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

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

2 · Foundations of Search

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

2.1 · Minimax and Negamax Framework

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

2.1.1 · Minimax Formulation

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

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

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

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

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

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

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

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

2.1.2 · Negamax Simplification:

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

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

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

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

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

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

The Negative Sign

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

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

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

2.2 · Pruning

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

2.2.1 · Branch-and-Bound Optimization

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

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

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

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

2.2.2 · Alpha Beta Pruning

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

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

Formally, Knuth and Moore define it as,

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

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

Deep Cutoffs

A siginificant advantage of alpha-beta over the single bounded approach is it's ability to do "deep cutoffs". Knuth and Moore demonstrated that $F_2(-\infty, +\infty)$ examines the same number of nodes

as $F_1(p, \infty)$ until the fourth look ahead level, but on the fourth and beyond levels, F_2 occasionally make deep cutoffs that F_1 isn't capable of finding. (Knuth and Moore, 1975, p.2, p.7).

Proof Of Optimality

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

In the best 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** postion, instead, they end at a certain depth and evalute the position using heuristic evaluation functions. As such, to acheive performance closer to the theoritical best case, chess programs emply different move ordering heuristics like examining captures or checks first, or iteratively deepening to prioritize moves that performed well in shallower searches.

2.3 · The Horizon Effect

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 it's maximum depth (the horizon). This problem emerges as a lack of computing power that force 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). A quiescent position is one that can be acessed accurately without needing further deepening (Björnsson and Marsland, 2000, p.7). This matters because positions at the horizon frequently occur with tactical sequences like captures, checks or other forcing moves, creating a situation that defies the accurate static evaluation (Björnsson and Marsland, 2000, p.19).

2.3.1 · Quiescience Search

Quiescience search is the principle approach to solving the horizon effect problem, by making sure that a position is stable before evaluation. It is a type of search extention that continues evaluation of all the forcing moves until a "quiet" position is reached. Rather than terminating the serach at a fixed depth regardless of the position's characterstics, quiescence search adapts, extending analysis in tactical positions.

Performance Impact

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)

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.

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 Two-Dimensional Array

This is arguably the most intuitive representation as it directly reflects a normal chess board. Despite its intuitiveness, this approach comes with performance costs. Indexing the array requires calculating the memory location `8 * rank + file` and performing multiple boundary checks, making it inefficient. (Bijl, Tiet and Bal, 2021, p.4; Vrzina, 2023, p.6). Bijl's testing showed that this 2D array approach was the slowest, coming in at 39.189mnps in Perft and 6.327mnps 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, where each element 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 million nodes per second. (Bijl, Tiet and Bal, 2021, p.19)

3.1.2 · Bitboard Revolution and Modern Realization

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. (Columbia *et al.*, 2023, p.16-p.26), (Rasmussen, 2004, p.47-p.50; Bijl, Tiet and Bal, 2021, p.5). 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. Since most CPUs today use 64-bit instructions, this representation proves to be a very efficient approach for chess engines. Consider the following position:

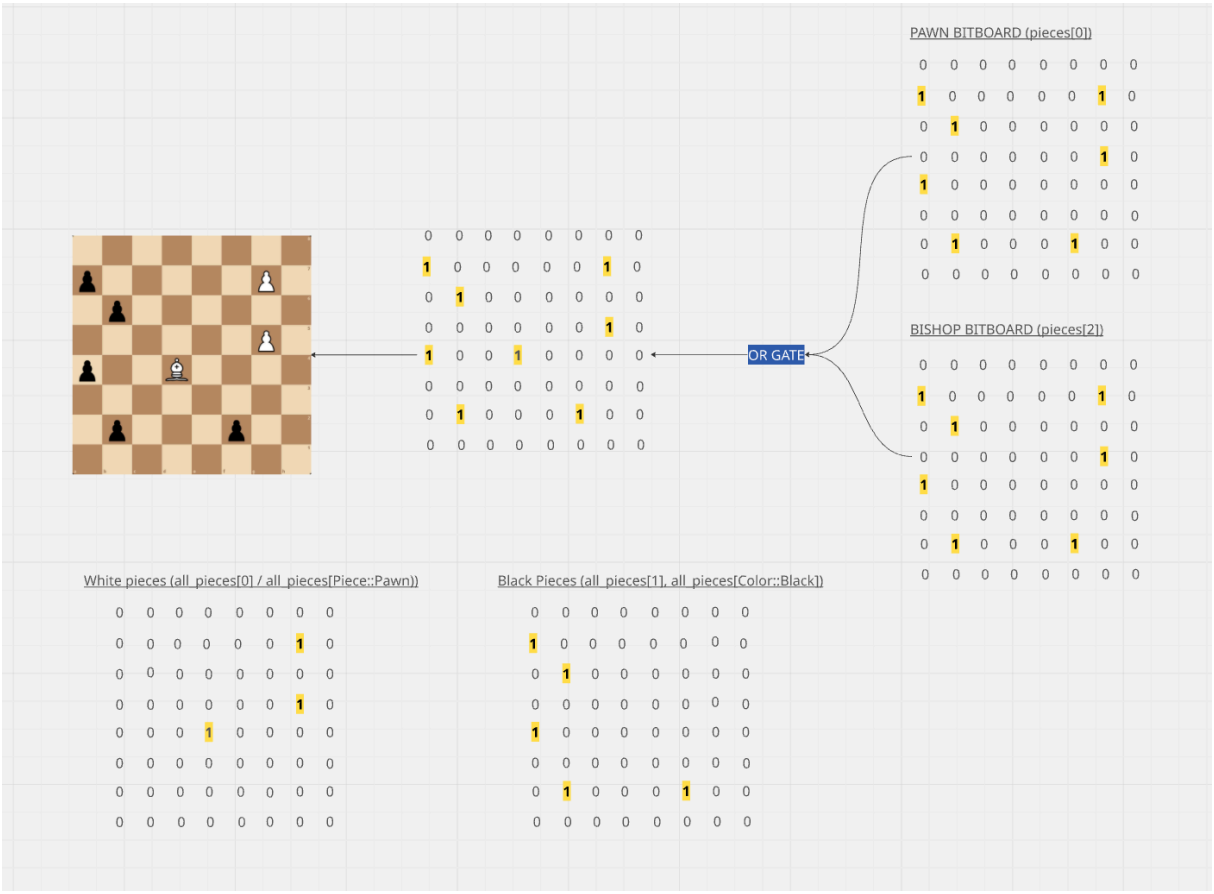

Figure 1: Bitboards Representing A Position

The bigger impact of this representation is that it enables other concepts in move generation, such as PEXT boards and Magic Bitboards, while also making filtering operations trivial. If we want all the white pieces, it's simply `all_pieces & white_pieces`. 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, which we'll examine after this section.

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).

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)

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:

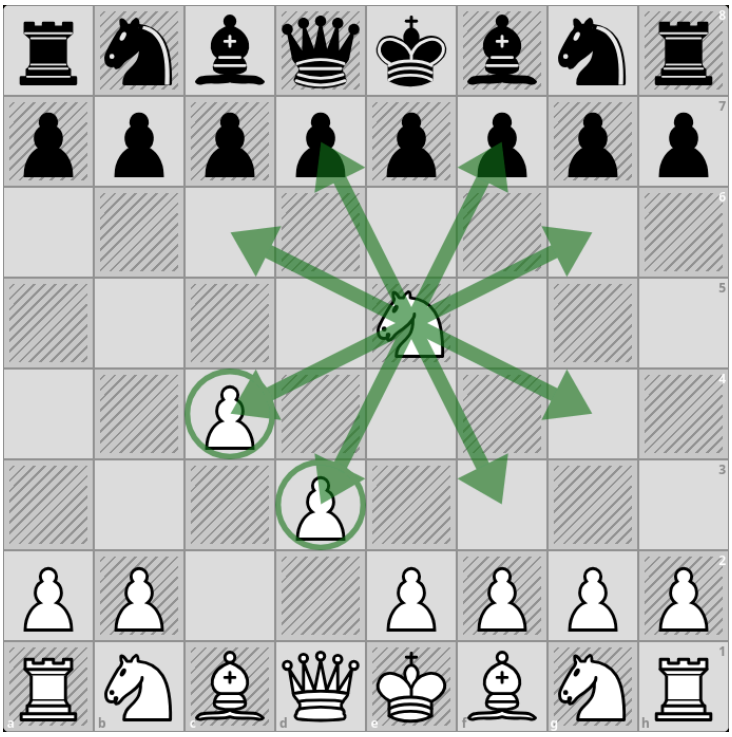

Figure 2: Attacks for a knight on e4

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] = ...

// where the number 44272527353856, displayed in the form of a bitboard,
is:
// note that the X's here represent 1's and the dots represent 0's
8 . . . . . . . .
7 . . . . . . . .
6 . . . X . X . .
5 . . X . . . X .
4 . . . . . . . .
3 . . X . . . X .
2 . . . X . X . .
1 . . . . . . . .
  a b c d e f g h
```

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), (Vrzina, 2023, p.7), (Bijl, Tiet and Bal, 2021, p.6). The same principle applies to kings as well—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 don't 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. 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, they're a hashing technique that uses the blocker configuration as a key to index the correct pseudo-legal attack bitboard. 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. Generally speaking, they employ the carry-rippler technique to enumerate across the variants. 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.

- These magic numbers transform 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 a constant-time algorithm for generating moves for sliding pieces and have thus 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, Fixed Shift Magics etc, do exist, they tackle the same fundamental problem. (Fiekas, 2018, p.30)

PEXT Boards

The PEXT instruction, part of the BMI2 instruction set, acts as an alternative to magic bitboards and multiplicative hashing. The PEXT instruction performs parallel bit extraction in a single CPU cycle:

```
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]
```

This approach is often preferred as it replaces the hashing algorithm with a single instruction that runs in one CPU cycle. (Bijl, Tiet and Bal, 2021, p.9-p.10; Vrzina, 2023, p.9-p.10)

Performance implications of these representations.

In Bijl & Tiet's tests, they found that in a complete engine, move generation accounts for only a small part of the entire processing time, consuming on average only about 10% of resources. The overall bottleneck is mainly during evaluation, not move generation. Their 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

Their findings are interesting because the general consensus that the main advantage of the bitboards approach is its speed in move generation (Vrzina, 2023, p.6, p.10), (Rasmussen, 2004, p.49; Columbia *et al.*, 2023, p.4). However, this study shows that mailbox approaches like `0x88` can still keep up. That said, in terms of evaluation, bitboards are still the fastest, yielding more nodes searched during actual gameplay. As such, bitboards remain the best option, but not solely because of move generation speed, but because of their overall performance benefits. (Bijl, Tiet and Bal, 2021, p.19)

Amongst the two bitboard approaches, although PEXT boards should have been faster, atleast theoritically, Bijl & Tiet's findings show no meaningful difference in engine speed between PEXT boards and Magic bitboards. However, PEXT boards do offer the benefit of not having to find and store magic numbers, thus avoiding that memory overhead.

However, it's worth noting that older chipsets (machines running pre-Haswell for Intel and some pre-Excavator/pre-Zen for AMD) don't support BMI2, making it mandatory to use magic bitboards if support for these systems is desired. (Vrzina, 2023, p.10)

3.2.4 · Move Representation

When working with PEXT or Magic bitboards, what we end up getting as pseudo-legal moves is raw bitboard. This won't 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 effecient structure.

The fundamental information that this structure has to capture is the `from` and the `to` square. As such, following the tradition of using unsigned integers to represent the entities, requires us to have an integer that is atleast 12 bits long at minimum, with each 6-bits representing the `from` and the `to` square . Generally, modern engines like Stockfish, use a 16 bit representation for the moves. (Vrzina, 2023, p.12), (Shannon, 1950, p.10; Bijl, Tiet and Bal, 2021, p. 8-9)

```
0000 000000 0000
____ ______ ______
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 effecient passing and manipulation
- Speed: With direct bit manipulation, parsing the `from`, `to` and `promo` values are significantly faster compared to struct fields.

- Cache Effeciency: 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 tessaract switch to this 16 bit encoding scheme, from a naive struct/classes implementation, it reported an almost 50% increase in move generation speed, which is quite significant. (Vrzina, 2023, p.12)

3.2.5 · Perft

Correctness and Validation

Perft, short for performance test, (also reffered 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 upto a given depth and counting all of the resulting nodes. (Herranz and Qiu, 2025, p.41), (Columbia *et al.*, 2023, p.67), (Vrzina, 2023, p.16). 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.

Peformance Indicator

Because of the branching factor of chess, just 9 plies deep from the starting position yileds 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)

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)

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) Although the exact combinations of these heuristics differed from engine to engine, the key factors Shannon suggested were:

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." 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 do 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 isn't everything; other factors such as king safety, mobility, and pawn structure determine whether a position is good or bad. 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:

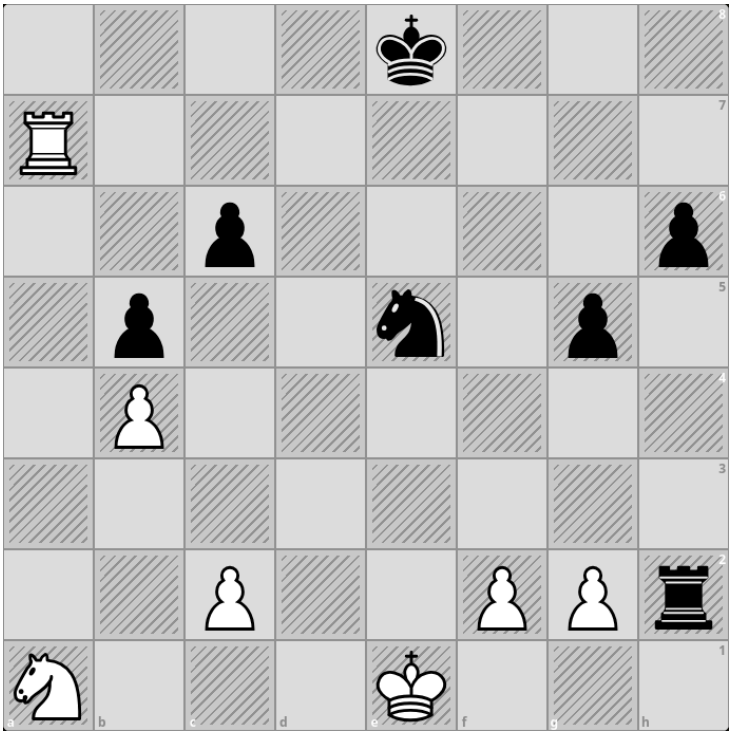

Figure 3: A Position With Equal Material Count

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)

Tessaract'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)

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. 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 implies that the findings of S. Droste and J. Furnkranz might've been more accurate. (Bijl, Tiet and Bal, 2021, p.12)

5 · Search Enhancements & Optimizations

This section now will expand upon the fundamentals and cover more advanced optimization techniques that modern engines implement.

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 it reached is inheretly 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 evaluatino 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)

Zobrist Hashing

Zobrist Hashing is the most popular way to generate the hash for game positions. 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 isn't 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 it's incremental nature, allowing the hash to be updated in just 2-4 `XOR` operations, rather than recalculating from scratch. (Zobrist, 1970, p.5, p.10; Björnsson and Marsland, 2000, p.14; Rasmussen, 2004, p.36; Brange, 2021, p.37; Vrzina, 2023, p.18; Herranz and Qiu, 2025, p.46).

Transposition Table Entry

Each entry in the table stores multiple things to maximize it's 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 collissions
- 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 represnts 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 atleast as good as the stored value, but it could also be better. This happens when a good move was found, (`score` >= β), 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` <= α). 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, loosing 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 · 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 a space-effecient alternatives to transposition tables. This table is often preffered 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. 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, it 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 · MDT(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). 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 check-mate, 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. 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)

The effectiveness of LMR is heavily dependent on move ordering quality. 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, though with a corresponding decrease in score from 8584 to 8124 (Vrzina, 2023, p.26). This trade off illustrates the delicate nature of LMR and other agressive pruning techniques.

5.6.3 · 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 againt 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 isn't 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 two prominent 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's 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 LazySMP.

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.

6 · Evaluation Optimizations & Enhancements

HCEs, despite of their historical value, have objective drawbacks. The main drawback of HCE is that it is inheretly designed on a human metric and heuristics. This limitation means that even the best HCE is limited to the same level as the best human player. These days the standard goto method to counteract this is NNUEs

6.1 · NNUE (ƎUИИ Efficiently Updatable Neural Networks)

The Efficiently Updatable Neural Network (NNUE) represents a major shift in how modern game engines—particularly in chess and shogi—approach evaluation functions. Originally proposed by Yu Nasu (2018) for computer shogi, NNUE introduced a hybrid paradigm that merges the pattern recognition strength of neural networks with the speed and deterministic precision of handcrafted linear evaluators. The architecture was later adapted to chess, first integrated into Stockfish in 2019, a performance leap unprecedented since the rise of alpha–beta–based engines.

Unlike deep convolutional or reinforcement-learning models such as AlphaZero, NNUE is designed explicitly for CPU execution rather than GPU acceleration. It employs a fully connected, shallow neural network, optimized for rapid, low-precision inference. The efficiency derives from three principles—input sparsity, input stability, and incremental updatability—which enable the network to update evaluations incrementally after each move, rather than recalculating from scratch.

A distinctive component of NNUE is its difference-based computation mechanism, where the system maintains an internal accumulator of first-layer activations. When a piece moves, only the relevant features—encoded through HalfKP (Half King–Piece) relationships—are updated. This yields 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.

In practical terms, NNUE's introduction bridged the gap between hand-engineered heuristics and learned evaluation, achieving both interpretability and adaptability. Its incorporation 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.

From a research perspective, NNUE has redefined what "efficiency" means in neural evaluation —emphasizing updatability and hardware-conscious design over brute-force scale. This paradigm continues to influence ongoing work in adaptive evaluation, incremental learning, and hybrid symbolic–neural systems beyond board games, pointing toward broader applications in real-time decision-making and embedded AI systems. (Nasu, 2018; Stockfish-Team, 2025)

Architecture

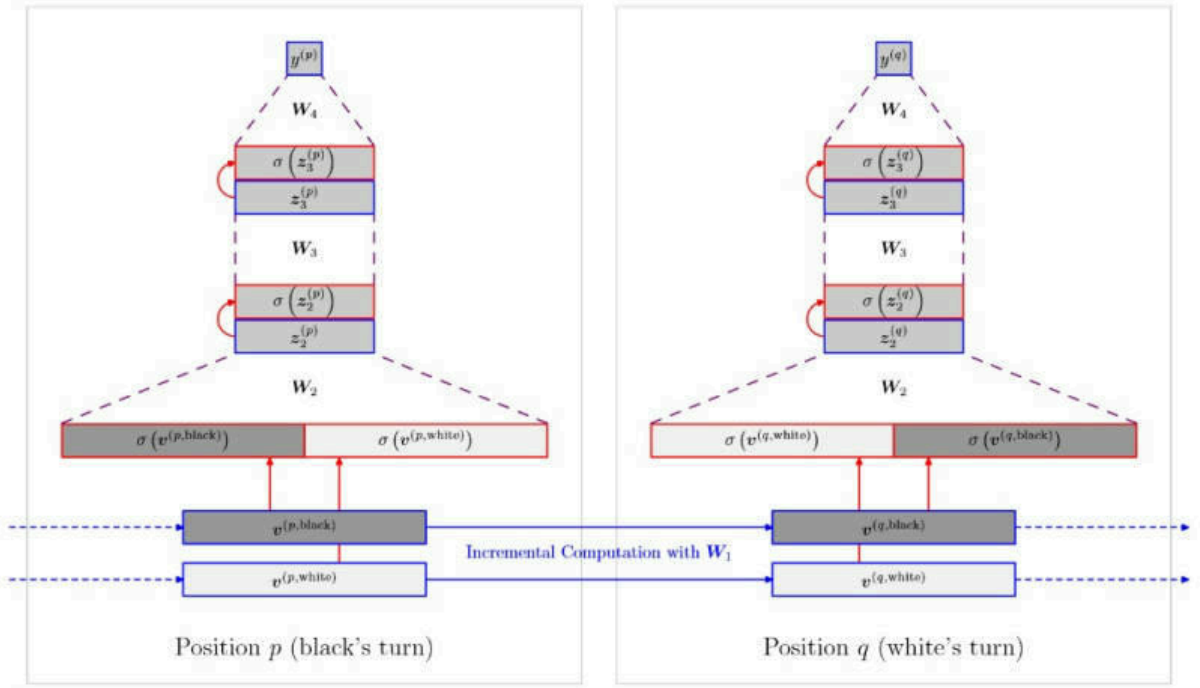

Figure 4: NNUE Stockfish Architecture

The NNUE architecture consists of four distinct layers designed specifically for efficient chess position evaluation. 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.

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. These inputs connect to a 256-neuron hidden layer per half, resulting in over 10 million weights in this feature transformer alone. 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.

The Accumulator: Efficient Updates

The critical innovation enabling NNUE's efficiency is the accumulator mechanism. Rather than recalculating the entire feature transformer output for each position during search, the engine maintains an "accumulator" - the weighted sum of active features. 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
```

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.

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 (Single Instruction, Multiple Data) operations using CPU instructions like AVX2, processing multiple values simultaneously. The final output undergoes scaling divisions (by 64 for hidden layers, by 16 for final output) to produce the evaluation score.

Modern versions (HalfKAv2) further optimize by using multiple sub-networks discriminated by piece count, allowing the network to specialize its evaluation based on material configuration, and by feeding some feature transformer outputs directly to the final layer to better handle imbalanced material situations.

6.2 · 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).

6.2.1 · 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.

6.2.2 · 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 (p)**: A probability distribution indicating which moves are most promising.
2. **Value (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).

6.2.3 · 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—nearly 1,000 times faster. 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 significantly fewer positions (Silver *et al.*, 2017, p.4). This represents a fundamental trade-off: traditional engines rely on examining massive numbers of positions with simpler evaluation, while AlphaZero examines relatively few positions with sophisticated learned evaluation.

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*.

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*.

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].