Handling:** Visit count-based exploration ensures underevaluated moves get reconsidered
- **Time Distribution:** Spends more time on critical positions, less on trivial ones
- **Search Instability:** Early search can dramatically shift as new variations are explored

This approach allows MCTS to explore promising lines deeply rather than every line uniformly, growing the tree asymmetrically. Unlike alpha-beta's deterministic best-first search, MCTS is inherently a probabilistic model. It gradually converges toward optimal play but initially explores suboptimal branches. This makes its play style appear "human-like" initially while finding "unconventional" moves that prove to be extremely strong several moves later.

Virtual Loss for Parallelization

The Problem

MCTS parallelization is challenging because multiple threads selecting moves simultaneously can all choose the same promising node, leading to redundant exploration and wasted computation.

The Solution

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 the neural network evaluation returns, the virtual loss is removed and replaced with the actual evaluation.

Tradeoffs

- **Exploration Diversity:** Threads naturally spread across different promising branches
- **Overhead:** Atomic operations required for thread-safe virtual loss updates
- **Tuning Sensitivity:** Virtual loss magnitude affects search behavior; too high causes over-diversification, too low causes redundant work
- **Batch Efficiency:** Enables gathering multiple positions for GPU batch inference, dramatically improving throughput

Virtual loss enables near-linear scaling up to 8-16 threads with a single GPU, with each thread exploring different variations. Combined with batched neural network inference (processing 256+ positions simultaneously), this achieves efficient GPU utilization.

Training via Self-Play

The Problem

Supervised learning from human games would limit the engine to human-level understanding. To surpass human play, the engine must discover novel strategies through exploration.

The Solution

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

Tradeoffs

- **Computational Cost:** Requires distributed infrastructure; LC0's training involved thousands of volunteers contributing GPU time
- **Training Stability:** Networks can plateau or even regress; requires careful hyperparameter tuning and network evaluation
- **Data Efficiency:** Learns from scratch without human knowledge, but requires millions of games to reach competitive strength
- **Emergent Understanding:** Discovers opening theory, endgame techniques, and tactical patterns independently

The distributed nature of LC0's training, with volunteers worldwide contributing self-play games, demonstrates both the power and challenge of this approach. Early networks (first few thousand games) play poorly, but strength improves dramatically as patterns emerge from accumulated experience.

Strength

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

Limitations

Despite its strength, LC0/AlphaZero has several constraints:

Computational Requirements: Due to their intrinsic model, these types of engines require high-end GPUs, making them inaccessible to many users. CPU-only inference is orders of magnitude slower.

Tactical Blindness: The probabilistic nature of MCTS occasionally misses forcing tactical sequences that alpha-beta would find instantly through exhaustive search. LC0 can overlook short, critical variations if they initially appear unpromising.

Opening Book Dependency: As these models converge, the early networks can struggle in sharp opening lines, requiring curated opening books to compensate and compete in tournaments.

Time Management: MCTS's gradual convergence makes it perform relatively worse in fast time controls where immediate evaluation is beneficial rather than deep exploration.

Explainability: The neural network's decision-making is opaque; unlike Stockfish's explicit evaluation terms, LC0 cannot explain why it prefers one move over another in human-understandable terms.

These limitations highlight the complementary nature of the two paradigms: 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 in chess programming, and the current state of the art: a hybrid approach.

7.3 · Hybrid Approach

- NNUE (2018), enabling classical engines to adopt neural networks
- Maintining CPU Effeciency & Neural Network benefits
- Current State Of The Art

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.

Groot, A.D. de (1965) Thought and Choice in Chess. The Hague: Mouton. Available at: https://iiif.library.cmu.edu/file/Simon_box00082_fld06608_bdl0002_doc0002/Simon_box00082_fld06608_bdl0002_doc0002.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.

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.

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].

Świechowski, M. et al. (2022) "Monte Carlo Tree Search: A Review of Recent Modifications and Applications," arXiv preprint arXiv:2103.04931 [Preprint].