Chess Engine Tutorial

Stochastic Batman

February 14, 2026

Abstract

This tutorial provides a comprehensive guide to implementing a chess engine using modern algorithmic techniques. I cover fundamental data structures (bitboards and magic bitboards), search algorithms (minimax with alpha-beta pruning), move generation strategies (pseudo-legal with legal filtering), hashing techniques (Zobrist hashing), principal variation tracking, evaluation heuristics (piece-square tables), and move ordering optimizations. The material is suitable for programmers with knowledge of C and chess rules who wish to implement a functional engine from scratch. These notes synthesize information from *Chess Programming Wiki* [1], Knuth and Moore's seminal work on alpha-beta [2], and practical implementation guides [3]. Gemini 3 Fast was used for grammar correction. For the implementation of this chess engine in `C`, check out [12].

1 Chess (Algebraic) Notation

Purpose

Chess notation is a standardized system for recording and describing chess moves. Understanding this notation is essential for implementing a chess engine that can communicate moves in a human-readable format. Throughout this document, all positions will be shown from White's perspective.

Algebraic Notation

Algebraic notation is the most common system. Each square on the chessboard is identified by a coordinate system:

- **Files**: Columns labeled a through h from left to right.
- **Ranks**: Rows numbered 1 through 8 from bottom to top.
- **Squares**: Identified by concatenating file and rank, e.g., $e4$, $a1$, $h8$.

Piece Designation

Each piece type has a letter designation:

- **K** $\leftarrow$ King
- **Q** $\leftarrow$ Queen
- **R** $\leftarrow$ Rook
- **B** $\leftarrow$ Bishop
- **N** $\leftarrow$ Knight (K is reserved for King)
- **(no letter)** $\leftarrow$ Pawn

Move Notation Rules

A move is recorded using the following conventions, where $\circ$ denotes concatenation:

1. **Simple moves**: Piece letter $\circ$ destination square
   - *Example:* $Nf3$ means ”Knight moves to $f3$”.
   - *Example:* $e4$ means ”pawn moves to $e4$”.
2. **Captures**: Piece letter $\circ\, x \circ$ destination square
   - *Example:* $Bxc4$ means ”Bishop captures on $c4$”.
   - *Example:* $exd5$ means ”pawn on e-file captures on $d5$”.
   - *Note*: The x symbol may be omitted in some notations, but definitely not in [12].
3. **Castling**:
   - O-O: Kingside castling (king and h-rook).
   - O-O-O: Queenside castling (king and a-rook).
4. **Pawn promotion**: Destination square $\circ = \circ$ promoted piece
   - *Example:* $e8 = Q$ means ”pawn moves to $e8$ and promotes to Queen”.
   - *Example:* $axb8 = N$ means ”pawn captures on $b8$ and promotes to Knight”.
5. **En passant**: Recorded as normal pawn capture
   - *Example:* $exd6$ *e.p.* (the notation *e.p.* may be appended for clarity).
6. **Disambiguation**: When multiple pieces of the same type can move to the same square

- Add the file of origin: $Nbd7$ (Knight from b-file to $d7$).
- Add the rank of origin: $R1a3$ (Rook from rank 1 to $a3$).
- Add both if necessary: $Qh4e1$ (Queen from $h4$ to $e1$).

7. **Check and checkmate annotations**:
   - +: Check (king is under attack).
   - *Example:* $Nf7+$ means "Knight to $f7$, check".
   - #: Checkmate (king is in check with no legal moves).
   - *Example:* $Qh5\#$ means "Queen to $h5$, checkmate".

Two-Column Game Recording

Complete games are typically recorded with move numbers in a two-column format, with White's moves on the left and Black's moves on the right:

Move	**White**	**Black**
1	e4	e5
2	Nf3	Nc6
3	Bb5	a6
4	Ba4	Nf6
5	O-O	Be7

This format clearly shows the sequence of play and is the standard used in chess databases and engine output.

2 Bitboards

Purpose

Bitboards are a highly efficient data structure for representing chess positions. Instead of using an 8×8 array to store the board state, bitboards use 64-bit integers where each bit represents a square on the chessboard. This representation enables extremely fast bitwise operations for move generation, attack detection, and position evaluation. The performance gain comes from the ability to operate on multiple squares simultaneously using single CPU instructions.

Formalism

Let $B \in \{0,1\}^{64}$ be a bitboard, represented as a 64-bit unsigned integer. Each bit position $i \in [0,63]$ corresponds to a square on the chessboard, where:

$$i = 8r + f$$

where $r \in [0,7]$ is the rank (row) and $f \in [0,7]$ is the file (column). The standard Little-Endian Rank-File (LERF) mapping is bit $0 \leftrightarrow a1$, bit $7 \leftrightarrow h1$, bit $56 \leftrightarrow a8$ and bit $63 \leftrightarrow h8$.

Basic Bitwise Operations

1. **Set bit at square** s:
$$B' = B \lor 2^s$$

2. **Clear bit at square** s:
$$B' = B \land \neg(2^s)$$

3. **Toggle bit at square** s:
$$B' = B \oplus 2^s$$

4. **Test bit at square** s:
$$\text{occupied}(s) = \begin{cases} \text{true} & \text{if } (B \land 2^s) \neq 0 \\ \text{false} & \text{otherwise} \end{cases}$$

5. **Population count** (count number of set bits):
$$|B| = \sum_{i=0}^{63} \left(\left\lfloor \frac{B}{2^i} \right\rfloor \bmod 2 \right)$$
Modern CPUs implement this as a single instruction: `POPCNT`.

6. **Least significant bit** (find lowest set bit):
$$\text{LSB}(B) = B \land (\neg B + 1)$$
This isolates the rightmost 1-bit using two's complement arithmetic.

7. **Remove least significant bit**:
$$B' = B \land (B - 1)$$
This clears the rightmost 1-bit, useful for iterating through set bits.

8. **Bit scan forward** (index of LSB):
$$\text{BSF}(B) = |(B \land (\neg B + 1)) - 1|$$
Returns the index of the least significant set bit.

Complete Board Representation

A complete chess position requires 12 piece bitboards (6 piece types × 2 colors):

$$\text{Let } \mathcal{P} = \{\overset{\text{(pawn)}}{P}, N, B, R, Q, K\} \text{ be the set of piece types} \tag{1}$$
$$\text{Let } \mathcal{C} = \{w, b\} \text{ be the set of colors (white, black)} \tag{2}$$

For each piece type $p \in \mathcal{P}$ and color $c \in \mathcal{C}$, we maintain:

$$B_{p,c} \in \{0,1\}^{64}$$

Additionally, we maintain composite bitboards for efficiency:

$$\text{Occupied}_w := \bigvee_{p \in \mathcal{P}} B_{p,w} \tag{3}$$

$$\text{Occupied}_b := \bigvee_{p \in \mathcal{P}} B_{p,b} \tag{4}$$

$$\text{O} := \text{Occupied}_w \vee \text{Occupied}_b \tag{5}$$

where $\bigvee$ denotes bitwise OR over all elements.

Shift Operations for Piece Attacks

Directional shifts allow the engine to calculate piece movement across the entire board simultaneously using bitwise arithmetic. Because the board is 8 squares wide, vertical movement corresponds to a shift of 8 bits, while horizontal movement corresponds to a shift of 1 bit.

The Wrapping Problem

Standard bit shifts do not respect the boundaries of an 8×8 grid. Without safety masks, bits on the edge of the board would *wrap* to the opposite side:

- **Eastward wrapping**: A bit on the h-file, when shifted left by 1, would incorrectly appear on the a-file of the next rank.
- **Westward wrapping**: A bit on the a-file, when shifted right by 1, would appear on the h-file of the previous rank.

To prevent this, we use **File Masks** (All squares on the a and h-files), usually represented in the hexadecimal format, to prune invalid moves that would otherwise cross the board's edge:

$$\text{FileA} := \texttt{0x0101010101010101} \tag{6}$$

$$\text{FileH} := \texttt{0x8080808080808080} \tag{7}$$

$$\text{NorthOne}(B) := B \ll 8 \tag{8}$$

$$\text{SouthOne}(B) := B \gg 8 \tag{9}$$

$$\text{EastOne}(B) := (B \ll 1) \wedge \neg\text{FileA} \tag{10}$$

$$\text{WestOne}(B) := (B \gg 1) \wedge \neg\text{FileH} \tag{11}$$

$$\text{NorthEast}(B) := (B \ll 9) \wedge \neg\text{FileA} \tag{12}$$

$$\text{NorthWest}(B) := (B \ll 7) \wedge \neg\text{FileH} \tag{13}$$

$$\text{SouthEast}(B) := (B \gg 7) \wedge \neg\text{FileA} \tag{14}$$

$$\text{SouthWest}(B) := (B \gg 9) \wedge \neg\text{FileH} \tag{15}$$

3 Magic Bitboards

Purpose

Magic bitboards solve the computational challenge of generating sliding piece attacks (rooks, bishops, queens) efficiently. The naive approach of ray-casting is too slow for deep search. Magic bitboards use perfect hashing [4] to pre-compute all possible attack patterns and retrieve them in $O(1)$ time using a single multiplication and shift operation.

The Sliding Piece Problem

Define the set of all square indices $\mathcal{S} = \{0, 1, \ldots, 63\}$. For a sliding piece on square $s \in \mathcal{S}$, the attack pattern depends on the occupancy of squares along its movement rays. Let $\mathcal{R}_s$ be the set of movement rays (orthogonal for rooks, diagonal for bishops). For each ray $r \in \mathcal{R}_s$, let the squares on that ray be ordered by their distance from s as $\{t_1, t_2, \ldots, t_k\}$. The challenge is to compute (in constant time):

$$A(s, O) := \bigcup_{r \in \mathcal{R}_s} \{t_i \in r \mid \forall j < i, t_j \notin O\}$$

where:

- $t_j \notin O$ denotes that square t_j is empty (`B[`t_j`] = 0`).
- The condition $\forall j < i, t_j \notin O$ ensures that the ray continues until it hits the first occupied square (the blocker), which is included in the attack set, but squares *beyond* the blocker are excluded.

The key insight: only a subset of squares (the *relevant occupancy*) actually affects the attack pattern.

Relevant Occupancy Mask

For any square $s \in \mathcal{S}$ with rank $r_s = \lfloor s/8 \rfloor$ and file $f_s = s \bmod 8$, we define the following foundational bitboard masks:

- $\text{RankMask}(s) := \sum\{2^i \mid i \in \mathcal{S}, \lfloor i/8 \rfloor = r_s\}$
- $\text{FileMask}(s) := \sum\{2^i \mid i \in \mathcal{S}, i \bmod 8 = f_s\}$
- $\text{DiagonalMask}(s)$: The set of squares where the difference between rank and file is constant (parallel to the $a1 - h8$ diagonal).

$$\text{DiagonalMask}(s) := \sum\{2^i \mid i \in \mathcal{S}, (\lfloor i/8 \rfloor - (i \bmod 8)) = (r_s - f_s)\}$$

- $\text{AntiDiagonalMask}(s)$: Same as above, but parallel to the $h1 - a8$ diagonal.

$$\text{AntiDiagonalMask}(s) := \sum\{2^i \mid i \in \mathcal{S}, (\lfloor i/8 \rfloor + (i \bmod 8)) = (r_s + f_s)\}$$

- Edges := RankMask(0) $\vee$ RankMask(56) $\vee$ FileMask(0) $\vee$ FileMask(7)

For a rook on square $s \in \mathcal{S}$, the relevant occupancy mask $M_R(s)$ includes all squares on its rank and file, *excluding* the board edges relative to the piece's rays. This is because a piece on the edge cannot block the sliding piece from reaching that edge, but it does not matter if there is a piece "behind" the edge square.

$$M_R(s) := (\text{RankMask}(s) \vee \text{FileMask}(s)) \wedge \neg(\text{Edges} \vee 2^s)$$

For a bishop:

$$M_B(s) := (\text{DiagonalMask}(s) \vee \text{AntiDiagonalMask}(s)) \wedge \neg(\text{Edges} \vee 2^s)$$

and for a queen:

$$M_Q(s) := M_R(s) \cup M_B(s)$$

Perfect Hashing via Magic Multiplication

The magic bitboards technique uses a multiplicative hash function [5]. For each square $s \in \mathcal{S}$, we find a magic number $\mu_s \in \mathbb{Z}_{2^{64}}$ such that:

$$h(s, O) = \frac{(O \wedge M(s)) \times \mu_s}{2^{64-n(s)}}$$

where:

- $M(s)$ is the relevant occupancy mask for square s (drop subscripts R, B, Q for simplicity).
- $n(s) = |M(s)|$ is the number of relevant occupancy bits.
- $h(s, O) \in [0, 2^{n(s)} - 1]$ is the hash index.

Attack Lookup

The complete lookup becomes:

$$A(s, O) = \text{AttackTable}_s \left[\frac{(O \wedge M(s)) \times \mu_s}{2^{64-n(s)}} \right]$$

where AttackTable_s is a pre-computed array of size $2^{n(s)}$ containing all possible attack patterns for square s given different occupancies.

Finding Magic Numbers

Magic numbers are found through brute-force trial-and-error search:

Candidates μ are typically chosen as random 64-bit integers with few set bits (sparse), as these tend to produce better hash distributions.

Algorithm 1 Finding Magic Numbers for Square s

```
Input: Square $s \in \mathcal{S}$, Relevant occupancy mask $M(s)$, Number of bits $n(s)|$
Output: Valid magic number $\mu_s$ and populated attack table AttackTable$_s$
 1: Initialize AttackTable$_s$ as empty array of size $2^{n(s)}$
 2: found ← false
 3: while not found do
 4:     $\mu \leftarrow$ random sparse 64-bit integer                ▷ Few set bits
 5:     collisionFree ← true
 6:     Clear AttackTable$_s$
 7:     for each occupancy pattern $O \in \mathcal{P}(M(s))$ do       ▷ $2^{n(s)}$ patterns
 8:         attacks ← $A(s, O)$                                  ▷ Compute actual attacks
 9:         index ← $\lfloor (O \wedge M(s)) \times \mu / 2^{64-n(s)} \rfloor$
10:         if AttackTable$_s$[index] is empty then
11:             AttackTable$_s$[index] ← attacks
12:         else if AttackTable$_s$[index] $\neq$ attacks then
13:             collisionFree ← false
14:             break                                  ▷ Collision detected, try next $\mu$
15:         end if
16:     end for
17:     if collisionFree then
18:         $\mu_s \leftarrow \mu$
19:         found ← true
20:     end if
21: end while
22: return $\mu_s$, AttackTable$_s$
```

4 Minimax Algorithm with Alpha-Beta Pruning

Purpose

Minimax [2] is the foundational search algorithm for two-player zero-sum games like chess. It explores the game tree to find the optimal move by assuming both players play perfectly. The algorithm alternates between maximizing and minimizing players, evaluating positions at leaf nodes. Alpha-beta pruning is a critical optimization that eliminates branches that cannot influence the final decision, often reducing the effective branching factor from b to $\sqrt{b}$.

Game Tree Formalism

Define a game tree $\mathcal{T} = (V, E)$ where:

- V is the set of positions (nodes)
- $E \subseteq V \times V$ is the set of moves (edges)
- Root position $p_0 \in V$ is the current position
- Leaf positions $L \subseteq V$ are terminal or depth-limited positions

Minimax Value Function

Let $V(p, d)$ be the minimax value of position p at remaining depth d. Define recursively:

$$V(p,d) = \begin{cases} \text{eval}(p) & \text{if } d = 0 \text{ or } p \in L \\ \max\limits_{m \in M(p)} V(m(p), d-1) & \text{if } p \text{ is MAX node} \\ \min\limits_{m \in M(p)} V(m(p), d-1) & \text{if } p \text{ is MIN node} \end{cases}$$

where:

- $M(p)$ is the set of legal moves in position p
- $m(p)$ denotes the position after applying move m to position p
- $\text{eval}(p) \in \mathbb{R}$ is the static evaluation function
- MAX nodes correspond to the side trying to maximize the score
- MIN nodes correspond to the side trying to minimize the score

Alpha-Beta Pruning

Alpha-beta maintains two bounds during search:

- $\alpha \leftarrow$ Best value the maximizer can guarantee so far (lower bound)
- $\beta \leftarrow$ Best value the minimizer can guarantee so far (upper bound)

Initially:

- $\alpha = -\infty$
- $\beta = +\infty$

The search window $[\alpha, \beta]$ narrows as the search progresses. When $\alpha \geq \beta$, the current branch can be pruned.

Alpha-Beta Algorithm

The enhanced minimax with alpha-beta pruning:

$$V_{AB}(p, d, \alpha, \beta) = \begin{cases} \text{eval}(p) & \text{if } d = 0 \text{ or } p \in L \\ \text{MaxValue}(p, d, \alpha, \beta) & \text{if MAX node} \\ \text{MinValue}(p, d, \alpha, \beta) & \text{if MIN node} \end{cases}$$

Algorithm 2 MaxValue (Maximizing Player in Alpha-Beta)

```
Input: Position $p$, depth $d$, bounds $\alpha$, $\beta$
Output: Best value $v$ for maximizing player
1: $v \leftarrow -\infty$
2: for each move $m \in M(p)$ do
3:     $v \leftarrow \max(v, V_{AB}(m(p), d-1, \alpha, \beta))$
4:     $\alpha \leftarrow \max(\alpha, v)$
5:     if $\beta \leq \alpha$ then
6:         break          ▷ $\beta$-cutoff: position too good, opponent won't allow it
7:     end if
8: end for
9: return $v$
```

Algorithm 3 MinValue (Minimizing Player in Alpha-Beta)

Input: Position p, depth d, bounds α, β
Output: Best value v for minimizing player

```
1: $v \leftarrow +\infty$
2: for each move $m \in M(p)$ do
3:     $v \leftarrow \min(v, V_{AB}(m(p), d-1, \alpha, \beta))$
4:     $\beta \leftarrow \min(\beta, v)$
5:     if $\beta \leq \alpha$ then
6:         break          ▷ $\alpha$-cutoff: position too bad, we have better alternative
7:     end if
8: end for
9: return $v$
```

Pruning Conditions

- **Beta Cutoff** (at MAX node): When $v \geq \beta$, the current position is too good for the maximizer; the minimizer will avoid this branch by choosing a different move earlier.
- **Alpha Cutoff** (at MIN node): When $v \leq \alpha$, the current position is too good for the minimizer; the maximizer will avoid this branch.

Formally, a branch rooted at position p can be pruned when:

$$\beta \leq \alpha \implies \text{prune}(p)$$

Correctness and Efficiency

Correctness: Alpha-beta returns the same value as minimax:

$$V_{AB}(p, d, -\infty, +\infty) = V(p, d) \quad \forall p, d$$

Best-case efficiency: With perfect move ordering (best move searched first), alpha-beta examines:

$$N_{best} = O(b^{d/2})$$

nodes instead of minimax's $O(b^d)$, effectively doubling the search depth.

Worst-case efficiency: With worst move ordering (best move searched last):

$$N_{worst} = O(b^d)$$

This highlights the critical importance of move ordering.

The Evaluation Function: Implementing eval(p)

The static evaluation function $\text{eval}(p) \in \mathbb{R}$ quantifies the material and positional advantage for a given position p, relative to the side to move. While modern engines like Stockfish utilize Neural Networks (NNUE) [6], a standard classical implementation (as well as my implementation [12]) uses a linear combination of heuristics.

Logic and Components

The function is typically structured as a weighted sum of various features:

$$\text{eval}(p) = \text{MaterialScore} + \text{PositionalScore} + \text{MobilityScore}$$

1. **Material Score**: Calculated using standard relative piece values. Let v_τ be the value assigned to each piece type $\tau \in \mathcal{P}$. These values represent the theoretical strength of a piece in centipawns (a unit of measurement in computer chess equivalent to $\frac{1}{100}$-th of a pawn, used to quantify positional advantages and engine evaluations). The total material score is defined as:
$$\text{MaterialScore} = \sum_{\tau \in \mathcal{P}} v_\tau (|B_{\tau,w}| - |B_{\tau,b}|)$$
where $|B|$ denotes the *Hamming weight* (or population count) of the bitboard:
$$|B| = \sum_{i=0}^{63} B \wedge 2^i$$

2. **Positional Score (Piece-Square Tables)**: $E_{\text{PST}}(p)$. Assigns specific values to squares based on the piece type and game phase (opening vs. endgame). For example, Knights are prioritized in the center, while King safety is weighted more heavily in the opening.

3. **Mobility Score**: Measures the number of legal or pseudo-legal moves available, rewarding positions with greater freedom.

Standard and Advanced Weights

Standard simplified weights (in centipawns) are:

Pawn	**Knight**	**Bishop**	**Rook**	**Queen**	**King**
100	320	330	500	900	20000

More advanced engines, such as Stockfish (pre-NNUE), use sophisticated ”tapered evaluation” [7] which interpolates weights between the middle-game and endgame to account for changing piece values as the board clears.

5 Move Generation

Purpose

Move generation is the process of enumerating the complete set of moves $M(p)$ for a position p. We employ a two-stage process: first generating *pseudo-legal* moves (M_{pseudo}) based on piece rules, then filtering for *legal* moves (M_{legal}) that do not leave the side to move in check.

Pseudo-Legal Move Generation

For the side to move with color $c \in \mathcal{C}$, we iterate through all piece types $\tau \in \mathcal{P}$. For each square $s \in \mathcal{S}$ where bit s is set in $B_{\tau,c}$, we generate moves to target squares t:

$$M_{\text{pseudo}}(s, \tau, c) = \{(s, t, f) \mid t \in A_\tau(s, O) \wedge t \notin \text{Occupied}_c\}$$

where $A_\tau(s, O)$ is the attack bitboard for type τ, and f encodes metadata such as captures (x) or promotion ($=$). Define $\neg c := (c = w)\,?\,b : w$.

Attack Generation by Piece Type

1. **Pawns**: Unlike other pieces, pawn moves depend on color c.
   - *Single Push*: $\text{NorthOne}(2^s) \wedge \neg O$.
   - *Double Push*: $\text{NorthOne}(\text{NorthOne}(2^s)) \wedge \neg O$ (if s is on Rank 2).
   - *Captures*: $(\text{NorthWest}(2^s) \vee \text{NorthEast}(2^s)) \wedge \text{Occupied}_{\neg c}$.
2. **Leaping Pieces (Knight/King)**: Targets are retrieved from pre-computed tables (based on their move patterns):

$$A_N(s) = \text{KnightTable}[s], \quad A_K(s) = \text{KingTable}[s]$$

3. **Sliding Pieces (Bishop/Rook/Queen)**: Targets are retrieved using the Magic Bitboard lookup defined in Section 3:

$$A_\tau(s, O) = \text{AttackTable}_{\tau,s}\,[h(s, O)]$$

Legal Move Filtering

A move m is legal if the resulting position $m(p)$ is not in check for color c:

$$M_{\text{legal}}(p) = \{m \in M_{\text{pseudo}}(p) \mid \neg\text{inCheck}(m(p), c)\}$$

The $\text{inCheck}(p, c)$ condition is true if the king of color c, located at $s_K = \text{BSF}(B_{K,c})$, is attacked by any opposing piece. $B_{\tau,\neg c}$ represents the set of all squares occupied by the opponent's pieces of type τ:

$$\text{inCheck}(p, c) \iff \bigvee_{\tau \in \mathcal{P}} (A_\tau(s_K, O) \cap B_{\tau,\neg c}) \neq \emptyset$$

6 Zobrist Hashing

Purpose

Zobrist hashing $h : \mathcal{S} \to \mathbb{Z}_{2^{64}}$ [8] maps a chess position p to a 64-bit integer $h(p) \in \{0, \ldots, 2^{64} - 1\}$. This enables $O(1)$ transposition table lookups and threefold repetition detection via incremental updates.

Hash Function Design

We define a table of 781 unique, pre-generated random 64-bit constants. Z stands for "Zobrist Table" or "Zobrist Constant":

- $Z_{\tau,c,s}$: One constant for each piece τ, color c, and square $s \in \mathcal{S}$ ($6 \times 2 \times 64 = 768$).
- $Z_{\text{ep},f}$: One constant for each file $f \in \{0, \ldots, 7\}$ where an en passant capture might be possible.
- $Z_{\text{castle},r}$: One constant for each castling right $r \in \{\text{WK, WQ, BK, BQ}\}$.
- Z_{side}: A single constant used to indicate it is Black's turn to move.

Complete Hash Computation

For a position p, the Zobrist hash is computed by XORing the constants corresponding to the features present in the position:

$$h(p) = \left(\bigoplus_{s \in B_{\tau,c}} Z_{\tau,c,s} \right) \oplus \left(\bigoplus_{r \in R(p)} Z_{\text{castle},r} \right) \oplus h_{\text{ep}}(p) \oplus h_{\text{side}}(p)$$

where the conditional terms are defined as:

$$h_{\text{ep}}(p) = \begin{cases} Z_{\text{ep},f} & \text{if en passant is available on file } f \\ 0 & \text{otherwise} \end{cases}$$

$$h_{\text{side}}(p) = \begin{cases} Z_{\text{side}} & \text{if side to move is Black} \\ 0 & \text{if side to move is White} \end{cases}$$

Hash Collision Analysis

The probability of collisions in a Zobrist hash table is an application of the *Birthday Problem.* Consider a table containing a set of k distinct chess positions $\{p_1, p_2, \ldots, p_k\}$.

Derivation of Expected Collisions

A collision occurs when a pair of distinct positions (p_i, p_j) results in the same hash value: $h(p_i) = h(p_j)$.

1. **Probability of a Single Pair Colliding:** Since each hash is a uniformly distributed random 64-bit integer, the probability that any two specific positions collide is:
$$P(h(p_i) = h(p_j)) = \frac{1}{2^{64}}$$
2. **Number of Possible Pairs:** In a set of k positions, the number of unique pairs (p_i, p_j) we can form is given by the binomial coefficient $\binom{k}{2}$:
$$\binom{k}{2} = \frac{k(k-1)}{2} = \frac{k^2 - k}{2}$$
3. **Linearity of Expectation:** Let $X_{i,j} = \mathbf{1}_{h(p_i)=h(p_j)}$. The total number of collisions is $C = \sum X_{i,j}$. By the linearity of expectation:
$$\mathbb{E}[C] = \sum \mathbb{E}[X_{i,j}] = \binom{k}{2} \times \frac{1}{2^{64}} = \frac{k^2 - k}{2 \cdot 2^{64}} = \frac{k^2 - k}{2^{65}}$$

For a standard transposition table size of $k = 2^{24} = 16,777,216$, the expected number of collisions is:

$$\mathbb{E}[C] = \frac{(2^{24})^2 - 2^{24}}{2^{65}} = \frac{2^{24}(2^{24} - 1)}{2^{65}} = \frac{2^{24} - 1}{2^{41}}$$

Expanding the fraction:

$$\mathbb{E}[C] = \frac{2^{24}}{2^{41}} - \frac{1}{2^{41}} = 2^{-17} - 2^{-41} \approx 0.0000076$$

This means that in a search tree of over 16 million nodes, there is only a $\approx 0.00076\%$ chance of seeing even *one* collision. Seems reasonable enough for me.

The Transposition Table

The Transposition Table (TT) is a large hash table that stores search results to avoid re-evaluating the same positions reached via different move orders (transpositions).

Entry Structure

Each entry in the table typically occupies 128 bits and contains the following data:

- **Key**: The full 64-bit Zobrist hash (to verify no collisions).

- **Score**: The evaluation value v found during search.
- **Depth**: The remaining search depth at which this score was found.
- **Flag**: Indicates if the score is an *Exact* value, a *Lower Bound* (β-cutoff), or an *Upper Bound* (α-cutoff).
- **Best Move**: The move that achieved this score, used to seed the next search's move ordering.

Replacement Strategy

Since the number of possible chess positions far exceeds the table size, we use a replacement strategy. When a collision occurs at an index, the engine typically uses a **Depth-Preferred** approach: the new entry overwrites the old one only if the new search depth is greater than or equal to the stored depth.

7 Principal Variation

Purpose

The Principal Variation is the sequence of moves that the engine considers to be the best play for both sides. It represents the "main line" of the game at any given search depth.

$$\text{PV} = \langle m_1, m_2, \ldots, m_d \rangle$$

Tracking this sequence is essential for two reasons:

- **User Feedback**: It shows the user what the engine is "thinking" and how it justifies its evaluation.
- **Search Stability**: It allows the engine to use an *Aspiration Window*, assuming the score of the current search will not deviate drastically from the score of the previous PV.

Formal Definition

Let p be a position and d be the search depth. We denote $m(p)$ as the position resulting from applying move $m \in M(p)$ to position p. The PV is defined recursively as the sequence starting with the best move m^*, concatenated with the PV of the resulting position $m^*(p)$:

$$\mathrm{PV}(p,d) = \begin{cases} \langle\rangle & \text{if } d = 0 \text{ or } p \text{ is terminal} \\ \langle m^* \rangle \circ \mathrm{PV}(m^*(p), d-1) & \text{otherwise} \end{cases}$$

The best move m^* is the one that leads to the state with the highest (or lowest) minimax value. Using our previously defined operators:

$$m^* = \begin{cases} arg\,max_{m \in M(p)}\{V_{AB}(m(p), d-1, \alpha, \beta)\} & \text{if MAX node} \\ arg\,min_{m \in M(p)}\{V_{AB}(m(p), d-1, \alpha, \beta)\} & \text{if MIN node} \end{cases}$$

where $\circ$ denotes the sequence concatenation operator. In this framework, $m^*(p)$ represents the board state that the opponent will face at the next ply (distance from root) of the search.

The Triangular PV Array

In a recursive Alpha-Beta search, maintaining the PV can be computationally expensive if using dynamic lists. Instead, engines use a **Triangular Array**, a structure where each ply has its own move buffer.

Define the array $\mathbf{PV}[D \times D]$, where D is the maximum search depth. If a move m improves α (causing a "PV-node" update), we update the array for the current ply i:

Algorithm 4 Update Principal Variation

```
Input: Current ply $i$, better move $m$ found at depth $d$, child PV at ply $i+1$
Output: Updated PV at ply $i$
1: $\mathbf{PV}[i][i] \leftarrow m$                              ▷ Store current best move
2: for $j \leftarrow i+1$ to length($\mathbf{PV}[i+1]$) do
3:     $\mathbf{PV}[i][j] \leftarrow \mathbf{PV}[i+1][j]$        ▷ Copy the continuation from the child
4: end for
```

By the time the search returns to the root (ply 0), the first row of the triangle $\mathbf{PV}[0][0 \ldots d-1]$ contains the full Principal Variation.

PV-Move Ordering

The most critical application of the PV is in **Move Ordering** during iterative deepening. If we have completed a search at depth d, the first move we should search in the next iteration (depth $d+1$) is the best move from the previous iteration. We denote the Principal Variation found at depth d as PV_d.

Searching $\text{PV}_d[0]$ (the first move of the previous best line) maximizes the probability of an early Alpha-Beta cutoff. We define the priority function for a move $m \in M(p)$ at iteration $d+1$ as:

$$\text{Priority}(m) = \begin{cases} \infty & \text{if } m = \text{PV}_d[0] \\ \text{HeuristicScore}(m) & \text{otherwise} \end{cases}$$

PV Stability and Aspiration

The efficiency of iterative deepening relies on **PV Stability**. We define this property as the probability that the Principal Variation at depth d remains a prefix of the Principal Variation at depth $d+1$:

$$\text{Stability}(d) = \mathbb{P}\left(\text{PV}_d[0 \ldots k] = \text{PV}_{d+1}[0 \ldots k]\right)$$

where k is the number of consistent moves (the "prefix length"). In stable positions, $k \geq 1$ with high probability.

When stability is high, we can utilize an **Aspiration Window**. Instead of searching the interval $(-\infty, \infty)$, we assume the new evaluation V_{d+1} will be close to the previous evaluation V_d. We define a search margin $\epsilon \in \mathbb{R}^+$, representing the maximum expected fluctuation in the position's value.

We initialize the Alpha-Beta bounds as:

$$\alpha = V_d - \epsilon, \quad \beta = V_d + \epsilon$$

If the search concludes with a score $V \in (\alpha, \beta)$, the result is valid and was found significantly faster due to the narrower window. However, we must handle two edge cases:

- **Fail Low**: If $V \leq \alpha$, the position is worse than expected. We must re-search with $[-\infty, \alpha]$.
- **Fail High**: If $V \geq \beta$, the position is better than expected. We must re-search with $[\beta, \infty]$.

8 Piece-Square Tables

Purpose

Piece-square tables (PSTs) encode positional knowledge by assigning a numerical value to each piece on each square. They enable the engine to differentiate between a Knight in the center (active) and a Knight on the edge (passive) without expensive geometric calculations. The total PST contribution, denoted as $E_{\mathrm{PST}}(p)$, is the primary component of the positional evaluation in eval(p).

Formalism

For each piece type $\tau \in \mathcal{P}$ and each square $s \in \mathcal{S}$, we define a static weight $\mathrm{PST}_\tau[s] \in \mathbb{Z}$. These values are predefined constants. The total contribution is:

$$E_{\mathrm{PST}}(p) = \sum_{c \in \{w,b\}} \sigma(c) \sum_{\tau \in \mathcal{P}} \sum_{s \in S_{\tau,c}(p)} \mathrm{PST}_\tau[s]$$

where $\sigma(w) = 1$ and $\sigma(b) = -1$. The initial value for a position p is calculated by iterating over all squares $s \in \mathcal{S}$ and summing the values of the pieces found.

Game Phase Interpolation

King safety is critical in the middlegame but centralization is key in the endgame. We interpolate between two tables, $\mathrm{PST}_K^{\mathrm{mg}}$ and $\mathrm{PST}_K^{\mathrm{eg}}$, using a phase factor $\phi \in [0, 1]$:

$$\mathrm{PST}_K[s] = (1 - \phi) \cdot \mathrm{PST}_K^{\mathrm{mg}}[s] + \phi \cdot \mathrm{PST}_K^{\mathrm{eg}}[s]$$

The phase ϕ is determined by the total non-pawn material currently on the board:

$$\phi = 1 - \frac{\mathrm{NonPawnMaterial}(p)}{\mathrm{NonPawnMaterial}_{\mathrm{start}}}$$

where $\mathrm{NonPawnMaterial}(p) = \sum_{c \in \mathcal{C}} \sum_{\tau \in \mathcal{P} \setminus \{P\}} v_\tau |B_{\tau,c}|$. At the start of a game, $\mathrm{NonPawnMaterial}_{\mathrm{start}}$ is usually 6400 centipawns. As pieces are traded, ϕ approaches 1 (Endgame).

Incremental PST Updates

To avoid re-summing the entire board, we update the PST score after every move $m : s_1 \to s_2$. If a piece τ of color c moves:

$$\Delta_{\mathrm{PST}}(m) = \sigma(c) \left(\mathrm{PST}_\tau[s_2] - \mathrm{PST}_\tau[s_1]\right) + \text{capture-bonus}$$

The capture-bonus ensures the evaluation is corrected for a piece removed from the board. If a piece of type τ_{cap} and color $\neg c$ was captured at s_2:

$$\text{capture-bonus} = -\sigma(\neg c) \cdot \mathrm{PST}_{\tau_{cap}}[s_2]$$

Note that because $\sigma(\neg c)$ is the opposite of $\sigma(c)$, this effectively becomes:

$$\text{capture-bonus} = \sigma(c) \cdot \text{PST}_{\tau_{cap}}[s_2]$$

Symmetry for Black Pieces

To save memory, we only store PSTs for White. Black's values are derived by mirroring the square vertically:

$$\text{PST}_{\tau,b}[s] = \text{PST}_{\tau,w}[s \oplus 56]$$

9 Move Ordering

Purpose

The efficiency of Alpha-Beta pruning is highly dependent on the order in which moves are searched. If the best move (the move that produces the highest evaluation) is searched first, the search window $[\alpha, \beta]$ narrows immediately, allowing the algorithm to prune the maximum number of subtrees. In the best-case scenario, perfect ordering reduces the search complexity from $O(b^d)$ to $O(b^{d/2})$.

Heuristic Scoring Function

We define a scoring function $S(m, p, i)$ that assigns a numerical priority to each move $m \in M_{\text{pseudo}}(p)$ at ply i. Before searching, moves are sorted in descending order of S. We define $S(m, p, i)$ to show that move priority depends on the position p and the current ply i, but we use $S(m)$ as a shorthand during sorting because the context of p and i is constant for all moves being considered at a specific node.

1. The Hash Move

If the Transposition Table contains an entry for the current position $h(p)$, the move that was previously stored as the best move in that entry is assigned the highest priority.

$$S(m) = 10^7 \quad \text{if } m \text{ matches the stored move for } h(p)$$

2. MVV-LVA (Most Valuable Victim - Least Valuable Attacker)

For capture moves, we prioritize those that capture high-value pieces with low-value attackers. For a move m where piece τ_a captures τ_v:

$$S(m) = 10^6 + (10 \cdot v_{\tau_v} - v_{\tau_a})$$

This requires no initialization as piece values are static constants.

3. Killer Heuristic

Killer moves are quiet moves (non-captures) that caused a beta-cutoff in a different branch at the same ply i. A table Killers$[i][0\ldots1]$ initialized to 0 for all plies.

$$S(m) = \begin{cases} 10^5 & \text{if } m = \text{Killers}[i][0] \\ 10^4 & \text{if } m = \text{Killers}[i][1] \end{cases}$$

4. History Heuristic

The history heuristic tracks the success of quiet moves globally. A table History[2][64][64] initialized to zero. When a quiet move $m : s_1 \to s_2$ causes a cutoff at depth d, we increment the score:

$$\text{History}[c][s_1][s_2] \leftarrow \text{History}[c][s_1][s_2] + d^2$$

10 The Optimized Search Framework

The Unified Alpha-Beta Algorithm

This implementation utilizes the **Negamax** formulation and integrates the Transposition Table logic using the hash $h(p)$.

Negamax Search and Transposition Table Integration

The search engine utilizes the **Negamax** formulation, a variant of Minimax that simplifies implementation by exploiting the mathematical property

$$\max(a, b) = -\min(-a, -b)$$

This allows a single function to handle both players by negating the scores returned by recursive calls. To optimize performance, we integrate the Transposition Table (TT) directly into the search flow using the Zobrist hash $h(p)$. Before move generation, the engine performs a TT lookup to check for previously searched results that might allow for an immediate cutoff

Iterative Deepening and Aspiration

The search is driven by an outer loop increasing depth d. We use an **Aspiration Window** around the previous result V_d:

$$\alpha = V_d - \epsilon, \quad \beta = V_d + \epsilon$$

If the result V_{d+1} falls outside (α, β), we widen the window and re-search.

Algorithm 5 Modern Alpha-Beta Search

```
Input: Position $p$, depth $d$, bounds $[\alpha, \beta]$, ply $i$
Output: Evaluation $v$
1: $h \leftarrow h(p)$
2: $entry \leftarrow$ lookupTranspositionTable($h$)
3: if $entry$.depth $\geq d$ then                              ▷ Transposition Table Cutoff
4:     if $entry$.flag = EXACT then return $entry$.score
5:     else if $entry$.flag = LOWER then $\alpha \leftarrow \max(\alpha, entry.\text{score})$
6:     else if $entry$.flag = UPPER then $\beta \leftarrow \min(\beta, entry.\text{score})$
7:     end if
8:     if $\alpha \geq \beta$ then return $entry$.score
9:     end if
10: end if
11: Base Case: If $d = 0$, return QuiescenceSearch($p, \alpha, \beta$)
12: $v \leftarrow -\infty$, $bestM \leftarrow$ null, $moveCount \leftarrow 0$
13: $M \leftarrow$ OrderMoves($M_{\text{pseudo}}(p), S$)
14: for each $m \in M$ do
15:     if isLegal($m, p$) then
16:         $moveCount \leftarrow moveCount + 1$
17:         $v_{cur} \leftarrow -V_{AB}(m(p), d-1, -\beta, -\alpha, i+1)$
18:         if $v_{cur} > v$ then
19:             $v \leftarrow v_{cur}, bestM \leftarrow m$
20:             if $v > \alpha$ then $\alpha \leftarrow v$
21:             end if
22:             if $\alpha \geq \beta$ then
23:                 UpdateKillersAndHistory(m, d, i)
24:                 break                                              ▷ Beta Cutoff
25:             end if
26:         end if
27:     end if
28: end for
29: if $moveCount = 0$ then return evaluateTerminalNode($p$)
30: end if
31: StoreEntry(hash: $h$, score: $v$, depth: $d$, move: $bestM$, flag:
    getFlag($\alpha, \beta, v$))
32: return $v$
```

References

[1] Chess Programming Wiki. *Chess Programming Wiki*. Accessed February 2026. https://www.chessprogramming.org/

[2] Knuth, D. E., & Moore, R. W. (1975). *An analysis of alpha-beta pruning*. Artificial Intelligence, 6(4), 293-326. https://doi.org/10.1016/0004-3702(75)90019-3

[3] Jonatan Pettersson. *Mediocre Chess: Guide to Writing a Chess Engine*. https://mediocrechess.blogspot.com/

[4] Wikipedia. Perfect Hash Function

[5] *Lecture 21: Hash Functions and Hash Tables*. CS 3110: Data Structures and Functional Programming, Cornell University. https://www.cs.cornell.edu/courses/cs3110/2008fa/lectures/lec21.html

[6] *Introducing NNUE Evaluation*. stockfishchess.org

[7] *Tapered Evaluation (pre-NNUE)*. Chess Programming Wiki

[8] Zobrist, A. L. (1970). *A new hashing method with application for game playing*. Technical Report 88, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin. Minds @ UW

[9] Shannon, C. E. (1950). *Programming a computer for playing chess*. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 41(314), 256-275. https://doi.org/10.1080/14786445008521796

[10] *Rotated Bitboards*. Chess Programming Wiki

[11] *Magic Bitboards*. Chess Programming Wiki

[12] *GitHub Repository*. Zugzwang