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 $b =$ branching factor (~35 in chess), $d =$ 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 $W^{\lceil \frac{D}{2} \rceil} + W^{\lfloor \frac{D}{2} \rfloor} - 1$ 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:

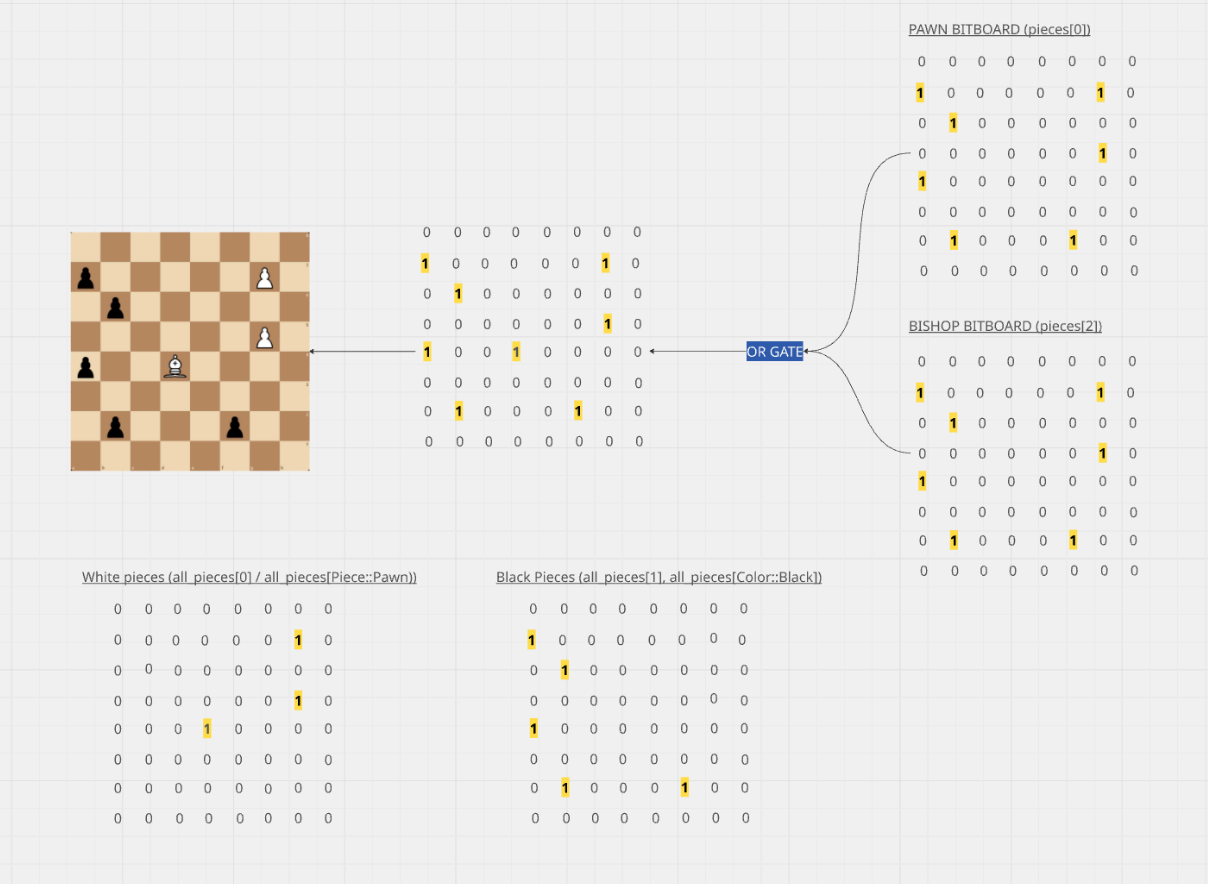

Figure 1: Bitboards Representing A Position

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:

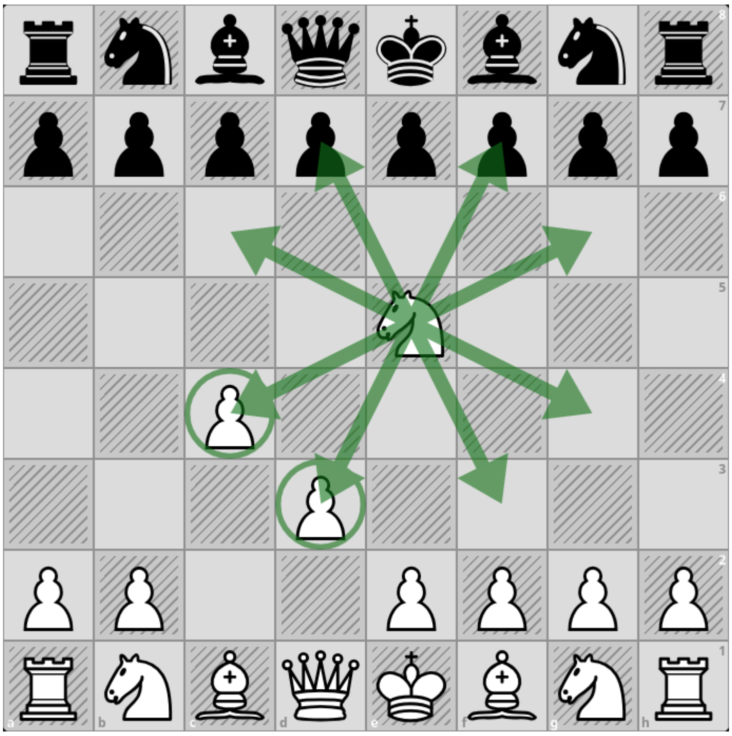

Figure 2: Attacks for a knight on e4

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:

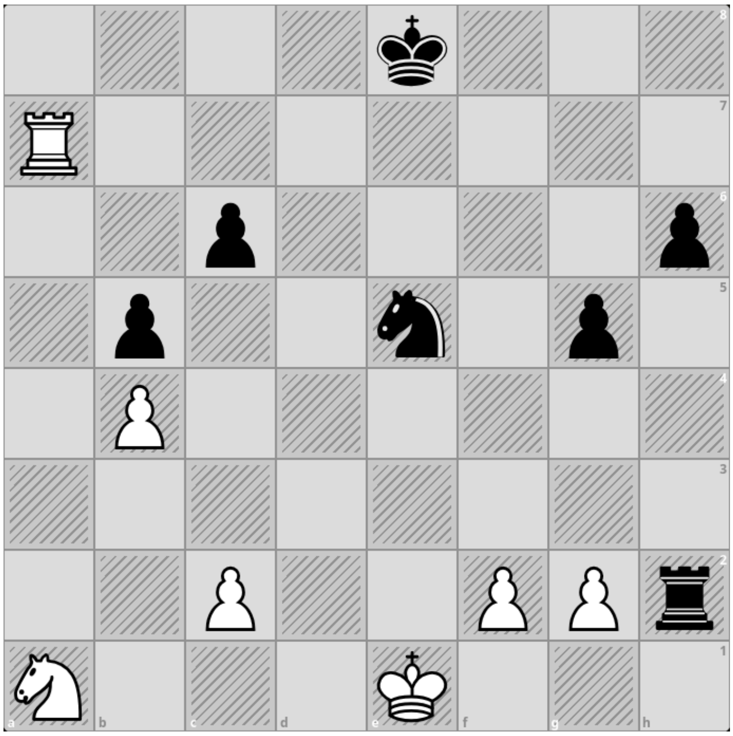

Figure 3: A Position With Equal Material Count

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:

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

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` $>= \beta$), 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, 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 remembers effective refutations. 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. (Björnsson and Marsland, 2000, p.16)

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 (Accessed: October 16, 2025).

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 (Accessed: October 18, 2025).

Fiekas, N. (2018) *Finding Hash Functions for Bitboard Based Move Generation*. Available at: https://backscattering.de/magics2.pdf (Accessed: October 16, 2025).

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.

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 (Accessed: October 18, 2025).

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.

Vrzina, S. (2023) *Piece By Piece Building a Strong Chess Engine.*. Available at: https://www.cs.vu.nl/~wanf/theses/vrzina-bscthesis.pdf (Accessed: October 16, 2025).

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].