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]. For the implementation of this chess engine in C, check out [10].

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 [10].
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:

$$i = 8r + f \tag{1}$$

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:
$$B' = B \vee 2^s \tag{2}$$

2. **Clear bit at square** s:
$$B' = B \wedge \neg(2^s) \tag{3}$$

3. **Toggle bit at square** s:
$$B' = B \oplus 2^s \tag{4}$$

4. **Test bit at square** s:
$$\text{occupied}(s) = \begin{cases} \text{true} & \text{if } (B \wedge 2^s) \neq 0 \\ \text{false} & \text{otherwise} \end{cases} \tag{5}$$

5. **Population count** (count number of set bits):
$$|B| = \sum_{i=0}^{63} \left(\left\lfloor \frac{B}{2^i} \right\rfloor \bmod 2 \right) \tag{6}$$
Modern CPUs implement this as a single instruction: `POPCNT`.

6. **Least significant bit** (find lowest set bit):
$$\text{LSB}(B) = B \wedge (\neg B + 1) \tag{7}$$
This isolates the rightmost 1-bit using two's complement arithmetic.

7. **Remove least significant bit**:
$$B' = B \wedge (B - 1) \tag{8}$$
This clears the rightmost 1-bit, useful for iterating through set bits.

8. **Bit scan forward** (index of LSB):
$$\text{BSF}(B) = |B \wedge (B - 1)| \tag{9}$$
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):

$$\text{Let } \mathcal{P} = \{\overset{\text{(pawn)}}{P}, N, B, R, Q, K\} \text{ be the set of piece types} \tag{10}$$
$$\text{Let } \mathcal{C} = \{w, b\} \text{ be the set of colors (white, black)} \tag{11}$$

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

$$B_{p,c} \in \{0,1\}^{64} \tag{12}$$

Additionally, we maintain composite bitboards for efficiency:

$$\text{Occupied}_w := \bigvee_{p \in \mathcal{P}} B_{p,w} \tag{13}$$

$$\text{Occupied}_b := \bigvee_{p \in \mathcal{P}} B_{p,b} \tag{14}$$

$$\text{O} := \text{Occupied}_w \vee \text{Occupied}_b \tag{15}$$

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:

$$\text{FileA} := \texttt{0x0101010101010101} \tag{16}$$

$$\text{FileH} := \texttt{0x8080808080808080} \tag{17}$$

$$\text{NorthOne}(B) := B \ll 8 \tag{18}$$

$$\text{SouthOne}(B) := B \gg 8 \tag{19}$$

$$\text{EastOne}(B) := (B \ll 1) \wedge \neg\text{FileA} \tag{20}$$

$$\text{WestOne}(B) := (B \gg 1) \wedge \neg\text{FileH} \tag{21}$$

$$\text{NorthEast}(B) := (B \ll 9) \wedge \neg\text{FileA} \tag{22}$$

$$\text{NorthWest}(B) := (B \ll 7) \wedge \neg\text{FileH} \tag{23}$$

$$\text{SouthEast}(B) := (B \gg 7) \wedge \neg\text{FileA} \tag{24}$$

$$\text{SouthWest}(B) := (B \gg 9) \wedge \neg\text{FileH} \tag{25}$$

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 precompute 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):

$$A(s, O) := \bigcup_{r \in \mathcal{R}_s} \{t_i \in r \mid \forall j < i, t_j \notin O\} \tag{26}$$

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:

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

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

- 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:

$$h(s, O) = \frac{(O \wedge M(s)) \times \mu_s}{2^{64-n(s)}} \tag{27}$$

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:

$$A(s, O) = \text{AttackTable}_s \left[\frac{(O \wedge M(s)) \times \mu_s}{2^{64-n(s)}} \right] \tag{28}$$

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$_s$
```

Space-Time Tradeoff

For rooks:

- Corner squares: $n(s) = 12$ bits $\Rightarrow$ 4096 entries
- Edge squares: typically $n(s) = 10 - 11$ bits
- Center squares: typically $n(s) = 10 - 12$ bits

For bishops:

- Corners: $n(s) = 5 - 7$ bits $\Rightarrow 32 - 128$ entries
- Center: $n(s) = 9$ bits $\Rightarrow$ 512 entries

Total memory for pre-computed tables: approximately 800KB for both rooks and bishops.

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:

$$V(p,d) = \begin{cases} \text{eval}(p) & \text{if } d = 0 \text{ or } p \in L \\ \max\limits_{m \in M(p)} V(m(p), d-1) & \text{if } p \text{ is MAX node} \\ \min\limits_{m \in M(p)} V(m(p), d-1) & \text{if } p \text{ is MIN node} \end{cases} \tag{29}$$

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:

$$V_{AB}(p, d, \alpha, \beta) = \begin{cases} \text{eval}(p) & \text{if } d = 0 \text{ or } p \in L \\ \text{MaxValue}(p, d, \alpha, \beta) & \text{if MAX node} \\ \text{MinValue}(p, d, \alpha, \beta) & \text{if MIN node} \end{cases} \tag{30}$$

For the maximizing player:

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 $v$
```

For the minimizing player:

Algorithm 3 MinValue (Minimizing Player in Alpha-Beta)

```
Input: Position $p$, depth $d$, bounds $\alpha$, $\beta$
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:

$$\beta \leq \alpha \implies \text{prune}(p) \tag{31}$$

Correctness and Efficiency

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

$$V_{AB}(p, d, -\infty, +\infty) = V(p, d) \quad \forall p, d \tag{32}$$

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

$$N_{best} = O(b^{d/2}) \tag{33}$$

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

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

$$N_{worst} = O(b^d) \tag{34}$$

This highlights the critical importance of move ordering (discussed in Section 8).

5 Move Generation

Purpose

Move generation is the process of enumerating all legal moves in a given chess position. A two-stage approach is standard: first generate *pseudo-legal* moves (moves that obey piece movement rules but may leave the king in check), then filter to obtain *legal* moves (those that don't leave one's own king in check). This separation improves performance by deferring expensive legality checks.

Pseudo-Legal Move Generation

For each piece type $\tau \in \{\text{Pawn, Knight, Bishop, Rook, Queen, King}\}$ and each square s occupied by a piece of type τ belonging to the side to move, generate candidate moves:

$$M_{\text{pseudo}}(s, \tau, c) = \{(s, t, f) \mid t \in A_\tau(s, O) \wedge \text{valid}(s, t, c)\} \tag{35}$$

where:

- $A_\tau(s, O)$ is the attack bitboard for piece type τ at square s given occupancy O
- $c \in \{w, b\}$ is the color of the moving side
- f encodes move flags (capture, en passant, castling, promotion)
- $\text{valid}(s, t, c)$ checks that square t is not occupied by a friendly piece

Attack Generation by Piece Type

1. **Pawn attacks** (color-dependent):

   For white pawns on square s:

   $$\begin{aligned} A_P^w(s) =& \text{NorthWest}(2^s) \vee \text{NorthEast}(2^s) \quad \text{(captures)} && (36)\\ & \vee \text{NorthOne}(2^s) \wedge \neg O \quad \text{(single push)} && (37)\\ & \vee (\text{NorthOne}(\text{NorthOne}(2^s)) \wedge \neg O) \quad \text{(double push if rank 2)} && (38) \end{aligned}$$

   Plus en passant captures and promotion flags when reaching rank 8.

2. **Knight attacks** (pre-computed table):

   $$A_N(s) = \text{KnightTable}[s] \tag{39}$$

   where KnightTable[s] contains all squares a knight can reach from s (up to 8 squares).

3. **Bishop attacks** (diagonal sliding):

$$A_B(s, O) = \text{DiagAttacks}(s, O) \vee \text{AntiDiagAttacks}(s, O) \tag{40}$$

   Computed using magic bitboards (Section 3) or classical ray-casting.

4. **Rook attacks** (orthogonal sliding):

$$A_R(s, O) = \text{RankAttacks}(s, O) \vee \text{FileAttacks}(s, O) \tag{41}$$

5. **Queen attacks** (combined):

$$A_Q(s, O) = A_B(s, O) \vee A_R(s, O) \tag{42}$$

6. **King attacks** (pre-computed table):

$$A_K(s) = \text{KingTable}[s] \tag{43}$$

   Plus special handling for castling moves (requires checking castling rights, empty squares, and non-attacked squares).

Castling Legality

Castling move $(e1, g1)$ for White kingside is pseudo-legal if and only if:

$$\text{CastlingRights}_w[\text{kingside}] = \text{true} \quad \wedge \tag{44}$$
$$(O \wedge \{f1, g1\}) = \emptyset \quad \wedge \tag{45}$$
$$\neg\text{isAttacked}(e1, b) \quad \wedge \tag{46}$$
$$\neg\text{isAttacked}(f1, b) \quad \wedge \tag{47}$$
$$\neg\text{isAttacked}(g1, b) \tag{48}$$

Similar conditions apply for queenside and Black castling.

Legal Move Filtering

A pseudo-legal move m is legal if and only if executing m does not leave the moving side's king in check:

$$\text{legal}(m, p) \iff \neg\text{inCheck}(p \circ m, \text{sideToMove}(p)) \tag{49}$$

where:

$$\text{inCheck}(p, c) \iff \exists s \in \text{KingSquare}(p, c) : \text{isAttacked}(p, s, \neg c) \tag{50}$$

The isAttacked(p, s, c) function returns true if square s is attacked by color c in position p:

$$\text{isAttacked}(p, s, c) \iff \bigvee_{\tau \in \mathcal{P}} (A_\tau(s, O) \wedge B_{\tau,c}) \neq \emptyset \tag{51}$$

This checks whether any piece of color c attacks square s.

Complete Legal Move Set

$$M_{\text{legal}}(p) = \{m \in M_{\text{pseudo}}(p) \mid \text{legal}(m, p)\} \tag{52}$$

Optimization: Early Legality Detection

For efficiency, some illegal moves can be detected without making the move:

- **Pinned pieces**: If a piece is absolutely pinned to the king, it can only move along the pin ray
- **King moves**: Only 8 possible destinations; check each for attacks
- **En passant**: Requires special check for horizontal pins

6 Zobrist Hashing

Purpose

Zobrist hashing provides an efficient method to compute a nearly-unique hash value for chess positions. This enables fast transposition table lookups (storing previously evaluated positions) and threefold repetition detection. The key advantage is incremental updating: the hash can be updated in $O(1)$ time when making or unmaking moves, rather than recomputing from scratch.

Hash Function Design

Let $h : \mathcal{S} \to \mathbb{Z}_{2^{64}}$ be a hash function from the space of chess positions $\mathcal{S}$ to 64-bit integers. We construct h using the XOR operation and pre-generated random numbers.

Define the following random 64-bit numbers (generated once at initialization):

- $Z_{\tau,c,s}$ for each piece type τ, color c, and square s: $6 \times 2 \times 64 = 768$ numbers
- $Z_{\text{ep},f}$ for en passant on each file f: 8 numbers
- $Z_{\text{castle},r}$ for each castling right $r \in \{\text{WK, WQ, BK, BQ}\}$: 4 numbers
- Z_{side} for side to move (1 number for black; white is implicit with hash unchanged)

Total: $768 + 8 + 4 + 1 = 781$ random 64-bit numbers.

Complete Hash Computation

For a position p, the Zobrist hash is:

$$\begin{aligned} h(p) = {} & \left(\bigoplus_{\substack{s\in[0,63] \\ \text{piece}(p,s)=(\tau,c)}} Z_{\tau,c,s} \right) \oplus \\ & \left(\bigoplus_{r\in R(p)} Z_{\text{castle},r} \right) \oplus \\ & \begin{cases} Z_{\text{ep},f} & \text{if en passant square exists on file } f \\ 0 & \text{otherwise} \end{cases} \oplus \\ & \begin{cases} Z_{\text{side}} & \text{if side to move is black} \\ 0 & \text{if side to move is white} \end{cases} \end{aligned} \tag{53}$$

where:

- $\text{piece}(p, s)$ returns the piece type and color at square s in position p
- $R(p)$ is the set of available castling rights in position p
- $\oplus$ denotes the XOR operation

Incremental Hash Updates

The power of Zobrist hashing lies in incremental updates. When making a move that changes the position from p to p':

1. Moving a piece from s_1 to s_2:

$$h(p') = h(p) \oplus Z_{\tau,c,s_1} \oplus Z_{\tau,c,s_2} \tag{54}$$

2. Capturing (piece (τ_1, c_1) at s_1 captures piece (τ_2, c_2) at s_2):

$$h(p') = h(p) \oplus Z_{\tau_1,c_1,s_1} \oplus Z_{\tau_1,c_1,s_2} \oplus Z_{\tau_2,c_2,s_2} \tag{55}$$

3. Castling rights change from R to R':

$$h(p') = h(p) \oplus \left(\bigoplus_{r\in R\triangle R'} Z_{\text{castle},r} \right) \tag{56}$$

where $R\triangle R'$ is the symmetric difference (rights that changed).

4. En passant square changes:

$$h(p') = h(p) \oplus Z_{\text{ep},f_{\text{old}}} \oplus Z_{\text{ep},f_{\text{new}}} \tag{57}$$

(XOR with 0 if no en passant square exists).

5. Side to move changes:

$$h(p') = h(p) \oplus Z_{\text{side}} \tag{58}$$

This is applied after every move.

Hash Collision Analysis

For a uniform random hash function over 2^{64} values and a transposition table storing k positions, the expected number of collisions is:

$$\mathbb{E}[\text{collisions}] = \frac{k(k-1)}{2 \cdot 2^{64}} \approx \frac{k^2}{2^{65}} \tag{59}$$

For $k = 2^{24}$ (approximately 16 million positions):

$$\mathbb{E}[\text{collisions}] \approx \frac{2^{48}}{2^{65}} = \frac{1}{2^{17}} \approx 0.0000076 \tag{60}$$

The collision probability remains negligible for reasonable table sizes.

Applications

1. **Transposition tables**: Store $\langle h(p), \text{depth}, \text{score}, \text{bestMove}\rangle$ tuples
2. **Repetition detection**: Maintain a history of hashes; threefold repetition occurs if $h(p)$ appears 3 times
3. **Opening books**: Index opening positions by hash for fast lookup

7 Principal Variation

Purpose

The principal variation (PV) is the sequence of best moves found by the search algorithm, representing the expected continuation if both sides play optimally according to the engine's evaluation. Tracking the PV serves multiple purposes: understanding the engine's reasoning, providing informative output to users, enabling iterative deepening schemes, and improving move ordering in subsequent searches.

Formal Definition

Let $\text{PV}(p, d)$ denote the principal variation from position p at depth d. It is a sequence of moves:

$$\text{PV}(p, d) = \langle m_1, m_2, \ldots, m_k \rangle \tag{61}$$

where $k \leq d$ and each m_i is the best move at depth $d - i + 1$.

Recursively:

$$\text{PV}(p, d) = \begin{cases} \langle\rangle & \text{if } d = 0 \text{ or } p \text{ is terminal} \\ \langle m^* \rangle \circ \text{PV}(p \circ m^*, d-1) & \text{otherwise} \end{cases} \tag{62}$$

where m^* is the best move at the current position:

$$m^* = \begin{cases} arg\,max_{m \in M(p)} V_{AB}(p \circ m, d-1, \alpha, \beta) & \text{if MAX node} \\ arg\,min_{m \in M(p)} V_{AB}(p \circ m, d-1, \alpha, \beta) & \text{if MIN node} \end{cases} \tag{63}$$

and $\circ$ denotes sequence concatenation.

Triangular PV Array

During alpha-beta search, the PV is stored in a triangular array to avoid expensive list operations:

$$\text{PV} : [0 \ldots d_{\max}] \times [0 \ldots d_{\max}] \to \text{Move} \tag{64}$$

where PV[i][j] contains the j-th move in the principal variation at depth i.

Update rule: When a better move m is found at depth d (ply p):

Algorithm 4 Update Principal Variation

```
Input: Ply $p$, better move $m$ found at depth $d$, child PV at $p+1$
Output: Updated PV at ply $p$
1: PV[$p$][0] $\leftarrow m$                                  ▷ Store best move
2: PV-length[$p$] $\leftarrow 1 +$ PV-length[$p+1$]           ▷ Length is 1 + child PV length
3: for $i \leftarrow 0$ to PV-length[$p+1$] $- 1$ do
4:     PV[$p$][$i+1$] $\leftarrow$ PV[$p+1$][$i$]              ▷ Copy child's PV
5: end for
```

This copies the child's PV and prepends the current move.

PV Extraction

After search completes, extract the PV from the root:

$$\text{FinalPV} = \langle \text{PV}[0][0], \text{PV}[0][1], \ldots, \text{PV}[0][\text{PV-length}[0] - 1] \rangle \tag{65}$$

Iterative Deepening and PV

In iterative deepening, the engine searches to increasing depths $d = 1, 2, 3, \ldots$ until time runs out. The PV from iteration d provides excellent move ordering for iteration $d + 1$:

$$\text{SearchOrder}_{d+1}(m) = \begin{cases} 10^9 - i & \text{if } m = \text{PV}_d[i] \\ \text{OtherHeuristics}(m) & \text{otherwise} \end{cases} \quad (66)$$

This ensures PV moves are searched first, leading to earlier cutoffs.

PV Stability

In stable positions, the PV often remains consistent across iterations:

$$\text{PV}_d[0 \dots k] = \text{PV}_{d+1}[0 \dots k] \quad \text{for small } k \quad (67)$$

This property can be exploited for aspiration windows and other search optimizations.

8 Piece-Square Tables

Purpose

Piece-square tables (PSTs) encode positional knowledge by assigning a numerical value to each piece on each square. They capture strategic principles such as "knights are stronger in the center," "rooks belong on open files," and "advanced passed pawns are valuable." PSTs enable fast position evaluation without complex pattern recognition, making them ideal for the leaf node evaluation function in alpha-beta search.

Formalism

For each piece type $\tau \in \mathcal{P} = \{P, N, B, R, Q, K\}$ and each square $s \in [0, 63]$, define a value:

$$\text{PST}_\tau[s] \in \mathbb{Z} \quad (68)$$

typically expressed in *centipawns* (hundredths of a pawn).

The piece-square contribution to the evaluation is:

$$E_{\text{PST}}(p) = \sum_{c \in \{w,b\}} \sigma(c) \sum_{\tau \in \mathcal{P}} \sum_{s \in S_{\tau,c}(p)} \text{PST}_\tau[s] \quad (69)$$

where:

- $\sigma(w) = +1$, $\sigma(b) = -1$ (score from White's perspective)
- $S_{\tau,c}(p)$ is the set of squares occupied by pieces of type τ and color c in position p

Symmetry for Black Pieces

Black's PST values are typically mirrored vertically from White's:

$$\text{PST}_\tau^b[s] = -\text{PST}_\tau^w[\text{mirror}(s)] \tag{70}$$

where:

$$\text{mirror}(s) = s \oplus 56 = (7 - \text{rank}(s)) \times 8 + \text{file}(s) \tag{71}$$

Example: Pawn PST

A typical pawn PST encourages central advancement (values in centipawns):

Rank 8 (promotion rank)							
0	0	0	0	0	0	0	0
Rank 7							
50	50	50	50	50	50	50	50
Rank 6							
10	10	20	30	30	20	10	10
Rank 5							
5	5	10	25	25	10	5	5
Rank 4							
0	0	0	20	20	0	0	0
Rank 3							
5	-5	-10	0	0	-10	-5	5
Rank 2							
5	10	10	-20	-20	10	10	5
Rank 1							
0	0	0	0	0	0	0	0

This table penalizes pawns on the second rank center squares (discouraging $e3/d3$ before development) and rewards central advancement.

Example: Knight PST

Knights are strongest in the center and weak on edges:

-50	-40	-30	-30	-30	-30	-40	-50
-40	-20	0	0	0	0	-20	-40
-30	0	10	15	15	10	0	-30
-30	5	15	20	20	15	5	-30
-30	0	15	20	20	15	0	-30
-30	5	10	15	15	10	5	-30
-40	-20	0	5	5	0	-20	-40
-50	-40	-30	-30	-30	-30	-40	-50

Phase-Dependent King PST

King safety requirements differ between middlegame and endgame. Define two separate PSTs:

$$\text{PST}_K^{\text{mg}}[s] \quad \text{(middlegame: encourage castling)} \tag{72}$$
$$\text{PST}_K^{\text{eg}}[s] \quad \text{(endgame: encourage centralization)} \tag{73}$$

Interpolate based on game phase $\phi \in [0, 1]$:

$$\text{PST}_K[s] = (1-\phi) \cdot \text{PST}_K^{\text{mg}}[s] + \phi \cdot \text{PST}_K^{\text{eg}}[s] \tag{74}$$

where ϕ is computed from remaining material:

$$\phi = 1 - \frac{\text{NonPawnMaterial}(p)}{\text{NonPawnMaterial}_{\text{start}}} \tag{75}$$

Incremental PST Updates

Like Zobrist hashing, PST evaluation can be updated incrementally. Maintain a running PST score:

$$\text{PST-score}(p') = \text{PST-score}(p) + \Delta_{\text{PST}}(m) \tag{76}$$

where for a move from s_1 to s_2:

$$\Delta_{\text{PST}}(m) = \sigma(c)\,(\text{PST}_\tau[s_2] - \text{PST}_\tau[s_1]) + \text{capture-bonus} \tag{77}$$

9 Move Ordering

Purpose

Move ordering is critical for alpha-beta efficiency. Searching better moves first increases the likelihood of early cutoffs, reducing the number of nodes examined. With perfect move ordering, alpha-beta achieves $O(b^{d/2})$ complexity; with random ordering, it degrades to $O(b^d)$. The goal is to approximate optimal ordering using fast heuristics.

Move Ordering Heuristics

Define a scoring function score : Move $\to \mathbb{R}$ that ranks moves by expected quality. Moves are sorted in descending order of score before search:

$$M_{\text{ordered}}(p) = \text{sort}(M(p), \lambda m_1, m_2.\text{score}(m_1) > \text{score}(m_2)) \tag{78}$$

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

For captures, prioritize taking high-value pieces with low-value pieces:

$$\text{score}_{\text{MVV-LVA}}(m) = 10 \cdot V(\text{captured}) - V(\text{attacker}) \tag{79}$$

where $V : \mathcal{P} \to \mathbb{R}$ is the piece value function:

$$\begin{aligned}
V(P) &= 1 && (80)\\
V(N) &= 3 && (81)\\
V(B) &= 3 && (82)\\
V(R) &= 5 && (83)\\
V(Q) &= 9 && (84)\\
V(K) &= 0 \quad \text{(king captures handled separately)} && (85)
\end{aligned}$$

Example ordering:

$$\text{Queen captures pawn} > \text{Pawn captures queen} > \text{Knight captures bishop} > \cdots \tag{86}$$

Killer Move Heuristic

Killer moves are quiet moves (non-captures) that caused beta cutoffs at the same ply in sibling positions. Maintain a table:

$$\text{Killers} : [0 \ldots d_{\max}] \times [0 \ldots k-1] \to \text{Move} \tag{87}$$

where Killers$[p][i]$ is the i-th killer move at ply p (typically $k = 2$).
When a quiet move m causes a beta cutoff at ply p:

$$\begin{aligned}
\text{Killers}[p][1] &\leftarrow \text{Killers}[p][0] && (88)\\
\text{Killers}[p][0] &\leftarrow m && (89)
\end{aligned}$$

Killer move scoring:

$$\text{score}_{\text{killer}}(m, p) = \begin{cases} 10^6 & \text{if } m = \text{Killers}[p][0] \\ 10^5 & \text{if } m = \text{Killers}[p][1] \\ 0 & \text{otherwise} \end{cases} \tag{90}$$

History Heuristic

The history heuristic tracks how often each move caused a cutoff across the entire search tree:

$$\text{History} : [0 \ldots 63] \times [0 \ldots 63] \to \mathbb{N} \tag{91}$$

where History$[s_1][s_2]$ counts cutoffs for moves from s_1 to s_2.
When move $m = (s_1, s_2, \cdots)$ causes a beta cutoff:

$$\text{History}[s_1][s_2] \leftarrow \text{History}[s_1][s_2] + \text{depth}^2 \tag{92}$$

Squaring the depth gives more weight to cutoffs closer to the root.
History move scoring:

$$\text{score}_{\text{history}}(m) = \text{History}[\text{from}(m)][\text{to}(m)] \tag{93}$$

Composite Move Ordering

Combine heuristics with decreasing priority:

$$\text{score}(m) = \begin{cases} 10^{10} & \text{if } m = \text{HashMove} \\ 10^9 + \text{score}_{\text{MVV-LVA}}(m) & \text{if } m \text{ is capture} \\ 10^6 + i & \text{if } m = \text{Killers}[p][i] \\ \text{score}_{\text{history}}(m) & \text{otherwise (quiet moves)} \end{cases} \tag{94}$$

where HashMove is the best move from the transposition table.

Expected Performance Gain

With good move ordering, the effective branching factor can be reduced by a factor of 2-4:

$$b_{\text{eff}} \approx \frac{b}{2} \quad \text{to} \quad \frac{b}{4} \tag{95}$$

For chess with $b \approx 35$, this means searching to depth 12 with $b_{\text{eff}} \approx 9$ instead of depth 6 with $b = 35$ in the same time.

Lazy Evaluation

To further optimize, generate and score moves lazily:

1. Generate hash move (if exists) and search it
2. Generate captures, score with MVV-LVA, search in order
3. Generate killer moves, search them
4. Generate remaining quiet moves, score with history, search in order

This avoids generating all moves when an early cutoff occurs.

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] 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

[7] 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

[8] *Rotated Bitboards*. https://www.chessprogramming.org/Rotated_Bitboards

[9] *Magic Bitboards*. https://www.chessprogramming.org/Magic_Bitboardsi

[10] *GitHub Repository*. Zugzwang