Building a Chess Engine in Rust

How cb2026 works, one technique at a time, from an empty board to master strength

The cb2026 authors

Created: 2026-10-30 08:48:13 MDT

Every chapter matches a stage of the engine's development and a git tag in the repository (`stage-01-board`, `stage-02-movegen`, ...), so you can check out any stage, run it, change it and measure the result.

Each chapter gives the idea, the real source, how it is tested, what it was worth in Elo, and where to read further. Chapters numbered with a decimal, such as 6.1 or 14.2, came after stage 20, as strength work towards version 1.0.

Bound from book/src in github.com/cjus/chessengine at commit c364c15. The code excerpts are the source at that commit.

Contents

Chapters 1 to 21 follow the engine's stages; decimal chapters came later and sit beside the stage they build on.

Figures 4

Introduction 5

How to use this book 6

PART I: THE RULES OF CHESS

1 Board representation and FEN 9

2 Move generation and perft 15

3 Magic bitboards 27

4 Zobrist hashing and talking to GUIs (UCI) 32

PART II: SEARCHING

5 Material, negamax and alpha-beta 43

6 Iterative deepening and time management 63

6.1 Spending time where it's needed 76

7 Quiescence search and MVV-LVA 81

8 The transposition table 89

8.1 Buckets: one cache line per key 97

9 Piece-square tables and tapered evaluation 106

10 Move ordering 113

10.1 Continuation history 121

10.2 Capture history 128

11 Principal variation search and aspiration windows 139

12 Null-move pruning 153

12.1 ProbCut 158

13 Late move reductions and pruning 165

13.1 History pruning 173
14 Futility, razoring and SEE pruning 179
14.1 Extensions and reductions 186
14.2 Correction history 225

PART III: EVALUATION

15 A full hand-crafted evaluation 237
16 Texel tuning on self-play games 246

PART IV: A COMPLETE ENGINE

17 Lazy SMP 261
18 Tablebases, opening books and Chess960 268
19 Strength levels 281
20 XBoard, pondering and playing on Lichess 305

PART V: NEURAL NETWORKS

21 NNUE 314

Index **357**

Figures

Numbered by chapter. Every figure is vector art, so it stays sharp at any zoom and in print.

2.1	In check, a piece may only land on the check mask; a pinned piece may only move along its pin	18
3.1	A rook on d4 depends on only 10 squares, so its attacks are one of 1,024 table entries	28
5.1	Once one reply refutes a move, alpha-beta skips that move's other replies	47
6.1	Depth 8 finished past the soft limit, so depth 9 was never started	73
8.1.1	A probe that misses the cache fetches a whole line of eight slots; buckets use all of them	98
14.1.1	Where the search without the hash move scores decides between two extensions, a cutoff and a reduction	194
21.1	A move changes the accumulator by a few rows of weights instead of recomputing it	319

Introduction

This book explains how **cb2026** works, from an empty board to a master-strength chess engine, one technique at a time.

Every chapter matches a *stage* of the engine's development and a git tag in the repository (`stage-01-board`, `stage-02-movegen`, ...). At each tag the engine builds, passes its tests and plays legal chess, so you can check out any stage, run it, change it and measure the result.

Each chapter covers:

- **The idea:** what the technique does and why it works.
- **The code:** the real source, included from the repository so it can't drift out of date.
- **Testing it:** how we know it's correct.
- **What it's worth:** the Elo gain measured when it was added, from the Elo ledger.
- **Further reading:** the Chess Programming Wiki, papers and other engines.

Chapters in the table of contents that can't be opened yet are the ones still to be written. They appear as the engine reaches each stage.

How to use this book

Reading the book locally

```
cargo install mdbook
mdbook serve book --open
```

Running the engine at any stage

```
git clone https://github.com/cjus/chessengine
cd chessengine
git checkout stage-05-alphabeta   # any stage tag
cargo build --release
./target/release/cb2026
```

The Rust toolchain version is pinned in `rust-toolchain.toml`; `rustup` installs it on first use.

Code excerpts

The code in each chapter is included straight from the source files, so it always shows the current code. Functions grow as later stages add to them: chapter 5's negamax, read today, already has the transposition table from chapter 8 in it. To see the code exactly as a chapter first described it, check out that chapter's stage tag.

Switching techniques off

From the stages that add search techniques onwards, every technique can be switched off at runtime (in the shell with `set <technique>`

`off`), so you can measure what each one is worth yourself. `techniques` lists them all, and `ab <technique> [depth]` does the measuring for you: it searches the current position with the technique on and off and compares nodes, time and the move chosen.

```
> ab nullmove 7
Null-move pruning (nullmove), depth 7, one thread, empty hash table:
              nodes      time  move      score
  on           9088    0.00 s  d6        -0.08
  off         13402    0.01 s  d6        -0.08
Switched off, the search needs 1.47× the nodes.
```

One position says little; try several (and `bench`, which searches a fixed set).

Looking inside the search

The shell has commands for watching the search decide:

- **`stats`** after a search: how often the transposition table answered, how often the first move searched caused the cutoff (a direct measure of move ordering), how often each pruning technique fired, and how often late move reductions had to search again.
- **`tree <depth> [plies]`** prints the first plies of a search tree: each node's window and score, the cutoffs, and the moves pruned or reduced, with the reason. Start small (`tree 4 2`): trees grow fast.
- **`explain <move>`** searches a move and compares it with the engine's choice, showing the engine's answer to it.
- **`eval`** breaks the evaluation into its terms; **`eval diff <move>`** shows which terms a move changes.
- **`see <move>`** shows the static exchange evaluation of a capture.

```
> tree 4 2
Na6  depth 3 [+0.32, +0.33] (beta cutoff by Na3 (move 1)) -> +0.55
  (pass)  depth 0 [-0.33, -0.32] -> -0.18
  Na3  depth 2 [-0.33, -0.32] -> -0.55
    b5  pruned (late move pruning)
    ...
```

Read that as (in the position after 1. d3): after 1...Na6, the null move (`(pass)`) didn't cut off, but White's first real move, Na3, did. Below it, late move pruning skipped most of Black's quiet replies.

Quick matches

`match <engine> <games> [time control]` plays cb2026, exactly as you've set it up in the shell (level, techniques switched off, hash size), against another UCI engine: an older stage's binary, or any engine you have. `selfplay <games>` plays it against a copy of cb2026 at default settings, which is a quick way to see what switching a technique off costs in games:

```
> set nullmove off
> selfplay 6 nodes=20000
...
Match over: cb2026 1.0.0 (shell) scored 1.5/6 against cb2026 1.0.0 (default) (+1 -4
=1).
Elo difference: -191 ± 1219 (95%). A few games say little: the margin shows how
little.
```

Six games, or even a hundred, can't tell a 30 Elo change from noise; the `match` tool's SPRT (see `docs/testing.md`) decides when enough games have been played. Games are saved as PGN (`cb2026-match.pgn` unless you give `pgn <file>`).

Board representation and FEN

Stage 1 · tag `stage-01-board` *· code in* `crates/chess`

Before an engine can search or evaluate anything, it needs a way to hold a chess position in memory. That choice shapes everything built on top of it, so it's worth getting right first.

The idea

A position is more than where the pieces stand. To know which moves are legal and how the game can end, an engine also needs:

- **whose turn it is**;
- **castling rights**: whether each king and rook have moved;
- the **en passant square**: where a pawn that just made a double step could be captured;
- the **halfmove clock**, for the fifty-move rule;
- the **fullmove number**.

Numbering the squares

cb2026 numbers the squares 0 to 63, starting at a1 and going along each rank:

```
8 | 56 57 58 59 60 61 62 63
7 | 48 49 50 51 52 53 54 55
...
2 |  8  9 10 11 12 13 14 15
1 |  0  1  2  3  4  5  6  7
     a  b  c  d  e  f  g  h
```

So `index = rank * 8 + file`, a step north is `+8` and a step east is `+1`. Most bitboard engines use this layout (the Chess Programming Wiki calls it *little-endian rank-file*).

```
/// One of the 64 squares, numbered as described in the [module docs](self).
#[derive(Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct Square(u8);

impl Square {
    /// The square on the given file and rank.
    pub const fn new(file: File, rank: Rank) -> Square {
        Square((rank as u8) * 8 + file as u8)
    }
```

Bitboards

A **bitboard** is a set of squares packed into one 64-bit integer: bit `i` set means square `i` is in the set. One bitboard can hold "all white pawns" or "every square a knight on g1 attacks", and set operations become single instructions:

Question	Bitboard code
Squares in either set	`a | b`
Squares in both sets	`a & b`
Squares in `a` but not `b`	`a & !b`
How many squares?	`a.count()` (one `popcnt` instruction)
Which is the first square?	`a.first()` (one "count trailing zeros" instruction)

```
/// A set of squares, one bit per square.
#[derive(Clone, Copy, Default, PartialEq, Eq, Hash)]
pub struct Bitboard(pub u64);

impl Bitboard {
    /// The empty set.
    pub const EMPTY: Bitboard = Bitboard(0);
    /// All 64 squares.
    pub const ALL: Bitboard = Bitboard(!0);

    /// The set containing just `square`.
    pub const fn from_square(square: Square) -> Bitboard {
        Bitboard(1 << square.index())
    }

    /// Whether `square` is in the set.
    pub const fn contains(self, square: Square) -> bool {
        self.0 & (1 << square.index()) != 0
    }
```

Iterating over a bitboard's squares takes one step per square *in the set*, not one per square on the board: find the lowest set bit, then clear it with `x & (x - 1)`.

Two views of the board

cb2026 keeps the pieces in two forms at once:

- **Bitboards**, one per piece type and one per colour. "Where are White's knights?" is the intersection of the knight set and the white set.
- **A mailbox**: a 64-entry array saying what is on each square. "What's on e4?" is one array read.

```
/// A chess position.
///
/// Build one from FEN with [`Position::from_fen`], or start a game with
/// [`Position::startpos`].
#[derive(Clone, PartialEq, Eq)]
pub struct Position {
    /// One bitboard per piece type, both colours together (indexed by
[`PieceType::index`]).
    by_type: [Bitboard; 6],
    /// One bitboard per colour, all piece types together (indexed by
[`Color::index`]).
    by_color: [Bitboard; 2],
    /// What stands on each square (indexed by [`Square::index`]).
    mailbox: [Option<Piece>; 64],
    side_to_move: Color,
    castling: CastlingRights,
    en_passant: Option<Square>,
    halfmove_clock: u32,
    fullmove_number: u32,
    /// The Zobrist key, kept up to date by every change (see [`crate::zobrist`]).
    hash: u64,
}
```

Every change to the board updates both, so they can never disagree:

```
/// Puts `piece` on the empty square `square`, updating both views of the board.
pub(crate) fn put_piece(&mut self, square: Square, piece: Piece) {
    debug_assert!(
        self.mailbox[square.index()].is_none(),
        "{square} is occupied"
    );
    self.mailbox[square.index()] = Some(piece);
    self.by_type[piece.piece_type.index()] |= Bitboard::from_square(square);
    self.by_color[piece.color.index()] |= Bitboard::from_square(square);
    self.hash ^= zobrist::piece_square(piece, square);
}
```

The cost is a little extra work per move; the gain is that both kinds of question are cheap.

Castling rights that also work for Chess960

In standard chess the castling rooks start on a and h, so four flags (`KQkq`) are enough. In Chess960 the rooks can start on other files, so cb2026 stores *the file of the rook* for each right. Standard chess is just the case where those files are a and h, and one piece of code handles both games.

FEN: a position as one line of text

Forsyth–Edwards Notation writes a whole position on one line. The starting position is:

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq – 0 1
```

Its six fields are:

1. the pieces, rank 8 first, with digits counting empty squares;
2. the side to move;
3. castling rights;
4. the en passant square;
5. the halfmove clock;
6. the fullmove number.

GUIs send positions to engines in FEN, test suites are written in it, and the shell's `fen` command uses it.

For Chess960, cb2026 reads both common extensions: **X-FEN**, where `K` / `Q` mean the outermost rook and a file letter names an inner one, and **Shredder-FEN**, which always uses file letters, like `HAha`. It writes X-FEN, which is plain `KQkq` in standard chess.

The parser also rejects positions that can't occur in a game:

- a side without exactly one king;
- a pawn on the first or last rank;
- castling rights without the king and rook in place;
- an en passant square with no pawn that could have just made a double step.

Each error explains itself, for example:

```
Invalid FEN: pawn on d1: pawns can't stand on the first or last rank
```

Try it

```
$ cargo run --release
> fen r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq - 0 1
> bb N          # White's knights as a bitboard
> bb occupied
> flip
> board ascii
```

Testing it

- Every square's name parses back to the same square, and `flip_rank` twice is the identity.
- Bitboard set operations, iteration order and the grid display.
- A list of positions, including Chess960 ones, survives FEN → position → FEN unchanged.
- Each kind of bad FEN gives the expected error.
- `Position::is_consistent` checks that the bitboards and the mailbox agree; every test position passes it.

What it's worth

Nothing yet in Elo: stage 1 can't play. It is the foundation that move generation (stage 2) builds on.

Further reading

- Chess Programming Wiki: Board Representation, Bitboards, Square Mapping Considerations, Mailbox, Forsyth-Edwards Notation, X-FEN.

Move generation and perft

Stage 2 · tag `stage-02-movegen` *· code in* `crates/chess`

An engine spends much of its time asking "what are the legal moves here?". Getting the answer exactly right is the first hard part of writing one, and **perft** is how we prove we did.

Attacks

Everything starts with attacks: which squares does each piece attack?

Leaping pieces: tables

A knight on g1 attacks e2, f3 and h3 whatever else is on the board. So the attacks of knights, kings and pawns are computed once, at compile time, into 64-entry tables:

```
/// `KNIGHT_ATTACKS[sq]`: the squares a knight on `sq` attacks.
pub const KNIGHT_ATTACKS: [Bitboard; 64] = leaper_table(&KNIGHT_STEPS);

/// `KING_ATTACKS[sq]`: the squares a king on `sq` attacks.
pub const KING_ATTACKS: [Bitboard; 64] = leaper_table(&KING_STEPS);
```

Sliding pieces: walking rays

Bishops, rooks and queens slide until they hit something, so their attacks depend on the occupied squares. Stage 2 computes them the obvious way: step out in each direction and stop after the first occupied square (which is attacked either way: an enemy piece can be captured, an own piece is defended).

```
/// Slides from `square` in each direction until the edge or the first occupied square
/// (included, since a piece there can be captured or is defended).
fn slider_attacks(square: Square, occupied: Bitboard, directions: &[(i8, i8)]) ->
Bitboard {
    let mut attacks = Bitboard::EMPTY;
    for &direction in directions {
        let mut index = square.index();
        while let Some(next) = step(index, direction) {
            let next_square = Square::from_index(next).expect("step stays on the
board");
            attacks = attacks.with(next_square);
            if occupied.contains(next_square) {
                break;
            }
            index = next;
        }
    }
    attacks
}
```

This is simple and clearly correct, which makes it the reference that stage 3's much faster *magic bitboards* are tested against.

Who attacks a square?

A neat trick turns "what does this piece attack?" around into "what attacks this square?": put each kind of piece on the square and see which enemy pieces of that kind it would attack. A knight on e4 attacks exactly the squares from which a knight attacks e4.

```
/// All pieces, of both colours, that attack `square`, if the occupied squares were
/// `occupied`.
///
/// The trick: put each kind of piece on `square` and see what it would attack. A
knight on
/// `square` attacks exactly the squares from which a knight attacks `square`, and the
same
/// holds for every piece (with the pawn directions swapped).
pub fn attackers_to(&self, square: Square, occupied: Bitboard) -> Bitboard {
    let rooks = self.pieces_of_type(PieceType::Rook) |
self.pieces_of_type(PieceType::Queen);
    let bishops =
        self.pieces_of_type(PieceType::Bishop) |
self.pieces_of_type(PieceType::Queen);
    (pawn_attacks(Color::White, square) & self.pieces(Color::Black, PieceType::Pawn))
        | (pawn_attacks(Color::Black, square) & self.pieces(Color::White,
PieceType::Pawn))
        | (knight_attacks(square) & self.pieces_of_type(PieceType::Knight))
        | (king_attacks(square) & self.pieces_of_type(PieceType::King))
        | (rook_attacks(square, occupied) & rooks)
        | (bishop_attacks(square, occupied) & bishops)
}
```

Legal moves, directly

The simplest generator makes *pseudo-legal* moves (each piece's moves, ignoring checks), plays each one and throws away those that leave the king in check. cb2026 instead generates only legal moves, using two facts about the position:

- **Checkers.** In double check only the king can move. In single check, every other move must capture the checker or block between it and the king: a *check mask* of allowed destinations.
- **Pins.** A piece standing alone between its king and an enemy slider may only move along that line.

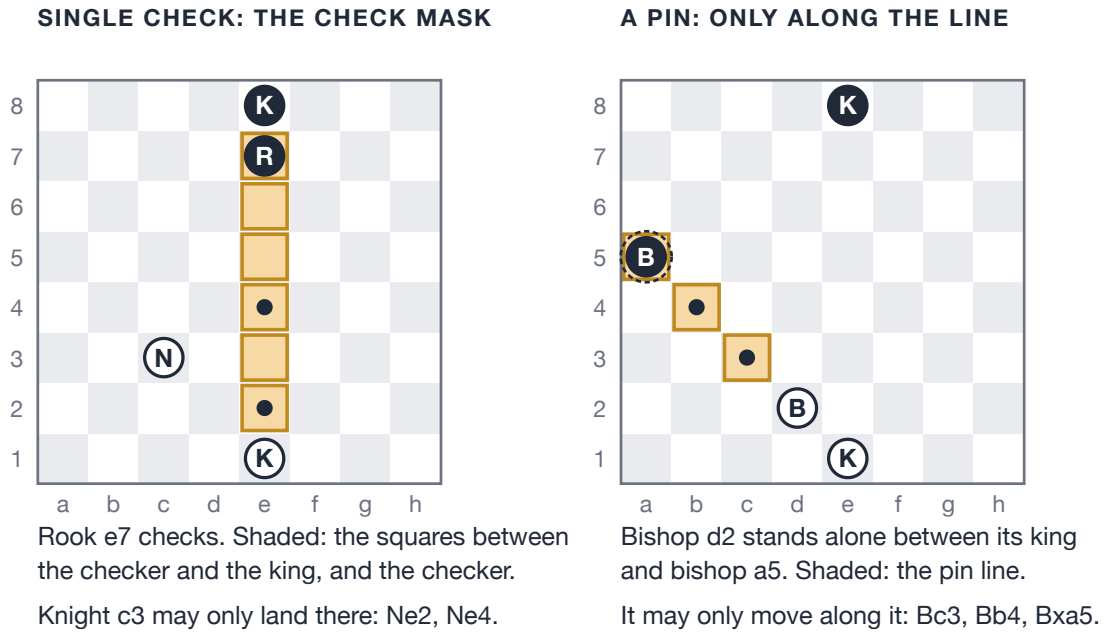

Figure 2.1 In check, a piece may only land on the check mask; a pinned piece may only move along its pin

```
fn generate(&self, list: &mut MoveList, gen_kind: Gen) {
    let us = self.side_to_move();
    let them = !us;
    let own = self.pieces_of_color(us);
    let enemy = self.pieces_of_color(them);
    let occupied = self.occupied();
    let king = self.king_square(us);
    let checkers = self.checkers();
    // Where pieces may move: any square not holding one of ours, or only enemy
pieces.
    let targets = match gen_kind {
        Gen::All => !own,
        Gen::Noisy => enemy,
    };

    // King moves: never onto a square the enemy attacks. Lift the king off the board
    // first, or it would seem to block a slider's attack on the square behind it.
    let without_king = occupied.without(king);
    for to in king_attacks(king) & targets {
        if (self.attackers_to(to, without_king) & enemy).is_empty() {
            list.push(Move::new(king, to, capture_or_quiet(enemy, to)));
        }
    }

    // In double check, only the king can move.
    if checkers.count() > 1 {
        return;
    }

    // With one checker, other moves must capture it or block the check.
    let check_mask = match checkers.first() {
        Some(checker) => between(king, checker).with(checker),
        None => Bitboard::ALL,
    };
    let pinned = self.pinned();
    // A pinned piece may only move along the line through its king and itself.
    let pin_mask = |from: Square| {
        if pinned.contains(from) {
            line(king, from)
        } else {
            Bitboard::ALL
        }
    };
```

Note the line that lifts the king off the board before testing its destination squares. Without it, a king in check from a rook on the same rank could "escape" one square further along the rank, because the king itself would seem to block the rook.

The awkward cases

Two moves are rare and subtle, so cb2026 checks them by playing them on a copy and looking at the king:

- **En passant** removes *two* pawns from one rank, which can expose the king to a rook on that rank even though neither pawn was pinned on its own.
- **Castling** in Chess960, where the castling rook itself might be what shields the king.

Castling gets one more idea: the move is stored as "king takes own rook" (`e1h1`). That describes Chess960 castling unambiguously, even when the king doesn't move at all, and the same code then handles standard chess.

```
/// Castling moves, for standard chess and Chess960 alike.
///
/// Castling needs: the right still held; every square the king and rook travel over
or
/// land on empty (apart from the king and rook themselves); and no square the king
stands
/// on, crosses or lands on attacked. The attack test runs with the king and castling
rook
/// lifted off the board, which catches the Chess960 case where the rook was shielding
the
/// king from an enemy rook or queen on the back rank.
fn generate_castling(&self, list: &mut MoveList) {
    let us = self.side_to_move();
    let king = self.king_square(us);
    for side in CastlingSide::ALL {
        let Some(rook_file) = self.castling().rook_file(us, side) else {
            continue;
        };
        let rook = Square::new(rook_file, king.rank());
        let (king_to, rook_to) = castling_destinations(us, side);

        let king_path = between(king, king_to).with(king_to).with(king);
        let rook_path = between(rook, rook_to).with(rook_to);
        let movers = Bitboard::from_square(king).with(rook);
        if !((king_path | rook_path) & self.occupied() & !movers).is_empty() {
            continue;
        }
        let lifted = self.occupied() & !movers;
        let enemy = self.pieces_of_color(!us);
        let safe = king_path
            .squares()
            .all(|square| (self.attackers_to(square, lifted) & enemy).is_empty());
        if safe {
            list.push(Move::new(king, rook, MoveKind::Castle));
        }
    }
}
```

Making moves

`make_move` updates the board and everything that depends on the move:

- captures, including en passant;
- promotions;
- castling rights, which are lost when a king or castling rook moves or a castling rook is captured;

- the en passant square;
- both clocks.

```
/// Plays `mv`, which must be legal in this position (for example one from
/// [`Position::legal_moves`]).
pub fn make_move(&mut self, mv: Move) {
    let us = self.side_to_move();
    let them = !us;
    let from = mv.from();
    let to = mv.to();
    let mut halfmove_clock = self.halfmove_clock() + 1;
    let mut en_passant = None;

    if mv.kind() == MoveKind::Castle {
        // `to` is the castling rook. Lift both pieces, then put them on their
destinations
        // (in Chess960 a destination may be the other piece's starting square).
        let side = if to.file() > from.file() {
            CastlingSide::KingSide
        } else {
            CastlingSide::QueenSide
        };
        let (king_to, rook_to) = castling_destinations(us, side);
        let king = self.remove_piece(from).expect("castling king");
        let rook = self.remove_piece(to).expect("castling rook");
        self.put_piece(king_to, king);
        self.put_piece(rook_to, rook);
        self.remove_castling_color(us);
    } else {
        let piece = self.remove_piece(from).expect("a piece on the from-square");

        if mv.is_capture() {
            // En passant captures the pawn beside the destination, not on it.
            let captured_square = match mv.kind() {
                MoveKind::EnPassant => Square::new(to.file(), from.rank()),
                _ => to,
            };
            self.remove_piece(captured_square)
                .expect("a piece to capture");
            // Capturing a castling rook removes that castling right.
            self.remove_rook_right(them, captured_square);
            halfmove_clock = 0;
        }

        let landing = match mv.promotion() {
            Some(piece_type) => Piece::new(us, piece_type),
            None => piece,
        };
        self.put_piece(to, landing);

        match piece.piece_type {
            PieceType::Pawn => {
                halfmove_clock = 0;
                if mv.kind() == MoveKind::DoublePawnPush {
                    // Record the skipped square only if an enemy pawn could capture
there,
                    // so equal positions always look equal.
                    let skipped = Square::new(from.file(), middle_rank(from.rank(),
to.rank()));
```

```
                    if !(pawn_attacks(us, skipped) & self.pieces(them,
PieceType::Pawn))
                        .is_empty()
                    {
                        en_passant = Some(skipped);
                    }
                }
            }
            PieceType::King => self.remove_castling_color(us),
            PieceType::Rook => self.remove_rook_right(us, from),
            _ => {}
        }
    }

    let fullmove_number = self.fullmove_number() + u32::from(us == Color::Black);
    self.set_clocks(halfmove_clock, fullmove_number);
    self.set_en_passant(en_passant);
    self.set_side_to_move(them);
}
```

To try a move and come back, cb2026 copies the position first ("copy-make"). A position is a few hundred bytes, so copying is cheap and there's no undo code to get wrong.

One detail: the en passant square is recorded only when an enemy pawn could actually capture there. Then two positions that are really the same always look the same, which matters once positions are compared and hashed (stage 4).

Perft: proving it

Perft counts the leaf positions of the full game tree to a given depth: every legal move, every reply, and so on.

```
/// The number of leaf positions `depth` plies below `pos`.
pub fn perft(pos: &Position, depth: u32) -> u64 {
    if depth == 0 {
        return 1;
    }
    let moves = pos.legal_moves();
    if depth == 1 {
        // Bulk counting: each legal move leads to exactly one leaf.
        return moves.len() as u64;
    }
    moves
        .iter()
        .map(|&mv| {
            let mut child = pos.clone();
            child.make_move(mv);
            perft(&child, depth - 1)
        })
        .sum()
}
```

The counts for standard test positions have been reproduced by many independent programs. A move generator that matches them exactly at every depth is almost certainly right, because any bug changes the count. When a count is wrong, `divide` splits it by first move, which points straight at the move whose subtree is wrong.

cb2026's suites live in `tests/data/`:

- `perft.epd`: the Chess Programming Wiki positions, including "Kiwipete", plus small positions aimed at classic bugs. Those bugs include an illegal en passant, castling that gives check, and underpromotion.
- `perft960.epd`: Chess960 start and middlegame positions.

The unit tests run every depth up to a million nodes. Every change is also checked at every depth in a release build (`cargo run --release -p cb2026-tools --bin perft-suite`), about 666 million positions.

Bulk counting. At depth 1 the count is just the number of legal moves, so they need not be played. That shortcut alone makes perft

several times faster.

Try it

```
> perft 5
Depth 5: 4865609 nodes in 0.030 s (160.0 million nodes/s)   # speed varies by machine
> e2e4
> legal
> divide 2
> fen r3k2r/8/8/8/2b5/8/8/R3K2R w KQkq - 0 1
> legal          # no e1g1: the bishop on c4 covers f1
> bb pinned
> attackers f1
```

What it's worth

Still nothing in Elo: there's no search yet. But every later stage relies on this code being exactly right.

Further reading

- Chess Programming Wiki: Move Generation, Pin, Classical Approach, Perft, Perft Results, Chess960 Perft Results.

Magic bitboards

Stage 3 · tag `stage-03-magics` *· code in* `crates/chess/src/magic.rs`

Stage 2 found a rook's attacks by walking each ray square by square. That's correct but slow: up to 14 steps, each with a branch the CPU can't predict well. This chapter replaces it with a table lookup that gives the same answer in four instructions.

The idea

A rook on d4 attacks along four rays. Which squares it attacks depends only on which squares **on those rays** are occupied, and not even all of those:

1. **Mask the relevant squares.** The last square of each ray never matters: the rook attacks it whether it's empty or not. For a rook on d4 that leaves 10 relevant squares, so there are only $2^{10} = 1024$ occupancy patterns to care about. `occupied & mask` keeps those bits.
2. **Hash them into an index.** Multiply by a carefully chosen 64-bit *magic number* and keep the top bits: `((occupied & mask) * magic) >> (64 - 10)`. The multiplication gathers the scattered occupancy bits into the top of the word.
3. **Look up the answer** in a table precomputed for every one of the 1024 patterns.

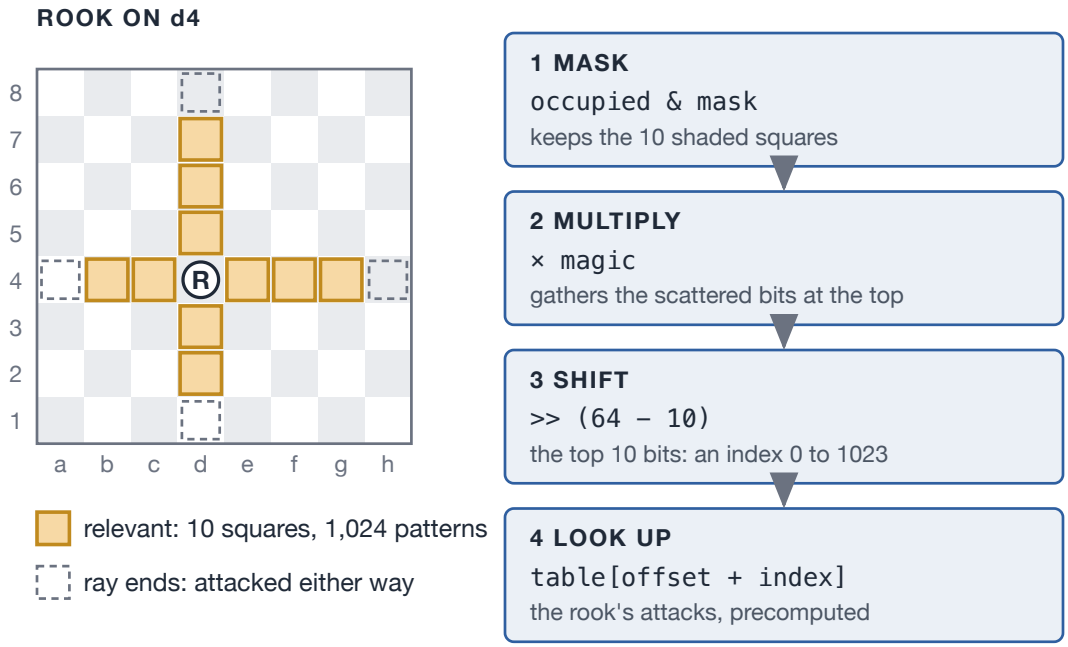

Figure 3.1 A rook on d4 depends on only 10 squares, so its attacks are one of 1,024 table entries

```
/// The magic data for one square and one kind of slider.
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub struct Magic {
    /// The relevant squares: the slider's rays, without the edge squares.
    pub mask: Bitboard,
    /// The magic multiplier.
    pub magic: u64,
    /// `64 - (number of relevant squares)`.
    pub shift: u32,
    /// Where this square's part of the shared attack table starts.
    pub offset: usize,
}

impl Magic {
    /// The attack-table index for the given occupied squares.
    pub fn index(&self, occupied: Bitboard) -> usize {
        let relevant = (occupied & self.mask).0;
        self.offset + (relevant.wrapping_mul(self.magic) >> self.shift) as usize
    }
}
```

```
/// The squares a rook on `square` attacks, given the occupied squares.
pub fn rook_attacks(square: Square, occupied: Bitboard) -> Bitboard {
    let tables = tables();
    tables.attacks[tables.rook[square.index()].index(occupied)]
}

/// The squares a bishop on `square` attacks, given the occupied squares.
pub fn bishop_attacks(square: Square, occupied: Bitboard) -> Bitboard {
    let tables = tables();
    tables.attacks[tables.bishop[square.index()].index(occupied)]
}
```

What makes a number "magic"

Two patterns that need *different* attack sets must never land on the same index. Two that need the *same* attack set may share one; that's a harmless, "constructive" collision. Nobody knows a formula for such numbers, so we search: try random 64-bit numbers with few bits set until one works for the square.

```
/// Searches for a magic number for `square`: random sparse candidates until one maps
every
/// relevant occupancy to an index without a harmful collision.
pub fn find_magic(square: Square, reference: Reference, rng: &mut Rng) -> u64 {
    let mask = relevant_mask(square, reference(square, Bitboard::EMPTY));
    let occupancies = occupancies(square, mask, reference);
    let mut table = vec![None; occupancies.len()];
    loop {
        let magic = rng.sparse();
        // Quick filter: a good magic moves enough mask bits into the top byte.
        if (mask.0.wrapping_mul(magic) >> 56).count_ones() < 6 {
            continue;
        }
        let candidate = Magic {
            mask,
            magic,
            shift: 64 - mask.count(),
            offset: 0,
        };
        if fill_table(&candidate, &occupancies, &mut table) {
            return magic;
        }
    }
}
```

The search takes about half a second for all 128 square/piece pairs. That's too slow to run every time the engine starts, because GUIs

expect an instant reply. So the `find-magics` tool runs the search once and writes the numbers to `magic/magic_numbers.rs`:

```
cargo run --release -p cb2026-tools --bin find-magics
```

At startup cb2026 just fills in the tables for those numbers, about 4 ms, and checks each number still works.

"Fancy" tables

Each square needs a table of `2^(relevant squares)` entries: 4096 for a rook in a corner, but only 1024 in the centre. Instead of 64 fixed-size tables, all of them share one array and each square remembers its *offset*. Rooks and bishops together need 107,648 entries, about 840 KB.

Testing it

Magic tables are easy to get subtly wrong, so the test is exhaustive. For every square, for every one of its relevant occupancy patterns (plus some irrelevant pieces that mustn't change anything), the magic lookup must equal stage 2's ray walk. That is 107,648 comparisons, and it runs in under a second. The perft suites from stage 2 still pass at every depth.

What it's worth

Perft speed on the full test suites (666 million positions), cloud machine, release build:

Sliding attacks	Speed
Walking rays (stage 2)	133 million positions/s
Magic bitboards	235 million positions/s

That's about 1.8× faster, and every later stage gets it for free, since search spends much of its time generating moves and testing attacks. Still no Elo yet: there is nothing to search with until stage 5.

Try it

```
> magic d4                    # step through the lookup for a rook and bishop on d4
> set sliders rays
> perft 5                     # note the speed
> set sliders magics
> perft 5                     # and compare
```

Going further

- **PEXT bitboards.** CPUs with the BMI2 instruction set have `pext`, which does step 2's gathering of bits in a single instruction, with no magic numbers needed. It's slow on AMD Zen 1 and 2, so it would be an optional build. The PRD lists it as a could-have.
- **Smaller tables.** Better magic numbers can use fewer index bits than the number of relevant squares; dedicated searches have found many such "black magics".

Further reading

- Chess Programming Wiki: Magic Bitboards, Looking for Magics (Tord Romstad's public-domain search, which `find_magic` follows), BMI2 / PEXT.

Zobrist hashing and talking to GUIs (UCI)

Stage 4 · tag `stage-04-uci` *· code in* `crates/chess`, `crates/engine`, `crates/protocol`

By the end of this stage cb2026 is a real, if hopeless, chess engine. A GUI can load it, send it positions and get a legal move back: for now a random one. Three pieces make that possible. **Hash keys** let us recognise positions, **algebraic notation** lets people read moves, and the **UCI protocol** lets GUIs talk to the engine.

Zobrist hashing

The search will meet the same position again and again, reached by different move orders, and the repetition rule needs to spot recurring positions. Comparing whole positions is slow, so each position gets a 64-bit fingerprint.

Zobrist hashing gives every feature a fixed random 64-bit number:

- one per (piece, square);
- one for "Black to move";
- one per castling right;
- one per en passant file.

The key is the XOR of the numbers for the features present:

```
/// The key for `piece` standing on `square`.
pub fn piece_square(piece: Piece, square: Square) -> u64 {
    let piece_index = piece.color.index() * 6 + piece.piece_type.index();
    KEYS[piece_index * 64 + square.index()]
}

/// The key XORed in when Black is to move.
pub fn black_to_move() -> u64 {
    KEYS[SIDE_INDEX]
}

/// The combined key for a set of castling rights.
pub fn castling(rights: CastlingRights) -> u64 {
    let mut key = 0;
    for color in Color::ALL {
        for side in CastlingSide::ALL {
            if rights.has(color, side) {
                key ^= KEYS[CASTLING_INDEX + color.index() * 2 + side.index()];
            }
        }
    }
    key
}

/// The key for an en passant square on `file`.
pub fn en_passant(file: File) -> u64 {
    KEYS[EN_PASSANT_INDEX + file.index()]
}
```

Because `x ^ k ^ k == x`, the key updates incrementally. Moving a knight from g1 to f3 is `key ^= N_g1 ^ N_f3`, a couple of instructions. cb2026 updates the key inside the same functions that move pieces, so it can never fall out of step. A test walks the move tree from every perft position and checks, at every node, that the incremental key equals one computed from scratch.

Different positions *can* share a key, but with 64 random bits it's rare enough that engines accept the risk. The transposition table in stage 8 also double-checks the moves it gets back.

The shell's `zobrist` command lists every part of the current key.

Algebraic notation

UCI speaks in `g1f3`, but people write `Nf3`. **Standard algebraic notation (SAN)** names the piece and destination:

- captures add `x` (`Bxe5`);
- promotions add `=Q`;
- castling is `O-O` or `O-O-O`;
- check and mate add `+` or `#`.

When two pieces of the same kind could reach the square, SAN adds the starting file (`Nbd7`), else the rank (`R1e2`), else both.

```
/// Writes `mv`, which must be legal here, in SAN.
///
/// ```
/// use cb2026_chess::Position;
///
/// let pos = Position::startpos();
/// let mv = pos.parse_uci_move("g1f3").unwrap();
/// assert_eq!(pos.to_san(mv), "Nf3");
/// ```
pub fn to_san(&self, mv: Move) -> String {
    let mut san = self.san_without_check(mv);
    let mut after = self.clone();
    after.make_move(mv);
    if after.in_check() {
        san.push(if after.legal_moves().is_empty() {
            '#'
        } else {
            '+'
        });
    }
    san
}
```

Reading SAN is deliberately forgiving: annotations are ignored, `0-0` works, and so do `e8Q` and over-specified moves like `Ng1f3`. The test is strict, though: every legal move in every perft position must survive writing and reading back.

The Engine API

The engine crate does no I/O, so the protocols and the shell drive it through one call:

```
/// Chooses a move in `position`.
///
/// `history` holds the hash keys of the earlier positions in the game. `stop` may be
set by
/// another thread to end the search early. `on_info` is called after each completed
depth,
/// and once more for a depth cut short if it found a move worth playing.
///
/// The search uses **iterative deepening**: it searches to depth 1, then 2, then 3,
and so
/// on, until a limit says stop. Each depth starts with the best move of the one
before,
/// and when time runs out mid-depth the engine still has a complete answer to fall
back on.
pub fn search(
    &mut self,
    position: &Position,
    history: &[u64],
    limits: &Limits,
    stop: &AtomicBool,
    on_info: &mut dyn FnMut(&SearchInfo),
) -> SearchResult {
    let start = Instant::now();
    let mut root_moves: Vec<Move> = position
        .legal_moves()
        .iter()
        .copied()
        .filter(|mv| limits.search_moves.is_empty() ||
limits.search_moves.contains(mv))
        .collect();
    if root_moves.is_empty() {
        return SearchResult::default();
    }

    // The opening book: play its move at once (unless asked to analyse).
    if self.own_book
        && !limits.infinite
        && limits.search_moves.is_empty()
        && let Some(book) = &self.book
    {
        // The dice: the clock's nanoseconds, mixed with the position.
        let nanos = std::time::SystemTime::now()
            .duration_since(std::time::UNIX_EPOCH)
            .map_or(0, |d| d.subsec_nanos());
        let random = u64::from(nanos) ^ position.hash();
        if let Some(mv) = book.pick(position, random)
            && root_moves.contains(&mv)
        {
            return SearchResult {
                best_move: Some(mv),
                ..SearchResult::default()
            };
        }
    }
    if self.techniques.mvv_lva {
        ordering::order_moves(position, &mut root_moves);
```

```
    }
    self.tt.set_buckets(self.techniques.tt_buckets);
    // The previous search often left a best move for this position.
    self.tt.new_search();
    self.ordering.new_search();
    if self.techniques.tt
        && let Some(mv) = self.tt.probe(position.hash()).and_then(|e| e.best_move)
        && let Some(at) = root_moves.iter().position(|&m| m == mv)
    {
        root_moves[..=at].rotate_right(1);
    }

    // In a tablebase position, keep only the move the tables recommend: it keeps the
best
    // result and makes progress under the fifty-move rule. The search still runs (and
    // reports), but can only choose that move.
    if let Some(tb) = &self.tablebase
        && tablebase::covers(tb.as_ref(), &self.tablebase_params, position)
        && let Some((_, mv)) = tb.probe_root(position)
        && root_moves.contains(&mv)
    {
        root_moves = vec![mv];
    }

    // A weaker level searches fewer nodes and several lines to choose from.
    let level = levels::settings_at(self.level());
    let mut level_limits = limits.clone();
    if let Some(nodes) = level.nodes {
        level_limits.nodes = Some(limits.nodes.map_or(nodes, |n| n.min(nodes)));
    }
    // Now and then, only as far ahead as a person would look (see `levels`), on one
    // thread. Never when analysing.
    let horizon = (!limits.infinite)
        .then(|| levels::horizon(&level, dice(position)))
        .flatten();
    if let Some(plies) = horizon {
        level_limits.depth = Some(plies);
    }
    let limits = &level_limits;
    let lines = self.multipv.max(level.candidates).max(1);
    // The levels were calibrated with the hand-crafted evaluation (chapter 19), so
below
    // full strength the search keeps it: their ratings stay true. Calibrating them
again
    // with a network is for later.
    let full_strength = self.level() >= f64::from(levels::FULL_STRENGTH);

    let budget = time::budget(limits, position.side_to_move(), &self.time);
    let unlimited = limits.depth.is_none()
        && limits.nodes.is_none()
        && limits.mate.is_none()
        && budget.hard.is_none()
        && !limits.infinite;
    let mut max_depth = limits
        .depth
        .unwrap_or(if unlimited { DEFAULT_DEPTH } else { MAX_PLY });
    if let Some(moves) = limits.mate {
```

```
        // A mate in N moves is 2N - 1 plies away.
        max_depth = max_depth.min(2 * moves);
    }
    let max_depth = max_depth.clamp(1, MAX_PLY);

    let deadline = budget.hard.map(|hard| start + hard);
    let root_fallback = root_moves.first().copied();
    // Lazy SMP (stage 17): helper threads search the same position, sharing only the
    // transposition table. They stop when the main thread is done.
    let helpers_stop = AtomicBool::new(false);
    let helper_nodes = AtomicU64::new(0);
    let Engine {
        params,
        techniques,
        search_params,
        tt,
        ordering,
        correction,
        threads,
        tablebase,
        tablebase_params,
        multipv,
        time,
        network,
        ..
    } = self;
    let tablebase = tablebase.as_deref();
    // The network, if the search is to evaluate with one.
    let network = network
        .as_deref()
        .filter(|_| techniques.nnue && full_strength);
    let self_multipv = (*multipv).max(1);
    let tablebase_params = *tablebase_params;
    let techniques = match horizon {
        Some(_) => Techniques::horizon(),
        None => *techniques,
    };
    let threads = if horizon.is_some() { 1 } else { *threads };
    let (params, search_params, tt) = (&*params, *search_params, &*tt);
    thread::scope(|scope| {
        let mut helper_threads = Vec::new();
        for helper in 1..threads {
            let root_moves = root_moves.clone();
            let (helpers_stop, helper_nodes) = (&helpers_stop, &helper_nodes);
            helper_threads.push(scope.spawn(move || {
                let mut ordering = OrderingTables::default();
                let mut correction = CorrectionHistory::default();
                let mut searcher = Searcher::new(
                    params,
                    search_params,
                    tt,
                    &mut ordering,
                    techniques,
                    StopConditions {
                        stop: helpers_stop,
                        deadline,
                        max_nodes: None,
```

```
                },
                history,
            );
            searcher.count_nodes_in(helper_nodes);
            searcher.use_correction(&mut correction);
            if let Some(tb) = tablebase {
                searcher.use_tablebase(tb, tablebase_params);
            }
            if let Some(net) = network {
                searcher.use_network(net);
            }
            run_helper(&mut searcher, position, root_moves, max_depth, helper);
            searcher.stats()
        }));
    }
    let mut searcher = Searcher::new(
        params,
        search_params,
        tt,
        ordering,
        techniques,
        StopConditions {
            stop,
            deadline,
            max_nodes: limits.nodes,
        },
        history,
    );
    searcher.use_correction(correction);
    if let Some(tb) = tablebase {
        searcher.use_tablebase(tb, tablebase_params);
    }
    if let Some(net) = network {
        searcher.use_network(net);
    }
    let result = main_search(
        &mut searcher,
        MainSearch {
            position,
            root_moves,
            max_depth,
            start,
            budget,
            time,
            helper_nodes: &helper_nodes,
            lines,
            reported: self_multipv,
        },
        on_info,
    );
    helpers_stop.store(true, Ordering::Relaxed);
    let mut stats = searcher.stats();
    for helper in helper_threads {
        if let Ok(helper_stats) = helper.join() {
            stats.add(&helper_stats);
        }
    }
```

```
        let Some(best) = result.first() else {
            return SearchResult {
                best_move: root_fallback,
                ponder_move: None,
                stats,
            };
        };
        // At full strength, the best line; at a lower level, one of the candidates.
        let candidates: Vec<(Move, i32)> = result
            .iter()
            .filter_map(|line| line.best_move.map(|mv| (mv, line.score)))
            .collect();
        let chosen = levels::choose(&candidates, &level,
dice(position)).or(best.best_move);
        let line = result
            .iter()
            .find(|line| line.best_move == chosen)
            .unwrap_or(best);
        let best_move = chosen.or(root_fallback);
        // The expected reply is the line's second move. A line cut short (by a
        // transposition table cutoff just below the root) may not have one; then the
        // table's best move for the position after ours is the next best guess.
        let ponder_move = line.pv.get(1).copied().or_else(|| {
            let mut after = position.clone();
            after.make_move(best_move?);
            tt.probe(after.hash())
                .and_then(|entry| entry.best_move)
                .filter(|mv| after.legal_moves().contains(mv))
        });
        SearchResult {
            best_move,
            ponder_move,
            stats,
        }
    })
}
```

- **`Limits`** says when to stop: a depth, a node count, a fixed time, the clock, or `infinite`.
- The **stop flag** is an `AtomicBool` that another thread can set at any moment.
- The **info callback** receives progress: depth, score, nodes and the best line. UCI prints it as `info` lines; the shell will show it as a live display.
- **`history`** carries the hash keys of earlier positions for repetition detection.

For now the "search" picks a random legal move. That's enough to test the whole path from a GUI and back; stage 5 replaces it with alpha-beta.

UCI

GUIs start the engine as a separate program and talk to it in lines of text on standard input and output. A session:

```
> uci
< id name cb2026 1.0.0
< id author the cb2026 authors
< option name UCI_Chess960 type check default false
< uciok
> isready
< readyok
> position startpos moves e2e4 e7e5
> go wtime 60000 btime 60000 winc 1000 binc 1000
< info depth 1 seldepth 1 score cp 0 nodes 29 time 0 pv g1f3
< bestmove g1f3
```

The GUI owns the game: before every search it sends the whole position, then `go` with the clocks. Two details matter for robustness:

- **The search runs on its own thread.** The command loop keeps reading, so `isready` is answered at once and `stop` takes effect immediately, even mid-search.
- **`go infinite` waits for `stop`** before printing `bestmove`, as the protocol requires, even if the engine has already decided.

Bad input is reported (`info string ...`) and never crashes the engine: an invalid FEN, an illegal move in `position`, an unknown command or option. The protocol tests in `crates/protocol/src/uci.rs` play scripted sessions covering all of this. That includes Chess960 castling notation (`e1h1`, "king takes rook") and Windows line endings.

One binary, three modes. GUIs send `uci` as their first line, so `cb2026` reads one line to decide whether to speak UCI, XBoard (stage 20) or open the shell.

Try it

In the shell:

```
> play black          # cb2026 plays Black
> e4                  # you move; it replies (randomly, for now)
> play both           # watch it play itself to the end of the game
```

In a GUI such as En Croissant, add `target/release/cb2026` as a UCI engine and play it.

Or by hand:

```
$ cargo run --release
uci
position startpos moves e2e4
go
```

What it's worth

A random mover is the zero point of the Elo scale. Every stage from here on is measured against the one before.

Further reading

- Chess Programming Wiki: Zobrist Hashing, Algebraic Chess Notation, UCI.
- The UCI protocol description by Stefan Meyer-Kahlen (distributed with most UCI GUIs and engines).

Material, negamax and alpha-beta

Stage 5 · tag `stage-05-alphabeta` *· code in* `crates/engine`

This is where cb2026 starts to play chess. It looks a few moves ahead, counts material at the end of each line, and picks the move whose worst case is best. That's enough to win material, avoid losing it, and find mates. Against the stage 4 random mover it won 285 of 286 games.

Evaluation: counting material

The search can't see to the end of the game, so at the edge of its tree it needs a number that says how good a position looks. The simplest useful one counts material in centipawns (hundredths of a pawn), from the point of view of the side to move:

```
/// The evaluation's tunable numbers, and switches for its term groups.
#[derive(Clone, Debug, PartialEq, Eq)]
pub struct EvalParams {
    /// Middlegame value of each piece type in centipawns, indexed by
[`PieceType::index`]
    /// (the king's value is never used: both sides always have one).
    pub mg_values: [i32; 6],
    /// Endgame value of each piece type.
    pub eg_values: [i32; 6],
    /// Middlegame piece-square tables, indexed by [`PieceType::index`].
    pub mg_pst: [Pst; 6],
    /// Endgame piece-square tables.
    pub eg_pst: [Pst; 6],
    /// How much each piece type counts toward the game phase.
    pub phase_weights: [i32; 6],
    /// Use the piece-square tables (off: material only).
    pub use_pst: bool,
    /// Blend middlegame and endgame values by phase (off: middlegame values only).
    pub tapered: bool,
    /// The positional terms of the full evaluation (stage 15).
    pub terms: TermParams,
}
```

```
/// Reports every term of the evaluation, for both sides, to `sink`.
pub fn collect<S: TermSink>(pos: &Position, params: &EvalParams, sink: &mut S) {
    for color in Color::ALL {
        for piece_type in PieceType::ALL {
            for square in pos.pieces(color, piece_type) {
                sink.term(color, Param::Material(piece_type), 1);
                if params.use_pst {
                    sink.term(color, Param::Pst(piece_type, pst_index(color, square)),
1);
                }
            }
        }
    }
    hce::collect(pos, &params.terms, sink);
}

/// The game phase: from 0 (bare kings and pawns) to [`MAX_PHASE`] (all the pieces).
pub fn phase(pos: &Position, params: &EvalParams) -> i32 {
    let phase: i32 = PieceType::ALL
        .iter()
        .map(|&pt| params.phase_weights[pt.index()] * pos.pieces_of_type(pt).count()
as i32)
        .sum();
    phase.min(MAX_PHASE)
}

/// Adds up the terms' values into an [`EvalTrace`].
struct TraceSink<'a> {
    params: &'a EvalParams,
    trace: EvalTrace,
}

impl TermSink for TraceSink<'_> {
    fn term(&mut self, side: Color, param: Param, count: i32) {
        let t = &mut self.trace;
        let slot = match param {
            Param::Material(_) => &mut t.material,
            Param::Pst(..) => &mut t.pst,
            Param::Mobility(_) => &mut t.mobility,
            Param::Doubled | Param::Isolated => &mut t.pawns,
            Param::Passed(_) => &mut t.passed,
            Param::BishopPair | Param::RookOpenFile | Param::RookSemiOpenFile => &mut
t.pieces,
            Param::PawnShield => &mut t.king_safety,
            Param::Tempo => &mut t.tempo,
        };
        let (mg, eg) = self.params.value(param);
        slot[side.index()].0 += mg * count;
        slot[side.index()].1 += eg * count;
    }

    fn king_danger(&mut self, side: Color, penalty: i32) {
        self.trace.king_safety[side.index()].0 -= penalty;
    }
}
```

```
/// Works out the evaluation's parts; the total is from White's point of view.
pub fn trace(pos: &Position, params: &EvalParams) -> EvalTrace {
    let mut sink = TraceSink {
        params,
        trace: EvalTrace::default(),
    };
    collect(pos, params, &mut sink);
    let mut t = sink.trace;
    t.phase = phase(pos, params);
    let (mg, eg) = t.net();
    t.total = if params.tapered {
        (mg * t.phase + eg * (MAX_PHASE - t.phase)) / MAX_PHASE
    } else {
        mg
    };
    t
}

/// The position's value in centipawns, from the side to move's point of view.
pub fn evaluate(pos: &Position, params: &EvalParams) -> i32 {
    let total = trace(pos, params).total;
    match pos.side_to_move() {
        Color::White => total,
        Color::Black => -total,
    }
}
```

The values are named parameters, not constants buried in code. Stage 16 tunes all the evaluation's numbers automatically, and that only works if they live in one place.

Minimax and negamax

To pick a move, imagine every move, every reply, every reply to that, down to some depth. Evaluate the positions at the bottom, then work back up assuming each side picks what's best for it: the side to move takes the maximum, its opponent the minimum. That's **minimax**.

Because chess scores are symmetric (+3 for White is −3 for Black), the two cases fold into one: a position's score is the best, over all moves, of *minus* the score after the move. That's **negamax**, and it's what the code below implements.

Alpha-beta

Most of the minimax tree can't change the answer. Suppose White's first move turns out to be worth +1. For the second move, the moment Black finds one reply that makes it worse than +1, White's second move is refuted: White would never play it, so Black's other replies needn't be examined.

Alpha-beta formalises this with a window `[alpha, beta]`:

- `alpha` is what the side to move can already guarantee elsewhere;
- `beta` is the most its opponent will allow.

As soon as a move scores `≥ beta`, the opponent won't let this position happen. The remaining moves are skipped: a **beta cutoff**.

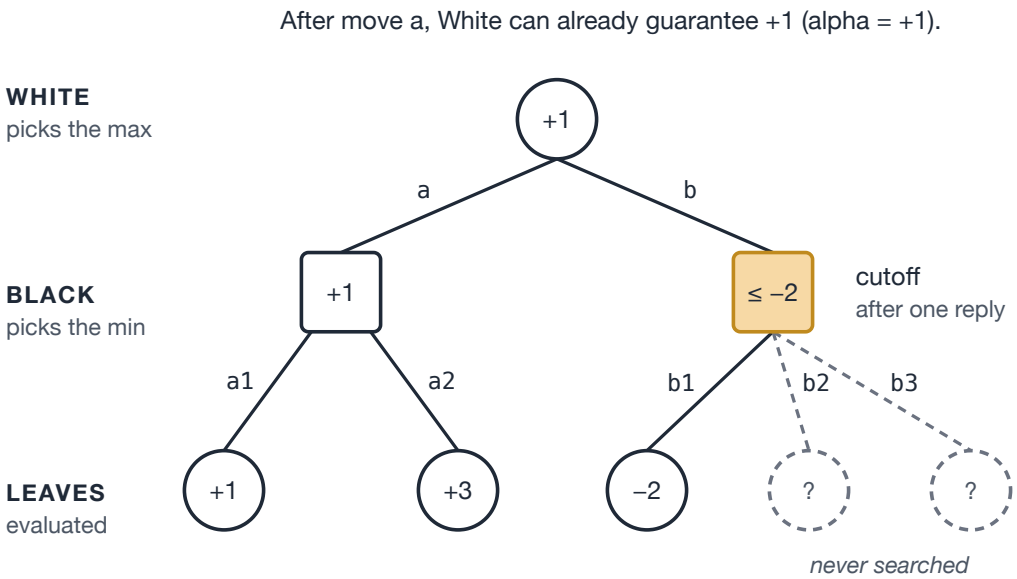

Figure 5.1 Once one reply refutes a move, alpha-beta skips that move's other replies

```
/// The negamax score of `pos` searched `depth` plies deep, `ply` plies from the root,
/// within the window `[alpha, beta]`. The best line found is written to `pv`.
`cut_node`
/// says whether the node is expected to fail high (an expected cut node; see
/// [`Searcher::search_move`]); a node with a full window never is.
// The search's state is in `self`; these are the node's own.
#[allow(clippy::too_many_arguments)]
fn negamax(
    &mut self,
    pos: &Position,
    depth: u32,
    ply: u32,
    mut alpha: i32,
    beta: i32,
    cut_node: bool,
    pv: &mut Vec<Move>,
) -> i32 {
    self.nodes += 1;
    self.note(Note::Enter { depth, alpha, beta });
    self.seldepth = self.seldepth.max(ply);
    pv.clear();
    if self.should_stop() {
        return 0;
    }

    // Set while a singular extension's search leaves this node's hash move out.
    let excluded = self.excluded[ply as usize];
    let moves = pos.legal_moves();
    if moves.is_empty() {
        // Checkmate (worse the sooner it happens) or stalemate.
        return if pos.in_check() {
            -MATE + ply as i32
        } else {
            0
        };
    }
    if self.is_draw(pos) {
        return 0;
    }
    if ply >= MAX_PLY {
        return self.evaluate(pos, ply);
    }
    if depth == 0 {
        return if self.techniques.quiescence {
            self.quiescence(pos, ply, alpha, beta, pv)
        } else {
            self.evaluate(pos, ply)
        };
    }

    // The fail highs two plies down are counted afresh under each node: the next
ply's
    // count then covers this node's children and its siblings'.
    self.cutoffs[ply as usize + 2] = 0;

    // Transposition table: a deep enough result may settle this position at once, and
```

```
in
    // any case its best move is searched first.
    // (Not in a singular extension's search: the entry is about the node with every
move.)
    let mut tt_move = None;
    let mut tt_entry = None;
    if self.techniques.tt
        && excluded.is_none()
        && let Some(entry) = self.tt.probe(pos.hash())
    {
        self.note(Note::TtHit);
        tt_entry = Some(entry);
        tt_move = entry.best_move.filter(|mv| moves.contains(mv));
        if u32::from(entry.depth) >= depth {
            let score = score_from_tt(entry.score, ply);
            let usable = match entry.bound {
                Bound::Exact => true,
                Bound::Lower => score >= beta,
                Bound::Upper => score <= alpha,
            };
            if usable {
                self.note(Note::TtCutoff);
                pv.extend(tt_move);
                return score;
            }
        }
    }

    let pv_node = beta - alpha > 1;
    // Is or has been on the principal variation: a PV node now, or one the table
remembers
    // as one. A singular extension's search, of the same node, keeps the node's flag.
    if excluded.is_none() {
        self.tt_pv[ply as usize] = pv_node || tt_entry.is_some_and(|entry| entry.pv);
    }
    let tt_pv = self.tt_pv[ply as usize];

    // Internal iterative reduction: with no hash move to try first, move ordering is
poor
    // and the node probably isn't important; search it a ply shallower.
    let depth = if self.techniques.iir
        && tt_move.is_none()
        && excluded.is_none()
        && depth >= self.search_params.iir_min_depth
    {
        depth - 1
    } else {
        depth
    };

    // Endgame tablebases: right after a capture or pawn move, a covered position's
result
    // is known exactly.
    if let Some((tb, tb_params)) = self.tablebase
        && ply > 0
        && depth >= tb_params.probe_depth
        && pos.halfmove_clock() == 0
```

```
        && tablebase::covers(tb, &tb_params, pos)
        && let Some(wdl) = tb.probe_wdl(pos)
    {
        self.tb_hits += 1;
        self.note(Note::Tablebase);
        let score = tablebase::score(wdl, ply);
        if self.techniques.tt {
            let stored = score_to_tt(score, ply);
            self.tt
                .store(pos.hash(), None, stored, MAX_PLY, Bound::Exact, tt_pv);
        }
        return score;
    }

    let p = self.search_params;
    let in_check = pos.in_check();
    // The static evaluation, for the pruning decisions (meaningless in check),
corrected by
    // what the search has learnt about positions with these pawns.
    let raw_eval = if in_check {
        -INFINITY
    } else {
        self.evaluate(pos, ply)
    };
    let static_eval = if in_check {
        -INFINITY
    } else {
        self.corrected_eval(pos, raw_eval)
    };
    // Improving: better than two plies ago, when this side last moved. Its evaluation
will
    // probably keep rising, which changes the bets the pruning techniques make.
    self.evals[ply as usize] = static_eval;
    let improving = self.techniques.improving
        && !in_check
        && ply >= 2
        && static_eval > self.evals[ply as usize - 2];
    // Selective pruning happens only off the principal variation and out of check.
    let selective = self.techniques.alpha_beta && !pv_node && !in_check;

    // Reverse futility pruning: far enough above beta, assume a search would stay
there.
    // An improving side gets a smaller margin, but never none at all.
    let rfp_margin = p.rfp_margin * depth as i32 - if improving { p.rfp_improving }
else { 0 };
    if self.techniques.reverse_futility
        && selective
        && depth <= p.rfp_max_depth
        && static_eval - rfp_margin >= beta
    {
        self.note(Note::ReverseFutility);
        return static_eval;
    }

    // Razoring: far below alpha near the leaves, only captures could save the node.
    if self.techniques.razoring
        && self.techniques.quiescence
```

```
        && selective
        && depth <= p.razor_max_depth
        && static_eval + p.razor_margin * (depth as i32) < alpha
    {
        let score = self.quiescence(pos, ply, alpha, alpha + 1, pv);
        if score <= alpha {
            self.note(Note::Razoring);
            return score;
        }
        pv.clear();
    }

    // Null-move pruning: if passing still keeps us at or above beta, a real move
would too.
    if self.techniques.null_move
        && selective
        && excluded.is_none()
        && depth >= p.null_min_depth
        && self.path.last() != Some(&None)
        && has_pieces(pos)
        && static_eval >= beta
    {
        self.note(Note::NullMove);
        let reduction = p.null_reduction + depth / p.null_depth_divisor;
        let mut child = pos.clone();
        child.make_null_move();
        self.push_move(pos, None);
        self.update_accumulator(pos, None, ply);
        self.keys.push(pos.hash());
        let mut null_pv = Vec::new();
        let score = -self.negamax(
            &child,
            depth.saturating_sub(1 + reduction),
            ply + 1,
            -beta,
            -beta + 1,
            !cut_node,
            &mut null_pv,
        );
        self.note(Note::Exit { score: -score });
        self.keys.pop();
        self.pop_move();
        if self.aborted {
            return 0;
        }
        if score >= beta {
            self.note(Note::NullMoveCutoff);
            // A mate found after passing isn't a real mate: claim no more than beta.
            return if score >= MATE - MAX_PLY as i32 {
                beta
            } else {
                score
            };
        }
    }

    // ProbCut: if a good capture beats beta by a margin even in a shallower search,
```

```
the
    // full-depth search would almost surely beat beta too, so the node is cut off.
Not if
    // the hash table already says that a search about as deep fell short of the
margin.
    let probcut_beta = beta + p.probcut_margin;
    if self.techniques.probcut
        && self.techniques.quiescence
        && selective
        && excluded.is_none()
        && depth >= p.probcut_min_depth
        && beta.abs() + p.probcut_margin < MATE - MAX_PLY as i32
        && !tt_entry.is_some_and(|entry| {
            entry.bound != Bound::Lower
                && u32::from(entry.depth) + p.probcut_reduction > depth
                && score_from_tt(entry.score, ply) < probcut_beta
        })
    {
        self.note(Note::ProbCut);
        let probcut_depth = depth.saturating_sub(p.probcut_reduction);
        let captures: Vec<Move> = moves.iter().copied().filter(|&mv|
is_noisy(mv)).collect();
        let previous = self.previous();
        let picker = MovePicker::new(
            pos,
            &captures,
            tt_move,
            ply,
            &previous,
            self.ordering,
            &self.techniques,
        );
        let mut cutoff = None;
        let mut probcut_pv = Vec::new();
        self.keys.push(pos.hash());
        for mv in picker {
            // Only captures that win, by static exchange, at least what the static
            // evaluation is short of the margin.
            if see(pos, mv) < probcut_beta - static_eval {
                continue;
            }
            let mut child = pos.clone();
            child.make_move(mv);
            self.push_move(pos, Some(mv));
            self.update_accumulator(pos, Some(mv), ply);
            // The quiescence search first: it is cheap, and if even it can't reach
the
            // margin, the shallower search seldom does.
            let mut score = -self.quiescence(
                &child,
                ply + 1,
                -probcut_beta,
                -probcut_beta + 1,
                &mut probcut_pv,
            );
            if score >= probcut_beta && probcut_depth > 0 && !self.aborted {
                score = self.search_child(
```

```
                    &child,
                    probcut_depth,
                    probcut_beta - 1,
                    probcut_beta,
                    !cut_node,
                    &mut probcut_pv,
                );
            }
            self.pop_move();
            if self.aborted {
                break;
            }
            if score >= probcut_beta {
                cutoff = Some((mv, score));
                break;
            }
        }
        self.keys.pop();
        if self.aborted {
            return 0;
        }
        if let Some((mv, score)) = cutoff {
            self.note(Note::ProbCutCutoff(mv));
            // What ProbCut found is a lower bound from its shallower search.
            if self.techniques.tt {
                let stored = score_to_tt(score, ply);
                self.tt.store(
                    pos.hash(),
                    Some(mv),
                    stored,
                    probcut_depth + 1,
                    Bound::Lower,
                    tt_pv,
                );
            }
            return score;
        }
    }

    // Singular extension: is the hash move much better than every other move? Search
the
    // node again, at half the depth and without the hash move, against a window a
little
    // below the hash move's score. If no other move reaches it, the hash move is
probably
    // forced, and it is searched a ply deeper.
    let mut singular_move = None;
    let mut double_move = None;
    let mut shallower_move = None;
    if self.techniques.singular_extension
        && depth >= p.singular_min_depth
        && ply < 2 * self.root_depth
        && let Some(entry) = tt_entry
        && let Some(hash_move) = tt_move
        && entry.bound != Bound::Upper
        && u32::from(entry.depth) + p.singular_tt_depth_margin >= depth
    {
```

```
        let tt_score = score_from_tt(entry.score, ply);
        if tt_score.abs() < MATE - MAX_PLY as i32 {
            let singular_beta = tt_score - p.singular_margin * depth as i32;
            self.note(Note::SingularSearch(hash_move));
            self.excluded[ply as usize] = Some(hash_move);
            let mut singular_pv = Vec::new();
            let score = self.negamax(
                pos,
                (depth - 1) / 2,
                ply,
                singular_beta - 1,
                singular_beta,
                cut_node,
                &mut singular_pv,
            );
            self.excluded[ply as usize] = None;
            self.note(Note::Exit { score });
            if self.aborted {
                return 0;
            }
            if score < singular_beta {
                self.note(Note::SingularExtension(hash_move));
                singular_move = Some(hash_move);
                // Double extension: nothing else comes close even well below
                // `singular_beta`, so the hash move is searched a second ply deeper.
Only
                // off the principal variation, and only a few times along a line, so
that
                // the search can't explode.
                if self.techniques.double_extension
                    && !pv_node
                    && score < singular_beta - p.double_extension_margin
                    && self.double_extensions < p.double_extension_limit
                {
                    self.note(Note::DoubleExtension(hash_move));
                    double_move = Some(hash_move);
                }
            } else if self.techniques.multi_cut && singular_beta >= beta {
                // Multi-cut: the hash move beat beta, and another move beats it too
even
                // at half the depth. With two moves that good, the node is a cutoff.
                self.note(Note::MultiCut);
                return singular_beta;
            } else if self.techniques.negative_extension && tt_score >= beta {
                // Negative extension: not singular, but its hash score beats beta.
The
                // node will probably cut off either way, so its line needs less care.
                self.note(Note::NegativeExtension(hash_move));
                shallower_move = Some(hash_move);
            }
        }
    }

    // In a singular extension's search, the hash move is left out. If it was the only
    // move, nothing else comes close: fail low.
    let without_excluded: Vec<Move>;
    let moves: &[Move] = match excluded {
```

```
        Some(left_out) => {
            without_excluded = moves.iter().copied().filter(|&mv| mv !=
left_out).collect();
            &without_excluded
        }
        None => &moves,
    };
    if moves.is_empty() {
        return alpha;
    }

    let previous = self.previous();
    let picker = MovePicker::new(
        pos,
        moves,
        tt_move,
        ply,
        &previous,
        self.ordering,
        &self.techniques,
    );
    let original_alpha = alpha;
    let mut best = -INFINITY;
    let mut best_move = None;
    let mut child_pv = Vec::new();
    // Quiet moves searched so far that didn't cause a cutoff (for the history
penalty),
    // and the same for captures and queen promotions (for capture history's).
    let mut quiets_tried = Vec::new();
    let mut noisy_tried = Vec::new();
    // Moves actually searched (not pruned) so far.
    let mut searched = 0;
    self.keys.push(pos.hash());
    for (index, mv) in picker.enumerate() {
        let quiet = !is_noisy(mv);
        let mut child = pos.clone();
        child.make_move(mv);
        let gives_check = child.in_check();
        let ordinary = quiet && !in_check && !gives_check;

        // Late move pruning: near the leaves, after enough quiet moves, skip the rest
        // (once a move has shown we aren't simply getting mated).
        if self.techniques.late_move_pruning
            && ordinary
            && !pv_node
            && depth <= p.lmp_max_depth
            && best > -MATE + MAX_PLY as i32
            && quiets_tried.len() as u32
                >= (p.lmp_base + depth * depth) * if improving { p.lmp_improving }
else { 1 }
        {
            self.note(Note::LateMovePruning(mv));
            continue;
        }

        // History pruning: near the leaves, skip a quiet move whose history says it
keeps
```

```
        // failing where other moves cut off, by more the deeper the node. As in
Stockfish
        // (GPL-3.0), which scores the move by continuation history alone.
        if self.techniques.history_pruning
            && ordinary
            && !pv_node
            && depth <= p.history_pruning_max_depth
            && best > -MATE + MAX_PLY as i32
            && self
                .ordering
                .quiet_score(pos, mv, &previous, &self.techniques)
                < -p.history_pruning_margin * depth as i32
        {
            self.note(Note::HistoryPruning(mv));
            continue;
        }

        // Futility pruning: a quiet move can't lift a hopeless static evaluation to
alpha.
        if self.techniques.futility
            && ordinary
            && !pv_node
            && depth <= p.futility_max_depth
            && best > -MATE + MAX_PLY as i32
            && static_eval + p.futility_base + p.futility_per_depth * depth as i32 <=
alpha
        {
            self.note(Note::Futility(mv));
            continue;
        }

        // SEE pruning: near the leaves, skip moves that lose material outright.
        if self.techniques.see_pruning
            && selective
            && index > 0
            && !gives_check
            && depth <= p.see_max_depth
            && best > -MATE + MAX_PLY as i32
        {
            let margin = if quiet {
                p.see_quiet_margin
            } else {
                p.see_capture_margin
            };
            if see(pos, mv) < -margin * (depth as i32) {
                self.note(Note::SeePruning(mv));
                continue;
            }
        }

        let how = if index == 0 {
            MoveSearch::First
        } else if self.techniques.lmr
            && ordinary
            && depth >= p.lmr_min_depth
            && index as u32 >= p.lmr_min_moves
        {
```

```
            let row = &self.lmr[(depth as usize).min(LMR_TABLE_SIZE - 1)];
            let mut reduction = i32::from(row[index.min(LMR_TABLE_SIZE - 1)]);
            // Reduce the principal variation less: its exact score matters.
            if pv_node {
                reduction -= 1;
            }
            // Reduce more at an expected cut node: one move should refute it, and the
late
            // ones seldom matter.
            if self.techniques.cut_node_reductions && cut_node {
                reduction += p.lmr_cut_node;
            }
            // Reduce more when many moves at this ply have been refuted: this node is
            // probably failing low, and its late moves along with it.
            if self.techniques.cutoff_reductions
                && self.cutoffs[ply as usize + 1] > p.lmr_cutoff_count
            {
                reduction += p.lmr_cutoff_plies;
                // More at an expected all-node, as in Stockfish 17.
                if !pv_node && !cut_node {
                    reduction += p.lmr_cutoff_all_node_plies;
                }
            }
            // Reduce more when the hash move is a capture: it is probably the best
move,
            // and a quiet move this late seldom beats it. Less (by default not at
all) on
            // the principal variation, whose exact score matters.
            if self.techniques.tt_capture_reductions && tt_move.is_some_and(is_noisy)
{
                reduction += if pv_node {
                    p.lmr_tt_capture_pv
                } else {
                    p.lmr_tt_capture
                };
            }
            // Reduce less at an expected all-node: every move has to fail low for its
            // score to hold, so a late move that is wrongly reduced into failing low
            // makes the parent's move a refutation it isn't.
            if self.techniques.all_node_reductions && !pv_node && !cut_node {
                reduction -= p.lmr_all_node;
            }
            // Reduce less at a node that has been on the principal variation: it may
be
            // again.
            if self.techniques.tt_pv && tt_pv && !pv_node {
                reduction -= p.lmr_tt_pv;
            }
            // Reduce a move with a good history less: it has caused cutoffs
elsewhere. And
            // one with a bad history more.
            if self.techniques.history_lmr {
                let score = self
                    .ordering
                    .quiet_score(pos, mv, &previous, &self.techniques);
                reduction -= p.lmr_history_adjustment(score);
            }
```

```
            MoveSearch::Later {
                // Never reduce into the quiescence search.
                reduction: reduction.clamp(0, depth as i32 - 2) as u32,
            }
        } else {
            MoveSearch::Later { reduction: 0 }
        };

        // Check extension: a move that gives check is searched one ply deeper. So is
a
        // singular hash move (above), and one singular by a wide margin a ply more.
        let doubled = Some(mv) == double_move;
        let extension = u32::from(
            (self.techniques.check_extension && gives_check && ply < MAX_PLY / 2)
                || Some(mv) == singular_move,
        ) + u32::from(doubled);
        // A hash move that isn't singular but beats beta is searched shallower
(above).
        let shallower = if Some(mv) == shallower_move {
            p.negative_extension_plies
        } else {
            0
        };

        searched += 1;
        self.push_move(pos, Some(mv));
        self.update_accumulator(pos, Some(mv), ply);
        let child_depth = (depth - 1 + extension).saturating_sub(shallower);
        self.double_extensions += u32::from(doubled);
        let score = self.search_move(
            &child,
            child_depth,
            alpha,
            beta,
            how,
            cut_node,
            &mut child_pv,
        );
        self.double_extensions -= u32::from(doubled);
        self.pop_move();
        if self.aborted {
            break;
        }
        if score > best {
            best = score;
            best_move = Some(mv);
            if score > alpha {
                alpha = score;
                pv.clear();
                pv.push(mv);
                pv.extend_from_slice(&child_pv);
            }
        }
        // Beta cutoff: the opponent won't allow this position, so stop looking.
        if self.techniques.alpha_beta && alpha >= beta {
            self.note(Note::BetaCutoff(mv, searched));
            // Counted for the node above: once more without a hash move, and less for
```

```
a
            // doubly extended move, as in Stockfish 17.
            let counts = if doubled {
                p.cutoff_count_double_extended
            } else {
                1
            };
            self.cutoffs[ply as usize] += counts + u32::from(tt_move.is_none());
            if quiet {
                self.ordering
                    .record_cutoff(pos, mv, &quiets_tried, ply, depth, &previous);
            }
            // The captures tried before it failed, whichever kind of move cut off.
            let noisy_best = (!quiet).then_some(mv);
            self.ordering
                .record_captures(pos, noisy_best, &noisy_tried, depth);
            break;
        }
        if quiet {
            quiets_tried.push(mv);
        } else {
            noisy_tried.push(mv);
        }
    }
    self.keys.pop();

    let bound = if best >= beta {
        Bound::Lower
    } else if best > original_alpha {
        Bound::Exact
    } else {
        Bound::Upper
    };
    // After a fail low under a node that is or has been on the principal variation,
the
    // opponent's last move was probably good, so this position may be on the next
one.
    if bound == Bound::Upper && ply > 0 && excluded.is_none() && depth >
p.tt_pv_fail_low_depth
    {
        self.tt_pv[ply as usize] |= self.tt_pv[ply as usize - 1];
    }
    let tt_pv = self.tt_pv[ply as usize];
    // (Not in a singular extension's search: its result leaves the best move out.)
    if self.techniques.tt && !self.aborted && excluded.is_none() {
        // After a fail low every move was refuted, so none of them is known to be
best.
        let move_to_store = if bound == Bound::Upper {
            None
        } else {
            best_move
        };
        let score = score_to_tt(best, ply);
        self.tt
            .store(pos.hash(), move_to_store, score, depth, bound, tt_pv);
    }
```

```
    // Correction history learns how far the static evaluation was off: not in check
(there
    // is none), not in a singular extension's search (its score leaves the best move
out),
    // and not when the best move is a capture or promotion (its gain is material the
static
    // evaluation isn't meant to see).
    let settings = self.correction_settings();
    if self.techniques.correction_history
        && !self.aborted
        && !in_check
        && excluded.is_none()
        && !best_move.is_some_and(is_noisy)
        && let Some(correction) = self.correction.as_deref_mut()
    {
        correction.update(pos, depth, raw_eval, best, bound, &settings);
    }
    best
}
```

Alpha-beta returns **exactly** the minimax result. It only skips work that can't change the answer. With perfect move ordering it searches about `b^(d/2)` positions instead of `b^d`, which means twice the depth for the same effort.

Try it in the shell on Kiwipete:

```
> fen r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq - 0 1
> set alphabeta off
> go depth 4
cb2026 plays Bxa6   (depth 4, +0.10, 4185552 nodes, 868 ms; line: Bxa6 hxg2 Qxg2 bxc3)
> undo
> set alphabeta on
> go depth 4
cb2026 plays Bxa6   (depth 4, +0.10, 131847 nodes, 39 ms; line: Bxa6 hxg2 Qxg2 bxc3)
```

It plays the same move with the same score, from 32 times fewer positions, and that's with moves in whatever order the generator produced them. Stage 10 orders moves well and gets much closer to the ideal.

Mate and draws

- **Checkmate** scores `-MATE + ply` for the side that is mated. Being mated sooner is worse, so the winner prefers the quickest mate and the loser the slowest. The score translates back to "mate in N" for GUIs.
- **Stalemate, the fifty-move rule, insufficient material and repetition** score 0. Inside the search a *single* repetition counts as a draw: if a position can come back once, it can come back again, so there's no point searching it twice.

Stopping

Stage 5 searches to a fixed depth: four plies unless told otherwise. It also stops early when any of these says so:

- the GUI's `stop`;
- a node limit;
- a time limit, about 1/30th of the remaining clock plus most of the increment.

When stopped early it plays the best move found so far. Stage 6 replaces this with iterative deepening and proper time management.

Bench

`cb2026 bench` searches 14 fixed positions to a fixed depth (4 at this stage, 10 from stage 14 on) and prints the node count:

```
$ cb2026 bench
Total time (ms) : 108
Nodes searched  : 556884
Nodes/second    : 5113843
```

Because the search is deterministic, that number is a fingerprint of the search. Every commit message ends with `Bench: <nodes>`, and a change meant to leave the search alone must leave it unchanged.

What it's worth

Test	Result
Stage 5 vs stage 4 (random mover), 8+0.08, SPRT [0, 5]	+285 −0 =1 over 286 games: passed

The Elo figure from such a lopsided match is meaningless (about +1100 ± 740), but the point stands: lookahead plus material counting beats random play every time. From now on each stage is measured against the one before.

Further reading

- Chess Programming Wiki: Minimax, Negamax, Alpha-Beta, Checkmate, Point Value.
- Donald Knuth and Ronald Moore, "An Analysis of Alpha-Beta Pruning", *Artificial Intelligence* 6 (1975): the classic proof of how much alpha-beta saves.

Iterative deepening and time management

Stage 6 · tag `stage-06-iterative` *· code in* `crates/engine/src/lib.rs` *and* `time.rs`

Stage 5 searched to a fixed depth. That's fine for a puzzle, but in a real game the engine has a clock, and it can't know in advance how long depth 7 will take in a given position. Too shallow wastes time it could have used, and too deep can lose on time. This stage fixes that with two ideas found in every modern engine:

- **iterative deepening:** search depth 1, then 2, then 3, and so on, until the time is up;
- **time management:** decide, from the clock, how long that is.

Iterative deepening

Searching depth 1, then depth 2, then depth 3 sounds wasteful, since each search repeats all the work of the one before. In practice it costs little. Each depth takes several times longer than the previous one (the *effective branching factor*: between 5 and 15 for cb2026 right now), so all the shallower searches together add only a fraction of the last one.

In return:

- **There's always an answer.** When time runs out in the middle of depth 8, the engine still has the complete result of depth 7 to fall back on.
- **Each depth helps the next.** The best move from depth *d* is searched first at depth *d + 1*. Alpha-beta prunes best when the best move comes first (chapter 5), so this ordering pays for much of the repeated work. Stages 8 and 10 add a transposition table and better move ordering, which use the shallower searches even more, until iterative deepening is *faster* than going straight to the final depth.
- **GUIs see progress.** After each depth the engine reports what it found, which is what scrolls by in a GUI's analysis pane.

```
/// Chooses a move in `position`.
///
/// `history` holds the hash keys of the earlier positions in the game. `stop` may be
set by
/// another thread to end the search early. `on_info` is called after each completed
depth,
/// and once more for a depth cut short if it found a move worth playing.
///
/// The search uses **iterative deepening**: it searches to depth 1, then 2, then 3,
and so
/// on, until a limit says stop. Each depth starts with the best move of the one
before,
/// and when time runs out mid-depth the engine still has a complete answer to fall
back on.
pub fn search(
    &mut self,
    position: &Position,
    history: &[u64],
    limits: &Limits,
    stop: &AtomicBool,
    on_info: &mut dyn FnMut(&SearchInfo),
) -> SearchResult {
    let start = Instant::now();
    let mut root_moves: Vec<Move> = position
        .legal_moves()
        .iter()
        .copied()
        .filter(|mv| limits.search_moves.is_empty() ||
limits.search_moves.contains(mv))
        .collect();
    if root_moves.is_empty() {
        return SearchResult::default();
    }

    // The opening book: play its move at once (unless asked to analyse).
    if self.own_book
        && !limits.infinite
        && limits.search_moves.is_empty()
        && let Some(book) = &self.book
    {
        // The dice: the clock's nanoseconds, mixed with the position.
        let nanos = std::time::SystemTime::now()
            .duration_since(std::time::UNIX_EPOCH)
            .map_or(0, |d| d.subsec_nanos());
        let random = u64::from(nanos) ^ position.hash();
        if let Some(mv) = book.pick(position, random)
            && root_moves.contains(&mv)
        {
            return SearchResult {
                best_move: Some(mv),
                ..SearchResult::default()
            };
        }
    }
    if self.techniques.mvv_lva {
        ordering::order_moves(position, &mut root_moves);
```

```
    }
    self.tt.set_buckets(self.techniques.tt_buckets);
    // The previous search often left a best move for this position.
    self.tt.new_search();
    self.ordering.new_search();
    if self.techniques.tt
        && let Some(mv) = self.tt.probe(position.hash()).and_then(|e| e.best_move)
        && let Some(at) = root_moves.iter().position(|&m| m == mv)
    {
        root_moves[..=at].rotate_right(1);
    }

    // In a tablebase position, keep only the move the tables recommend: it keeps the
best
    // result and makes progress under the fifty-move rule. The search still runs (and
    // reports), but can only choose that move.
    if let Some(tb) = &self.tablebase
        && tablebase::covers(tb.as_ref(), &self.tablebase_params, position)
        && let Some((_, mv)) = tb.probe_root(position)
        && root_moves.contains(&mv)
    {
        root_moves = vec![mv];
    }

    // A weaker level searches fewer nodes and several lines to choose from.
    let level = levels::settings_at(self.level());
    let mut level_limits = limits.clone();
    if let Some(nodes) = level.nodes {
        level_limits.nodes = Some(limits.nodes.map_or(nodes, |n| n.min(nodes)));
    }
    // Now and then, only as far ahead as a person would look (see `levels`), on one
    // thread. Never when analysing.
    let horizon = (!limits.infinite)
        .then(|| levels::horizon(&level, dice(position)))
        .flatten();
    if let Some(plies) = horizon {
        level_limits.depth = Some(plies);
    }
    let limits = &level_limits;
    let lines = self.multipv.max(level.candidates).max(1);
    // The levels were calibrated with the hand-crafted evaluation (chapter 19), so
below
    // full strength the search keeps it: their ratings stay true. Calibrating them
again
    // with a network is for later.
    let full_strength = self.level() >= f64::from(levels::FULL_STRENGTH);

    let budget = time::budget(limits, position.side_to_move(), &self.time);
    let unlimited = limits.depth.is_none()
        && limits.nodes.is_none()
        && limits.mate.is_none()
        && budget.hard.is_none()
        && !limits.infinite;
    let mut max_depth = limits
        .depth
        .unwrap_or(if unlimited { DEFAULT_DEPTH } else { MAX_PLY });
    if let Some(moves) = limits.mate {
```

```
        // A mate in N moves is 2N - 1 plies away.
        max_depth = max_depth.min(2 * moves);
    }
    let max_depth = max_depth.clamp(1, MAX_PLY);

    let deadline = budget.hard.map(|hard| start + hard);
    let root_fallback = root_moves.first().copied();
    // Lazy SMP (stage 17): helper threads search the same position, sharing only the
    // transposition table. They stop when the main thread is done.
    let helpers_stop = AtomicBool::new(false);
    let helper_nodes = AtomicU64::new(0);
    let Engine {
        params,
        techniques,
        search_params,
        tt,
        ordering,
        correction,
        threads,
        tablebase,
        tablebase_params,
        multipv,
        time,
        network,
        ..
    } = self;
    let tablebase = tablebase.as_deref();
    // The network, if the search is to evaluate with one.
    let network = network
        .as_deref()
        .filter(|_| techniques.nnue && full_strength);
    let self_multipv = (*multipv).max(1);
    let tablebase_params = *tablebase_params;
    let techniques = match horizon {
        Some(_) => Techniques::horizon(),
        None => *techniques,
    };
    let threads = if horizon.is_some() { 1 } else { *threads };
    let (params, search_params, tt) = (&*params, *search_params, &*tt);
    thread::scope(|scope| {
        let mut helper_threads = Vec::new();
        for helper in 1..threads {
            let root_moves = root_moves.clone();
            let (helpers_stop, helper_nodes) = (&helpers_stop, &helper_nodes);
            helper_threads.push(scope.spawn(move || {
                let mut ordering = OrderingTables::default();
                let mut correction = CorrectionHistory::default();
                let mut searcher = Searcher::new(
                    params,
                    search_params,
                    tt,
                    &mut ordering,
                    techniques,
                    StopConditions {
                        stop: helpers_stop,
                        deadline,
                        max_nodes: None,
```

```
                },
                history,
            );
            searcher.count_nodes_in(helper_nodes);
            searcher.use_correction(&mut correction);
            if let Some(tb) = tablebase {
                searcher.use_tablebase(tb, tablebase_params);
            }
            if let Some(net) = network {
                searcher.use_network(net);
            }
            run_helper(&mut searcher, position, root_moves, max_depth, helper);
            searcher.stats()
        }));
    }
    let mut searcher = Searcher::new(
        params,
        search_params,
        tt,
        ordering,
        techniques,
        StopConditions {
            stop,
            deadline,
            max_nodes: limits.nodes,
        },
        history,
    );
    searcher.use_correction(correction);
    if let Some(tb) = tablebase {
        searcher.use_tablebase(tb, tablebase_params);
    }
    if let Some(net) = network {
        searcher.use_network(net);
    }
    let result = main_search(
        &mut searcher,
        MainSearch {
            position,
            root_moves,
            max_depth,
            start,
            budget,
            time,
            helper_nodes: &helper_nodes,
            lines,
            reported: self_multipv,
        },
        on_info,
    );
    helpers_stop.store(true, Ordering::Relaxed);
    let mut stats = searcher.stats();
    for helper in helper_threads {
        if let Ok(helper_stats) = helper.join() {
            stats.add(&helper_stats);
        }
    }
```

```
        let Some(best) = result.first() else {
            return SearchResult {
                best_move: root_fallback,
                ponder_move: None,
                stats,
            };
        };
        // At full strength, the best line; at a lower level, one of the candidates.
        let candidates: Vec<(Move, i32)> = result
            .iter()
            .filter_map(|line| line.best_move.map(|mv| (mv, line.score)))
            .collect();
        let chosen = levels::choose(&candidates, &level,
dice(position)).or(best.best_move);
        let line = result
            .iter()
            .find(|line| line.best_move == chosen)
            .unwrap_or(best);
        let best_move = chosen.or(root_fallback);
        // The expected reply is the line's second move. A line cut short (by a
        // transposition table cutoff just below the root) may not have one; then the
        // table's best move for the position after ours is the next best guess.
        let ponder_move = line.pv.get(1).copied().or_else(|| {
            let mut after = position.clone();
            after.make_move(best_move?);
            tt.probe(after.hash())
                .and_then(|entry| entry.best_move)
                .filter(|mv| after.legal_moves().contains(mv))
        });
        SearchResult {
            best_move,
            ponder_move,
            stats,
        }
    })
}
```

When the search is stopped in the middle of a depth, the moves already searched at that depth are still useful. The first one is last depth's best move, searched deeper, so if any move finished, the engine plays the best of those. If none finished, it plays the best move from the previous depth.

The loop also ends early when it finds a forced mate: a mate within d plies found by a full-width search can't be improved by searching deeper.

Time management

With a clock, the engine has to share its remaining time among all the moves still to play. cb2026 sets two limits per move:

- **Soft limit:** after finishing a depth, don't start the next one if this much time has gone. The next depth would probably take several times longer, so starting it late mostly wastes time.
- **Hard limit:** stop at once, even mid-depth. It is the safety net for a depth that takes far longer than expected.

```
/// The time manager's tunable numbers.
#[derive(Clone, Debug, PartialEq, Eq)]
pub struct TimeParams {
    /// When the time control doesn't say, plan as if this many moves remain.
    pub moves_to_go: u32,
    /// Percentage of the increment added to each move's budget.
    pub increment_percent: u32,
    /// The hard limit is this many times the soft limit.
    pub hard_factor: u32,
    /// Time held back for communication delays (the UCI option `Move Overhead`).
    pub move_overhead: Duration,
    /// Scale the soft limit by how long the best move has stayed the same (`set
tmstability`).
    pub best_move_stability: bool,
    /// The soft limit's scale, in percent, when the best move has stayed the same for
0, 1, 2,
    /// 3, and 4 or more depths in a row.
    pub stability_percent: [u32; 5],
    /// Think longer when the score falls from one depth to the next (`set
tmscore`)...
    pub falling_score: bool,
    /// ...by this many percent per centipawn it fell...
    pub falling_score_percent_per_cp: u32,
    /// ...counting at most this many centipawns.
    pub falling_score_max_cp: u32,
}

impl Default for TimeParams {
    fn default() -> TimeParams {
        TimeParams {
            moves_to_go: 30,
            increment_percent: 75,
            hard_factor: 3,
            move_overhead: Duration::from_millis(30),
            best_move_stability: true,
            stability_percent: [180, 140, 110, 90, 75],
            falling_score: true,
            falling_score_percent_per_cp: 1,
            falling_score_max_cp: 100,
        }
    }
}
```

(The fields after `move_overhead` came later, in chapter 6.1.)

```
/// The time budget for a search with `limits`, with `side` to move.
pub fn budget(limits: &Limits, side: Color, params: &TimeParams) -> TimeBudget {
    if let Some(movetime) = limits.movetime {
        let usable = movetime.saturating_sub(params.move_overhead);
        return TimeBudget {
            soft: Some(usable),
            hard: Some(usable),
            adaptive: false,
        };
    }
    let Some(remaining) = limits.time[side.index()] else {
        return TimeBudget::default();
    };
    let remaining = remaining.saturating_sub(params.move_overhead);
    let increment = limits.increment[side.index()].unwrap_or(Duration::ZERO);
    let moves_left = limits
        .moves_to_go
        .unwrap_or(params.moves_to_go)
        .clamp(1, params.moves_to_go);
    let ceiling = remaining / 2;
    let soft = remaining / moves_left + increment * params.increment_percent / 100;
    TimeBudget {
        soft: Some(soft.min(ceiling)),
        hard: Some((soft * params.hard_factor).min(ceiling)),
        adaptive: true,
    }
}
```

For example, with 60 seconds left and a 1-second increment:

- the soft limit is (60 s − 30 ms) / 30 + 0.75 s = 2.75 s;
- the hard limit is three times that, 8.25 s.

Neither limit may exceed half the remaining time, so a short clock with a large increment can't flag. `Move Overhead` (a UCI option, 30 ms by default) is held back for the time the GUI and operating system take to pass messages. Raise it if cb2026 ever loses on time in a GUI or on a server.

Here's what that looks like in UCI with `go wtime 60000 btime 60000 winc 1000 binc 1000` from the starting position:

```
info depth 5 seldepth 5 score cp 100 nodes 38434 time 5 pv b2b3 b8a6 c1b2 a6b4 b2g7
info depth 6 seldepth 6 score cp -100 nodes 289817 time 40 pv b2b3 b7b6 b1a3 c8a6 g1f3
a6e2
info depth 7 seldepth 7 score cp 100 nodes 1522091 time 194 pv b2b3 b8a6 b1a3 a6b4
a3b5 b4d5 b5a7
info depth 8 seldepth 8 score cp -100 nodes 21698796 time 3096 pv b2b3 b8c6 b1a3 g8f6
g1f3 c6b4 a3b1 b4a2
bestmove b2b3 ponder b8c6
```

Depth 7 finished at 0.19 s, well under the 2.75 s soft limit, so depth 8 started. It finished at 3.1 s, past the soft limit, so the engine stopped there instead of starting depth 9, which would have taken far longer. The scores also flip between +1 and −1 from one depth to the next. That is the **horizon effect**: the search stops in the middle of an exchange and counts a piece that is about to be recaptured. Stage 7's quiescence search fixes it.

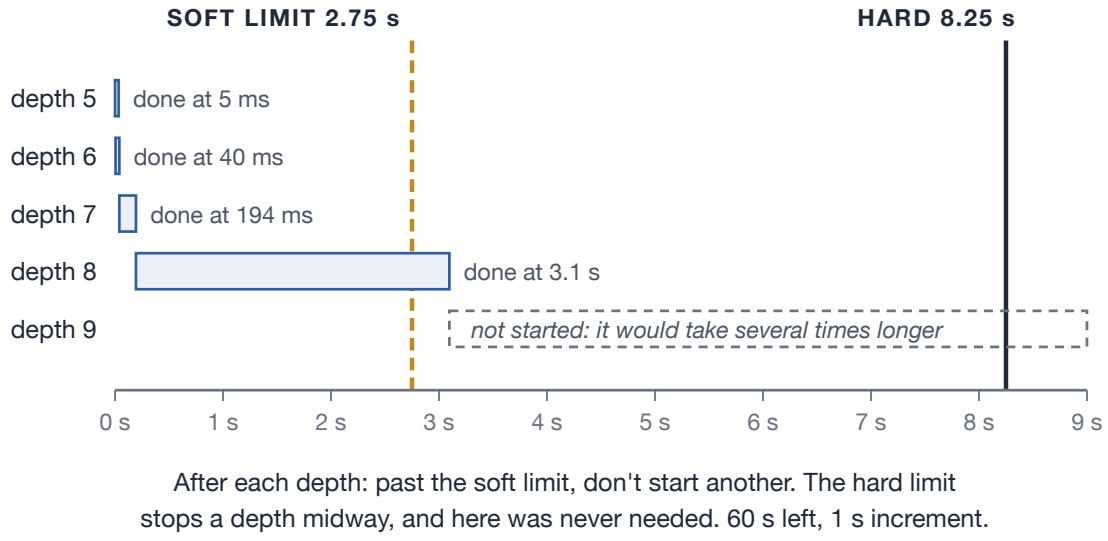

Figure 6.1 Depth 8 finished past the soft limit, so depth 9 was never started

In the shell

The shell now thinks for one second per move by default and shows each depth as it goes:

```
> fen r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq - 0 1
> go
  depth  1    +3.30          48 nodes       0 ms  Bxa6
  depth  2    +0.10         202 nodes       0 ms  Bxa6 bxc3
  depth  3    +3.30        7466 nodes       1 ms  Bxa6 bxc3 Qxf6
  depth  4    +0.10       29664 nodes       5 ms  Bxa6 hxg2 Qxg2 bxc3
  depth  5    +2.30     2105269 nodes     401 ms  Bxa6 hxg2 Rg1 bxc3 Qxf6
  depth  6    -0.90     4734976 nodes     970 ms  Bxa6 bxc3 bxc3 exd5 Nd3 Nxe4
cb2026 plays Bxa6   (depth 6, -0.90, 4734976 nodes, 970 ms)
```

Depth 6 didn't finish within the second. The line shown for it covers only the moves searched before time ran out, starting with Bxa6.

- `set movetime MS` sets the thinking time;
- `set depth N` switches back to a fixed depth;
- `think off` hides the per-depth lines.

Pitfalls

- **Checking the clock too often is slow.** Reading the time costs far more than visiting a node, so the search checks it every 1024 nodes.
- **Trusting an unfinished depth too much.** Its result is only reliable for the moves it finished. That's why the previous best move is always searched first, and why an iteration that finished no move is thrown away.
- **Forgetting the overhead.** An engine that plans to use exactly its remaining time loses on time, because the GUI measures from when it sent `go` to when it received `bestmove`.

What it's worth

Test	Result
Stage 6 vs stage 5, 8+0.08, SPRT [0, 5]	+33 −0 =8 over 41 games: passed (about +390 ± 140 Elo)

Most of the gain comes from using the clock well: stage 5 always searched to depth 4, which took a few milliseconds in quiet positions and far too long in sharp ones.

Further reading

- Chess Programming Wiki: Iterative Deepening, Time Management, Horizon Effect.
- David Slate and Larry Atkin, "CHESS 4.5 – The Northwestern University Chess Program", in *Chess Skill in Man and Machine* (1977): where iterative deepening became standard.

Spending time where it's needed

Milestone M5, after stage 20 · on `main`, no stage tag · code in `crates/engine/src/time.rs` and `lib.rs`

Chapter 6's time manager gives every move the same budget: a share of the clock, plus most of the increment. But moves aren't alike. In some, the best move is obvious after a few milliseconds and every deeper search agrees. In others, the search changes its mind from one depth to the next, or its score keeps falling as it sees further. A strong player spends little time on the first kind and a lot on the second, and so can an engine: the search itself shows which kind of move it is facing.

So after each depth, the soft limit (when to stop starting new depths) is scaled by two signals. Each has its own switch: `set tmstability` and `set tmscore`.

Best-move stability

If the best move has stayed the same for several depths in a row, one more depth is unlikely to change it. If it has just changed, the search hasn't settled, and the next depth may change it again. The scale, by how many depths in a row the best move has stayed the same:

Depths unchanged	0 (just changed)	1	2	3	4 or more
Soft limit	180%	140%	110%	90%	75%

A falling score

If the score dropped since the last depth, the search has found a problem, a threat or a line that doesn't work, and the next depth may find the answer to it. The soft limit grows by 1% per centipawn the score fell, up to twice as long (100 centipawns). A rising score changes nothing.

The code

Both scales multiply, and the result is capped at the hard limit:

```
/// The soft limit after a depth, adapted to how the search is going: `stable` is how
many
/// depths in a row the best move has stayed the same, `fall` how many centipawns the
score
/// dropped since the last depth (negative if it rose). Never past the hard limit.
pub fn adapted_soft(
    budget: &TimeBudget,
    stable: u32,
    fall: i32,
    params: &TimeParams,
) -> Option<Duration> {
    let soft = budget.soft?;
    if !budget.adaptive {
        return Some(soft);
    }
    let mut percent = 100;
    if params.best_move_stability {
        let row = (stable as usize).min(params.stability_percent.len() - 1);
        percent = percent * params.stability_percent[row] / 100;
    }
    if params.falling_score && fall > 0 {
        let fall = (fall as u32).min(params.falling_score_max_cp);
        percent = percent * (100 + fall * params.falling_score_percent_per_cp) / 100;
    }
    let adapted = soft * percent / 100;
    Some(budget.hard.map_or(adapted, |hard| adapted.min(hard)))
}
```

The iterative deepening loop keeps count of how long the best move has been stable and compares each depth's score with the last one's:

```
// Starting another depth now would probably not finish in time. How soon "now" is
// depends on how settled the search is (see `time::adapted_soft`).
let (best_move, score) = (lines[0].best_move, lines[0].score);
let fall = match before {
    Some((previous_move, previous_score)) => {
        stable = if previous_move == best_move {
            stable + 1
        } else {
            0
        };
        previous_score - score
    }
    None => 0,
};
let soft = time::adapted_soft(&budget, stable, fall, time);
if soft.is_some_and(|soft| start.elapsed() >= soft) {
    break;
}
```

The hard limit doesn't move, so the clock is exactly as safe as before. Time saved on easy moves isn't lost either: each move's budget is a share of what remains, so the saving is spent on later moves. Two cases are left alone. With a fixed `movetime` the GUI asked for that time exactly. And after a `ponderhit`, the engine already got its thinking time on the opponent's clock, and it stops after a fixed share of its own.

What it changes

`ab` can't measure this: it searches to a fixed depth, and `bench` does too, so its node count doesn't change. In a game it shows in how the time is spread. One self-play game of 60 moves each at 8+0.08, before and after, measured twice: on the engine of chapter 14.1 before continuation history, and again with continuation history and singular extensions (each pair is two different games, so the positions differ):

	Mean per move	Median	Slowest 10% from	Longest
Before (fixed soft limit)	210 ms	146 ms	527 ms	857 ms
After	210 ms	169 ms	433 ms	648 ms
Before, with singular extensions	210 ms	121 ms	467 ms	916 ms
After, with singular extensions	210 ms	156 ms	459 ms	851 ms

The total is the same, as it must be: both use their clock. The typical think gets longer and the longest ones a little shorter. One game is a small sample, and the two pairs differ in how much. The games say it plays better, by about **33 Elo** at 8+0.08 and about 14 at 40+0.4 (see "What it's worth"): spending the same time in different places is worth as much as singular extensions at the short time control.

Pitfalls

- **Never past the hard limit.** Scaling the soft limit up is safe only because the hard limit still stops the search mid-depth. Scaling the hard limit too would risk the clock.
- **Scores wobble.** Aspiration windows, and the odd/even effect (a search that ends on the side to move's ply tends to score a little higher than one that ends on the opponent's), move the score by a few centipawns from one depth to the next with nothing wrong. At 1% per centipawn that costs a few percent of time; a steeper scale would chase noise.
- **Respect what the GUI asked for.** `go movetime 500` means 500 ms. A time manager that stops early "because the move is

obvious" breaks it.

- **The numbers are a starting point.** The scale's values are reasonable, not tuned. Like the rest of the time manager, they are named parameters, and an SPRT at one time control says little about another.

What it's worth

Test	Result
Adaptive soft limit vs `main` without it (with singular extensions), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**+33.5 ± 14.3 Elo** over 1,062 games (+299 −197 =566): passed (LLR 2.97), no time losses
Experiment: the adaptive soft limit off vs `main` with it (with history in LMR's second version), 40+0.4, test Mac (M3 Max), SPRT [−5, 0]	**−13.8 ± 8.6 Elo** over 2,599 games (+482 −585 =1532): failed (LLR −2.97), no time losses; it stays on

The gain was first measured at 8+0.08. At longer time controls each depth takes longer and the trade-offs shift, so a second test switched both scales off at 40+0.4 and asked, as a simplification with bounds [−5, 0], whether that loses (cjus/chessengine#76). It does: the gain holds at the long time control, though it is smaller, about 14 Elo against 33. An SPRT stops as soon as it is sure, so either estimate could be some way off, but both say it gains. The scale's numbers were set by hand, so tuning them, at the long time control, is a candidate for later.

Further reading

- Chess Programming Wiki: Time Management.
- Robert Hyatt, "Using Time Wisely", *ICCA Journal* 7 (1984).

Quiescence search and MVV-LVA

Stage 7 · tag `stage-07-quiescence` *· code in* `crates/engine/src/search.rs` *and* `ordering.rs`

Chapter 6 ended with a warning sign. From the starting position the stage 6 engine's score flipped between +1 and −1 from one depth to the next, as if a pawn appeared and disappeared with every ply. Nothing of the sort happens in the real game. The search simply stopped in the middle of exchanges. This chapter fixes that, and the fix is one of the biggest single gains an engine gets.

The horizon effect

A fixed-depth search stops at its horizon and evaluates whatever position is there, even if a piece is hanging. Take this position: White has a queen, Black two pawns.

```
> fen 4k3/8/4p3/3p4/8/8/8/3QK3 w - - 0 1
> set depth 1
> set quiescence off
> go
  depth  1    +8.00          18 nodes       0 ms  Qxd5
cb2026 plays Qxd5   (depth 1, +8.00, 18 nodes, 0 ms)
```

At depth 1 the search sees Qxd5, counts material (queen against one pawn, +8), and stops, without seeing that exd5 wins the queen. Searching deeper doesn't cure this. It only moves the horizon: at any depth, the last move can be a capture whose recapture lies beyond it.

Quiescence search

The fix is to keep searching past the horizon, but only moves that change the material balance: captures and queen promotions. Captures run out quickly, so this ends. The search goes on until the position is **quiet** (in Latin *quiescens*), and only then is it evaluated.

```
/// The quiescence score of `pos`, `ply` plies from the root, within `[alpha, beta]`:
/// captures (and queen promotions) only, until the position is quiet. In check, every
/// evasion is searched instead.
fn quiescence(
    &mut self,
    pos: &Position,
    ply: u32,
    mut alpha: i32,
    beta: i32,
    pv: &mut Vec<Move>,
) -> i32 {
    self.nodes += 1;
    self.stats.count(Note::Quiescence);
    self.seldepth = self.seldepth.max(ply);
    pv.clear();
    if self.should_stop() {
        return 0;
    }

    // In check every evasion counts; otherwise only captures and promotions are
    // generated. (So a stalemate reached by captures isn't noticed: a rare error that
all
    // engines accept for the speed.)
    let in_check = pos.in_check();
    let moves = if in_check {
        let evasions = pos.legal_moves();
        if evasions.is_empty() {
            return -MATE + ply as i32;
        }
        evasions
    } else {
        pos.noisy_moves()
    };
    if ply >= MAX_PLY {
        return self.evaluate(pos, ply);
    }

    let mut best = -INFINITY;
    if !in_check {
        // Stand pat: not capturing is an option, so the static score is a lower
bound. It
        // is corrected, as in the main search.
        best = self.corrected_eval(pos, self.evaluate(pos, ply));
        if self.techniques.alpha_beta && best >= beta {
            return best;
        }
        alpha = alpha.max(best);
    }

    // Under-promotions are generated but almost never matter: skip them.
    let moves: Vec<Move> = moves
        .iter()
        .copied()
        .filter(|&mv| in_check || is_noisy(mv))
        .collect();
    // Capture history is taught by the main search only, since a cutoff's bonus is
```

```
depth².
    // Unless `capture_history_quiescence` is on, the quiescence search keeps MVV-
LVA's order.
    let techniques = Techniques {
        capture_history: self.techniques.capture_history
            && self.techniques.capture_history_quiescence,
        ..self.techniques
    };
    let picker = MovePicker::new(
        pos,
        &moves,
        None,
        ply,
        &Previous::default(),
        self.ordering,
        &techniques,
    );
    let stand_pat = best;
    let mut child_pv = Vec::new();
    for mv in picker {
        if !in_check {
            // Delta pruning: even winning the piece outright wouldn't reach alpha.
            if self.techniques.delta_pruning && mv.promotion().is_none() {
                let gain = pos
                    .piece_at(mv.to())
                    .map_or(SEE_VALUES[0], |p| SEE_VALUES[p.piece_type.index()]);
                if stand_pat + gain + self.search_params.delta_margin <= alpha {
                    self.stats.count(Note::DeltaPruning(mv));
                    continue;
                }
            }
            // SEE pruning: a capture that loses material can't help.
            if self.techniques.see_pruning && see(pos, mv) < 0 {
                self.stats.count(Note::SeePruning(mv));
                continue;
            }
        }
        let mut child = pos.clone();
        child.make_move(mv);
        self.update_accumulator(pos, Some(mv), ply);
        let score = -self.quiescence(&child, ply + 1, -beta, -alpha, &mut child_pv);
        if self.aborted {
            break;
        }
        if score > best {
            best = score;
            if score > alpha {
                alpha = score;
                pv.clear();
                pv.push(mv);
                pv.extend_from_slice(&child_pv);
            }
        }
        if self.techniques.alpha_beta && alpha >= beta {
            break;
        }
    }
```

```
    best
}
```

Three details make it work:

- **Stand pat.** The side to move never *has* to capture. Its static evaluation is a lower bound on its score, the "I'll just stop here" option. If that's already `≥ beta`, the opponent won't allow this position and the search returns at once. Otherwise it raises `alpha`, and only captures that do better than standing still matter.
- **Ordering.** The quiescence search is mostly cutoffs, and those depend on trying the best capture first (see MVV-LVA below).
- **Checks.** In check, standing pat isn't an option. The position may be lost, so every legal move is searched, not just captures. That's how the quiescence search spots a mate at the horizon.

Now the same position:

```
> set quiescence on
> go
  depth  1    +7.00          41 nodes       0 ms  Kf1
cb2026 plays Kf1   (depth 1, +7.00, 41 nodes, 0 ms)
```

The score is right (a queen against two pawns is +7) and the queen stays home. (Kf1 is as good a quiet move as any to an engine that only counts material. Stage 9 gives it reasons to prefer one square over another.)

From the starting position the scores stop flipping:

```
info depth 4 seldepth 7 score cp 0 nodes 3830 time 0 pv b1a3 b8a6 g1f3 a6b4
info depth 5 seldepth 11 score cp 0 nodes 31783 time 5 pv b1a3 b8a6 g1f3 a6b4 a3b1
info depth 6 seldepth 11 score cp 0 nodes 205774 time 42 pv b1a3 b8a6 g1f3 a6b4 a3b1
b4d5
```

`seldepth`, the deepest ply reached, is now larger than `depth`: that's the quiescence search following captures.

MVV-LVA

Alpha-beta prunes most when the best move is searched first (chapter 5). Without any knowledge of the position, the best guess is that captures come first, and among captures: **Most Valuable Victim, Least Valuable Attacker**. Take the biggest piece you can, and use your smallest piece to do it. Pawn takes queen comes first, queen takes pawn last.

```
/// The MVV-LVA key of `mv` in `pos`: higher means search earlier. Quiet moves score
0;
/// captures and queen promotions score above 0, by victim (or promoted piece) first
and by
/// attacker second.
///
/// The key uses the pieces' *rank* (pawn 1 … queen 5), not their values: only the
order
/// matters, and ranks keep the victim strictly more important than the attacker.
pub fn mvv_lva(pos: &Position, mv: Move) -> i32 {
    let rank = |pt: PieceType| pt.index() as i32 + 1;
    let mut gain = 0;
    if mv.is_capture() {
        let victim = match mv.kind() {
            MoveKind::EnPassant => PieceType::Pawn,
            _ => pos
                .piece_at(mv.to())
                .map_or(PieceType::Pawn, |p| p.piece_type),
        };
        gain += rank(victim);
    }
    if mv.promotion() == Some(PieceType::Queen) {
        gain += rank(PieceType::Queen);
    }
    if gain == 0 {
        return 0;
    }
    let attacker = pos
        .piece_at(mv.from())
        .map_or(PieceType::Pawn, |p| p.piece_type);
    // Six attacker ranks fit between consecutive victim ranks, so the victim always
decides.
    gain * 8 - rank(attacker)
}
```

In the quiescence search, ordering is not a nicety. In a sharp position like Kiwipete, searching captures in generation order makes the quiescence search explode: try `set mvvlva off` and `go depth 3` on

it. The ordered search finishes depth 3 in 15,786 nodes and 4 ms. The unordered one, after a second and more than 4 million nodes, has only got through depth 1. In a quiet opening position at depth 4 the difference is 12,242 nodes against 178,336.

MVV-LVA has a known blind spot: QxP is ordered before the other captures of a pawn even when the pawn is defended and the exchange loses the queen. Stage 14 adds static exchange evaluation (SEE), which works out the whole exchange on a square. Quiet moves keep their generation order for now; stage 10 orders them with killer moves and the history heuristic.

Pitfalls

- **Searching all promotions.** Under-promotions almost never matter and multiply the tree, so only queen promotions are searched.
- **Standing pat in check.** That would let a mated side "stand pat" at a good score. In check, cb2026 searches every evasion and recognises mate.
- **Comparing with minimax.** With alpha-beta off, the quiescence search has no cutoffs at all and takes minutes on positions like Kiwipete. The alpha-beta-vs-minimax test in `search.rs` switches quiescence off for that reason.

What it's worth

Test	Result
Stage 7 vs stage 6, 8+0.08, SPRT [0, 5]	+123 −31 =128 over 282 games: passed (about +118 ± 30 Elo)

Further reading

- Chess Programming Wiki: Quiescence Search, Horizon Effect, MVV-LVA, Move Ordering.

The transposition table

Stage 8 · tag `stage-08-tt` *· code in* `crates/engine/src/tt.rs` *and* `search.rs`

1. e4 e5 2. Nf3 and 1. Nf3 e5 2. e4 reach the same position by different routes. Chess is full of these **transpositions**, and a plain alpha-beta search treats each route as a new position and searches it again. Iterative deepening makes it worse: depth 7 repeats all of depth 6's work. The **transposition table** (TT) remembers what the search found in every position it visited, so it can reuse the result or at least start from the best move.

What gets stored

The TT is one large array of entries indexed by the position's Zobrist key (chapter 4):

```
/// What a stored score means.
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum Bound {
    /// The exact score.
    Exact,
    /// A lower bound: the real score is at least this (the search failed high).
    Lower,
    /// An upper bound: the real score is at most this (the search failed low).
    Upper,
}

/// One stored result.
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub struct Entry {
    /// The position's Zobrist key.
    pub key: u64,
    /// The best move found, if any (a fail-low node may have none).
    pub best_move: Option<Move>,
    /// The score, with mates relative to this position (see [`score_to_tt`]).
    pub score: i32,
    /// The depth the position was searched to.
    pub depth: u8,
    /// What kind of score it is.
    pub bound: Bound,
    /// The search that stored it, for replacement.
    pub generation: u8,
    /// Whether the position is or has been on the principal variation (the search's
`tt_pv`
    /// flag, after Stockfish's).
    pub pv: bool,
}
```

Two parts of an entry are useful in different ways:

- **The best move.** Even when the stored result is too shallow to reuse, its best move is probably still best, and searching it first gives alpha-beta its early cutoffs (chapter 5). With iterative deepening, depth d leaves a best move behind for almost every position depth $d + 1$ visits. Most of the TT's value comes from this.
- **The score**, if it was searched at least as deep as the search now needs. Then the position needn't be searched at all: a **TT cutoff**.

Bounds

Alpha-beta rarely learns a position's exact score. When a move fails high (`≥ beta`), the search stops early: it only knows the score is *at least* that. When every move fails low, it only knows the score is *at most* `alpha`. So each entry records which kind of score it holds, and a later search uses it only when it settles the question that search is asking:

Bound	What it says	A TT cutoff when
`Exact`	score = stored	always
`Lower`	score ≥ stored	stored ≥ beta: still fails high
`Upper`	score ≤ stored	stored ≤ alpha: still fails low

Here is the probe at the top of negamax, then the store at the end:

```
// Transposition table: a deep enough result may settle this position at once, and in
// any case its best move is searched first.
// (Not in a singular extension's search: the entry is about the node with every
move.)
let mut tt_move = None;
let mut tt_entry = None;
if self.techniques.tt
    && excluded.is_none()
    && let Some(entry) = self.tt.probe(pos.hash())
{
    self.note(Note::TtHit);
    tt_entry = Some(entry);
    tt_move = entry.best_move.filter(|mv| moves.contains(mv));
    if u32::from(entry.depth) >= depth {
        let score = score_from_tt(entry.score, ply);
        let usable = match entry.bound {
            Bound::Exact => true,
            Bound::Lower => score >= beta,
            Bound::Upper => score <= alpha,
        };
        if usable {
            self.note(Note::TtCutoff);
            pv.extend(tt_move);
            return score;
        }
    }
}
```

```
// (Not in a singular extension's search: its result leaves the best move out.)
if self.techniques.tt && !self.aborted && excluded.is_none() {
    // After a fail low every move was refuted, so none of them is known to be best.
    let move_to_store = if bound == Bound::Upper {
        None
    } else {
        best_move
    };
    let score = score_to_tt(best, ply);
    self.tt
        .store(pos.hash(), move_to_store, score, depth, bound, tt_pv);
}
```

(The `excluded` conditions came later, with singular extensions in chapter 14.1.)

The stored move is checked against the legal moves before it is used. A wrong move could come from two different positions sharing a key, which is very rare with 64-bit keys but not impossible.

Mate scores

A mate score counts plies from the root: `MATE - 7` means mate in 7 plies from where the search started. But the TT stores positions, and the same position can turn up at ply 3 in one line and ply 5 in another. So mate scores are stored relative to the position itself ("mated in 2 plies from here") and converted back when read. Forget this, and the engine reports wrong mate distances or shuffles its pieces instead of mating.

Indexing and replacement

The table has a fixed size, set with the UCI `Hash` option (16 MB by default) or `set hash` in the shell. A key picks a slot with one multiplication instead of a division:

```
let index = ((key as u128 * len as u128) >> 64) as usize;
```

This treats the key as a fraction between 0 and 1 and scales it to the table size, which works for any size. The trick is from Daniel Lemire, "A fast alternative to the modulo reduction" (2016).

Slots fill up fast, so the table needs a **replacement strategy** for when a new result maps to an occupied slot:

```
/// Stores a search result for the position with `key`, and whether the position is or
has
/// been on the principal variation (`pv`).
pub fn store(
    &self,
    key: u64,
    best_move: Option<Move>,
    score: i32,
    depth: u32,
    bound: Bound,
    pv: bool,
) {
    let generation = self.generation.load(Ordering::Relaxed);
    let depth = depth.min(u8::MAX as u32) as u8;
    let slot = self.slot_to_store(key, generation);
    let old = slot.entry();
    let replace = match old {
        None => true,
        Some(old) => old.key == key || old.generation != generation || depth >=
old.depth,
    };
    if replace {
        // A fail-low result has no best move; keep the one already known for this
position.
        let best_move = best_move.or_else(|| {
            old.filter(|old| old.key == key)
                .and_then(|old| old.best_move)
        });
        slot.write(key, pack(best_move, score, depth, bound, generation, pv));
    }
}

/// Where a result for `key` goes: the slot that already holds the position, else an
empty
/// one, else the one whose entry is worth least (see [`worth`]). With buckets off
there is
/// only one candidate.
fn slot_to_store(&self, key: u64, generation: u8) -> &Slot {
    let slots = self.slots_for(key);
    slots
        .iter()
        .find(|slot| slot.entry_for(key).is_some())
        .unwrap_or_else(|| {
            // `min_by_key` returns the first of equals, so the first empty slot.
            slots
                .iter()
                .min_by_key(|slot| slot.entry().map_or(i32::MIN, |e| worth(e,
generation)))
                .expect("a key always has at least one slot")
        })
}
```

cb2026 replaces an entry if it is for the same position, from an earlier search, or searched no deeper than the new result. Deep results cost

the most to recompute, so they're kept until they go stale. Stronger engines give each key a bucket of several slots and weigh age against depth; chapter 8.1 adds both, which is why the code above picks a slot first. The Chess Programming Wiki lists the options.

What it saves

Each pair below has the same move and score, with and without the table (`set tt off`), at depth 6:

Position	With TT	Without	Saved
Kiwipete	694,370 nodes	1,606,180 nodes	57%
An open game after 2...Nc6	190,506	475,421	60%
A bishop endgame	15,237	42,065	64%

The table also lasts from one move to the next. When the engine moves and the opponent replies, the new position was already searched two plies deep in the last search, so the next search starts with good moves for most positions it visits. `ucinewgame` clears it.

GUIs show how full the table is: `info ... hashfull 312` means 31.2%, estimated from the first thousand slots.

Pitfalls

- **Storing garbage.** A search stopped by the clock returns made-up scores. Nothing is stored once the search has been stopped.
- **Graph history interaction.** The TT doesn't know how a position was reached. A position that is a draw by repetition on one path might be stored as a draw and reused on a path where it isn't. Every engine accepts this small inaccuracy.

- **Truncated principal variations.** A TT cutoff ends the line at that position, so the PV can come out shorter than the depth. Later stages can rebuild it from the table.
- **Minimax comparisons.** The TT's bounds come from alpha-beta, so the minimax comparison test switches it off.

What it's worth

Test	Result
Stage 8 vs stage 7, 8+0.08, SPRT [0, 5]	+48 −9 =230 over 287 games: passed (about +48 ± 17 Elo)

Further reading

- Chess Programming Wiki: Transposition Table, Replacement Strategies, Graph History Interaction.
- Albert Zobrist, "A New Hashing Method with Application for Game Playing" (1970): the hashing behind the table.
- Daniel Lemire, "A fast alternative to the modulo reduction" (2016).

Buckets: one cache line per key

Milestone M5, after stage 20 · on `main`, no stage tag · code in `crates/engine/src/tt.rs`

Chapter 8's table gives every key one slot. Two positions whose keys pick the same slot fight over it, and the replacement rule decides which result is lost. That costs most when the loser was a deep search: a result from depth 12 can be thrown away to make room for one from depth 2, just because it came from the previous move.

Buckets give each key several slots to choose from. The number of slots isn't arbitrary: it comes from how the processor reads memory.

Like chapter 17's threads, this is about the machine the engine runs on, not chess. It came after stage 20, as strength work towards version 1.0, so no stage tag contains it. Switch it with `set ttbuckets on|off`.

cb2026 ships with it off. In games it made the engine weaker, about 9 Elo at 8+0.08, and at 40+0.4 it came out about even (see "What it's worth"). The table really does keep more of what it is given, as this chapter shows; what that bought in the games was a bigger search, not a better one.

Cache lines

A table of 16 MB doesn't fit in the processor's caches, so most probes wait for main memory, and memory doesn't arrive a byte at a time. It comes in **cache lines**, blocks of a fixed size at aligned addresses. Asking for one byte fetches the whole line around it.

Apple Silicon's lines are 128 bytes (`sysctl hw.cachelinesize` prints 128 on an M1 Max). Most x86 processors use 64. A slot is 16 bytes, the two 64-bit words of chapter 17's lock-free entries. So in chapter 8's table, a probe that misses the caches fetches eight slots and looks at one of them.

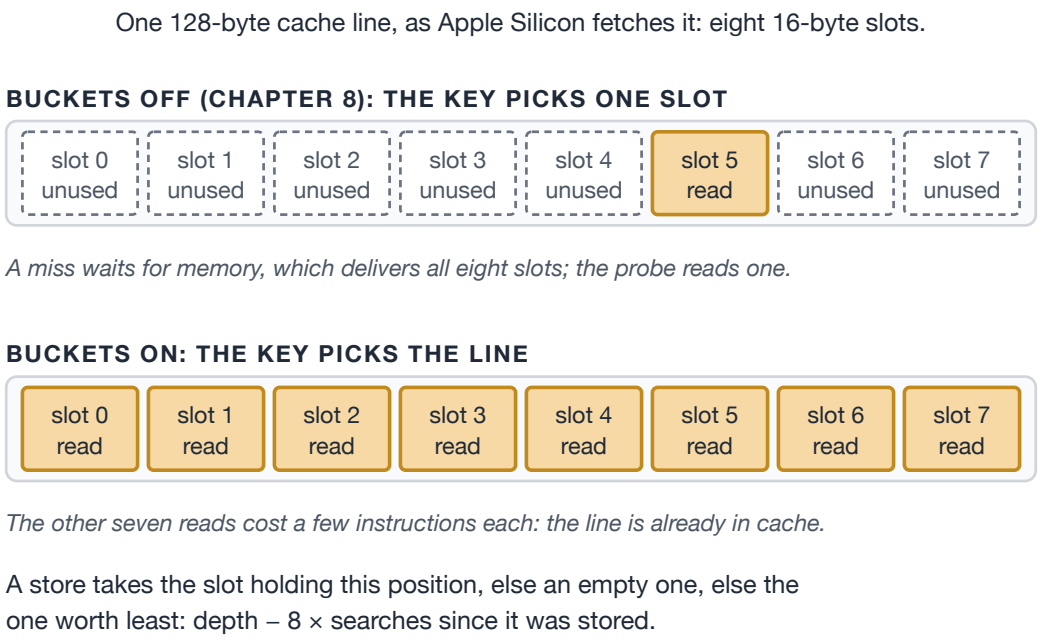

Figure 8.1.1 A probe that misses the cache fetches a whole line of eight slots; buckets use all of them

A bucket uses all eight:

```
/// [`BUCKET_SLOTS`] slots, aligned so that they fill exactly one cache line. (`align` only
/// takes a number, not a constant; the assertion below checks it matches.)
#[derive(Clone, Debug, Default)]
#[repr(align(128))]
struct Bucket([Slot; BUCKET_SLOTS]);
```

`repr(align(128))` makes every bucket start on a line boundary, so a bucket is exactly one line, never two halves of two lines. A `Vec` of buckets is aligned the same way by the allocator, so this needs no `unsafe` code. A compile-time assertion checks the size and alignment.

Finding a position

A key now picks a bucket instead of a slot, with the same multiplication as chapter 8:

```
/// The slots where the position with `key` may be stored: its bucket, or with buckets off
/// a single slot.
///
/// Taking the key modulo the table size would work, but division is slow. Instead the key
/// is treated as a fraction of 2^64 and scaled to the number of buckets (or slots):
/// `(key × count) >> 64`. It is one multiplication, works for any table size, and spreads
/// keys evenly because Zobrist keys are uniformly random. (Daniel Lemire, "A fast
/// alternative to the modulo reduction", 2016.) There are [`BUCKET_SLOTS`] times as many
/// slots as buckets, so a key's slot is always one of the slots of its bucket.
fn slots_for(&self, key: u64) -> &[Slot] {
    let scale = |count: usize| ((key as u128 * count as u128) >> 64) as usize;
    if self.use_buckets {
        &self.buckets[scale(self.buckets.len())].0
    } else {
        let index = scale(self.len());
        std::slice::from_ref(&self.buckets[index / BUCKET_SLOTS].0[index % BUCKET_SLOTS])
    }
}
```

A probe reads the bucket's slots in turn until one has the key. A miss reads all eight, but the first read already brought the whole line into

the cache, so the other seven cost a few instructions each, not another wait for memory. `bench` measured the same speed both ways: about 1.63 million nodes per second over five runs each on the M1 Max.

Take a key as a fraction x of 2^64 and a table of B buckets. Its bucket is `⌊Bx⌋`, and with buckets off its slot is `⌊8Bx⌋`, since there are eight times as many slots. Slot `s` is in bucket `⌊s / 8⌋`, and `⌊⌊8Bx⌋ / 8⌋ = ⌊Bx⌋`, so a key's single slot is always one of the slots of its bucket. Switching buckets on in the middle of a game still finds every entry. Switching them off only finds the entries that happen to sit in their key's own slot.

Making room

A store goes to the slot that already holds the position, or else an empty slot. When all eight slots hold other positions, it goes to the slot whose entry is **worth least**:

```
/// Stores a search result for the position with `key`, and whether the position is or
has
/// been on the principal variation (`pv`).
pub fn store(
    &self,
    key: u64,
    best_move: Option<Move>,
    score: i32,
    depth: u32,
    bound: Bound,
    pv: bool,
) {
    let generation = self.generation.load(Ordering::Relaxed);
    let depth = depth.min(u8::MAX as u32) as u8;
    let slot = self.slot_to_store(key, generation);
    let old = slot.entry();
    let replace = match old {
        None => true,
        Some(old) => old.key == key || old.generation != generation || depth >=
old.depth,
    };
    if replace {
        // A fail-low result has no best move; keep the one already known for this
position.
        let best_move = best_move.or_else(|| {
            old.filter(|old| old.key == key)
                .and_then(|old| old.best_move)
        });
        slot.write(key, pack(best_move, score, depth, bound, generation, pv));
    }
}

/// Where a result for `key` goes: the slot that already holds the position, else an
empty
/// one, else the one whose entry is worth least (see [`worth`]). With buckets off
there is
/// only one candidate.
fn slot_to_store(&self, key: u64, generation: u8) -> &Slot {
    let slots = self.slots_for(key);
    slots
        .iter()
        .find(|slot| slot.entry_for(key).is_some())
        .unwrap_or_else(|| {
            // `min_by_key` returns the first of equals, so the first empty slot.
            slots
                .iter()
                .min_by_key(|slot| slot.entry().map_or(i32::MIN, |e| worth(e,
generation)))
                .expect("a key always has at least one slot")
        })
}
```

An entry's worth is its depth, less 8 plies for every search since it was stored. Deep results cost the most to search again, and old ones are

the least likely to be needed. Take a depth-14 entry from the previous move and a depth-2 entry from this one: they are worth $14 - 8 = 6$ and 2, so the shallow one goes. Two moves later the old entry is worth $14 - 16 = -2$, and it goes first.

Chapter 8's rule then decides whether the chosen entry is replaced. It always is unless it comes from this search and is deeper than the new result. In a bucket the chosen entry is the least valuable of eight, so that is rare.

The age weighting is Stockfish's (GPL-3.0), where an entry one search old also counts as 8 plies shallower. Stockfish packs three 10-byte entries into 32 bytes; cb2026 keeps its 16-byte slots with full keys and puts eight in a line. The code is cb2026's own.

What it changes

The six positions of chapters 10 to 14, each searched from an empty table:

Table	Buckets	Depth 12 nodes	Depth 15 nodes
1 MB	off	1,670,589	5,123,837
1 MB	on	1,583,256 (−5%)	4,900,688 (−4%)
16 MB	off	1,537,930	4,979,436
16 MB	on	1,583,673 (+3%)	5,318,128 (+7%)

A 1 MB table has 65,536 slots, fewer than these searches have results to store, so it is under pressure, and buckets shrink the tree. At depth 12, with buckets, the 1 MB table does almost exactly what the 16 MB one does: the share of nodes that find an entry is the same to a tenth of a percent in every position, and the trees differ by 417 nodes.

At 16 MB the tree gets *bigger*. The table keeps more deep entries, and every singular extension (chapter 14.1) starts from a deep enough entry. At depth 15, with buckets on, the six positions run 9% more singular searches (49,042 against 44,877) and extend 12% more hash moves. More extensions mean a bigger tree. As chapter 10.1 found, node counts can't say whether that makes the engine stronger. Only games can, and here they said no: about **−9 Elo** at 8+0.08. The likeliest reason is the one above. At 16 MB and this time control the table is under little pressure, so buckets save few results that matter, and the extra deep entries start extensions that cost more time than they repay. That is a reading of the node counts, not something the games showed.

A test at 40+0.4 fits it, without proving it. With five times as long per move the table fills more, and the loss went away: **+1.6 ± 6.3 Elo** after 4,630 games, stopped with no verdict either way. It looks as if what buckets save there is worth about what the bigger trees cost, and the difference is too small to measure in 18 hours of games.

With `set ttbuckets on`, `bench` counts 1,114,877 nodes instead of 1,098,926 (+1.5%). Off, the default, the table behaves exactly as in chapter 8, and `bench` is 1,098,926.

Pitfalls

- **Other processors.** With 64-byte lines, a bucket spans two. They are next to each other, and many processors fetch such pairs together (Intel's "adjacent line" prefetcher does). The bucket size is the same everywhere so that `bench`, which identifies a version of the engine, counts the same nodes on every machine.

- **Alignment.** `repr(align(...))` takes a number, not a constant, so the 128 is written twice. The compile-time assertion catches a mismatch.
- **Nodes aren't Elo.** See above: a better table made the 16 MB tree bigger.
- **Light pressure in short games.** At 8+0.08 a move searches a few hundred thousand nodes, and the default table has about a million slots. Buckets matter more when the table fills: longer time controls, more threads, or a smaller `Hash`. Even at 40+0.4 they weren't worth a measurable gain; a smaller `Hash` at 8+0.08 would be the cheaper way to test a full table.

What it's worth

Test	Result
Buckets vs `main` without them, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−8.8 ± 8.6 Elo** over 2,685 games (+536 −604 =1545): failed (LLR −2.96); off by default
Buckets vs `main` without them, 40+0.4, test Mac (M3 Max), SPRT [0, 5]	**+1.6 ± 6.3 Elo** over 4,630 games (+916 −895 =2819): no verdict (LLR −0.45), stopped; off by default (cjus/chessengine#92)

Further reading

- Chess Programming Wiki: Transposition Table, Replacement Strategies.
- Stockfish's source (GPL-3.0), `tt.h` and `tt.cpp`: clusters of three entries and replacement by depth minus age.

- Ulrich Drepper, "What Every Programmer Should Know About Memory" (2007): cache lines, and why a miss costs what it does.

Piece-square tables and tapered evaluation

Stage 9 · tag `stage-09-pst` *· code in* `crates/engine/src/eval.rs`

Until now cb2026 counted material and nothing else. All quiet moves looked equally good to it, so it played whichever one the move generator listed first: in chapter 7 it kept its queen safe with Kf1, and from the starting position it played Na3. This chapter teaches it where its pieces belong.

Piece-square tables

A **piece-square table** (PST) gives a piece type a bonus or penalty on each of the 64 squares. The idea is old and cheap: the evaluation adds one table lookup per piece. What goes into the tables is basic chess knowledge:

- **Knights** belong in the centre. On e5 a knight reaches eight squares, on a1 only two ("a knight on the rim is dim").
- **Bishops** like long, open diagonals and dislike corners.
- **Rooks** like the seventh rank, where they attack pawns and hem in the king.
- **Pawns** gain value as they advance, the centre pawns most. Unmoved d- and e-pawns are penalised, since they block the bishops.

- **The king** hides behind its pawns in the corner while there are pieces around to attack it.

cb2026's tables are written like a printed board, from White's side with rank 8 at the top, so they can be read at a glance. Here is the knight's, in centipawns:

```
-50 -35 -25 -20 -20 -25 -35 -50      8
-35 -15   0   5   5   0 -15 -35      7
-25   5  15  20  20  15   5 -25      6
-20   5  20  25  25  20   5 -20      5
-20   0  15  25  25  15   0 -20      4
-25   5  10  15  15  10   5 -25      3
-35 -15   0   5   5   0 -15 -35      2
-50 -30 -25 -20 -20 -25 -30 -50      1
 a   b   c   d   e   f   g   h
```

A black piece uses the same table, flipped: Black's back rank is its rank 1. The flip is one XOR on the square number:

```
pub fn pst_index(color: Color, square: Square) -> usize {
    match color {
        Color::White => square.index() ^ 56,
        Color::Black => square.index(),
    }
}
```

The numbers are cb2026's own, set by hand from the principles above. They don't need to be exact. Stage 16 tunes all of them automatically on hundreds of thousands of positions from self-play games.

Tapered evaluation

The king's table shows the problem with a single set of values. In the middlegame the king must hide, but in an endgame it is a strong piece that should march to the centre. Pawns, too, matter much more as the board empties and promotion gets closer.

So every value comes in two versions, **middlegame** and **endgame**, and the evaluation blends them by the **game phase**: a count of the pieces left on the board.

```
phase = knights + bishops + 2 × rooks + 4 × queens        24 at the start
score = (mg × phase + eg × (24 − phase)) / 24
```

At the start the score is all middlegame. With only kings and pawns left it is all endgame. After one rook pair is traded it is 20/24 middlegame. The blend changes a little with each trade, so the score never jumps, and the search has no reason to seek or avoid a trade just to cross a threshold.

```
/// The evaluation's tunable numbers, and switches for its term groups.
#[derive(Clone, Debug, PartialEq, Eq)]
pub struct EvalParams {
    /// Middlegame value of each piece type in centipawns, indexed by
[`PieceType::index`]
    /// (the king's value is never used: both sides always have one).
    pub mg_values: [i32; 6],
    /// Endgame value of each piece type.
    pub eg_values: [i32; 6],
    /// Middlegame piece-square tables, indexed by [`PieceType::index`].
    pub mg_pst: [Pst; 6],
    /// Endgame piece-square tables.
    pub eg_pst: [Pst; 6],
    /// How much each piece type counts toward the game phase.
    pub phase_weights: [i32; 6],
    /// Use the piece-square tables (off: material only).
    pub use_pst: bool,
    /// Blend middlegame and endgame values by phase (off: middlegame values only).
    pub tapered: bool,
    /// The positional terms of the full evaluation (stage 15).
    pub terms: TermParams,
}
```

```
/// Reports every term of the evaluation, for both sides, to `sink`.
pub fn collect<S: TermSink>(pos: &Position, params: &EvalParams, sink: &mut S) {
    for color in Color::ALL {
        for piece_type in PieceType::ALL {
            for square in pos.pieces(color, piece_type) {
                sink.term(color, Param::Material(piece_type), 1);
                if params.use_pst {
                    sink.term(color, Param::Pst(piece_type, pst_index(color, square)),
1);
                }
            }
        }
    }
    hce::collect(pos, &params.terms, sink);
}

/// The game phase: from 0 (bare kings and pawns) to [`MAX_PHASE`] (all the pieces).
pub fn phase(pos: &Position, params: &EvalParams) -> i32 {
    let phase: i32 = PieceType::ALL
        .iter()
        .map(|&pt| params.phase_weights[pt.index()] * pos.pieces_of_type(pt).count()
as i32)
        .sum();
    phase.min(MAX_PHASE)
}

/// Adds up the terms' values into an [`EvalTrace`].
struct TraceSink<'a> {
    params: &'a EvalParams,
    trace: EvalTrace,
}

impl TermSink for TraceSink<'_> {
    fn term(&mut self, side: Color, param: Param, count: i32) {
        let t = &mut self.trace;
        let slot = match param {
            Param::Material(_) => &mut t.material,
            Param::Pst(..) => &mut t.pst,
            Param::Mobility(_) => &mut t.mobility,
            Param::Doubled | Param::Isolated => &mut t.pawns,
            Param::Passed(_) => &mut t.passed,
            Param::BishopPair | Param::RookOpenFile | Param::RookSemiOpenFile => &mut
t.pieces,
            Param::PawnShield => &mut t.king_safety,
            Param::Tempo => &mut t.tempo,
        };
        let (mg, eg) = self.params.value(param);
        slot[side.index()].0 += mg * count;
        slot[side.index()].1 += eg * count;
    }

    fn king_danger(&mut self, side: Color, penalty: i32) {
        self.trace.king_safety[side.index()].0 -= penalty;
    }
}
```

```
/// Works out the evaluation's parts; the total is from White's point of view.
pub fn trace(pos: &Position, params: &EvalParams) -> EvalTrace {
    let mut sink = TraceSink {
        params,
        trace: EvalTrace::default(),
    };
    collect(pos, params, &mut sink);
    let mut t = sink.trace;
    t.phase = phase(pos, params);
    let (mg, eg) = t.net();
    t.total = if params.tapered {
        (mg * t.phase + eg * (MAX_PHASE - t.phase)) / MAX_PHASE
    } else {
        mg
    };
    t
}

/// The position's value in centipawns, from the side to move's point of view.
pub fn evaluate(pos: &Position, params: &EvalParams) -> i32 {
    let total = trace(pos, params).total;
    match pos.side_to_move() {
        Color::White => total,
        Color::Black => -total,
    }
}
```

The piece values are tapered too: pawns and rooks are worth more in the endgame.

In the shell

`eval` shows the evaluation term by term. Here's the Italian Game with White's king walked to e2 instead of castled:

```
> fen r1bq1rk1/pppp1ppp/2n2n2/2b1p3/2B1P3/2N2N2/PPPPKPPP/R1BQ3R w - - 0 1
> eval
Evaluation, from White's side, in pawns:
                      Middle       End
  Material             +0.00     +0.00
  Piece-squares        -0.55     +0.35
  Sum                  -0.55     +0.35
Phase 24/24: 100% middlegame, 0% endgame
Total: -0.55 for White, so -0.55 for White to move
```

The king on e2 costs White about half a pawn in the middlegame and would be a plus in an endgame. With all the pieces on the board, the middlegame figure counts.

`set pst off` and `set tapered off` switch the two ideas off separately, to measure each.

From the starting position cb2026 now develops its knights instead of playing Na3:

```
info depth 7 seldepth 18 score cp 25 nodes 644665 nps 6024906 time 107 hashfull 47 pv
b1c3 b8c6 g1f3 g8f6 d2d4 d7d5 c1e3
```

Pitfalls

- **Asymmetry bugs.** A mistake in flipping tables for Black makes the evaluation favour one colour. The test `mirrored_positions_score_the_same_for_the_side_to_move` evaluates every bench position and its colour-flipped mirror and requires the same score.
- **Too much positional weight.** Big table values let the engine give away material for "good squares". Tables stay small next to piece values until tuning says otherwise.
- **Phase jumps.** Switching tables at a fixed amount of material, instead of blending, makes the evaluation jump when one piece is traded, and the search plays for that jump.

What it's worth

Test	Result
Stage 9 vs stage 8, 8+0.08, SPRT [0, 5]	+14 −0 =0 over 14 games: passed

The test stopped after 14 straight wins, too few games for a meaningful Elo figure. A material-only engine that doesn't know where its pieces belong has no chance against one that does.

Further reading

- Chess Programming Wiki: Piece-Square Tables, Tapered Eval, Game Phase, Simplified Evaluation Function (a well-known hand-written set of tables, by Tomasz Michniewski).

Move ordering

Stage 10 · tag `stage-10-ordering` *· code in* `crates/engine/src/ordering.rs`

Alpha-beta's speed depends on searching the best move first. With perfect ordering it visits about the square root of the positions minimax does; with bad ordering it saves almost nothing. The stages so far order moves like this:

- the transposition table's move first (chapter 8);
- then captures, by MVV-LVA (chapter 7);
- then quiet moves, in whatever order the generator produced them.

Most moves are quiet, and at many nodes none of the captures is any good, so that last category matters. This chapter adds three heuristics that learn, from the search itself, which quiet moves tend to be good. It also changes *how* moves are ordered, so the work isn't wasted at nodes that cut off early.

Killer moves

Say White threatens mate with Qh7. At ply 3 the search tries Black's first move, finds that Qh7# refutes it, and cuts off. Then it tries Black's second move, and Qh7# probably refutes that too. A quiet move that caused a cutoff at a given ply is a **killer**, and it is tried early at the

same ply in other branches. cb2026 keeps two per ply. When a new killer arrives, the older one is dropped.

Countermoves

Many moves have a natural reply: ...a6 is often met by a4, and Bb5 by ...a6. The **countermove** table remembers, for each opponent move (the piece that moved and the square it went to), the quiet move that last refuted it, and tries that reply early.

The history heuristic

Killers and countermoves remember a handful of moves. The **history** table scores every quiet move, by side, from-square and to-square. When a quiet move causes a cutoff with `depth` plies left, its score rises by `depth²`, since cutoffs deep in the tree save the most work. The quiet moves searched before it at that node, which failed to cut off, lose the same amount. Quiet moves are then searched in order of their history score.

```
/// Records that the quiet move `best` caused a beta cutoff at `ply` with `depth`
left,
/// after the quiet moves in `tried` failed to. `previous` are the moves that led
here.
pub fn record_cutoff(
    &mut self,
    pos: &Position,
    best: Move,
    tried: &[Move],
    ply: u32,
    depth: u32,
    previous: &Previous,
) {
    let killers = &mut self.killers[ply as usize];
    if killers[0] != Some(best) {
        killers[1] = killers[0];
        killers[0] = Some(best);
    }
    if let Some(prev) = previous.mv
        && let Some(slot) = piece_slot(pos, prev)
    {
        self.countermoves[slot][prev.to().index()] = Some(best);
    }
    let side = pos.side_to_move().index();
    let bonus = cutoff_bonus(depth);
    let mut update = |mv: Move, change: i32| {
        gravity(
            &mut self.history[side][mv.from().index()][mv.to().index()],
            change,
        );
        // Continuation history: the same change, after each earlier move.
        let Some(this) = PieceTo::new(pos, mv) else {
            return;
        };
        for (back, earlier) in previous.moved.iter().enumerate() {
            if let Some(earlier) = *earlier {
                let index = continuation_index(back, earlier, this);
                gravity(&mut self.continuation[index], change);
            }
        }
    };
    update(best, bonus);
    for &mv in tried {
        update(mv, -bonus);
    }
}
```

(The `previous` argument and the continuation history lines came later, in chapter 10.1.) The bonus and the update are two small functions, which chapter 10.2's capture history shares:

```
/// The history bonus for a cutoff with `depth` plies left (and the penalty for the
moves that
/// failed before it): `depth²`, since cutoffs deep in the tree save the most work.
fn cutoff_bonus(depth: u32) -> i32 {
    (depth * depth).min(HISTORY_MAX as u32) as i32
}

/// Adds `change` to a history score with gravity: the nearer the cap, the smaller the
change,
/// so scores stay within ±[`HISTORY_MAX`].
fn gravity(entry: &mut i32, change: i32) {
    *entry += change - *entry * change.abs() / HISTORY_MAX;
}
```

The update uses **gravity**: `change − score × |change| / MAX`. A bonus to a score near the maximum hardly moves it, while a penalty to it moves it a lot. Scores stay within ±16,384, and the table keeps adapting instead of saturating. Between searches every score is halved, so old knowledge fades.

The order, and picking moves lazily

The full order is:

1. the transposition table move;
2. captures and queen promotions, by MVV-LVA;
3. the first killer, the second killer, the countermove;
4. the remaining quiet moves, by history.

Sorting the whole list at every node would waste time: most nodes that cut off do so on the first or second move, and the order of the rest never matters. So a **move picker** computes each move's key, then on each request picks the best remaining move. That is one pass of a selection sort, done only as far as the search actually goes:

```
/// Hands out moves best first, choosing each one only when it is asked for.
pub struct MovePicker {
    moves: Vec<(Move, i32)>,
    next: usize,
}

impl MovePicker {
    /// Orders `moves` in `pos` for the search at `ply`. `tt_move` comes first;
`previous` are
    /// the moves that led here (for the countermove and continuation history).
    pub fn new(
        pos: &Position,
        moves: &[Move],
        tt_move: Option<Move>,
        ply: u32,
        previous: &Previous,
        tables: &OrderingTables,
        techniques: &Techniques,
    ) -> MovePicker {
        let killers = if techniques.killers {
            tables.killers(ply)
        } else {
            [None; 2]
        };
        let countermove = if techniques.countermoves {
            tables.countermove(pos, previous.mv)
        } else {
            None
        };
        let key = |mv: Move| {
            if Some(mv) == tt_move {
                TT_MOVE_KEY
            } else if techniques.mvv_lva && is_noisy(mv) {
                let score = tables.noisy_score(pos, mv, techniques);
                // Captures that lose material in the exchange go after the quiet
moves.
                if techniques.see_ordering && see(pos, mv) < 0 {
                    BAD_CAPTURE_KEY + score
                } else {
                    NOISY_KEY + score
                }
            } else if Some(mv) == killers[0] {
                KILLER_KEYS[0]
            } else if Some(mv) == killers[1] {
                KILLER_KEYS[1]
            } else if Some(mv) == countermove {
                COUNTERMOVE_KEY
            } else {
                tables.quiet_score(pos, mv, previous, techniques)
            }
        };
        MovePicker {
            moves: moves.iter().map(|&mv| (mv, key(mv))).collect(),
            next: 0,
        }
    }
```

```
}

impl Iterator for MovePicker {
    type Item = Move;

    /// The best move not yet handed out: one pass of a selection sort. Ties go to the
move
    /// generated first, so the order is deterministic.
    fn next(&mut self) -> Option<Move> {
        let rest = self.moves.get(self.next..)?;
        let mut best = 0;
        for (i, &(_, key)) in rest.iter().enumerate() {
            if key > rest[best].1 {
                best = i;
            }
        }
        let (mv, _) = *rest.get(best)?;
        // Move the chosen one to the front; keep the others in their order.
        self.moves[self.next..=self.next + best].rotate_right(1);
        self.next += 1;
        Some(mv)
    }
}
```

The quiescence search takes the same idea further, with **staged generation**. Unless the side to move is in check, it only wants captures and promotions, so the move generator now has a mode that generates only those (`Position::noisy_moves`). The pieces' target squares are restricted to enemy pieces, and pawns only generate captures and promoting pushes. Most of the positions a search visits are quiescence nodes, so this is where the cheaper generation pays off. A test checks that this mode generates exactly the captures and promotions among the legal moves, in every position within two plies of each perft position.

What it saves

Total nodes to search six positions (openings, middlegames, Kiwipete and an endgame) to depth 7:

Setting	Nodes	vs all on
All on	16,836,067	—
`set killers off`	16,275,312	−3%
`set countermoves off`	16,800,724	0%
`set history off`	18,126,807	+8%
All three off	28,854,516	+71%

Together they cut the tree by 42%. Each one adds little on top of the others, because they often agree on the same move: with history and countermoves on, killers even cost a few percent here. These numbers are from one small set of positions. The real test is games, below. Later stages (late move reductions, above all) lean on good quiet-move ordering even more, because they search the late moves less deeply.

Pitfalls

- **Killers that aren't quiet.** A capture that cuts off is already ordered by MVV-LVA. Storing it as a killer would waste the slot, so only quiet moves are recorded.
- **Stale killers.** Killers belong to a ply of one search tree. They are cleared at the start of each search, while history is only halved.
- **Unbounded history.** Adding bonuses without gravity or a cap lets a few moves' scores run away, and the table stops reflecting recent results.
- **Determinism.** Ties keep generation order, so the search, and the bench count, are reproducible.

What it's worth

Test	Result
Stage 10 vs stage 9, 8+0.08, SPRT [0, 5]	+251 −110 =116 over 477 games: passed (about +106 ± 28 Elo)

Further reading

- Chess Programming Wiki: Move Ordering, Killer Heuristic, Countermove Heuristic, History Heuristic, Move Generation: staged.
- Jonathan Schaeffer, "The History Heuristic and Alpha-Beta Search Enhancements in Practice", *IEEE Transactions on Pattern Analysis and Machine Intelligence* 11 (1989).
- Jos Uiterwijk, "The Countermove Heuristic", *ICCA Journal* 15 (1992).

Continuation history

Milestone M5, after stage 20 · on `main`, no stage tag · code in `crates/engine/src/ordering.rs`

Chapter 10's history table scores a quiet move by its side, from-square and to-square: it asks whether the move has caused cutoffs *anywhere* in the tree. But whether a move is good depends on what just happened. After the opponent's ...Bg4, h3 is often the right reply; after ...Nf6 it rarely is. History adds up both cases into one number.

Continuation history keeps the context. It scores a quiet move *after an earlier move*: the table is indexed by the earlier move (the piece that moved and its destination) and the candidate (its piece and destination). cb2026 keeps two such tables:

- **one ply back**, after the opponent's last move. This is a countermove table that scores every reply instead of remembering one;
- **two plies back**, after the side to move's own move before that. This one catches plans: after Nf3, a knight move to g5 or e5 is often the follow-up, whatever the opponent did in between.

Like chapter 14.1's techniques, it came after stage 20, as strength work towards version 1.0, so no stage tag contains it. Switch it with `set conthist on|off`.

The tables

A move is described by its piece and destination, which is what the tables are indexed by:

```
/// A move as continuation history sees it: the piece that moved and where it went.
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub struct PieceTo {
    /// The piece, `color × 6 + piece type`.
    piece: usize,
    /// The destination square.
    to: usize,
}

impl PieceTo {
    /// `mv`, about to be played in `pos`. `None` if no piece stands on its from-
square, which
    /// a legal move never does.
    pub fn new(pos: &Position, mv: Move) -> Option<PieceTo> {
        let piece = pos.piece_at(mv.from())?;
        Some(PieceTo {
            piece: piece.color.index() * 6 + piece.piece_type.index(),
            to: mv.to().index(),
        })
    }

    /// A number from 0 to 767, for indexing.
    fn index(self) -> usize {
        self.piece * 64 + self.to
    }
}

/// The moves that led to a node, for the heuristics that look back at them.
#[derive(Clone, Copy, Debug, Default, PartialEq, Eq)]
pub struct Previous {
    /// The move that led here, for the countermove (`None` at the root and after a
null move).
    pub mv: Option<Move>,
    /// The last moves by piece and destination, most recent first, for continuation
history.
    pub moved: [Option<PieceTo>; CONTINUATION_PLIES],
}
```

There are 12 pieces and 64 squares, so 768 (piece, destination) pairs, and each table has 768 × 768 scores: about 2.4 MB each, as against 32 KB for the history table. The search keeps the pieces that moved alongside the moves on its path, so at every node it can say what the

last two moves were. At the root, and after a null move, there is no earlier move, and that part of the score is 0.

Ordering and updating

A quiet move's ordering key, after the hash move, the captures, the killers and the countermove, is now the sum of three scores: its history, and its continuation scores after the moves one and two plies back. Each stays within ±16,384, so the sum stays below the countermove's band.

When a quiet move causes a cutoff, all three tables get the same bonus, `depth²`, and the quiet moves searched before it the same penalty, with chapter 10's gravity:

```
/// Records that the quiet move `best` caused a beta cutoff at `ply` with `depth`
left,
/// after the quiet moves in `tried` failed to. `previous` are the moves that led
here.
pub fn record_cutoff(
    &mut self,
    pos: &Position,
    best: Move,
    tried: &[Move],
    ply: u32,
    depth: u32,
    previous: &Previous,
) {
    let killers = &mut self.killers[ply as usize];
    if killers[0] != Some(best) {
        killers[1] = killers[0];
        killers[0] = Some(best);
    }
    if let Some(prev) = previous.mv
        && let Some(slot) = piece_slot(pos, prev)
    {
        self.countermoves[slot][prev.to().index()] = Some(best);
    }
    let side = pos.side_to_move().index();
    let bonus = cutoff_bonus(depth);
    let mut update = |mv: Move, change: i32| {
        gravity(
            &mut self.history[side][mv.from().index()][mv.to().index()],
            change,
        );
        // Continuation history: the same change, after each earlier move.
        let Some(this) = PieceTo::new(pos, mv) else {
            return;
        };
        for (back, earlier) in previous.moved.iter().enumerate() {
            if let Some(earlier) = *earlier {
                let index = continuation_index(back, earlier, this);
                gravity(&mut self.continuation[index], change);
            }
        }
    };
    update(best, bonus);
    for &mv in tried {
        update(mv, -bonus);
    }
}
```

Between searches every score is halved, as history's is.

The idea is from Stockfish (GPL-3.0), which calls it continuation history; its first versions called the two tables counter-move history

and follow-up history. Stockfish also looks four and six plies back. The code here is cb2026's own.

What it saves

The six positions of chapters 10 to 14 at depth 8, with the share of beta cutoffs caused by the first move searched (the `stats` command shows it):

Setting	Nodes	vs defaults	First-move cutoffs
Defaults (continuation history on)	145,410	—	92.7%
`set conthist off`	148,075	+2%	92.8%
`set history off`	118,729	−18%	93.7%
Both off	112,503	−23%	94.3%

Continuation history takes 2% off the tree. Then the table says something odd: with both history tables off, the tree is almost a quarter *smaller*, and more of its cutoffs come from the first move. In chapter 10, before late move reductions, turning history off made the tree 8% bigger.

Since chapter 13, late quiet moves are searched less deeply (LMR) or not at all (late move pruning). With a worse order, the moves that would have caused a cutoff come late, so they get reduced or skipped, and the node gives up sooner. The tree shrinks because good moves were missed, not because they were found quickly. The first-move figure misleads the same way: the cutoffs that a bad order would have found late are the ones that never happen.

So in a search with late move reductions and pruning, node counts can't say whether move ordering got better. Only games can. That

raises a question about the history table itself: with continuation history to order the quiet moves, does it still help? An SPRT with history switched off answers it (below).

Pitfalls

- **Size and speed.** The tables are large, so most lookups miss the cache. In `bench`, cb2026 searches about 3% fewer nodes per second with them. A gain in ordering has to pay for that.
- **Slow learning.** A table with 590,000 scores needs many cutoffs before most of them mean anything; the history table, with 8,192, learns quickly. Summing them helps: history gives a rough order straight away, and continuation history refines it as it learns.
- **Building big tables.** `Box::new([[...]; N])` can build its array on the stack before moving it to the heap, and 4.7 MB is more than a search thread's stack. The tables are one flat `Vec`, indexed by hand.
- **Which piece moved.** The piece that made the move two plies back may since have been captured, so it can't be looked up in the current position. The search records each move's piece as the move is made.
- **Nodes aren't Elo.** See above: with heavy pruning, a smaller tree can mean a worse order.

What it's worth

Test	Result
Continuation history vs `main` without it, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**+10.7 ± 7.3 Elo** over 4,532 games (+1257 −1117 =2158): passed (LLR 2.96)

It took 4,532 games: a gain of about 10 Elo sits close to the test's upper bound of 5, and the closer the true value is to a bound, the longer an SPRT needs to decide. None of the games was lost on time.

Test	Result
History off (continuation history alone orders quiet moves) vs `main` with both, 8+0.08, test Mac (M3 Max), SPRT [−5, 0]	**−17.7 ± 9.9 Elo** over 2,124 games (+427 −535 =1162): failed (LLR −2.96), so history stays on

That second test is a simplification test: its bounds ask whether switching history off loses 5 Elo or more. It does, about 18. So the history table still earns its place, on top of continuation history and in a search that prunes late moves, even though switching it off made the tree 18% smaller. That is the plainest case in this book of fewer nodes meaning a weaker engine: the moves a worse order puts last are the ones that get reduced and pruned, and some of them were good.

Further reading

- Chess Programming Wiki: History Heuristic, Countermove Heuristic.
- Stockfish's source (GPL-3.0), `movepick.h` and `search.cpp`: the continuation history tables and their updates.

Capture history

Milestone M5, after stage 20 · on `main`, no stage tag · code in `crates/engine/src/ordering.rs` and `crates/engine/src/search.rs`

Chapter 7 orders captures by MVV-LVA: the biggest victim first and, among captures of the same piece, the smallest attacker first. It is a good rule, and a fixed one. Bxf7 comes before Rxf7 in every position, whether or not the bishop's capture works there, and a capture of a pawn always waits behind every capture of a knight, even where taking the pawn is what refutes the opponent's last move.

Chapter 10 had the same problem with quiet moves and solved it with the history table, which learns from the search which moves cause cutoffs. **Capture history** does the same for captures. Its table is indexed by the piece that captures, the square it captures on and the piece it takes, and it learns the way history does. Like chapter 14.1's techniques, it came after stage 20, so no stage tag contains it. Its three versions all measured weaker, the second in Stockfish's form and the third the second in the main search only, so it is off by default. `set capthist on` switches the third version on; `set capthistq on` with it gives the second.

The table

```
/// Where `mv`, a capture or queen promotion in `pos`, is in the capture history
table: by the
/// piece that moves, its destination and what it captures. `None` if no piece stands
on its
/// from-square, which a legal move never does.
fn capture_index(pos: &Position, mv: Move) -> Option<usize> {
    let victim = match mv.kind() {
        MoveKind::EnPassant => PieceType::Pawn.index(),
        _ => pos
            .piece_at(mv.to())
            .map_or(NO_VICTIM, |p| p.piece_type.index()),
    };
    Some(PieceTo::new(pos, mv)?.index() * VICTIMS + victim)
}
```

Twelve pieces, 64 squares and six victims make 4,608 scores, 18 KB. The victim keeps apart moves that mean different things: a rook arriving on d5 to take a pawn and one arriving to take a knight. The sixth victim is nothing, for a queen promotion that captures nothing, which the ordering treats as a capture (chapter 7). A king is never captured, so its slot is free for it. An en passant capture takes a pawn that isn't on the destination square, so its victim is set by hand.

Ordering

A capture starts from what it wins, by the piece values of static exchange evaluation (chapter 14): the victim's value, and a queen less a pawn for a queen promotion.

```
/// What `mv`, a capture or queen promotion in `pos`, wins in material if nothing is
taken back,
/// by static exchange evaluation's piece values: the victim's value, plus a queen
less a pawn
/// for a queen promotion. Capture history's ordering starts from this.
pub fn capture_value(pos: &Position, mv: Move) -> i32 {
    let value = |pt: PieceType| SEE_VALUES[pt.index()];
    let mut gain = 0;
    if mv.is_capture() {
        let victim = match mv.kind() {
            MoveKind::EnPassant => PieceType::Pawn,
            _ => pos
                .piece_at(mv.to())
                .map_or(PieceType::Pawn, |p| p.piece_type),
        };
        gain += value(victim);
    }
    if mv.promotion() == Some(PieceType::Queen) {
        gain += value(PieceType::Queen) - value(PieceType::Pawn);
    }
    gain
}
```

Its ordering key is that value times 32, plus its capture history score:

```
/// The score that orders `mv`, a capture or queen promotion in `pos`, among the
others:
/// its MVV-LVA key, or with capture history on, the material it wins scaled up plus
its
/// capture history score, less the attacker's rank (pawn 1 … king 6) to break ties.
pub fn noisy_score(&self, pos: &Position, mv: Move, techniques: &Techniques) -> i32 {
    if techniques.capture_history {
        // The attacker's rank is smaller than any history update from depth 3 on (a
bonus
        // of 9), so it only decides between captures that history can't tell apart,
at
        // first all of them: then the smaller attacker goes first, as in MVV-LVA.
        let attacker = pos
            .piece_at(mv.from())
            .map_or(1, |p| p.piece_type.index() as i32 + 1);
        capture_value(pos, mv) * CAPTURE_VALUE_SCALE + self.capture_history(pos, mv) -
attacker
    } else {
        mvv_lva(pos, mv)
    }
}
```

Times 32, a pawn is worth 3,200, a fifth of the 16,384 that a history score can reach; a knight 10,240, a rook 16,000 and a queen 28,800. So:

- **Among captures of the same piece, capture history decides.** The attacker is left out, as in Stockfish, except to break ties: its rank, 1 to 6, is less than one cutoff's bonus from depth 3 on. At a node the table knows nothing about, every capture of the same piece ties, and the smaller attacker goes first, as in MVV-LVA.
- **Across victims it can move a capture a long way, but not all the way.** A capture of a pawn with the best history there is (3,200 + 16,384) comes before a capture of a rook with none (16,000), but never before a capture of a queen. A knight and a bishop are only 320 points apart (10 centipawns), so between them history decides soon.
- **The static exchange split stays.** Captures that lose material by SEE still come after the quiet moves, in this same order among themselves.

Stockfish 16 (GPL-3.0) adds 7 times the victim's middlegame value to a capture history limited to 10,692. That puts a queen at 1.66 times the limit, where cb2026's scale puts it at 1.76. Its pawn is worth less (126 against a knight's 781, where SEE's values give 100 and 320), so a pawn's capture moves further on its history there.

The first version

The first version kept MVV-LVA's attacker term. Its key was the MVV-LVA key times 1,024 plus capture history. MVV-LVA's key rises by 8 for each rank of victim and falls by 1 for each rank of attacker, so a victim rank was worth 8,192, half of what a history score can reach, and an attacker rank 1,024. Among captures of the same piece, history decided only once it disagreed with MVV-LVA by more than the attackers' difference (3,072 between a pawn and a rook), and across

victims it moved a capture two ranks at most. It took 11.5% off the bench and lost about 19 Elo (see "What it's worth").

The main search only

The quiescence search doesn't teach the table: its nodes have no depth left, and the bonus is `depth²`. The first two versions let it order its captures by the table all the same. The third doesn't, unless `capture_history_quiescence` is on (`set capthistq on`): the quiescence search keeps MVV-LVA's order, and the table orders only the captures of the search that teaches it.

```
// Capture history is taught by the main search only, since a cutoff's bonus is
depth².
// Unless `capture_history_quiescence` is on, the quiescence search keeps MVV-LVA's
order.
let techniques = Techniques {
    capture_history: self.techniques.capture_history
        && self.techniques.capture_history_quiescence,
    ..self.techniques
};
```

The quiescence search is more than half of the nodes, 55% to 58% over the six positions of chapters 10 to 14 (bench positions 2 to 7) at depths 9 to 11. Stockfish (GPL-3.0) orders its quiescence search's captures by capture history too, so this third version departs from it.

Learning

When a move causes a beta cutoff, the captures searched before it at that node get a penalty, and the move gets a bonus if it is a capture:

```
// Beta cutoff: the opponent won't allow this position, so stop looking.
if self.techniques.alpha_beta && alpha >= beta {
    self.note(Note::BetaCutoff(mv, searched));
    // Counted for the node above: once more without a hash move, and less for a
    // doubly extended move, as in Stockfish 17.
    let counts = if doubled {
        p.cutoff_count_double_extended
    } else {
        1
    };
    self.cutoffs[ply as usize] += counts + u32::from(tt_move.is_none());
    if quiet {
        self.ordering
            .record_cutoff(pos, mv, &quiets_tried, ply, depth, &previous);
    }
    // The captures tried before it failed, whichever kind of move cut off.
    let noisy_best = (!quiet).then_some(mv);
    self.ordering
        .record_captures(pos, noisy_best, &noisy_tried, depth);
    break;
}
if quiet {
    quiets_tried.push(mv);
} else {
    noisy_tried.push(mv);
}
```

```
/// Records a beta cutoff with `depth` left in capture history: `best` is the move
that
/// caused it if that was a capture or queen promotion (`None` if it was quiet), and
`tried`
/// are the captures and queen promotions searched before it, which failed to.
pub fn record_captures(
    &mut self,
    pos: &Position,
    best: Option<Move>,
    tried: &[Move],
    depth: u32,
) {
    let bonus = cutoff_bonus(depth);
    let mut update = |mv: Move, change: i32| {
        if let Some(index) = capture_index(pos, mv) {
            gravity(&mut self.captures[index], change);
        }
    };
    if let Some(best) = best {
        update(best, bonus);
    }
    for &mv in tried {
        update(mv, -bonus);
    }
}
```

The penalty goes to the captures tried before a *quiet* move's cutoff too. Every capture that doesn't lose material is searched before the killers and the other quiet moves, so when a quiet move cuts off, those captures all failed, and that is worth knowing. Without the penalties, scores could only rise. The bonus is chapter 10's `depth²`, with the same gravity, and the table is halved between searches like the others.

Credit

The Chess Programming Wiki credits capture history to Stefan Geschwentner (2016) and describes it as replacing MVV-LVA's attacker term with the history score. Stockfish (GPL-3.0) does that: it scores a capture by its victim's value and its capture history. The second version follows it. The first kept the attacker term as a large part of the key; the second keeps it only to break ties. Stockfish generates pawn captures first, then the knights', bishops', rooks' and queens', so its ties tend to go to the smaller attacker too. The code here is cb2026's own.

What it changes

The first version made the bench 11.5% smaller: 943,210 nodes instead of 1,065,754. The second makes it 12.1% smaller, 998,700 instead of 1,136,500 (`main` by then had double extensions, chapter 14.1). With `set capthist off` it is `main`'s exactly, so the change does nothing when switched off. The speed is unchanged: over four alternating pairs of bench runs, the second version's speed was 0.996 to 1.004 times `main`'s. The table is small enough to stay in the cache.

The bench is one depth. Over the six positions of chapters 10 to 14 (bench positions 2 to 7), the second version's trees:

Depth	`set capthist off`	Second version	Cutoffs by the first move (off, on)
9	447,273	347,024 (−22%)	89.8%, 90.1%
10	703,714	570,471 (−19%)	89.7%, 89.7%
11	885,651	1,110,833 (+25%)	89.7%, 90.0%

So the trees shrink at one depth and grow at the next, and the share of cutoffs made by the first move searched hardly moves. Nothing here says whether the order is better.

The third version, in the main search only, makes the bench 11.7% smaller (1,003,431 nodes), and its trees are close to the second version's: 363,572 nodes at depth 9 (−19% against `set capthist off`), 581,326 at depth 10 (−17%) and 1,015,664 at depth 11 (+15%), with 89.5% to 90.0% of cutoffs by the first move. The speed is the same as `main`'s.

Chapter 10.1 warned that with late move reductions and pruning, a smaller tree doesn't prove better ordering: a worse order can push good moves into the reductions and shrink the tree by missing them. The warning seemed to apply less here. Captures are never reduced or pruned for coming late, so a capture moved later is still searched in full, and the 11.5% looked like cutoffs found sooner. The games said otherwise (below): the tree shrank and the engine got weaker. Why isn't measured. One candidate: at a principal variation node, the order decides which capture gets the first move's full window, while the others get null windows, where null-move pruning and the rest are allowed and a good capture can be missed.

Pitfalls

- **Swamping the victim's value.** With too large a history score, a capture of a pawn that happened to cut off a few times would be searched before a free queen. The scale keeps a queen out of history's reach, and lets it move a pawn's capture past a rook's.
- **Ties are most of the table at first.** Without the tie-break, the second version made the bench 6.5% bigger (1,210,252 nodes), not 12.1% smaller: at a node the table knows nothing about, captures of the same piece tied, and the order fell back to the order the moves were generated in, rooks before pawns. A rule that only breaks ties can still change the tree a lot.
- **Bad captures.** History doesn't move a capture across the static exchange split, so a losing capture with a good history still waits behind the quiet moves.
- **Under-promotions** are quiet moves to the ordering (chapter 7), so their history is in the quiet tables, not here.

What it's worth

Test	Result
Capture history vs `main` without it, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−19.2 ± 11.9 Elo** over 1,561 games (+328 −414 =819): failed (LLR −2.96), cjus/chessengine#115, so it is off by default
Capture history's second version (the victim's value plus history, the attacker only breaking ties) vs `main` without it (with double extensions), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−6.8 ± 7.8 Elo** over 3,512 games (+771 −840 =1901): failed (LLR −2.96), cjus/chessengine#129, so it stays off by default
Capture history's third version (the second in the main search only, the quiescence search keeping MVV-LVA's order) vs `main` without it, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−2.9 ± 5.9 Elo** over 6,301 games (+1460 −1512 =3329): failed (LLR −2.97), cjus/chessengine#134, so it stays off by default

Like switching history off in chapter 10.1, the first version made the tree smaller and the engine weaker. The switch and the table stayed for a second version, in Stockfish's form: the victim's value plus capture history, with the attacker only breaking ties. It lost too, by less: about **−7 Elo** (cjus/chessengine#129), again with a smaller bench. The third version left the quiescence search on MVV-LVA's order and lost by less again: about **−3 Elo** over 6,301 games (cjus/chessengine#134). That is close to zero, but the SPRT settled on no gain rather than the 5 Elo it tested for. The second and third versions differ by 4 ± 10 Elo, so the games don't show that the quiescence search's order was what cost the second. All three made the bench about 12% smaller and the engine no stronger; why isn't measured. Still untested:

- **History only among captures of the same piece**: a scale large enough that a victim rank outweighs any history score, so history only reorders the attackers.
- **No penalty for the captures tried before a quiet move's cutoff**, so that a capture is marked down only when another capture did better.

Further reading

- Chess Programming Wiki: History Heuristic, section "Capture History".
- Stockfish's source (GPL-3.0), `movepick.cpp` and `search.cpp`: how captures are scored, and the capture history updates.

Principal variation search and aspiration windows

Stage 11 · tag `stage-11-pvs` *· code in* `crates/engine/src/search.rs`

Chapters 7 to 10 made the first move searched at a node very likely to be the best one. This chapter puts that to work. Both ideas here change the **window** `[alpha, beta]` a search is given, and both rest on the same fact: the narrower the window, the more alpha-beta can prune.

Null-window searches

What does a search with the window `[alpha, alpha + 1]` do? No integer score fits strictly between alpha and alpha + 1, so it can't compute an exact score. It can only answer a yes/no question, *is this position worth more than alpha?*, and it answers with a bound. That is enough to prune almost everything: every move that is clearly worse fails low at once, and the first move that is better fails high and ends the search.

Principal variation search

At a node where move ordering has done its job, the first move is the best and every other move only needs to be *proven worse* than it, not

given an exact score. **Principal variation search** (PVS) does exactly that:

1. Search the first move with the full window `[alpha, beta]`.
2. Search every later move with the null window `[alpha, alpha + 1]`.
3. If a later move unexpectedly scores above alpha, the bet failed: search that move again with the full window to get its real score.

```
/// The score of the move that led to `child` (the last move on the path), from the
/// mover's side, searched to `depth`.
///
/// The first move gets the full window. With PVS on, a later move gets a null window
/// first, at a reduced depth if LMR chose a reduction; if it beats `alpha` it is
searched
/// again at full depth, and then with the full window if it is inside it.
///
/// `cut_node` is the parent's flag. The child of a null-window search is an expected
cut
/// node if its parent is an expected all-node, and the other way round; a child with
the
/// full window is a principal variation node, never a cut node. A late move's reduced
/// search counts as a cut node whatever the parent, as in Stockfish (GPL-3.0).
#[allow(clippy::too_many_arguments)]
fn search_move(
    &mut self,
    child: &Position,
    depth: u32,
    alpha: i32,
    beta: i32,
    how: MoveSearch,
    cut_node: bool,
    pv: &mut Vec<Move>,
) -> i32 {
    let pvs = self.techniques.pvs && self.techniques.alpha_beta;
    let null_window = beta - alpha == 1;
    let reduction = match how {
        MoveSearch::Later { reduction } if pvs => reduction,
        _ => return self.search_child(child, depth, alpha, beta, null_window &&
!cut_node, pv),
    };
    let mv = self.path.last().copied().flatten();
    if reduction > 0
        && let Some(mv) = mv
    {
        self.note(Note::Reduced(mv, reduction));
    }
    let mut score = self.search_child(
        child,
        depth - reduction,
        alpha,
        alpha + 1,
        reduction > 0 || !cut_node,
        pv,
    );
    if score > alpha && reduction > 0 && !self.aborted {
        if let Some(mv) = mv {
            self.note(Note::ReductionResearch(mv));
        }
        score = self.search_child(child, depth, alpha, alpha + 1, !cut_node, pv);
    }
    if score > alpha && score < beta && !self.aborted {
        if let Some(mv) = mv {
            self.note(Note::PvsResearch(mv));
        }
```

```
        score = self.search_child(child, depth, alpha, beta, false, pv);
    }
    score
}

/// The score of `child` for the side that moved there, searched to `depth` within the
/// mover's window `[alpha, beta]`.
fn search_child(
    &mut self,
    child: &Position,
    depth: u32,
    alpha: i32,
    beta: i32,
    cut_node: bool,
    pv: &mut Vec<Move>,
) -> i32 {
    let ply = self.path.len() as u32;
    let score = -self.negamax(child, depth, ply, -beta, -alpha, cut_node, pv);
    self.note(Note::Exit { score: -score });
    score
}
```

The re-searches cost something, but with good ordering they are rare, and the null-window searches are much cheaper than full ones. The name comes from the **principal variation**: the line both sides are expected to play, which is the only one that gets full-window searches.

Aspiration windows

At the root, iterative deepening gives a good guess before each depth starts: the previous depth's score. Scores usually move by a few centipawns from one depth to the next. So instead of the infinite window, each depth from 5 on starts with **aspiration window** `[previous - 25, previous + 25]`.

- If the score lands inside, the search was cheaper than with the full window, and the answer is exact.
- If it **fails low** (the position is worse than thought) or **fails high** (better), the window is widened on that side, the margin doubled,

and the depth searched again. After a fail high, the move that failed high goes first in the re-search. Past a margin of 800 the search gives up on guessing and uses the full window.

```
/// Searches the root to `depth`, using an aspiration window around `previous` (the
last
/// depth's score) when there is one. Fail-high moves are moved to the front of
/// `root_moves` for the re-search.
pub(crate) fn search_depth(
    &mut self,
    pos: &Position,
    root_moves: &mut Vec<Move>,
    depth: u32,
    previous: Option<i32>,
) -> RootResult {
    let p = self.search_params;
    let center = previous.filter(|score| {
        self.techniques.aspiration
            && self.techniques.alpha_beta
            && depth >= p.aspiration_min_depth
            && score.abs() < MATE - MAX_PLY as i32
    });
    let Some(center) = center else {
        return self.search_root(pos, root_moves, depth, -INFINITY, INFINITY);
    };
    let mut delta = p.aspiration_delta;
    let (mut alpha, mut beta) = (center - delta, center + delta);
    let mut fail_high: Option<RootResult> = None;
    loop {
        let result = self.search_root(pos, root_moves, depth, alpha, beta);
        if !result.complete {
            // Stopped during a re-search before any move finished: the move that
failed
            // high is still the best information there is.
            return match fail_high {
                Some(high) if result.score == -INFINITY => RootResult {
                    complete: false,
                    nodes: result.nodes,
                    seldepth: result.seldepth,
                    ..high
                },
                _ => result,
            };
        }
        if result.score <= alpha {
            alpha = (result.score - delta).max(-INFINITY);
        } else if result.score >= beta {
            beta = (result.score + delta).min(INFINITY);
            if let Some(mv) = result.best_move {
                root_moves.retain(|&m| m != mv);
                root_moves.insert(0, mv);
            }
            fail_high = Some(result);
        } else {
            return result;
        }
        delta *= 2;
        if delta > p.aspiration_max_delta {
            (alpha, beta) = (-INFINITY, INFINITY);
        }
```

```
    }
}
```

```
/// The search's tunable numbers.
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub struct SearchParams {
    /// Aspiration windows start at the previous score ± this many centipawns.
    pub aspiration_delta: i32,
    /// Aspiration windows are used from this depth on (shallow scores jump around too
much).
    pub aspiration_min_depth: u32,
    /// Once the margin grows past this, the full window is used.
    pub aspiration_max_delta: i32,
    /// Null moves are tried with at least this much depth left.
    pub null_min_depth: u32,
    /// The null move's search is reduced by this many plies...
    pub null_reduction: u32,
    /// ...plus one more per this many plies of depth.
    pub null_depth_divisor: u32,
    /// Late move reductions start at this depth...
    pub lmr_min_depth: u32,
    /// ...for moves after this many have been searched.
    pub lmr_min_moves: u32,
    /// The reduction's constant part, in hundredths of a ply.
    pub lmr_base: u32,
    /// The divisor of `ln(depth) × ln(move number)`, in hundredths.
    pub lmr_divisor: u32,
    /// A quiet move's reduction shrinks by a ply for each this much history score it
has
    /// (and grows by a ply for each this much below zero)...
    pub lmr_history_divisor: i32,
    /// ...by at most this many plies either way.
    pub lmr_history_max: i32,
    /// Late move reductions are this many plies larger at an expected cut node.
    pub lmr_cut_node: i32,
    /// Late move reductions are this many plies smaller at an expected all-node.
    pub lmr_all_node: i32,
    /// Late move reductions are this many plies smaller at a node that is or has been
on the
    /// principal variation (but isn't a PV node now, which has its own ply off).
    pub lmr_tt_pv: i32,
    /// A node that fails low takes its parent's "is or has been on the PV" flag only
above
    /// this depth.
    pub tt_pv_fail_low_depth: u32,
    /// Late move reductions are `lmr_cutoff_plies` larger when the next ply has
counted more
    /// than this many fail highs (see [`Techniques::cutoff_reductions`])...
    pub lmr_cutoff_count: u32,
    /// ...this many (and `lmr_cutoff_all_node_plies` more at an expected all-node).
    pub lmr_cutoff_plies: i32,
    /// After many fail highs, late move reductions at an expected all-node are this
many plies
    /// larger again (see `lmr_cutoff_plies`).
    pub lmr_cutoff_all_node_plies: i32,
    /// What a fail high by a doubly extended move adds to its ply's count of fail
highs, besides
    /// the one more for a node without a hash move (any other move adds 1).
    pub cutoff_count_double_extended: u32,
```

```
    /// Late move reductions are this many plies larger when the hash move is a
capture (see
    /// [`Techniques::tt_capture_reductions`])...
    pub lmr_tt_capture: i32,
    /// ...but this many at a PV node.
    pub lmr_tt_capture_pv: i32,
    /// Late move pruning applies up to this depth...
    pub lmp_max_depth: u32,
    /// ...once `lmp_base + depth²` quiet moves have been tried...
    pub lmp_base: u32,
    /// ...times this when the side to move is improving (1: no change).
    pub lmp_improving: u32,
    /// History pruning applies up to this depth...
    pub history_pruning_max_depth: u32,
    /// ...to a quiet move whose quiet-move score is below minus this much per ply of
depth.
    pub history_pruning_margin: i32,
    /// Reverse futility pruning applies up to this depth...
    pub rfp_max_depth: u32,
    /// ...when the static evaluation beats beta by this much per ply of depth...
    pub rfp_margin: i32,
    /// ...less this much when the side to move is improving. Keep it below
`rfp_margin`, so
    /// the margin never reaches zero.
    pub rfp_improving: i32,
    /// Razoring applies up to this depth...
    pub razor_max_depth: u32,
    /// ...when the static evaluation is below alpha by this much per ply of depth.
    pub razor_margin: i32,
    /// Futility pruning applies up to this depth...
    pub futility_max_depth: u32,
    /// ...when the static evaluation plus this...
    pub futility_base: i32,
    /// ...plus this much per ply of depth is still at or below alpha.
    pub futility_per_depth: i32,
    /// SEE pruning applies up to this depth...
    pub see_max_depth: u32,
    /// ...to quiet moves losing more than this much per ply of depth...
    pub see_quiet_margin: i32,
    /// ...and to captures losing more than this much per ply of depth.
    pub see_capture_margin: i32,
    /// Delta pruning: a capture is skipped if winning the piece plus this margin
still
    /// leaves the score at or below alpha.
    pub delta_margin: i32,
    /// Internal iterative reductions apply from this depth on.
    pub iir_min_depth: u32,
    /// Singular extensions apply from this depth on...
    pub singular_min_depth: u32,
    /// ...when the hash move's entry is at most this many plies shallower than the
node...
    pub singular_tt_depth_margin: u32,
    /// ...and no other move comes within this many centipawns per ply of depth of its
score.
    pub singular_margin: i32,
    /// A singular hash move is searched two plies deeper if no other move comes
within this
```

```
    /// many centipawns of `singular_beta` either (a double extension)...
    pub double_extension_margin: i32,
    /// ...at most this many times along a line.
    pub double_extension_limit: u32,
    /// A hash move that isn't singular but whose hash score beats beta is searched
this many
    /// plies shallower (a negative extension).
    pub negative_extension_plies: u32,
    /// Correction history moves an entry towards a node's error by at most this many
256ths of
    /// the way (`depth + 1` below it)...
    pub correction_max_weight: i32,
    /// ...and no entry corrects by more than this many centipawns either way.
    pub correction_limit: i32,
    /// With the piece tables on, the pawn table's correction counts this many
256ths...
    pub correction_pawn_share: i32,
    /// ...and each side's piece table's this many, added to it.
    pub correction_non_pawn_share: i32,
    /// ProbCut applies from this depth on...
    pub probcut_min_depth: u32,
    /// ...and asks whether a capture beats beta by this many centipawns...
    pub probcut_margin: i32,
    /// ...in a search this many plies shallower.
    pub probcut_reduction: u32,
}

impl Default for SearchParams {
    fn default() -> SearchParams {
        SearchParams {
            aspiration_delta: 25,
            aspiration_min_depth: 5,
            aspiration_max_delta: 800,
            null_min_depth: 3,
            null_reduction: 3,
            null_depth_divisor: 6,
            lmr_min_depth: 3,
            lmr_min_moves: 3,
            lmr_base: 75,
            lmr_divisor: 225,
            // About one reduced move in seven has a quiet-move score of at least 128
either way
            // (measured over a self-play game; see the module documentation). The
first
            // version used 256 and history alone, and changed about one in ten.
            lmr_history_divisor: 128,
            lmr_history_max: 2,
            // 1 gained +11.6 ± 7.7 against none (cjus/chessengine#145); 2, as in
Stockfish
            // (GPL-3.0), gained +8.7 ± 6.4 against 1 (cjus/chessengine#155).
Stockfish adds
            // them alongside reductions that cb2026 doesn't have. Not tuned further.
            lmr_cut_node: 2,
            // Set by hand, not tuned; it failed at 1 (cjus/chessengine#158).
            lmr_all_node: 1,
            // Set by hand, not tuned; no verdict at 1 (cjus/chessengine#160).
Stockfish
```

```
            // (GPL-3.0) takes off 1 to 3 plies, depending on the hash entry.
            lmr_tt_pv: 1,
            // As in Stockfish 16 (GPL-3.0): above depth 3.
            tt_pv_fail_low_depth: 3,
            // As in Stockfish 16 (GPL-3.0); passed at these values
(cjus/chessengine#165), not
            // tuned.
            lmr_cutoff_count: 3,
            lmr_cutoff_plies: 1,
            // Stockfish 17's version (GPL-3.0), which passed against Stockfish 16's,
+13.7 ± 8.5
            // (cjus/chessengine#167); 0 and 1 give Stockfish 16's
(cjus/chessengine#165).
            lmr_cutoff_all_node_plies: 1,
            cutoff_count_double_extended: 0,
            // As in Stockfish 16 and 17 (GPL-3.0), which apply it at every node. With
1 at PV
            // nodes too it failed (cjus/chessengine#171), and with 0 there too
            // (cjus/chessengine#179).
            lmr_tt_capture: 1,
            lmr_tt_capture_pv: 0,
            lmp_max_depth: 3,
            lmp_base: 3,
            // Doubling it when improving was part of both failed versions (#60, #63).
            lmp_improving: 1,
            // Below one depth-4 cutoff's penalty (16) per ply of depth, up to depth
4: set from
            // measurements over a game's positions, not tuned (see chapter 13.1).
            history_pruning_max_depth: 4,
            history_pruning_margin: 16,
            rfp_max_depth: 7,
            rfp_margin: 80,
            rfp_improving: 40,
            razor_max_depth: 2,
            razor_margin: 300,
            futility_max_depth: 6,
            futility_base: 100,
            futility_per_depth: 100,
            see_max_depth: 6,
            see_quiet_margin: 60,
            see_capture_margin: 100,
            delta_margin: 200,
            iir_min_depth: 4,
            // The first version started at 8 (cjus/chessengine#71); 6 gained +5.0 ±
4.1
            // (cjus/chessengine#78); 4 gained +6.1 ± 4.8 more (cjus/chessengine#80).
            singular_min_depth: 4,
            // 3, as in Stockfish (GPL-3.0); 4 failed, -7.4 ± 8.0 against 3
            // (cjus/chessengine#123).
            singular_tt_depth_margin: 3,
            // 2 until 4 became the start depth; 3 gained +5.2 ± 4.2
(cjus/chessengine#82); 4
            // failed, -1.8 ± 5.3 against 3 (cjus/chessengine#84).
            singular_margin: 3,
            // 20 at first, as in Stockfish (GPL-3.0), had no verdict
(cjus/chessengine#96); 50
            // gained +13.0 ± 8.3 (cjus/chessengine#125); 100 failed, -7.3 ± 8.0
```

```
against 50
            // (cjus/chessengine#127); 70 failed too, -20.5 ± 12.2 against 50
            // (cjus/chessengine#138).
            double_extension_margin: 50,
            double_extension_limit: 6,
            negative_extension_plies: 1,
            correction_max_weight: 16,
            correction_limit: 64,
            // All of the pawn table's and an eighth of each piece table's, added: set
by hand,
            // not tuned. The first version averaged them, 128 and 64, and measured
weaker
            // (cjus/chessengine#119).
            correction_pawn_share: 256,
            correction_non_pawn_share: 32,
            // Set by hand, not tuned: a pawn above beta, four plies shallower.
            probcut_min_depth: 5,
            probcut_margin: 100,
            probcut_reduction: 4,
        }
    }
}
```

What it saves

Total nodes over six positions at depth 8 (the same positions as chapter 10):

Setting	Nodes	vs both on
Both on	40,634,830	—
`set pvs off`	46,234,900	+14%
`set aspiration off`	40,783,218	0%
Both off	56,198,161	+38%

PVS does most of the work. Aspiration windows hardly change the totals on their own, but they and PVS help each other: the narrower root window makes PVS's re-searches rarer. Aspiration windows matter more in long searches, where every saved node at the root multiplies down a deep tree.

Pitfalls

- **Search instability.** With a transposition table, a re-search doesn't always reproduce the result that triggered it: a move fails high on the null window, then scores below alpha on the full one. The search must cope (cb2026 simply keeps the result of the re-search), not loop forever.
- **Mate scores and aspiration.** A window around a mate score makes no sense, since mate scores jump by large amounts between depths. Aspiration is skipped when the previous score was a mate.
- **Fail-soft bounds.** When a root search fails, the returned score (beyond the window) is used to place the next window, which wastes fewer re-searches than widening blindly.
- **Stopping mid-re-search.** If time runs out while re-searching after a fail high, the move that failed high is still the best information there is, so it is kept.

What it's worth

Test	Result
Stage 11 vs stage 10, 8+0.08, SPRT [0, 5], cloud	+340 −297 =213 over 850 games, about +18 ± 20 Elo: no verdict (LLR 0.71) when the run hit the session's time limit
Stage 11 vs stage 10, 8+0.08, SPRT [0, 5], test Mac (M1 Max, 7 games at a time)	+818 −654 =529 over 2001 games, **+28.5 ± 13.1 Elo**: passed (LLR 2.95)

PVS and aspiration windows are worth much less than the early stages, so proving the gain took two thousand games: the first stage

whose test ran on the faster test Mac. The cloud run's estimate (+18 ± 20) and the Mac's (+28.5 ± 13.1) agree within their margins.

Further reading

- Chess Programming Wiki: Principal Variation Search, Null Window, Aspiration Windows, Search Instability.
- Tony Marsland and Murray Campbell, "Parallel Search of Strongly Ordered Game Trees", *ACM Computing Surveys* 14 (1982): the principal variation idea.

Null-move pruning

Stage 12 · tag `stage-12-nullmove` *· code in* `crates/engine/src/search.rs`

Everything so far has been **sound**: with alpha-beta, the transposition table, move ordering and PVS, cb2026 finds the same score minimax would, only faster. From this chapter on, the search starts taking calculated risks. It prunes moves that are *probably* irrelevant, accepting the occasional mistake in exchange for searching much deeper. Null-move pruning was the first of these techniques to become universal, and it is still one of the most effective.

The idea

In chess, having the move is almost always an advantage. Imagine the side to move could pass, a **null move**, and the opponent then gets a full search to show what they can do. If even after passing the score stays at or above beta, then a real move would surely do at least as well. The opponent won't allow this position, and the node can be cut off without searching a single real move.

The test is cheap for two reasons:

- the null move's search is **reduced**: 3 plies less than normal, and one more for every 6 plies of depth;

- it uses a **null window** around beta, since it only asks "still ≥ beta?".

```
// Null-move pruning: if passing still keeps us at or above beta, a real move would
too.
if self.techniques.null_move
    && selective
    && excluded.is_none()
    && depth >= p.null_min_depth
    && self.path.last() != Some(&None)
    && has_pieces(pos)
    && static_eval >= beta
{
    self.note(Note::NullMove);
    let reduction = p.null_reduction + depth / p.null_depth_divisor;
    let mut child = pos.clone();
    child.make_null_move();
    self.push_move(pos, None);
    self.update_accumulator(pos, None, ply);
    self.keys.push(pos.hash());
    let mut null_pv = Vec::new();
    let score = -self.negamax(
        &child,
        depth.saturating_sub(1 + reduction),
        ply + 1,
        -beta,
        -beta + 1,
        !cut_node,
        &mut null_pv,
    );
    self.note(Note::Exit { score: -score });
    self.keys.pop();
    self.pop_move();
    if self.aborted {
        return 0;
    }
    if score >= beta {
        self.note(Note::NullMoveCutoff);
        // A mate found after passing isn't a real mate: claim no more than beta.
        return if score >= MATE - MAX_PLY as i32 {
            beta
        } else {
            score
        };
    }
}
```

(The `excluded` condition came later, with singular extensions in chapter 14.1.)

Most positions where the side to move is far ahead fail this test for the opponent quickly, and whole subtrees vanish. On the six test positions at depth 8, null-move pruning cuts the nodes from 40.6 million to 10.0 million, a 75% saving. That's roughly two extra plies of depth in the same time.

When passing would be better: zugzwang

The idea rests on one assumption: that moving is better than passing. In **zugzwang** it is false: every legal move makes the position worse, and the side to move would love to pass. In such positions null-move pruning cuts off nodes that are actually lost.

Zugzwang is rare in the middlegame and common in pawn endings, so cb2026 doesn't try a null move when the side to move has only pawns (and its king). Stronger engines add a **verification search** for deep nodes, which re-searches without null moves to confirm a cutoff. cb2026 may add this later if testing shows it helps.

The other conditions

A null move is tried only when all of these hold:

- **Not in check:** passing would leave the king in check, which is illegal even in this imaginary world.
- **Not twice in a row:** two passes would cancel out and just waste a reduced search.
- **Not on the principal variation:** at full-window nodes the exact score matters and a risky cutoff could corrupt the main line, so null moves are only tried at null-window nodes. This is why null-move pruning needs PVS (chapter 11).

- **Enough depth left:** at least 3 plies.
- **The static evaluation is already ≥ beta:** if the side to move isn't even ahead before passing, the test is unlikely to succeed and costs nodes for nothing.

The chess crate's `make_null_move` passes the turn. It clears the en passant square and resets the halfmove clock, so the repetition check doesn't treat positions on either side of the pass as repetitions.

A mate found after a null move isn't real (the side that passed broke the rules), so a null-move cutoff never claims a mate score; it returns beta instead.

Pitfalls

- **Zugzwang.** Covered above. If cb2026 misplays a pawn ending, `set nullmove off` is the first thing to try.
- **Threats beyond the reduced depth.** The search after the pass is shallower than a normal one, so it can miss a deep threat and cut off a node that needed searching. These errors are the price of the extra depth, and the SPRT shows whether the trade is worth it.
- **Test positions.** Many "mate in N" puzzle solutions involve quiet moves that null-move pruning makes harder to find, so engines can do worse on puzzle suites while playing stronger games.

What it's worth

Test	Result
Stage 12 vs stage 11, 8+0.08, SPRT [0, 5]	+155 −39 =61 over 255 games: passed (about +170 ± 40 Elo)

Further reading

- Chess Programming Wiki: Null Move Pruning, Null Move Observation, Zugzwang.
- Chrilly Donninger, "Null Move and Deep Search: Selective-Search Heuristics for Obtuse Chess Programs", *ICCA Journal* 16 (1993): the adaptive reduction.
- Omid David and Nathan Netanyahu, "Verified Null-Move Pruning", *ICGA Journal* 25 (2002).

ProbCut

Milestone M5, after stage 20 · on `main`, no stage tag · code in `crates/engine/src/search.rs`

Null-move pruning (chapter 12) cuts a node off when passing keeps the score at or above beta, in a search a few plies shallower. **ProbCut** makes a bet of the same shape with a real move: if some capture beats beta by a clear margin, even in a search four plies shallower, then the full-depth search would almost surely beat beta too, and the node is cut off.

The bet rests on a fact about search scores: a shallow search's score and a deep search's score of the same position are closely related (Buro measured it, for Othello). So a shallow score well above beta says, with high probability, that the deep score is above beta as well. The name is short for *probabilistic cut*. Like chapter 14.1's techniques, it came after stage 20, so no stage tag contains it. It measured weaker, so it is **off by default**; `set probcut on` switches it on.

The test

```
// ProbCut: if a good capture beats beta by a margin even in a shallower search, the
// full-depth search would almost surely beat beta too, so the node is cut off. Not if
// the hash table already says that a search about as deep fell short of the margin.
let probcut_beta = beta + p.probcut_margin;
if self.techniques.probcut
    && self.techniques.quiescence
    && selective
    && excluded.is_none()
    && depth >= p.probcut_min_depth
    && beta.abs() + p.probcut_margin < MATE - MAX_PLY as i32
    && !tt_entry.is_some_and(|entry| {
        entry.bound != Bound::Lower
            && u32::from(entry.depth) + p.probcut_reduction > depth
            && score_from_tt(entry.score, ply) < probcut_beta
    })
{
    self.note(Note::ProbCut);
    let probcut_depth = depth.saturating_sub(p.probcut_reduction);
    let captures: Vec<Move> = moves.iter().copied().filter(|&mv|
is_noisy(mv)).collect();
    let previous = self.previous();
    let picker = MovePicker::new(
        pos,
        &captures,
        tt_move,
        ply,
        &previous,
        self.ordering,
        &self.techniques,
    );
    let mut cutoff = None;
    let mut probcut_pv = Vec::new();
    self.keys.push(pos.hash());
    for mv in picker {
        // Only captures that win, by static exchange, at least what the static
        // evaluation is short of the margin.
        if see(pos, mv) < probcut_beta - static_eval {
            continue;
        }
        let mut child = pos.clone();
        child.make_move(mv);
        self.push_move(pos, Some(mv));
        self.update_accumulator(pos, Some(mv), ply);
        // The quiescence search first: it is cheap, and if even it can't reach the
        // margin, the shallower search seldom does.
        let mut score = -self.quiescence(
            &child,
            ply + 1,
            -probcut_beta,
            -probcut_beta + 1,
            &mut probcut_pv,
        );
        if score >= probcut_beta && probcut_depth > 0 && !self.aborted {
```

```
            score = self.search_child(
                &child,
                probcut_depth,
                probcut_beta - 1,
                probcut_beta,
                !cut_node,
                &mut probcut_pv,
            );
        }
        self.pop_move();
        if self.aborted {
            break;
        }
        if score >= probcut_beta {
            cutoff = Some((mv, score));
            break;
        }
    }
    self.keys.pop();
    if self.aborted {
        return 0;
    }
    if let Some((mv, score)) = cutoff {
        self.note(Note::ProbCutCutoff(mv));
        // What ProbCut found is a lower bound from its shallower search.
        if self.techniques.tt {
            let stored = score_to_tt(score, ply);
            self.tt.store(
                pos.hash(),
                Some(mv),
                stored,
                probcut_depth + 1,
                Bound::Lower,
                tt_pv,
            );
        }
        return score;
    }
}
```

ProbCut asks about `probcut_beta`, beta plus `probcut_margin` (100 centipawns), and it keeps the cost down three ways:

- **Captures only.** A move that beats beta by a pawn in a shallow search is usually one that wins material. Quiet moves are left to the full search.
- **Only captures that could make it.** A capture's static exchange evaluation (chapter 14) has to make up what the static evaluation is short of `probcut_beta`. In a position evaluated at beta, that is a

capture winning a pawn or more. Ten points below, it has to win 110.

- **The quiescence search first.** Each capture gets a quiescence search against `probcut_beta`, which costs little. Only if that reaches the margin does the capture get the real test, a null-window search `probcut_reduction` (4) plies shallower than the node.

A capture that passes the test cuts the node off. The cutoff is stored in the transposition table as a lower bound, at the depth of ProbCut's search plus one: one ply more because the capture was made before that search.

The conditions

- **Off the principal variation and out of check**, like the pruning of chapters 12 to 14.
- **From depth 5** (`probcut_min_depth`). At depth 4, ProbCut's own search would be only the quiescence search.
- **Not near a mate score.** A margin added to a mate score is no longer a score.
- **Not when the hash table already knows better.** If the table holds an exact score or an upper bound for this position, from a search at least as deep as ProbCut's own, and that score is below `probcut_beta`, ProbCut's search would probably fail too, so it isn't tried. A lower bound below `probcut_beta` doesn't say the true score is below it, so it doesn't stop the test.
- **Not in a singular extension's search**, which leaves the hash move out and asks a different question.

Where it comes from

Michael Buro invented ProbCut for Othello, in his thesis (1994) and in "ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm" (ICCA Journal, 1995). His version is statistical. It measures, over many positions, how a deep search's score relates to a shallow one's, and sets the margin from the spread, so that a cut is wrong with a chosen, small probability. It worked very well in Othello. In chess, attempts in Crafty in 2003 and 2004 found it no better, or worse.

The version that works in chess is Gary Linscott's for Stockfish (GPL-3.0, 2013): a fixed margin, captures only, and a search a few plies shallower. cb2026's follows that form and adds the quiescence search first and the hash table check, as later Stockfish versions do. The code is cb2026's own. The three numbers, 5, 100 and 4, were set by hand, not tuned.

What it changes

The bench is 9.6% smaller with it: 963,637 nodes instead of 1,065,754. With `set probcut off` it is `main`'s exactly. Three alternating bench runs gave 1.55 to 1.62 million nodes a second for `main` and 1.63 to 1.65 million with ProbCut.

The `stats` command shows how often it fires. After a search to depth 12:

Position	ProbCut tried	Cut off	Nodes with it ÷ without (`ab probcut 12`)
Kiwipete	3,677 nodes	2,512 (68%)	1.00
An Italian Game middlegame	947	334 (35%)	1.01
A queen's pawn middlegame	1,209	424 (35%)	1.02

On those three, ProbCut cuts off hundreds or thousands of nodes, yet the tree comes out about the same size. Presumably its own searches cost about what its cutoffs save there; on the bench positions it saves 9.6%. The games (below) found it weaker all the same.

Pitfalls

- **The margin is a bet.** Too small, and ProbCut cuts nodes whose deep score would have fallen below beta. Too large, and it seldom fires and only costs time. The right size depends on the evaluation: a pawn here, on cb2026's centipawn scale.
- **Cost at nodes that don't cut off.** Each capture tried costs a quiescence search, and some a shallower search too. Where no capture reaches the margin, that work is wasted. The static exchange filter and the quiescence search first keep it small.
- **Mate scores.** A margin added to a score near mate makes no sense, so ProbCut isn't tried there. A mate it finds through the capture is a real one, so it is returned as it is.
- **The hash table.** A ProbCut result is a lower bound from a shallower search, stored at that depth, so a later search that needs

more depth searches the node again.

What it's worth

Test	Result
ProbCut vs `main` without it, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−10.2 ± 9.1 Elo** over 2,593 games (+562 −638 =1393): failed (LLR −2.95), cjus/chessengine#117, so it is off by default

Like capture history (chapter 10.2), it made the bench smaller and the engine weaker. Why isn't measured: the cutoffs it makes may be wrong too often at a margin of a pawn, or the searches it spends where no capture reaches the margin may cost more in games than in the bench. The switch and the three parameters stay. Untested candidates for another try:

- **A larger margin**, 150 or 200 centipawns: fewer cutoffs, and fewer wrong ones.
- **A deeper start**, `probcut_min_depth` 6 or 7, so that ProbCut's own search is deeper and its score closer to the full search's.
- **The long time control**, 40+0.4, where searches are deeper and more nodes are at the depths ProbCut works at.

Further reading

- Chess Programming Wiki: ProbCut, with Multi-ProbCut and the Crafty experiments.
- Michael Buro, "ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm", ICCA Journal 18(2), 1995.
- Stockfish's source (GPL-3.0), `search.cpp`, "ProbCut".

Late move reductions and pruning

Stage 13 · tag `stage-13-lmr` *· code in* `crates/engine/src/search.rs`

Good move ordering (chapter 10) means a cutoff almost always comes from one of the first few moves searched. So most of the effort at a node goes into proving that the *other* moves are no better, and those moves are usually poor. This chapter spends less on them. It is the single biggest step in depth an engine takes.

Late move reductions

Late move reductions (LMR) search the late moves at a reduced depth. The later a move comes in the ordering and the deeper the search, the bigger the reduction:

```
reduction = 0.75 + ln(depth) × ln(move number) / 2.25      (rounded down)
```

At depth 10, the 20th move is searched 3 plies less deep. At depth 3, the third move loses 1 ply. The table is computed once:

```
/// The late move reduction for each depth and move number:
/// `base + ln(depth) × ln(move number) / divisor`, rounded down.
fn lmr_table(p: &SearchParams) -> Vec<[u8; LMR_TABLE_SIZE]> {
    let base = f64::from(p.lmr_base) / 100.0;
    let divisor = f64::from(p.lmr_divisor) / 100.0;
    (0..LMR_TABLE_SIZE)
        .map(|depth| {
            let mut row = [0; LMR_TABLE_SIZE];
            for (index, r) in row.iter_mut().enumerate().skip(1) {
                if depth > 0 {
                    let value = base + (depth as f64).ln() * (index as f64).ln() /
divisor;
                    *r = value.max(0.0) as u8;
                }
            }
            row
        })
        .collect()
}
```

A reduced search is a guess, so it has a safety net. If the reduced, null-window search says the move beats alpha after all, the move is searched again at full depth. If it still beats alpha and lies within the window, PVS searches it once more with the full window:

```
/// The score of the move that led to `child` (the last move on the path), from the
/// mover's side, searched to `depth`.
///
/// The first move gets the full window. With PVS on, a later move gets a null window
/// first, at a reduced depth if LMR chose a reduction; if it beats `alpha` it is
searched
/// again at full depth, and then with the full window if it is inside it.
///
/// `cut_node` is the parent's flag. The child of a null-window search is an expected
cut
/// node if its parent is an expected all-node, and the other way round; a child with
the
/// full window is a principal variation node, never a cut node. A late move's reduced
/// search counts as a cut node whatever the parent, as in Stockfish (GPL-3.0).
#[allow(clippy::too_many_arguments)]
fn search_move(
    &mut self,
    child: &Position,
    depth: u32,
    alpha: i32,
    beta: i32,
    how: MoveSearch,
    cut_node: bool,
    pv: &mut Vec<Move>,
) -> i32 {
    let pvs = self.techniques.pvs && self.techniques.alpha_beta;
    let null_window = beta - alpha == 1;
    let reduction = match how {
        MoveSearch::Later { reduction } if pvs => reduction,
        _ => return self.search_child(child, depth, alpha, beta, null_window &&
!cut_node, pv),
    };
    let mv = self.path.last().copied().flatten();
    if reduction > 0
        && let Some(mv) = mv
    {
        self.note(Note::Reduced(mv, reduction));
    }
    let mut score = self.search_child(
        child,
        depth - reduction,
        alpha,
        alpha + 1,
        reduction > 0 || !cut_node,
        pv,
    );
    if score > alpha && reduction > 0 && !self.aborted {
        if let Some(mv) = mv {
            self.note(Note::ReductionResearch(mv));
        }
        score = self.search_child(child, depth, alpha, alpha + 1, !cut_node, pv);
    }
    if score > alpha && score < beta && !self.aborted {
        if let Some(mv) = mv {
            self.note(Note::PvsResearch(mv));
        }
```

```
        score = self.search_child(child, depth, alpha, beta, false, pv);
    }
    score
}

/// The score of `child` for the side that moved there, searched to `depth` within the
/// mover's window `[alpha, beta]`.
fn search_child(
    &mut self,
    child: &Position,
    depth: u32,
    alpha: i32,
    beta: i32,
    cut_node: bool,
    pv: &mut Vec<Move>,
) -> i32 {
    let ply = self.path.len() as u32;
    let score = -self.negamax(child, depth, ply, -beta, -alpha, cut_node, pv);
    self.note(Note::Exit { score: -score });
    score
}
```

Some moves are never reduced, because they are too often important:

- the first move (usually the TT move);
- captures and promotions;
- moves that give check, and every move when in check;
- moves at depths below 3, or among the first 3 moves.

On the principal variation, where the exact score matters, reductions are one ply smaller.

```
let how = if index == 0 {
    MoveSearch::First
} else if self.techniques.lmr
    && ordinary
    && depth >= p.lmr_min_depth
    && index as u32 >= p.lmr_min_moves
{
    let row = &self.lmr[(depth as usize).min(LMR_TABLE_SIZE - 1)];
    let mut reduction = i32::from(row[index.min(LMR_TABLE_SIZE - 1)]);
    // Reduce the principal variation less: its exact score matters.
    if pv_node {
        reduction -= 1;
    }
    // Reduce more at an expected cut node: one move should refute it, and the late
    // ones seldom matter.
    if self.techniques.cut_node_reductions && cut_node {
        reduction += p.lmr_cut_node;
    }
    // Reduce more when many moves at this ply have been refuted: this node is
    // probably failing low, and its late moves along with it.
    if self.techniques.cutoff_reductions
        && self.cutoffs[ply as usize + 1] > p.lmr_cutoff_count
    {
        reduction += p.lmr_cutoff_plies;
        // More at an expected all-node, as in Stockfish 17.
        if !pv_node && !cut_node {
            reduction += p.lmr_cutoff_all_node_plies;
        }
    }
    // Reduce more when the hash move is a capture: it is probably the best move,
    // and a quiet move this late seldom beats it. Less (by default not at all) on
    // the principal variation, whose exact score matters.
    if self.techniques.tt_capture_reductions && tt_move.is_some_and(is_noisy) {
        reduction += if pv_node {
            p.lmr_tt_capture_pv
        } else {
            p.lmr_tt_capture
        };
    }
    // Reduce less at an expected all-node: every move has to fail low for its
    // score to hold, so a late move that is wrongly reduced into failing low
    // makes the parent's move a refutation it isn't.
    if self.techniques.all_node_reductions && !pv_node && !cut_node {
        reduction -= p.lmr_all_node;
    }
    // Reduce less at a node that has been on the principal variation: it may be
    // again.
    if self.techniques.tt_pv && tt_pv && !pv_node {
        reduction -= p.lmr_tt_pv;
    }
    // Reduce a move with a good history less: it has caused cutoffs elsewhere. And
    // one with a bad history more.
    if self.techniques.history_lmr {
        let score = self
            .ordering
            .quiet_score(pos, mv, &previous, &self.techniques);
```

```
        reduction -= p.lmr_history_adjustment(score);
    }
    MoveSearch::Later {
        // Never reduce into the quiescence search.
        reduction: reduction.clamp(0, depth as i32 - 2) as u32,
    }
} else {
    MoveSearch::Later { reduction: 0 }
};
```

(The history adjustment came later, in chapter 14.1.)

Late move pruning

Near the leaves, LMR's reductions would leave little depth to search. **Late move pruning** (LMP) goes one step further. At depth 3 or less, outside the principal variation, once `3 + depth²` quiet moves have been tried, the remaining quiet moves are skipped entirely: 4 at depth 1, 7 at depth 2, 12 at depth 3.

```
// Late move pruning: near the leaves, after enough quiet moves, skip the rest
// (once a move has shown we aren't simply getting mated).
if self.techniques.late_move_pruning
    && ordinary
    && !pv_node
    && depth <= p.lmp_max_depth
    && best > -MATE + MAX_PLY as i32
    && quiets_tried.len() as u32
        >= (p.lmp_base + depth * depth) * if improving { p.lmp_improving } else { 1 }
{
    self.note(Note::LateMovePruning(mv));
    continue;
}
```

Pruning only starts once some move has been found that doesn't lose to a mate. Otherwise a side that is under a mating attack could prune the one quiet move that defends.

What it saves

Six positions to depth 8, as in chapters 10 to 12:

Setting	Nodes	vs both on
Both on	1,389,854	—
`set lmr off`	3,814,336	2.7×
`set lmp off`	2,488,721	1.8×
Both off	9,996,424	7.2×

The tree is seven times smaller. From the starting position cb2026 now reaches depth 16 in about two seconds. At stage 9 the same time bought depth 9. The depth numbers are no longer comparable with the early stages: a "depth 16" search with LMR has many lines searched far less deeply than 16 plies.

Pitfalls

- **Reductions that are never verified.** Without the full-depth re-search, LMR would silently miss every late move that turns out to be good. The re-search is what makes it safe enough.
- **Reducing the wrong moves.** Captures, checks and evasions often change the evaluation sharply. Reducing them hides tactics.
- **Ordering matters more than ever.** A good move ordered late now gets a shallower search. Every improvement to ordering (chapter 10, and better history in later stages) pays off twice.
- **Tuning.** The 0.75 and 2.25 in the formula, the 3 move and depth thresholds and LMP's `3 + depth²` are reasonable starting values, not measured optima. Each is a named parameter in `SearchParams`, ready for tuning.

What it's worth

Test	Result
Stage 13 vs stage 12, 8+0.08, SPRT [0, 5]	+73 −5 =35 over 113 games: passed (about +242 ± 59 Elo)

Further reading

- Chess Programming Wiki: Late Move Reductions, Futility Pruning: Move Count Based Pruning, Reductions.
- Tord Romstad, "An Introduction to Late Move Reductions" (2006): the article that made LMR widely known.

History pruning

Milestone M5, after stage 20 · on `main`, no stage tag · code in `crates/engine/src/search.rs`

Late move pruning (chapter 13) skips quiet moves near the leaves by how late they come: at depth 3, after twelve quiet moves, the rest are skipped. It doesn't look at the moves themselves. **History pruning** does: near the leaves, it skips a quiet move whose history says it keeps failing where other moves cut off. The worse the history, the deeper it may prune.

The history tables (chapters 10 and 10.1) already order the quiet moves, and history in late move reductions (chapter 14.1) reduces a move with a bad history more. History pruning takes the last step and doesn't search such a move at all. Like chapter 14.1's techniques, it came after stage 20, so no stage tag contains it. It measured weaker, so it is **off by default**; `set histprune on` switches it on.

The test

```
// History pruning: near the leaves, skip a quiet move whose history says it keeps
// failing where other moves cut off, by more the deeper the node. As in Stockfish
// (GPL-3.0), which scores the move by continuation history alone.
if self.techniques.history_pruning
    && ordinary
    && !pv_node
    && depth <= p.history_pruning_max_depth
    && best > -MATE + MAX_PLY as i32
    && self
        .ordering
        .quiet_score(pos, mv, &previous, &self.techniques)
        < -p.history_pruning_margin * depth as i32
{
    self.note(Note::HistoryPruning(mv));
    continue;
}
```

A move is skipped when its quiet-move score, history plus continuation history (the score that orders the quiet moves), is below `-history_pruning_margin` (16) per ply of depth, up to `history_pruning_max_depth` (4). A cutoff at depth 4 gives the quiet moves tried before it a penalty of 16 each, so at depth `d` the move is pruned when its score is worse than `d` such penalties. Like late move pruning and futility pruning, it never applies:

- on the principal variation, whose exact score matters;
- in check, or to a move that gives check;
- before a move has been searched: the first move never goes, and if every move searched so far is mated, nothing is pruned, so a mate can't be missed by skipping the only defence.

Which score

Stockfish (GPL-3.0) prunes by continuation history alone. In cb2026 that score turned out to be a poor sign. The test was over the positions of one game: each searched to depth 10, one after another in

the same shell, so that the tables fill as they do in a game. At the nodes where history pruning could apply, each quiet move searched was counted by its score and by whether it raised alpha:

Depth	All	Quiet-move score below −16 × depth	Quiet-move score 0	Quiet-move score above 0	Continuation history below 0	Continuation history 0
1	6.1%	6.0%	1.0%	7.0%	3.7%	1.6%
2	4.6%	2.9%	1.3%	6.6%	2.4%	1.2%
3	2.2%	0.5%	0.7%	5.0%	1.3%	0.4%
4	0.7%	0.3%	0.3%	3.0%	0.4%	0.1%

A move with a negative continuation score raised alpha two to three times as often as one with no score at all, at every depth. A score of zero mostly means the move was never tried in this context. A negative one means it was tried at a node where another quiet move then cut off: only those nodes hand out penalties. So negative scores pile up at nodes that cut off, where any move is more likely to fail high, and the sign says as much about the node as about the move.

The quiet-move score, which adds the history table, does better at depths 3 and 4. There, a move below the margin raised alpha about as rarely as a move with no score, against 3% to 5% for a move with a positive score. At depths 1 and 2 it still raised alpha often. But those are late in the order there, and late move pruning skips most of them anyway. The margin and the depth limit were set from these measurements, not tuned.

Where it comes from

The Chess Programming Wiki traces the idea to an option in a 2005 version of Fruit, by Fabien Letouzey: *history leaf pruning*, which skipped moves near the leaves below a history threshold. Stockfish (GPL-3.0) uses the form here: a threshold that grows with the depth, on the continuation history of the moves one, two and four plies back. cb2026 keeps only two of those and adds the history table. The code is cb2026's own.

What it changes

With it on, the bench is 10.0% smaller: 1,022,919 nodes instead of 1,136,500. Off, it is `main`'s exactly. Three alternating pairs of bench runs gave speed ratios of 0.98 to 1.00: the score is a few table reads, and the history tables are in the cache.

The six positions of chapters 10 to 14, each searched from empty tables:

Depth	`set histprune off`	History pruning	Moves it skipped
9	447,273	425,452 (−5%)	16,720
10	703,714	580,284 (−18%)	38,355
11	885,651	830,827 (−6%)	90,711

Over the game's 58 positions at depth 10, with the tables carried over, it skipped 261,901 moves, against 1,494,283 skipped by late move pruning and 378,100 by futility pruning. Even so, the trees came out about the same size: 5% more nodes in all, smaller in 30 of the 58 positions. Other limits did no better: up to depth 6, or with half the margin, the average change per position (a geometric mean) went

from 10% smaller to 22% bigger. Pruning moves reshuffles the tree as much as it shrinks it, and tree sizes have not predicted the games here (chapters 10.2, 12.1 and 14.1).

Pitfalls

- **The scores are small.** A cutoff's bonus is `depth²`, so most scores stay within a few hundred of zero, far from the limit of 16,384. A margin copied from an engine with other bonuses would prune nothing, or everything. Stockfish's bonus is up to 1,561 a cutoff, and its threshold is 3,832 a ply.
- **A bad score isn't always a bad move.** The table in "Which score" shows a negative continuation score marking moves that matter more often than unscored ones.
- **The tables have to fill.** A short search from empty tables has few scores that low: the unit test, eight plies with a material-only evaluation, needs a margin of 1 to see any pruning. The six positions at depth 9 already prune thousands of moves.
- **Node counts at one depth swing.** With history pruning on, IIR's unit test found IIR's tree 8% bigger on Kiwipete at depth 8, though IIR saves nodes at depths 7, 9 and 10 and over the six positions. The test now searches to depth 9.

What it's worth

Test	Result
History pruning vs `main` without it, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−5.3 ± 7.1 Elo** over 4,548 games (+1087 −1156 =2305): failed (LLR −2.97), cjus/chessengine#140, so it is off by default

Stockfish's own estimate for its version is about 2 Elo, in an engine where many techniques already overlap, so a small result was expected; this one leans the other way. Like capture history, ProbCut and others (chapters 10.2, 12.1 and 14.1), it made the bench smaller and the engine no stronger. The switch and both parameters stay. Untested candidates for another try:

- **From depth 3 only.** At depth 1 the moves it skips raised alpha as often as quiet moves in general, and at depth 2 more than half as often ("Which score"), so pruning them there may cost the most. A `history_pruning_min_depth` would be a new parameter.
- **A larger margin**, 32 a ply: a quarter as many moves skipped over the game's positions.
- **The long time control**, 40+0.4, where the history tables are fuller and the searches deeper.

Further reading

- Chess Programming Wiki: History Leaf Pruning, History Heuristic.
- Stockfish's source (GPL-3.0), `search.cpp`, "Continuation history based pruning".

Futility, razoring and SEE pruning

Stage 14 · tag `stage-14-pruning` *· code in* `crates/engine/src/search.rs` *and* `see.rs`

Chapters 12 and 13 cut whole subtrees (null move) and searched late moves less deeply (LMR). This chapter adds the last family of selective techniques in cb2026's hand-crafted search: pruning near the leaves, decided by how far the static evaluation is from the window. Each one is a small, cheap guess. Together they make the tree nearly five times smaller.

They all share the same ground rules. None of them applies on the principal variation, in check, or (for the move-level ones) before a move has been found that doesn't lose to a mate.

Static exchange evaluation

Several of these techniques need to know whether a move loses material, so first a tool: **static exchange evaluation** (SEE). It plays out every capture and recapture on one square, each side using its least valuable piece first and free to stop when continuing would lose more. It runs without any search, so it is cheap enough to call on every capture.

```
/// The material `mv` wins (or loses, if negative) for the side making it, once the
exchange
/// on its destination square is played out. A quiet move scores 0 or less: less if
the piece
/// can be taken for free or for less than it is worth.
pub fn see(pos: &Position, mv: Move) -> i32 {
    let to = mv.to();
    let from = mv.from();
    let value = |pt: PieceType| SEE_VALUES[pt.index()];

    // What the first move captures, and the piece that will stand on the square after
it.
    let mut occupied = pos.occupied().without(from);
    let first_gain = match mv.kind() {
        MoveKind::EnPassant => {
            let captured = Square::new(to.file(), from.rank());
            occupied = occupied.without(captured);
            value(PieceType::Pawn)
        }
        MoveKind::Castle => return 0,
        _ => pos.piece_at(to).map_or(0, |p| value(p.piece_type)),
    };
    let mover = pos.piece_at(from).expect("a piece on the from-square");
    let mut on_square = match mv.promotion() {
        Some(pt) => pt,
        None => mover.piece_type,
    };
    let promotion_gain = mv
        .promotion()
        .map_or(0, |pt| value(pt) - value(PieceType::Pawn));

    // gains[i]: material for the side that made capture i, if capture i is the last
one.
    let mut gains = vec![first_gain + promotion_gain];
    let mut side = !mover.color;
    let mut attackers = pos.attackers_to(to, occupied) & occupied;
    loop {
        let ours = attackers & pos.pieces_of_color(side);
        let Some((piece_type, square)) = least_valuable(pos, ours) else {
            break;
        };
        // This side captures the piece standing on the square.
        let previous = *gains.last().expect("at least one entry");
        gains.push(value(on_square) - previous);
        on_square = piece_type;
        occupied = occupied.without(square);
        // Recompute, so sliders behind the piece that just moved join in (x-rays).
        attackers = pos.attackers_to(to, occupied) & occupied;
        side = !side;
        // A king can't capture into an attacked square: if it just did and the other
side
        // has an attacker left, the king's capture was illegal. Drop it and stop.
        if piece_type == PieceType::King && !(attackers &
pos.pieces_of_color(side)).is_empty() {
            gains.pop();
            break;
```

```
        }
    }

    // Work backwards: each side may decline to continue the exchange.
    while gains.len() > 1 {
        let last = gains.pop().expect("more than one entry");
        let before = gains.last_mut().expect("at least one entry");
        *before = (*before).min(-last);
    }
    gains[0]
}
```

The `gains` list records, after each capture, what the capturing side would have gained if the exchange stopped there. Walking it backwards lets each side choose between stopping and going on. When a piece leaves the square's line, the attackers are recomputed with it removed from the occupancy, which reveals **x-ray** attackers: the rook behind a rook, the bishop behind a queen.

SEE improves ordering too. MVV-LVA put QxP before every quiet move even when the pawn is defended. Now captures that lose material are searched last, after the quiet moves.

Reverse futility pruning

If the static evaluation is above beta by a margin that grows with depth (80 centipawns per ply, up to depth 7), the side to move is probably doing so well that the opponent can't bring the score back down within the remaining depth. The node returns the static evaluation straight away. This is also called the **static null move**: it is null-move pruning (chapter 12) without the search.

```
// Reverse futility pruning: far enough above beta, assume a search would stay there.
// An improving side gets a smaller margin, but never none at all.
let rfp_margin = p.rfp_margin * depth as i32 - if improving { p.rfp_improving } else {
0 };
if self.techniques.reverse_futility
    && selective
    && depth <= p.rfp_max_depth
    && static_eval - rfp_margin >= beta
{
    self.note(Note::ReverseFutility);
    return static_eval;
}
```

The `improving` part of the margin came later, in chapter 14.1.

Razoring

The mirror image: at depth 1 or 2, if the static evaluation is far *below* alpha (300 centipawns per ply), quiet moves are unlikely to rescue the position. Only a capture could, so the node asks the quiescence search. If even the captures can't reach alpha, the node is abandoned.

```
// Razoring: far below alpha near the leaves, only captures could save the node.
if self.techniques.razoring
    && self.techniques.quiescence
    && selective
    && depth <= p.razor_max_depth
    && static_eval + p.razor_margin * (depth as i32) < alpha
{
    let score = self.quiescence(pos, ply, alpha, alpha + 1, pv);
    if score <= alpha {
        self.note(Note::Razoring);
        return score;
    }
    pv.clear();
}
```

Futility pruning

The same idea per move. Within 6 plies of the horizon, if the static evaluation plus a generous margin (100 + 100 per ply) still doesn't reach alpha, a quiet move is "futile". It changes no material, so it can

hardly gain that much, and it is skipped without being searched. Captures and checks are still searched.

```
// Futility pruning: a quiet move can't lift a hopeless static evaluation to alpha.
if self.techniques.futility
    && ordinary
    && !pv_node
    && depth <= p.futility_max_depth
    && best > -MATE + MAX_PLY as i32
    && static_eval + p.futility_base + p.futility_per_depth * depth as i32 <= alpha
{
    self.note(Note::Futility(mv));
    continue;
}
```

SEE pruning

Within 6 plies of the horizon, moves that lose material by SEE are skipped: quiet moves that put a piece where it can be taken for more than 60 centipawns per ply of depth, and captures that lose more than 100 per ply. In the quiescence search every losing capture is skipped: captures there are meant to resolve the position, and a capture that loses material can't improve on standing pat.

```
// SEE pruning: near the leaves, skip moves that lose material outright.
if self.techniques.see_pruning
    && selective
    && index > 0
    && !gives_check
    && depth <= p.see_max_depth
    && best > -MATE + MAX_PLY as i32
{
    let margin = if quiet {
        p.see_quiet_margin
    } else {
        p.see_capture_margin
    };
    if see(pos, mv) < -margin * (depth as i32) {
        self.note(Note::SeePruning(mv));
        continue;
    }
}
```

Delta pruning

In the quiescence search, a capture is skipped if even winning the captured piece for free, plus a 200-centipawn safety margin, would leave the score at or below alpha. A side that is a rook down won't be saved by winning a pawn.

What it saves

The six positions at depth 8, as in the earlier chapters:

Setting	Nodes	vs all on
All on	300,790	—
`set rfp off`	323,534	+8%
`set razoring off`	306,230	+2%
`set futility off`	314,032	+4%
`set seeprune off`	583,815	+94%
`set delta off`	343,890	+14%
`set seeorder off`	266,118	−12%
All six off	1,389,854	+362%

SEE pruning, mostly in the quiescence search, does the most. Ordering losing captures last *costs* nodes on these six positions. Moving a losing capture behind the quiet moves isn't always right: some of those captures are sacrifices that work. Whether it's worth keeping is for the SPRT to say, not a node count.

The bench now searches to depth 10 instead of 4: at depth 4 the pruning left too little of the tree to be a useful fingerprint.

Pitfalls

- **Margins.** Every technique here trades accuracy for depth through a margin. Too small, and the engine prunes good moves; too large, and it prunes nothing. The values here are common starting points, and stage 16 and later tuning will revisit them.
- **Mates.** Pruning by static evaluation is blind to mating attacks. That's why none of it applies in check, and the move-level pruning waits until a move has been found that doesn't lose to a mate.
- **SEE's blind spots.** SEE ignores pins and checks, and it only looks at one square. A capture it calls losing can win if the recapturing piece is pinned.
- **Interactions.** These techniques overlap with each other and with null move and LMR: turning one off shows only what it adds *on top of* the others.

What it's worth

Test	Result
Stage 14 vs stage 13, 8+0.08, SPRT [0, 5]	+174 −55 =99 over 328 games: passed (about +132 ± 33 Elo)

Further reading

- Chess Programming Wiki: Static Exchange Evaluation, Reverse Futility Pruning, Razoring, Futility Pruning, Delta Pruning.
- Ernst Heinz, "Extended Futility Pruning", *ICCA Journal* 21 (1998).

Extensions and reductions

Milestone M5, after stage 20 · on `main`, no stage tag · code in `crates/engine/src/search.rs`

Chapters 12 to 14 made the tree smaller by guessing which moves and positions don't need a full search. This chapter moves depth in both directions. An **extension** searches a move *deeper* than its siblings; a **reduction** searches a whole node *shallower* (chapter 13's late move reductions work on single moves). One fact about a position, whether the side to move is **improving**, changes how boldly the pruning of chapters 13 and 14 guesses. And one fact about a move, its **history** score, changes how much late move reductions take off it.

The techniques in this chapter came after stage 20, as strength work towards version 1.0 (milestone M5), so no stage tag contains them: read this chapter against `main`. Like everything before them, each can be switched off and on: `set checkext`, `set singular`, `set doubleext`, `set multicut`, `set negext`, `set iir`, `set improving` and `set histlmr`.

Check extensions

A move that gives check is searched one ply deeper than the moves around it. A check leaves the opponent few replies, so the extra ply costs little, and checks are where the horizon hides mates and lost

pieces: a mating attack is a run of checks, and stopping halfway through one evaluates a position whose point hasn't happened yet.

```
// Check extension: a move that gives check is searched one ply deeper. So is a
// singular hash move (above), and one singular by a wide margin a ply more.
let doubled = Some(mv) == double_move;
let extension = u32::from(
    (self.techniques.check_extension && gives_check && ply < MAX_PLY / 2)
        || Some(mv) == singular_move,
) + u32::from(doubled);
// A hash move that isn't singular but beats beta is searched shallower (above).
let shallower = if Some(mv) == shallower_move {
    p.negative_extension_plies
} else {
    0
};
```

The child is then searched at `depth - 1 + extension`. (`singular_move` is the next section's.) The `ply < MAX_PLY / 2` condition matters: an extension that can apply at every ply never reaches the horizon, and a long series of checks would grow the search without bound.

Philidor's legacy shows what it buys. White mates in four, and every move is a check: 1. Nf7+ Kg8 2. Nh6+ Kh8 3. Qg8+ Rxg8 4. Nf7#.

```
> fen 4r2k/6pp/8/3Q2N1/8/8/6PP/6K1 w - - 0 1
> ab checkext 8
```

With the extension the engine sees the mate at depth 8; without it, it needs a deeper search.

cb2026 ships with it off. Extending every check made the bench search about 24% bigger, and an SPRT with the extension on against the same engine without it found no gain (see "What it's worth" below). Most of what it finds is found anyway: late move reductions never reduce a move that gives check (chapter 13), and the quiescence search resolves the captures. Strong engines extend more selectively, for example only checks that don't lose material, or only **singular**

moves: a move that a reduced search, with that move left out, shows to be much better than every alternative. The switch stays, so the experiment can be repeated.

Singular extensions

The check extension extends a kind of move. A **singular extension** extends a move because of how it compares with the others: the hash move, if it is much better than every alternative. A move like that is probably forced, a recapture or the one defence to a threat, and a mistake anywhere in its line costs the most, so it is worth a ply more.

Finding out costs a search. Before the moves are searched, the node is searched once more, at half the depth, **without the hash move**, with a null window just below the hash move's score: `singular_beta`, the hash table's score less 3 centipawns per ply of depth. If no other move reaches `singular_beta`, the hash move is singular and is searched with one ply more.

```
// Singular extension: is the hash move much better than every other move? Search the
// node again, at half the depth and without the hash move, against a window a little
// below the hash move's score. If no other move reaches it, the hash move is probably
// forced, and it is searched a ply deeper.
let mut singular_move = None;
let mut double_move = None;
let mut shallower_move = None;
if self.techniques.singular_extension
    && depth >= p.singular_min_depth
    && ply < 2 * self.root_depth
    && let Some(entry) = tt_entry
    && let Some(hash_move) = tt_move
    && entry.bound != Bound::Upper
    && u32::from(entry.depth) + p.singular_tt_depth_margin >= depth
{
    let tt_score = score_from_tt(entry.score, ply);
    if tt_score.abs() < MATE - MAX_PLY as i32 {
        let singular_beta = tt_score - p.singular_margin * depth as i32;
        self.note(Note::SingularSearch(hash_move));
        self.excluded[ply as usize] = Some(hash_move);
        let mut singular_pv = Vec::new();
        let score = self.negamax(
            pos,
            (depth - 1) / 2,
            ply,
            singular_beta - 1,
            singular_beta,
            cut_node,
            &mut singular_pv,
        );
        self.excluded[ply as usize] = None;
        self.note(Note::Exit { score });
        if self.aborted {
            return 0;
        }
        if score < singular_beta {
            self.note(Note::SingularExtension(hash_move));
            singular_move = Some(hash_move);
            // Double extension: nothing else comes close even well below
            // `singular_beta`, so the hash move is searched a second ply deeper. Only
            // off the principal variation, and only a few times along a line, so that
            // the search can't explode.
            if self.techniques.double_extension
                && !pv_node
                && score < singular_beta - p.double_extension_margin
                && self.double_extensions < p.double_extension_limit
            {
                self.note(Note::DoubleExtension(hash_move));
                double_move = Some(hash_move);
            }
        } else if self.techniques.multi_cut && singular_beta >= beta {
            // Multi-cut: the hash move beat beta, and another move beats it too even
            // at half the depth. With two moves that good, the node is a cutoff.
            self.note(Note::MultiCut);
            return singular_beta;
        } else if self.techniques.negative_extension && tt_score >= beta {
```

```
            // Negative extension: not singular, but its hash score beats beta. The
            // node will probably cut off either way, so its line needs less care.
            self.note(Note::NegativeExtension(hash_move));
            shallower_move = Some(hash_move);
        }
    }
}
```

The test needs a hash entry worth comparing against: a lower bound or an exact score, at most 3 plies shallower than the node (`singular_tt_depth_margin`, see below). It applies from depth 4 on (`singular_min_depth`), so only with a few plies of the search still to come. And no extension is made beyond twice the root depth, so that a chain of them can't grow the search without limit.

The first version started at depth 8. That is cautious: Stockfish (GPL-3.0) starts lower. Each test costs a search at half the depth, so starting lower runs many more of them, and the extensions and multi-cuts they find change the tree in both directions. The bench and the six positions of "What it saves" at depth 10, by the depth singular extensions start from:

Start from depth	Bench	Six positions at depth 10	Tests	Extended	Multi-cuts
8 (first version)	1,100,720	668,426	485	125	142
7	1,031,076	630,754	808	236	282
6 (second version)	1,131,028	647,169	1,738	708	635
5	1,134,904	722,245	3,436	1,371	1,413
4	1,343,312	750,416	6,449	3,107	2,484

Starting two plies lower runs the test more than three times as often, but the trees hardly grow, and at depth 10 they shrink: the extra tests

find multi-cuts as well as extensions, and the two pull in opposite directions. (These counts include negative extensions, below. Before them, starting from depth 6 made the depth-10 tree 4% bigger, not 3% smaller.) Node counts don't say which plays better, so depth 6, the bigger step towards Stockfish's, went to the games: about **+5 Elo** (see "What it's worth").

That suggested going lower still, and the next two-ply step is depth 4, close to where Stockfish starts. From there the trees do grow: the test runs almost four times as often as from depth 6, the bench is 19% bigger and the depth-10 tree 16%. The evidence gets thinner, too. At depth 4 the test is a search of a single ply, against an entry that may come from a search of one ply, so "nothing else comes close" means less. The games say the many extra extensions and cutoffs are worth it all the same: about **+6 Elo** more (see "What it's worth").

One effect shows already. The check extension's test position above, Philidor's legacy, is a chain of forced moves, each the only good one. With IIR off, singular extensions from depth 4 find them singular and extend them, so at depth 7 the engine finds the mate with the check extension off too; from depth 6 it doesn't. (With IIR on, the defaults, neither finds it at depth 7 or 8 without the check extension.) Where it applies, the more general extension does the special one's job. The unit test for the check extension now switches singular extensions off, as it does IIR.

The same search answers a second question for free. If another move does reach `singular_beta`, and `singular_beta` is at least beta, then two moves beat beta: the hash move, and this one even at half the depth. The node is cut off on the spot. That is **multi-cut**.

And a third. If the hash move isn't singular, but its hash score beats beta (while `singular_beta` doesn't, or multi-cut would have cut the node off), the node will most likely fail high whichever good move it searches. A cutoff needs any move that beats beta, not the best line after the hash move, so that line needs less care: the hash move is searched a ply *shallower* than usual (`negative_extension_plies`, 1). That is a **negative extension**: the same test that extends a forced move a ply reduces one with good alternatives.

How far `singular_beta` sits below the hash score is `singular_margin`: 2 centipawns per ply of depth at first, 3 now. One number moves all three outcomes. A wider margin lowers `singular_beta`, so it asks more of a singular move and fewer moves are extended. Multi-cut is pulled both ways: another move reaches the lower `singular_beta` more easily, but the lower `singular_beta` is at least beta less often. And more hash moves fall between the two, not singular but beating beta, and are searched a ply shallower. From depth 4, a margin of 3 makes the bench 18% smaller (1,098,926 nodes against 1,343,312), and over the six positions of "What it saves" at depth 10 it runs a fifth fewer tests and finds about a quarter fewer extensions and multi-cuts. A margin of 1 makes the bench 6% smaller too (1,266,996), from the other side: more extensions, but more multi-cuts as well. Node counts don't say which plays better, so 3 went to the games: about **+5 Elo** (see "What it's worth").

A margin of 4 makes the bench 8% smaller again (1,006,594), but the six positions at depth 10 grow by 3%: it extends 4% fewer hash moves than 3 and makes 13% more multi-cuts. That is the first pull winning: from 3 to 4, multi-cuts went up, where from 2 to 3 they went down. In

the games, 4 was no better than 3, about **−2 Elo**, so the margin stays at 3 (see "What it's worth").

The test's other number is how deep the hash entry has to be. With `singular_tt_depth_margin` at 3, as in Stockfish (GPL-3.0), an entry counts if it is at most 3 plies shallower than the node, so at depth 4 an entry from a one-ply search will do. cb2026 stores no entries from the quiescence search, so every entry has a depth of at least 1, and a margin of 4 changes nothing at depth 4: from depth 5 on, it lets in entries a ply shallower than before. That makes the bench 3.5% smaller (1,028,535 nodes against 1,065,754), and the six positions of "What it saves" at depth 10 9% smaller (574,643 against 634,430). They run 3% more tests and extend 4% fewer hash moves, but make 11% more multi-cuts and search 19% more hash moves shallower: a score from a shallower search is a rougher guess, and more of the extra tests end in a cutoff or a reduction than in an extension. A smaller tree is no promise. A `singular_margin` of 4 made the bench 8% smaller and failed, and so did capture history, ProbCut and the first version of correction history's piece tables (chapters 10.2, 12.1 and 14.2). In the games, 4 was worse than 3, about **−7 Elo**, so the margin stays at 3 (see "What it's worth").

The search answers one more question. Its score is the best any other move managed, so it says not only whether the hash move is singular but by how much. When no other move comes within `double_extension_margin` (50 centipawns; 20 at first, see below) of `singular_beta` either, the hash move is close to the only move: a recapture with nothing else to do, the one defence that doesn't lose a piece. A mistake in its line costs the most of all, so it is searched two plies deeper instead of one. That is a **double extension**.

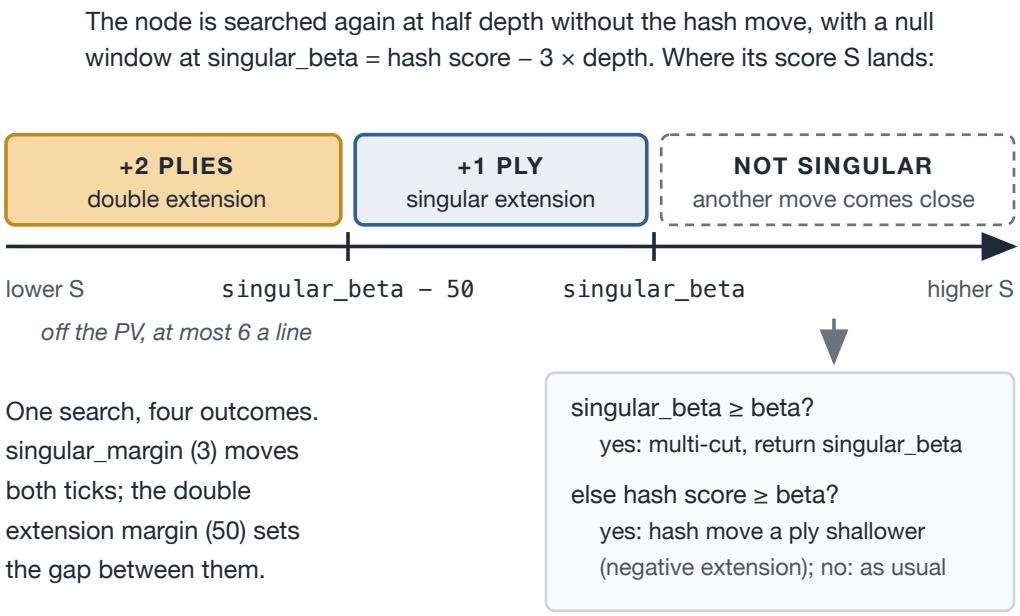

Figure 14.1.1 Where the search without the hash move scores decides between two extensions, a cutoff and a reduction

Two plies at a time feed on themselves faster than one, so double extensions have two limits of their own, on top of the limit of twice the root depth that every extension has. They apply only off the principal variation, in the null-window searches that make up most of the tree. And a line can have at most `double_extension_limit` (6) of them: the searcher counts them on the way down and takes them off on the way up.

They apply more often than the name suggests, because the search's score is usually not just under `singular_beta` but far below it. Over the six positions of "What it saves" at depth 10, seven singular hash moves in ten are extended twice, and with a margin of 200 centipawns a third still would be. With the margin at 20 the depth-10 tree is 44% bigger and the bench 14% (1,248,666 nodes against 1,098,926). A margin of 50 makes the tree 30% bigger, and 100 only 9%. At this depth the limit hardly matters: at a margin of 50, a limit of

12 gives the same tree as 6, and only a limit of 3 takes the margin of 20's growth down to 23%. The first margin, 20, was small, as in Stockfish.

At a margin of 20, double extensions had no verdict. An SPRT against the same engine without them found about +4 Elo, give or take 5, after 8,670 games (see "What it's worth"). Like the first version of history in LMR (below), that sits between the test's two hypotheses, 0 and 5 Elo, where an SPRT takes longest, so the test was stopped and double extensions were switched off by default. The candidates for another test were the ones that grow the tree less, a margin of 50 or a limit of 3, and the long time control, 40+0.4.

The second test is at a margin of 50 (cjus/chessengine#125), with double extensions on again (`set doubleext off` switches them off). Since correction history (chapter 14.2), the trees grow less than they did. At a margin of 20 the depth-10 tree is 29% bigger than without double extensions (819,182 nodes against 634,430) and the bench 12% (1,189,432 against 1,065,754). At 50 they are 11% bigger (703,714) and 7% (1,136,500). A little over half of the singular hash moves are extended twice at 50 (1,487 of 2,762), against seven in ten at 20.

At a margin of 50, double extensions passed: about **+13 Elo** after 3,167 games (see "What it's worth"), so they are on by default. The two tests aren't quite a comparison of the margins alone, since correction history came in between them, and their error bars overlap. Together they suggest, no more, that extending about half of the singular moves twice does better than extending most of them.

A margin of 100 went a step further the same way (cjus/chessengine#127). Over the six positions at depths 9 to 11 it

extends about two singular hash moves in five twice, where 50 extends a little over half. The trees don't follow in a straight line. Total nodes over the six positions:

Depth	Without double extensions	Margin 50	Margin 100
9	359,248	447,273 (+25%)	388,698 (+8%)
10	634,430	703,714 (+11%)	530,504 (−16%)
11	969,728	885,651 (−9%)	895,527 (−8%)

At depth 11 both margins make the tree smaller than without double extensions. One likely reason: a hash move searched two plies deeper comes back with a more reliable score, and fewer of the moves around it need searching again. The bench, at depth 10, is 17% smaller at 100 than at 50 (945,934 nodes against 1,136,500), which says more about depth 10 than about the margin. In the games, 100 was worse than 50, about **−7 Elo**, so the margin stays at 50 (see "What it's worth").

A limit of 3 double extensions a line instead of 6 hardly binds at a margin of 50. Over the six positions it takes the share of singular hash moves extended twice down by about 3 points at depths 9 and 10 (52% to 49%, 54% to 51%) and by 0.4 to 1.1 points at depths 11 to 13: a line rarely has more than three. It wasn't tested.

A margin of 70, between 50 and 100, was the next try (cjus/chessengine#138). The three results before it, 20 with no verdict, 50 passing against none and 100 failing against 50, put the best margin somewhere between 20 and 100. A curve through them peaks at about 60, but each point is give or take 8 Elo, and the first was measured before correction history (chapter 14.2). At 70, 45% to 53%

of the singular hash moves are extended twice at depths 9 to 12, against 52% to 57% at 50 and 60% to 63% at 35. Total nodes over the six positions:

Depth	Margin 35	Margin 50	Margin 70
9	412,163 (−8%)	447,273	352,048 (−21%)
10	661,909 (−6%)	703,714	639,141 (−9%)
11	915,464 (+3%)	885,651	841,482 (−5%)
12	1,352,793 (−13%)	1,554,248	1,216,722 (−22%)

The bench is 7.2% smaller at 70 (1,054,909 nodes against 1,136,500) and 3.4% smaller at 35 (1,098,095). Smaller trees haven't meant stronger play here before (100 made the bench 17% smaller and lost), so the games decided. In them, 70 was worse than 50 too, about **−20 Elo** over 1,407 games, so the margin stays at 50 (see "What it's worth"). That is more than 100 lost, but the error bars overlap (±12 and ±8), and an SPRT that stops early tends to overstate the difference. Together they suggest that a margin wider than 50 does worse; a narrower one, like 35, is untested.

The search without the hash move asks a different question from the node's own search, so it must not mix with it. It neither reads nor writes the transposition table (an entry about the node with all its moves would answer the wrong question, and its result would poison the entry), it tries no null move, and IIR doesn't reduce it further. If the hash move is the only legal move, the search has nothing to search: it fails low, since nothing else comes close.

Singular extensions go back to Deep Thought, the Carnegie Mellon machine that led to Deep Blue (Anantharaman, Campbell and Hsu,

1988). This version, multi-cut, and negative and double extensions are as in Stockfish (GPL-3.0); the code is cb2026's own.

Internal iterative reductions

The hash move, the best move the transposition table remembers from an earlier search of the position, is usually the best move, and searching it first is what makes alpha-beta fast (chapter 10). A node with no hash move has never been searched, every search of it failed low, or its entry has been replaced. Its move ordering will be poor, so a full-depth search of it is expensive. And nodes like that are rarely on the principal variation, which every iteration searches first and leaves in the table.

Internal iterative reduction (IIR) searches such a node, from depth 4 on, one ply shallower:

```
// Internal iterative reduction: with no hash move to try first, move ordering is poor
// and the node probably isn't important; search it a ply shallower.
let depth = if self.techniques.iir
    && tt_move.is_none()
    && excluded.is_none()
    && depth >= self.search_params.iir_min_depth
{
    depth - 1
} else {
    depth
};
```

If the node does matter, little is lost: the next iteration of iterative deepening reaches it again, now with the move this search stored, and searches it at full depth.

The older technique, **internal iterative deepening** (IID), did more work instead of less: before searching a node without a hash move, it ran a shallower search of the same node just to find a move to try

first. IIR gets that move from the next iteration instead, and is cheaper and about as good. Stockfish (GPL-3.0) has replaced IID with it.

Improving

A side whose static evaluation is higher than two plies ago, the last time it was to move, is **improving**: its moves are working, and its evaluation will probably keep rising.

```
// Improving: better than two plies ago, when this side last moved. Its evaluation
will
// probably keep rising, which changes the bets the pruning techniques make.
self.evals[ply as usize] = static_eval;
let improving = self.techniques.improving
    && !in_check
    && ply >= 2
    && static_eval > self.evals[ply as usize - 2];
```

That changes the bet reverse futility pruning makes.

Reverse futility pruning (chapter 14) bets that a node far above beta stays there. If the side to move is improving, the bet is safer, so the margin shrinks by half a ply's worth (`rfp_improving`, 40 centipawns):

```
// Reverse futility pruning: far enough above beta, assume a search would stay there.
// An improving side gets a smaller margin, but never none at all.
let rfp_margin = p.rfp_margin * depth as i32 - if improving { p.rfp_improving } else {
0 };
if self.techniques.reverse_futility
    && selective
    && depth <= p.rfp_max_depth
    && static_eval - rfp_margin >= beta
{
    self.note(Note::ReverseFutility);
    return static_eval;
}
```

Late move pruning (chapter 13) could use it too. In an improving position quiet moves are more often good, so it could try more of

them before giving up. That is the parameter `lmp_improving`, and it is 1: no change.

```
// Late move pruning: near the leaves, after enough quiet moves, skip the rest
// (once a move has shown we aren't simply getting mated).
if self.techniques.late_move_pruning
    && ordinary
    && !pv_node
    && depth <= p.lmp_max_depth
    && best > -MATE + MAX_PLY as i32
    && quiets_tried.len() as u32
        >= (p.lmp_base + depth * depth) * if improving { p.lmp_improving } else { 1 }
{
    self.note(Note::LateMovePruning(mv));
    continue;
}
```

This is the third version. **The first failed.** It shrank the margin by a whole ply, as Stockfish 11 (GPL-3.0) did. At depth 1 that left no margin at all: the node was cut off whenever its static evaluation alone reached beta, and cb2026's evaluation, unlike Stockfish's, knows nothing about pieces under attack. In an SPRT against the same engine without it, cb2026 lost about 32 Elo (see "What it's worth"). **The second failed too**, by about 18 Elo. It took half a ply off the margin, as here, so at depth 1 a node still needs 40 centipawns to spare. Both also doubled late move pruning's threshold when improving (`lmp_improving` = 2). The version above leaves that out, so it tests the margin on its own.

The third version failed as well, by about 33 Elo, so **cb2026 ships with improving off**. Taken together, the three results point at reverse futility pruning: with the margin alone smaller, the engine lost more than with the same margin and late move pruning doubled, so the late move pruning part seems to have made up some of the loss. Testing the parts one at a time is what showed that. The switch and both parameters stay, so either part can be tried again.

Both tests also ran with a bug, found afterwards: the root is searched by its own function, which didn't store its static evaluation, so every node two plies down compared its evaluation with `-INFINITY` and counted as improving. Fixing it changed the six test positions' tree by 31 nodes, so it is probably not why improving lost.

History in late move reductions

Late move reductions (chapter 13) reduce a quiet move by how late it comes in the ordering. But that order is only as good as the history table, which ranks the quiet moves, and it puts the killers and the countermove first whatever their history. A move with a high history score that still came late has caused cutoffs elsewhere, and it deserves a closer look.

So the reduction shrinks by a ply for every `lmr_history_divisor` points of the move's score, by at most two plies (`lmr_history_max`). A negative score makes it grow the same way. The score is the one that ordered the quiet moves, history plus continuation history; the first version used history alone.

```
let how = if index == 0 {
    MoveSearch::First
} else if self.techniques.lmr
    && ordinary
    && depth >= p.lmr_min_depth
    && index as u32 >= p.lmr_min_moves
{
    let row = &self.lmr[(depth as usize).min(LMR_TABLE_SIZE - 1)];
    let mut reduction = i32::from(row[index.min(LMR_TABLE_SIZE - 1)]);
    // Reduce the principal variation less: its exact score matters.
    if pv_node {
        reduction -= 1;
    }
    // Reduce more at an expected cut node: one move should refute it, and the late
    // ones seldom matter.
    if self.techniques.cut_node_reductions && cut_node {
        reduction += p.lmr_cut_node;
    }
    // Reduce more when many moves at this ply have been refuted: this node is
    // probably failing low, and its late moves along with it.
    if self.techniques.cutoff_reductions
        && self.cutoffs[ply as usize + 1] > p.lmr_cutoff_count
    {
        reduction += p.lmr_cutoff_plies;
        // More at an expected all-node, as in Stockfish 17.
        if !pv_node && !cut_node {
            reduction += p.lmr_cutoff_all_node_plies;
        }
    }
    // Reduce more when the hash move is a capture: it is probably the best move,
    // and a quiet move this late seldom beats it. Less (by default not at all) on
    // the principal variation, whose exact score matters.
    if self.techniques.tt_capture_reductions && tt_move.is_some_and(is_noisy) {
        reduction += if pv_node {
            p.lmr_tt_capture_pv
        } else {
            p.lmr_tt_capture
        };
    }
    // Reduce less at an expected all-node: every move has to fail low for its
    // score to hold, so a late move that is wrongly reduced into failing low
    // makes the parent's move a refutation it isn't.
    if self.techniques.all_node_reductions && !pv_node && !cut_node {
        reduction -= p.lmr_all_node;
    }
    // Reduce less at a node that has been on the principal variation: it may be
    // again.
    if self.techniques.tt_pv && tt_pv && !pv_node {
        reduction -= p.lmr_tt_pv;
    }
    // Reduce a move with a good history less: it has caused cutoffs elsewhere. And
    // one with a bad history more.
    if self.techniques.history_lmr {
        let score = self
            .ordering
            .quiet_score(pos, mv, &previous, &self.techniques);
```

```
        reduction -= p.lmr_history_adjustment(score);
    }
    MoveSearch::Later {
        // Never reduce into the quiescence search.
        reduction: reduction.clamp(0, depth as i32 - 2) as u32,
    }
} else {
    MoveSearch::Later { reduction: 0 }
};
```

The first version

The first version used the history score alone, and a divisor of 256. The divisor came from measuring. History scores can run from −16,384 to +16,384, but those of the moves LMR actually reduces are mostly tiny. Over a 40-move self-play game at 200 ms a move (measured before continuation history, chapter 10.1, came in):

History score of a reduced move	Share
−256 or below	almost none
−256 to −64	9%
−64 to +64	74%
+64 to +256	7%
+256 to +512	3%
+512 and above	7%

So that version reduces about one late move in ten less, and hardly ever reduces one more. The scores lean positive because the penalties go only to the quiet moves searched before a quiet move's cutoff. At a node where no quiet move cuts off, which is most of the nodes LMR works on, none of them is penalised.

Stockfish (GPL-3.0) adjusts its reductions by history too, using its history and continuation history added up, on a scale of its own; the

scale here is cb2026's.

The first version had no verdict. An SPRT against the same engine without it found no clear difference: about +2 Elo, give or take 5, after 8,910 games (see "What it's worth"). An effect that small sits between the test's two hypotheses, 0 and 5 Elo, which is where an SPRT takes longest; it could have needed tens of thousands more games, so the test was stopped. That fits the table above: the adjustment changed about one reduction in ten, by a ply or two, and almost never made one bigger.

The second version

The second version scores a move as the move picker does, history plus continuation history (`OrderingTables::quiet_score`), so the reduction and the order agree about which moves look good. Adding continuation history doesn't widen the scores much at the moves LMR reduces. Over the same kind of game, after continuation history came in (and before singular extensions):

History plus continuation history of a reduced move	Share
−256 or below	0.5%
−256 to −128	5%
−128 to +128	86%
+128 to +256	3%
+256 and above	6%

So the divisor comes down to 128. Then about one reduced move in seven changes: about 9% are reduced less and 5% more. For the first

time the adjustment works both ways: a move with a bad score, one that failed where it was tried before, is reduced a ply more.

This version passed, by about **+22 Elo** (see "What it's worth"). The first changed too few reductions to show any effect in a reasonable number of games; changing one in seven, in both directions, and with the same score that orders the moves, made the difference. cb2026 ships with it on.

Reductions at expected cut nodes

Off the principal variation, every node is searched with a null window: the question is only whether its score is above or below one value. Such a node is expected either to **fail high**, when one move should be enough to refute the opponent's last move, or to **fail low**, when every move has to be searched and none is good enough. Knuth and Moore described the kinds in their analysis of alpha-beta (1975), as types 2 and 3; Tony Marsland and Fred Popowich called them *cut nodes* and *all-nodes*. In a perfectly ordered tree they alternate: the children of a cut node are all-nodes, and the children of an all-node are cut nodes.

The search can't know a node's type, but it can pass its guess down the tree:

```
#[allow(clippy::too_many_arguments)]
fn search_move(
    &mut self,
    child: &Position,
    depth: u32,
    alpha: i32,
    beta: i32,
    how: MoveSearch,
    cut_node: bool,
    pv: &mut Vec<Move>,
) -> i32 {
    let pvs = self.techniques.pvs && self.techniques.alpha_beta;
    let null_window = beta - alpha == 1;
    let reduction = match how {
        MoveSearch::Later { reduction } if pvs => reduction,
        _ => return self.search_child(child, depth, alpha, beta, null_window &&
!cut_node, pv),
    };
    let mv = self.path.last().copied().flatten();
    if reduction > 0
        && let Some(mv) = mv
    {
        self.note(Note::Reduced(mv, reduction));
    }
    let mut score = self.search_child(
        child,
        depth - reduction,
        alpha,
        alpha + 1,
        reduction > 0 || !cut_node,
        pv,
    );
    if score > alpha && reduction > 0 && !self.aborted {
        if let Some(mv) = mv {
            self.note(Note::ReductionResearch(mv));
        }
        score = self.search_child(child, depth, alpha, alpha + 1, !cut_node, pv);
    }
    if score > alpha && score < beta && !self.aborted {
        if let Some(mv) = mv {
            self.note(Note::PvsResearch(mv));
        }
        score = self.search_child(child, depth, alpha, beta, false, pv);
    }
    score
}
```

The root and every search with the full window are principal variation nodes, never cut nodes. Every other null-window search flips its parent's guess, and a late move's reduced search counts as a cut node whatever its parent, as in Stockfish (GPL-3.0). The null

move's search flips the guess too, and the singular extension's search, of the same node, keeps it. At an expected cut node the late moves seldom matter, because one of the first should already refute the node, so late move reductions take `lmr_cut_node` (2; 1 at first) plies more off them:

```
// Reduce more at an expected cut node: one move should refute it, and the late
// ones seldom matter.
if self.techniques.cut_node_reductions && cut_node {
    reduction += p.lmr_cut_node;
}
```

The guess is a good one. Over the positions of a game, each searched to depth 10 one after another so that the tables fill as they do in a game, 88% of the expected cut nodes that reached their moves failed high, against 42% of the expected all-nodes (35% from depth 3). Nodes cut off before their moves, by the null move or reverse futility pruning, aren't counted.

Stockfish passes the same flag and adds 2 plies at cut nodes, alongside reductions that cb2026 doesn't have and that take plies off again elsewhere. cb2026 adds 1, set by hand, not tuned. The code is cb2026's own. `set cutnode off` switches it off, and gives the search without it exactly.

The bench is 5.1% smaller with it: 1,078,861 nodes instead of 1,136,500. The trees don't shrink consistently, though. Over the six positions of chapters 10 to 14 they are 6% and 10% smaller at depths 9 and 10 and 7% bigger at depth 11 (421,160, 634,668 and 946,605 nodes against 447,273, 703,714 and 885,651). Over the game's 58 positions at depth 10 they are 3.5% bigger in all, and smaller in 27 of them. With 2 plies, as in Stockfish, the six positions give 437,022, 657,366 and 939,364, and the game 0.4% more nodes. A reduction that fails high is

searched again at full depth, so reducing more doesn't always mean searching less.

It passed, by about **+12 Elo** (see "What it's worth"), so it is on by default. Stockfish's own estimate for its version is about 3 Elo, in an engine where many other reductions overlap with it; in cb2026 it had more to add. The other half of the same guess, a smaller reduction at an expected all-node, came third (below).

The second test was 2 plies, as in Stockfish (cjus/chessengine#155). The six positions above show what it does to the trees: about as much as 1 ply did, and no more consistently. The bench is 1,072,184 nodes, against 1,078,861 at 1 ply. **It passed too**, by about **+9 Elo** against 1 ply (see "What it's worth"), so `lmr_cut_node` is 2. Together the two steps are worth about 20 Elo in self-play, more than Stockfish's estimate for the whole technique. Nothing larger than 2 has been tried.

Smaller reductions at expected all-nodes

At an expected all-node every move has to fail low, and the node's score holds only if they all do. A late move reduced into failing low, when a full-depth search would have found it good enough, makes the node fail low wrongly, and the move that led to it then looks like a refutation it isn't. So the third test (cjus/chessengine#158) takes `lmr_all_node` (1) ply less off the late moves there:

```
// Reduce less at an expected all-node: every move has to fail low for its
// score to hold, so a late move that is wrongly reduced into failing low
// makes the parent's move a refutation it isn't.
if self.techniques.all_node_reductions && !pv_node && !cut_node {
    reduction -= p.lmr_all_node;
}
```

Stockfish doesn't do this; Stockfish 17 (GPL-3.0) even reduces more at an all-node when the next ply has had many fail highs. The idea is

cb2026's own, and the ply was set by hand. `set allnode on|off` switches it, and off gives the search without it exactly.

It costs a lot. A cut node usually stops after its first move or two, but an all-node searches all of its moves, so it has many more late moves to reduce. The bench grows by a fifth, from 1,072,184 to 1,284,421 nodes. The six positions grow by 30%, 25% and 42% at depths 9 to 11 (568,646, 819,202 and 1,333,493 nodes against 437,022, 657,366 and 939,364), and the game's 58 positions by 22.6% at depth 10, smaller in only 19 of them. At a fixed time that is most of a ply, so the better searches at all-nodes have to be worth more than a ply of depth.

It failed, by about **−9 Elo** (see "What it's worth"), so it is off by default. Untested: the smaller reduction only at greater depths, or only for moves with a good history, either of which would cost fewer nodes.

Reductions where the principal variation was

The principal variation changes as the search deepens. A move that was best at depth 9 may not be at depth 10, and the positions along the old line are then searched with null windows like any others. They are still more likely than most to matter again. Stockfish (GPL-3.0) remembers them: each transposition table entry has a bit that says whether the position is or has been on the principal variation (its `ttPv`), and late move reductions are smaller there. cb2026 does the same, with code of its own (cjus/chessengine#160).

A node has the flag if it is a PV node now, or if the table says it was one:

```
// Is or has been on the principal variation: a PV node now, or one the table
remembers
// as one. A singular extension's search, of the same node, keeps the node's flag.
if excluded.is_none() {
    self.tt_pv[ply as usize] = pv_node || tt_entry.is_some_and(|entry| entry.pv);
}
let tt_pv = self.tt_pv[ply as usize];
```

The flag goes into the table with every result stored for the position, so it lasts from one iteration to the next and from one move to the next. One more rule, also Stockfish's, spreads it. A node that fails low under a flagged parent, above depth `tt_pv_fail_low_depth` (3), takes the flag too: every move here failed, so the opponent's last move was probably good, and this position may be on the next principal variation.

```
// After a fail low under a node that is or has been on the principal variation, the
// opponent's last move was probably good, so this position may be on the next one.
if bound == Bound::Upper && ply > 0 && excluded.is_none() && depth >
p.tt_pv_fail_low_depth
{
    self.tt_pv[ply as usize] |= self.tt_pv[ply as usize - 1];
}
let tt_pv = self.tt_pv[ply as usize];
```

At a flagged node that isn't a PV node now, late moves are reduced `lmr_tt_pv` (1) ply less. A PV node already has a ply off.

```
// Reduce less at a node that has been on the principal variation: it may be
// again.
if self.techniques.tt_pv && tt_pv && !pv_node {
    reduction -= p.lmr_tt_pv;
}
```

Stockfish 17 takes 1 to 3 plies off, depending on the hash entry's score and depth, at PV nodes too, and its own comment rates it about 7 Elo. cb2026 takes 1, set by hand. `set ttpv on|off` switches it, and off gives the search without it exactly.

The flag is rare. Over the game's positions at depth 10, about 2% of the non-PV nodes from depth 3 had it. But they are near the principal

variation, where the subtrees are large, so it costs nodes:

- The bench grows by 11.8%, from 1,072,184 to 1,198,340 nodes.
- The six positions of chapters 10 to 14 grow by 3%, 19% and 12% at depths 9 to 11 (450,688, 784,751 and 1,054,117 nodes, against 437,022, 657,366 and 939,364).
- The game's 58 positions at depth 10 grow by 8.0% in all, and 28 of them get smaller.

No verdict: about +0 Elo, give or take 5, after 8,100 games, when the run was stopped (see "What it's worth"), so it is off by default. Whatever it adds is too small to measure at this time control. The flag is still kept in the table, for another try. Untested: Stockfish's larger, graded reductions (2 or 3 plies, by the hash entry's score and depth), and the other uses Stockfish makes of the flag, such as skipping reverse futility pruning at flagged nodes.

Reductions after many fail highs

The node types above are guesses passed down before a node's moves are searched. The search also learns as it goes. When the moves at a node are refuted one after another, every refutation is a fail high one ply down, and a node whose moves keep being refuted is probably failing low: its late moves are unlikely to do better than the first ones. Stockfish (GPL-3.0) counts those fail highs and reduces more once there are several (cjus/chessengine#165). cb2026 does the same, with code of its own.

Each ply has a count. Entering a node clears the count two plies down, so the next ply's count, the one a node reads, covers its own children and its siblings':

```
// The fail highs two plies down are counted afresh under each node: the next ply's
// count then covers this node's children and its siblings'.
self.cutoffs[ply as usize + 2] = 0;
```

A fail high adds to its ply's count, and adds twice when the node had no hash move to try first, as in Stockfish 16 (its code doesn't say why).

```
// Counted for the node above: once more without a hash move, and less for a
// doubly extended move, as in Stockfish 17.
let counts = if doubled {
    p.cutoff_count_double_extended
} else {
    1
};
self.cutoffs[ply as usize] += counts + u32::from(tt_move.is_none());
```

When the count is more than `lmr_cutoff_count` (3), late moves are reduced `lmr_cutoff_plies` (1) more:

```
// Reduce more when many moves at this ply have been refuted: this node is
// probably failing low, and its late moves along with it.
if self.techniques.cutoff_reductions
    && self.cutoffs[ply as usize + 1] > p.lmr_cutoff_count
{
    reduction += p.lmr_cutoff_plies;
    // More at an expected all-node, as in Stockfish 17.
    if !pv_node && !cut_node {
        reduction += p.lmr_cutoff_all_node_plies;
    }
}
```

Stockfish's own comment rates it about 5 Elo. `set cutoffs off` switches it off, and gives the search without it exactly.

It applies often. Over the game's positions at depth 10, the count was over 3 for 60% of the moves late move reductions reduced (and for 47% at expected all-nodes), so the trees shrink:

- The bench shrinks by 28%, from 1,072,184 to 767,859 nodes.
- The six positions of chapters 10 to 14 shrink by 26%, 24% and 12% at depths 9 to 11 (324,275, 500,421 and 828,983 nodes, against 437,022, 657,366 and 939,364).

- The game's 58 positions at depth 10 shrink by 28.4% in all, and 36 of them get smaller.

Smaller reductions at expected all-nodes, the opposite direction, failed (above). **This one passed**, by about **+13 Elo** (see "What it's worth"), so it is on by default: where the search has seen the moves refuted, reducing more paid. Its numbers are Stockfish's, not tuned.

The second test, Stockfish 17's version, **passed too**, by about **+14 Elo** against the first (cjus/chessengine#167, see "What it's worth"), so it is the default. It changes two things:

- At an expected all-node the reduction is `lmr_cutoff_all_node_plies` (1) larger again, 2 plies in all: the node expected to fail low, and seen to, is reduced the most.
- A fail high by a move extended twice (singular by a wide margin) counts once less: it adds `cutoff_count_double_extended` (0), plus 1 without a hash move. Stockfish 16, and the first version here, added 1 for it like any other move.

The trees change little and not one way. Against the first version, the bench is 2.4% smaller (749,181 nodes against 767,859), and the six positions 2%, 5% and 8% smaller at depths 9 to 11 (319,002, 473,898 and 764,000 against 324,275, 500,421 and 828,983), but the game's 58 positions at depth 10 are 14.9% bigger in all, and only 20 of them smaller. Each change alone does some of both: the extra all-node ply alone gives a bench of 741,257 and the game +9.1%; the count alone, 757,332 and +3.9%. The games decided, as with cut-node reductions: a bigger tree in the game positions, and still a gain.

Reductions when the hash move is a capture

One more signal comes from the transposition table. Stockfish 16 and 17 (GPL-3.0) reduce late moves a ply more when the move the table remembers as best here is a capture (or a promotion to a queen). Their comment rates it about 3 Elo and doesn't say why (cjus/chessengine#171). A plausible reason: the node's best move is probably that capture, and a quiet move searched this late seldom does better. cb2026's first version did the same at every node, with code of its own. `lmr_tt_capture` is 1, as in Stockfish, and `set ttcapture off` gives the search without it exactly. It applies less often than the reductions above: over the game's positions at depth 10, the hash move was a capture for 12.5% of the moves late move reductions reduced. The trees still shrink:

- The bench shrinks by 4.0%, from 749,181 to 719,329 nodes.
- The six positions of chapters 10 to 14 shrink by 8.5%, 5.0% and 8.5% at depths 9 to 11 (292,044, 450,207 and 698,814 nodes, against 319,002, 473,898 and 764,000).
- The game's 58 positions at depth 10 shrink by 14.7% in all, and 30 of them get smaller.

It failed, by about **−6 Elo** (see "What it's worth"), so it went off by default. A smaller tree isn't a stronger engine when the moves cut away mattered. To see where it cut, the extra ply was tried at one kind of node at a time:

Extra ply when the hash move is a capture	Bench	The game's 58 positions at depth 10
Nowhere (`main`)	749,181	1,898,454
At every node (the first version)	719,329 (−4.0%)	1,618,971 (−14.7%)
Except at expected cut nodes	719,458 (−4.0%)	1,681,317 (−11.4%)
On the principal variation only	715,028 (−4.6%)	1,695,304 (−10.7%)
At expected all-nodes only	755,028 (+0.8%)	1,818,833 (−4.2%)
Off the principal variation (all-nodes and cut nodes)	754,886 (+0.8%)	1,927,897 (+1.6%)

Leaving out expected cut nodes, the obvious narrowing, changes almost nothing. Their late moves are reduced 2 plies more already, so the extra ply mostly runs into the limit of depth − 2. Most of the first version's saving was on the principal variation, where cb2026 otherwise reduces late moves a ply less because the exact score matters; the extra ply took that back. That may be what cost the Elo.

The second version: off the principal variation

So the second version leaves the principal variation alone. At a PV node the extra reduction is `lmr_tt_capture_pv` (0), and 1 gives the first version:

```
// Reduce more when the hash move is a capture: it is probably the best move,
// and a quiet move this late seldom beats it. Less (by default not at all) on
// the principal variation, whose exact score matters.
if self.techniques.tt_capture_reductions && tt_move.is_some_and(is_noisy) {
    reduction += if pv_node {
        p.lmr_tt_capture_pv
    } else {
        p.lmr_tt_capture
    };
}
```

It barely changes the size of the tree:

- The bench grows by 0.8%, to 754,886 nodes.
- The six positions change by −0.05%, +1.2% and +1.7% at depths 9 to 11 (318,853, 479,481 and 777,375 nodes).
- The game's positions grow by 1.6% in all: 26 get smaller and 29 bigger.

More reductions and a bigger tree aren't a contradiction. A reduced search that fails high is searched again at full depth, and every search changes the moves and scores the table hands to the rest. Whatever the second version gains has to come from where the effort goes, not from a smaller tree.

It failed too, by about **−7.5 Elo** (see "What it's worth"), so both versions are off by default. A smaller tree didn't help, and neither did a bigger one. cb2026 already reduces late moves more at expected cut nodes and after many fail highs. Off the principal variation, one more ply when the hash move is a capture seems to reduce some of them too far. Stockfish tunes its reductions together, where cb2026 adds them one at a time, so the ply may only pay off alongside Stockfish's other numbers.

What it saves

The six positions of chapters 10 to 14 (bench positions 2 to 7) at depth 8, measured while singular extensions started at depth 8, so that none applied:

Setting	Nodes	vs defaults
Defaults (IIR, continuation history and history in LMR on; improving and the check extension off)	148,427	—
`set histlmr off`	145,410	−2%
`set iir off`	194,771	+31%
`set improving on`	117,842	−21%
`set checkext on`	152,496	+3%

IIR takes about a quarter off the tree. Improving takes off over a fifth more; its second version, which also made late move pruning wait longer, took off 7%, and its first 31%. All three made the engine weaker: the games below decide, not the node counts.

History in LMR's first version hardly changed these trees: history scores grow with depth (the bonus is `depth²`) and over a game (each search keeps half of the last one's), and a search to depth 8 from empty tables leaves few moves at 256 or more. The second version, with half the divisor and both directions, makes the tree 2% bigger here. With singular extensions in, it makes the bench 1.1% bigger, and the six positions at depth 10 3% smaller (674,643 nodes against 694,595): reducing some moves more and others less doesn't add up the same way at every depth.

Singular extensions don't appear in that table: at depth 8, no node below the root had 8 plies left. From depth 4 they do, and make the depth-8 tree 57% bigger (232,328 nodes against 148,427; from depth 6, 242,552). At depth 10, starting from depth 8, they made the tree **85% bigger** (694,595 nodes against 375,334). Over the six positions the search tried 472 hash moves, extended 120 and cut off 138 nodes by multi-cut; without multi-cut the tree is 7% bigger again. That is the price of extending near the root: each extension adds a ply to a large part of the tree. The games say it is worth it: about **+30 Elo** (see "What it's worth"), the biggest gain of anything in this chapter, with a tree almost twice the size. Nodes aren't Elo, in this direction too.

Negative extensions came later, on top of history in LMR's second version. At depth 10 the six positions tried 485 hash moves for singularity, extended 125, cut off 142 nodes by multi-cut and searched 56 hash moves a ply shallower. That takes 0.9% off the tree (668,426 nodes against 674,643) and 2.3% off the bench: a small change, since it applies only where the singular test has already run and the hash move is expected to cut off anyway. The games agree that it is small, about **+5 Elo**, and it took over 14,000 of them to say so.

Pitfalls

- **Search explosion.** An extension that can apply at every ply never reaches the horizon, and one that applies often can double the tree. Limit extensions (here, to the first half of the maximum ply), and count nodes as well as games.
- **Overlap.** Every technique here works on top of all the others. The check extension found little because LMR already spares checks; turning a technique off shows only what it adds *on top of* the rest.

- **Store what you searched.** IIR replaces `depth` for the rest of the node, so the table entry records the reduced depth. Storing the original depth would let the table answer later, deeper searches with a result it doesn't have.
- **Nodes aren't Elo.** IIR saves nodes by searching some nodes less deeply. Whether that costs more than it saves is for an SPRT to say, not a node count.
- **A margin that reaches zero.** A margin scaled down by a flag can disappear altogether: at depth 1, the first version of improving took reverse futility pruning's margin to zero. Make sure every margin stays positive at the smallest depth it applies to; a unit test now checks that `rfp_improving` stays below `rfp_margin`.
- **Nodes aren't Elo, again.** The first version of improving made the tree almost a third smaller and lost 32 Elo. A big node saving is a reason to run an SPRT, not a result.
- **Every ply stores its evaluation, the root included.** A heuristic that looks back two plies reads what earlier nodes stored. A node searched by a different function, like the root here, is easy to miss, and nothing fails: the comparison just gives the wrong answer. A unit test now checks the root's entry.
- **Measure the scale before choosing a divisor.** A history score's range says nothing about where the scores actually sit. Here they sit within ±64 for three moves in four, so a divisor set from the range (say 8,192, two plies at the cap) would almost never change a reduction.
- **A second search of the same node.** The search without the hash move runs at the same ply as the node itself. Anything stored per

ply or per position, the hash table above all, has to know that it is a different search, or it will mix the two results.

- **A negative extension trusts the hash score.** It reduces the move the node expects to cut off with, on the strength of a score from an earlier search. If that score was too optimistic, the move's shallower search is the one that has to find out. It applies only where the singular test has run, with an entry at most `singular_tt_depth_margin` plies shallower than the node.
- **Count along the line, and take the count off on every way out.** The limit on double extensions is per line, so the searcher keeps a count that each double-extended move adds to before its search and takes off after it. A count that some exit skips (a cutoff, a stopped search) only grows, until no move can be extended twice. A unit test checks that the count is back at zero after a search.
- **Two plies back may be a check.** In check, cb2026 computes no static evaluation (it is recorded as `-INFINITY`), so a side that was in check two plies ago always counts as improving. Some engines look four plies back in that case.

What it's worth

Test	Result
Check extensions on vs off (stage 20), 8+0.08, cloud, SPRT [0, 5]	−8.3 ± 17.7 Elo over 840 games (+229 −249 =362), no verdict when the session's two-hour limit stopped it: no gain, so it is off by default
IIR vs `main` without it, 8+0.08, test Mac, SPRT [0, 5]	**+19.2 ± 10.4 Elo** over 2,337 games (+705 −576 =1056): passed
Improving, first version (a whole ply off the margin), vs `main` without it (with IIR), 8+0.08, test Mac (M1 Max), SPRT [0, 5]	**−32.0 ± 15.1 Elo** over 1,142 games (+267 −372 =503): failed (LLR −2.96)
Improving, half a ply off the margin, late move pruning doubled, vs `main` without it (with IIR), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−17.7 ± 11.5 Elo** over 1,691 games (+362 −448 =881): failed (LLR −2.97)
Improving, half a ply off the margin only, vs `main` without it (with IIR), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−33.0 ± 15.3 Elo** over 1,036 games (+221 −319 =496): failed (LLR −2.96), so improving is off by default
History in LMR vs `main` without it (with continuation history), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	+2.1 ± 5.0 Elo over 8,910 games (+2188 −2134 =4588): no verdict (LLR −0.30), stopped; off by default
Singular extensions and multi-cut vs `main` without them (with continuation history), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**+29.5 ± 13.3 Elo** over 1,357 games (+409 −294 =654): passed (LLR 2.96)
History in LMR, second version (history plus continuation history, divisor 128), vs `main` without it (with singular extensions and	**+22.2 ± 11.4 Elo** over 1,661 games (+437 −331 =893): passed (LLR 2.95)

Test	Result
adaptive time), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	
Negative extensions vs `main` without them (with adaptive time), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**+4.9 ± 4.0 Elo** over 14,211 games (+3518 −3317 =7376): passed (LLR 2.96)
Singular extensions from depth 6 vs `main` (from depth 8, with negative extensions), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**+5.0 ± 4.1 Elo** over 12,282 games (+2758 −2580 =6944): passed (LLR 2.97)
Singular extensions from depth 4 vs `main` (from depth 6), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**+6.1 ± 4.8 Elo** over 9,057 games (+2127 −1969 =4961): passed (LLR 2.96)
`singular_margin` 3 vs `main` (2, from depth 4), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**+5.2 ± 4.2 Elo** over 11,400 games (+2552 −2381 =6467): passed (LLR 2.96)
`singular_margin` 4 vs `main` (3), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−1.8 ± 5.3 Elo** over 7,476 games (+1670 −1709 =4097): failed (LLR −2.95); the margin stays at 3
`singular_tt_depth_margin` 4 vs `main` (3), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−7.4 ± 8.0 Elo** over 3,345 games (+735 −806 =1804): failed (LLR −2.97); the margin stays at 3
Double extensions (margin 20, at most 6 a line) vs `main` without them, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	+3.6 ± 4.9 Elo over 8,670 games (+1988 −1897 =4785): no verdict (LLR 0.92), stopped; off by default
Double extensions (margin 50, at most 6 a line) vs `main` without them (with correction history), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**+13.0 ± 8.3 Elo** over 3,167 games (+798 −680 =1689): passed (LLR 2.94); on by default

Test	Result
Double extensions at a margin of 100 vs `main` (50), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−7.3 ± 8.0 Elo** over 3,368 games (+747 −818 =1803): failed (LLR −2.95); the margin stays at 50
Double extensions at a margin of 70 vs `main` (50), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−20.5 ± 12.2 Elo** over 1,407 games (+278 −361 =768): failed (LLR −2.97); the margin stays at 50
Larger reductions at expected cut nodes (1 ply more) vs `main` without them, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**+11.6 ± 7.7 Elo** over 3,687 games (+925 −802 =1960): passed (LLR 2.97); on by default
Cut-node reductions of 2 plies vs `main` (1), 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**+8.7 ± 6.4 Elo** over 5,803 games (+1551 −1405 =2847): passed (LLR 2.95); `lmr_cut_node` is 2
Smaller reductions at expected all-nodes (1 ply less) vs `main` without them, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−8.5 ± 8.4 Elo** over 2,993 games (+649 −722 =1622): failed (LLR −2.97); off by default
Smaller reductions where the principal variation was (Stockfish's `ttPv`, 1 ply less) vs `main` without them, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	+0.3 ± 5.2 Elo over 8,100 games (+1939 −1933 =4228): no verdict (LLR −1.57), stopped; off by default
Larger reductions after many fail highs (1 ply more when the next ply has counted more than 3) vs `main` without them, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**+12.8 ± 8.2 Elo** over 3,144 games (+776 −660 =1708): passed (LLR 2.95); on by default
Stockfish 17's version of them (2 plies at an expected all-node; a doubly extended move's fail high counted once less) vs Stockfish 16's, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**+13.7 ± 8.5 Elo** over 2,953 games (+741 −625 =1587): passed (LLR 2.96); the default
Larger reductions when the hash move is a capture (1 ply more, at every node) vs `main`	**−6.5 ± 7.7 Elo** over 3,668 games (+819 −888 =1961): failed (LLR −2.95); off by default

Test	Result
without them, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	
Its second version, off the principal variation (1 ply more at expected all-nodes and cut nodes, none at PV nodes) vs `main` without them, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−7.5 ± 8.0 Elo** over 3,031 games (+608 −673 =1750): failed (LLR −2.96); off by default

Further reading

- Chess Programming Wiki: Extensions, Node Types, Check Extensions, Singular Extensions, Internal Iterative Deepening, Internal Iterative Reductions, Improving.

Correction history

Milestone M5, after stage 20 · on `main`, no stage tag · code in `crates/engine/src/correction.rs` and `crates/engine/src/search.rs`

Chapters 12 to 14 prune by guessing from the static evaluation. Reverse futility pruning returns when the evaluation is far above beta, razoring gives up when it is far below alpha, null-move pruning needs it at beta or above, and futility pruning skips quiet moves that can't lift it to alpha. The quiescence search stands pat on it at every leaf. All of them are only as good as that one number.

Chapter 15's evaluation is hand-crafted, and it is wrong in patterns. In some pawn structures it is too optimistic, in others too pessimistic, by much the same amount each time the structure comes up: a weakness it doesn't know about, or a pawn chain it overrates. The search finds out as it goes. A node's searched score is a better estimate than its static evaluation, and the difference between them is the evaluation's error in that position.

Correction history remembers those errors. It keeps a running average of the difference in a table indexed by the side to move and the pawn structure, and adds the average to every later static evaluation of a position with the same pawns. Two more tables do the same by each side's pieces (below): averaged with the pawn table

they measured weaker, and added to it they made no difference, so they are off by default. Like chapter 14.1's techniques, it came after stage 20, so no stage tag contains it. Switch it with `set correction on|off`, and the piece tables with `set corrpieces on|off`.

The table

A pawn structure is two bitboards, White's pawns and Black's. There are far too many structures to give each its own entry, so the two bitboards are mixed into one number, and its low bits pick one of 16,384 entries per side:

```
/// The pawn structure as one number: both sides' pawn bitboards, mixed so that every
bit of
/// them affects every bit of the result. Black's are mixed first, so a white pawn and
a black
/// pawn on the same square don't count as the same structure.
pub fn pawn_key(pos: &Position) -> u64 {
    let white = pos.pieces(Color::White, PieceType::Pawn).0;
    let black = pos.pieces(Color::Black, PieceType::Pawn).0;
    mix(white ^ mix(black))
}

/// SplitMix64's finaliser (Steele, Lea and Flood, 2014): a cheap, thorough bit mixer.
fn mix(mut x: u64) -> u64 {
    x ^= x >> 30;
    x = x.wrapping_mul(0xBF58_476D_1CE4_E5B9);
    x ^= x >> 27;
    x = x.wrapping_mul(0x94D0_49BB_1331_11EB);
    x ^ (x >> 31)
}
```

Black's pawns are mixed before they are combined with White's, so that a white pawn on a2 and a black pawn on a2 don't count as the same structure. The mixer is SplitMix64's finaliser, a few shifts and multiplications that make every bit of the input affect every bit of the output, so similar structures land far apart. Engines usually keep a pawn hash key up to date move by move, as the position's Zobrist key is (chapter 4). Computing it from the bitboards instead costs

much less than the evaluation it corrects and leaves the position as it is.

The entries are kept in 256ths of a centipawn:

```
/// How far the static evaluation tends to be off, by side to move and pawn structure,
and by
/// side to move and each side's other pieces.
#[derive(Clone, Debug)]
pub struct CorrectionHistory {
    /// `2 × CORRECTION_ENTRIES` corrections in 256ths of a centipawn, by pawn
structure: White
    /// to move's, then Black to move's.
    pawn: Vec<i32>,
    /// The same, by White's pieces other than pawns, then by Black's.
    non_pawn: [Vec<i32>; 2],
}
```

Correcting

Every node in the main search computes its static evaluation, then corrects it:

```
// The static evaluation, for the pruning decisions (meaningless in check), corrected
by
// what the search has learnt about positions with these pawns.
let raw_eval = if in_check {
    -INFINITY
} else {
    self.evaluate(pos, ply)
};
let static_eval = if in_check {
    -INFINITY
} else {
    self.corrected_eval(pos, raw_eval)
};
```

```
/// `raw_eval`, the static evaluation of `pos`, corrected (see [`Self::correction`]),
and
/// kept clear of the mate scores.
pub fn corrected(&self, pos: &Position, raw_eval: i32, settings: &CorrectionSettings)
-> i32 {
    let highest = MATE - MAX_PLY as i32 - 1;
    (raw_eval + self.correction(pos, settings)).clamp(-highest, highest)
}
```

The quiescence search corrects its stand pat the same way. The uncorrected evaluation is kept too, because that is what the table learns about.

Learning

When a node's search is done, its result says something about the evaluation, but not always:

```
// Correction history learns how far the static evaluation was off: not in check
(there
// is none), not in a singular extension's search (its score leaves the best move
out),
// and not when the best move is a capture or promotion (its gain is material the
static
// evaluation isn't meant to see).
let settings = self.correction_settings();
if self.techniques.correction_history
    && !self.aborted
    && !in_check
    && excluded.is_none()
    && !best_move.is_some_and(is_noisy)
    && let Some(correction) = self.correction.as_deref_mut()
{
    correction.update(pos, depth, raw_eval, best, bound, &settings);
}
```

```
/// Learns from a search of `pos` to `depth` that scored `score` as a `bound`, where
the
/// uncorrected static evaluation was `raw_eval`: the pawn table, and the piece tables
too
/// if they are on. The caller has already left out the nodes that can't say anything
about
/// the evaluation (see the module documentation); this leaves out mate scores and
bounds on
/// the wrong side of the evaluation.
pub fn update(
    &mut self,
    pos: &Position,
    depth: u32,
    raw_eval: i32,
    score: i32,
    bound: Bound,
    settings: &CorrectionSettings,
) {
    if score.abs() >= MATE - MAX_PLY as i32 {
        return;
    }
    // A fail high shows the evaluation too low, a fail low shows it too high; nothing
else.
    let informative = match bound {
        Bound::Exact => true,
        Bound::Lower => score > raw_eval,
        Bound::Upper => score < raw_eval,
    };
    if !informative {
        return;
    }
    let weight = (depth as i32 + 1).min(settings.max_weight);
    let limit = settings.limit * GRAIN;
    let target = ((score - raw_eval) * GRAIN).clamp(-limit, limit);
    let learn = |entry: &mut i32| {
        *entry = (*entry * (WEIGHT_SCALE - weight) + target * weight) / WEIGHT_SCALE;
    };
    learn(&mut self.pawn[index(pos, pawn_key(pos))]);
    if settings.non_pawn_share > 0 {
        for color in Color::ALL {
            learn(&mut self.non_pawn[color.index()][index(pos, non_pawn_key(pos,
color))]);
        }
    }
}
```

- **Captures and promotions.** If the best move wins material, the score is higher than the evaluation because of the capture. The evaluation of a position isn't meant to include material that is about to change hands; the quiescence search is there for that.

- **Bounds.** A fail high is a lower bound: the true score is at least `score`. If that is above the evaluation, the evaluation was too low by at least the difference; if it is below, the true score could be anything, and the node says nothing. A fail low is the mirror image.
- **Mate scores** are about a forced line, not the position's evaluation.
- **No static evaluation in check**, and a singular extension's search leaves the best move out on purpose, so neither counts.

An entry then moves towards the new difference by a fraction of the way: `min(depth + 1, 16)` 256ths. A deep search's result is worth more than a shallow one's, but even the deepest moves the entry only a sixteenth of the way, so the entry is an average over many nodes and no single surprising result dominates. No entry goes past 64 centipawns either way. Both numbers are search parameters, `correction_max_weight` and `correction_limit`.

What it changes

The bench, which starts every position with an empty table, is 3.0% smaller with it: 1,065,754 nodes instead of 1,098,926. The speed is unchanged: computing the pawn key at every node costs nothing measurable.

The corrections themselves are small. After searching the 14 bench positions to depth 13 one after another, without clearing the table, 2,111 of the 32,768 entries were in use. Their average correction was 3.5 centipawns, 90% were within 8 and 99% within 25, and none reached the limit of 64. Reverse futility pruning, razoring and futility pruning have margins of 80 centipawns a ply or more, so a typical

correction changes few of their decisions. Null-move pruning, which only needs the evaluation at beta, and the stand pat feel it more directly.

Even so, it gained about as much as singular extensions and multi-cut (+29.5, chapter 14.1), among the largest of the changes made after stage 20. The numbers above don't explain that on their own, and two things they leave out probably do. In a game the table carries over from move to move and the same structures come up again and again, so its entries learn from far more nodes than in that measurement. And most of the gain may come from the few structures where the evaluation is consistently wrong by more, the 1% beyond 25 centipawns, which are also where a wrong evaluation costs the most. Neither is measured.

More tables

The pawns aren't the only part of a position whose errors repeat. A knight stuck on the rim, a king in an awkward spot, a bishop pair that the evaluation over- or underrates: those are about the pieces. Two more tables, indexed the same way by the side to move and a key, use each side's pieces other than pawns as the key, White's knights, bishops, rooks, queens and king in one and Black's in the other:

```
/// `color`'s pieces other than pawns as one number, made like [`pawn_key`]: each
piece type's
/// bitboard is mixed into the key in turn, so a knight and a bishop on the same
square differ.
pub fn non_pawn_key(pos: &Position, color: Color) -> u64 {
    NON_PAWN_PIECES
        .iter()
        .fold(0, |key, &piece| mix(key ^ pos.pieces(color, piece).0))
}
```

Every update teaches all three tables the same difference. A move changes one side's pieces at most, or both sides' with a capture, so the positions a search visits share their piece keys with many others nearby: after a White move, Black's key is the one it was. The piece tables learn from the positions the search is in.

Stockfish (GPL-3.0) keeps tables like these, one keyed by each side's pieces other than pawns, among others. The code here is cb2026's own. With `set corrpieces off`, the pawn table corrects alone, as before.

The first version: an average

The first version's correction was the tables' weighted average: half the pawn table's, and a quarter each piece table's (`correction_pawn_share` 128 and `correction_non_pawn_share` 64, of 256). An average, not a sum, because each table learns the whole error: summed, three tables that agree would correct three times over.

The bench was 4.3% smaller with it, 1,019,663 nodes instead of 1,065,754, and the search ran about 2.5% fewer nodes a second: each lookup computes two more keys, of five bitboards each, and the two tables add 256 KB that compete for the cache. It lost in games (below).

The piece tables weren't silent. Over the 14 bench positions searched to depth 13 one after another, the corrections applied at lookups averaged 2.4 centipawns from the pawn table and 2.4 from the piece tables' average, and 2.25 combined. So the piece tables replaced half of the pawn table's correction with corrections as large and, by the result, worse. Why isn't measured. A side's piece key changes whenever one of its pieces moves, the king included, so each entry

may learn from fewer and more varied positions than a pawn entry, and average more noise than error.

The second version: added

The second version keeps the pawn table's correction whole and adds an eighth of each piece table's (`correction_pawn_share` 256, `correction_non_pawn_share` 32), keeping the sum within `correction_limit`:

```
/// The correction for `pos`'s static evaluation, in centipawns. With the piece tables
off
/// it is the pawn table's; on, the pawn table's counts `pawn_share` 256ths and each
side's
/// piece table's `non_pawn_share` 256ths, added up and kept within the limit.
pub fn correction(&self, pos: &Position, settings: &CorrectionSettings) -> i32 {
    let pawn = self.pawn[index(pos, pawn_key(pos))];
    let non_pawn_share = settings.non_pawn_share;
    if non_pawn_share == 0 {
        return pawn / GRAIN;
    }
    let pieces: i32 = Color::ALL
        .iter()
        .map(|&color| self.non_pawn[color.index()][index(pos, non_pawn_key(pos,
color))])
        .sum();
    let sum = pawn * settings.pawn_share + pieces * non_pawn_share;
    (sum / (SHARE_SCALE * GRAIN)).clamp(-settings.limit, settings.limit)
}
```

Where the pawn table alone knows the position, it corrects as before. Where all three agree, the correction is a quarter bigger. A piece table that is wrong can move the evaluation by an eighth of its error at most, where in the first version it moved it by a quarter and took half of the pawn table's say away.

The bench is 2.7% bigger with it: 1,094,228 nodes instead of 1,065,754, and with `set corrpieces off` it is `main`'s exactly. The search runs about 1% fewer nodes a second (two sets of five alternating pairs of bench runs, one sample left out as an outlier).

It made no difference in games (below): −0.4 ± 4.3 Elo after 11,433. Kept small, the piece tables did no harm, and showed no gain either. So the pawn table corrects alone, as before, and the piece tables are off by default.

Pitfalls

- **Learning from the correction itself.** The searched score comes from a tree whose leaves were corrected. If the table learnt the difference from the *corrected* evaluation, each correction would build on the last. Measured against the raw evaluation, an entry is an average of how far the raw evaluation was off, and it settles.
- **Averaging dilutes.** In the first version the pawn table counted half as much as it did alone, so where only it had learnt something, the correction was half as big. Adding keeps it whole, at the cost of correcting by more than any one table where they agree; the limit still holds.
- **Collisions.** Every pawn structure the search meets shares 16,384 entries per side, so unrelated structures can share an average. A bigger table collides less, but each entry learns more slowly.
- **The mate range.** A corrected evaluation is clamped short of the mate scores, so that a correction can never make a large evaluation look like a forced mate.
- **Helper threads** (chapter 17) start every search with an empty table of their own; only the main thread's table carries over from one move to the next. What they find still reaches the main thread through the shared transposition table.
- **Comparisons at equal scores.** Correction history changes the static evaluation, so it changes scores, not only node counts.

`Techniques::sound()`, which keeps only the techniques that don't change the result, switches it off.

What it's worth

Test	Result
Correction history vs `main` without it, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**+28.0 ± 13.0 Elo** over 1,308 games (+362 −257 =689): passed (LLR 2.95), cjus/chessengine#112
The piece tables' first version, averaged with the pawn table, vs `main` with the pawn table alone, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−20.7 ± 12.3 Elo** over 1,362 games (+263 −344 =755): failed (LLR −2.96), cjus/chessengine#119
The second version, an eighth of each piece table added to the whole pawn table, vs `main` with the pawn table alone, 8+0.08, test Mac (M3 Max), SPRT [0, 5]	**−0.4 ± 4.3 Elo** over 11,433 games (+2605 −2617 =6211): failed (LLR −2.97), cjus/chessengine#121, so they are off by default

That second test took 11,433 games, about 9 hours: an SPRT needs many games to tell a change worth about nothing from one worth a few Elo.

Other candidates, untested:

- **Keys without the king**, which moves often and makes a side's key change more than its pieces do.
- **Fewer pieces per key**, for example knights and bishops in one key and rooks and queens in another, so that each entry is shared by more positions.

Further reading

- Chess Programming Wiki: Static Evaluation Correction History.

- The idea comes from Caissa (MIT licence, 2023), and Stockfish (GPL-3.0) and most strong engines have adopted it, several with more tables than the pawn one. The code here is cb2026's own.
- Guy L. Steele, Doug Lea and Christine H. Flood, "Fast Splittable Pseudorandom Number Generators", OOPSLA 2014: SplitMix64, whose finaliser mixes the pawn key.

A full hand-crafted evaluation

Stage 15 · tag `stage-15-hce` *· code in* `crates/engine/src/hce.rs` *and* `eval.rs`

The search is now deep and selective. What it finds at the leaves is still judged by chapter 9's evaluation: material and piece-square tables. Those know that a knight likes the centre, but nothing about how the pieces relate to each other. A bishop on its best square counts the same whether its diagonals are open or blocked by its own pawns. A king behind three pawns counts the same as one whose shelter has been torn away. This chapter adds the classic positional terms of a hand-crafted evaluation (HCE).

The terms

```
/// The positional terms' numbers (each `(middlegame, endgame)` in centipawns) and
switches.
#[derive(Clone, Debug, PartialEq, Eq)]
pub struct TermParams {
    /// Mobility: value per safe square, by piece type (pawn and king unused).
    pub mobility: [MgEg; 6],
    /// Mobility is counted relative to this typical number of safe squares, by piece
type.
    pub mobility_typical: [i32; 6],
    /// Each pawn beyond the first on a file.
    pub doubled: MgEg,
    /// Each pawn with no friendly pawn on a neighbouring file.
    pub isolated: MgEg,
    /// A passed pawn, by rank from its own side (index 1 = second rank ... 6 =
seventh).
    pub passed: [MgEg; 8],
    /// Two bishops or more.
    pub bishop_pair: MgEg,
    /// A rook on a file with no pawns.
    pub rook_open_file: MgEg,
    /// A rook on a file with no pawns of its own side.
    pub rook_semi_open_file: MgEg,
    /// Each friendly pawn in the 3×2 block in front of the king (at most 3 count).
    pub pawn_shield: MgEg,
    /// King danger units per attacked square around the king, by attacking piece
type.
    pub king_attack_weights: [i32; 6],
    /// King danger penalty (middlegame) = units² × this / 100...
    pub king_danger_scale: i32,
    /// ...up to this much.
    pub king_danger_max: i32,
    /// For the side to move.
    pub tempo: MgEg,
    /// Score mobility.
    pub use_mobility: bool,
    /// Score pawn structure and passed pawns.
    pub use_pawns: bool,
    /// Score the bishop pair and rook files.
    pub use_pieces: bool,
    /// Score king safety.
    pub use_king_safety: bool,
    /// Score the tempo.
    pub use_tempo: bool,
}

impl Default for TermParams {
    /// The hand-set values (stage 15). The engine uses the tuned ones (see
[`crate::tuned`]).
    fn default() -> TermParams {
        TermParams {
            mobility: [(0, 0), (4, 4), (5, 5), (2, 4), (1, 2), (0, 0)],
            mobility_typical: [0, 4, 6, 7, 13, 0],
            doubled: (-10, -20),
```

```
            isolated: (-10, -15),
            passed: [
                (0, 0),
                (0, 5),
                (5, 10),
                (10, 20),
                (20, 35),
                (35, 60),
                (60, 90),
                (0, 0),
            ],
            bishop_pair: (30, 50),
            rook_open_file: (25, 10),
            rook_semi_open_file: (12, 8),
            pawn_shield: (15, 0),
            king_attack_weights: [0, 2, 2, 3, 5, 0],
            king_danger_scale: 35,
            king_danger_max: 500,
            tempo: (10, 5),
            use_mobility: true,
            use_pawns: true,
            use_pieces: true,
            use_king_safety: true,
            use_tempo: true,
        }
    }
}

impl TermParams {
    /// All positional terms switched off.
    pub fn none() -> TermParams {
        TermParams {
            use_mobility: false,
            use_pawns: false,
            use_pieces: false,
            use_king_safety: false,
            use_tempo: false,
            ..TermParams::default()
        }
    }
}
```

Mobility

A piece is worth more when it has more places to go. For each knight, bishop, rook and queen, cb2026 counts the squares it attacks that are neither occupied by its own side nor covered by an enemy pawn: squares it could actually use. Each piece type scores per square above or below a typical count. A bishop with 11 safe squares beats one with

3, and that's how the evaluation learns about "good" and "bad" bishops without a rule for each.

Pawn structure

- **Doubled pawns**, two on one file, can't protect each other, and the rear one is often blocked.
- **Isolated pawns**, with no friendly pawn on either neighbouring file, can only be defended by pieces.
- **Passed pawns** have no enemy pawn ahead on their own or a neighbouring file, so nothing but pieces can stop them. Their bonus climbs steeply with each rank, and more steeply in the endgame, where few pieces remain to stop them.

Pieces

- **The bishop pair**: two bishops cover both square colours and work well together in open positions, worth about a third of a pawn more in the middlegame and half a pawn in the endgame.
- **Rooks on open files** (no pawns) or **semi-open files** (no pawns of their own) can reach the enemy position.

King safety

In the middlegame the king needs shelter, and the danger it faces depends on how many enemy pieces are bearing down on it.

- The **pawn shield** scores each friendly pawn in the 3×2 block in front of the king.
- **King attacks** count, for each enemy piece, how many squares of the **king zone** (the king's square and its neighbours) it attacks, weighted by piece type: knight and bishop 2, rook 3, queen 5. The

penalty is the *square* of the total, scaled and capped. One piece alone rarely mates, but several together often do, and the quadratic shape captures that.

King safety only has middlegame values. As pieces come off, the phase blend fades it out by itself.

Tempo

The side to move gets a small bonus: having the move is worth something. The evaluation is still symmetric, since mirroring a position (swapping colours and flipping the board) also swaps the side to move.

```
/// Reports every switched-on positional term, for both sides, to `sink`: which
parameter
/// applies and how many times (see [`TermSink`]).
pub fn collect<S: TermSink>(pos: &Position, p: &TermParams, sink: &mut S) {
    let occupied = pos.occupied();
    for us in Color::ALL {
        let them = !us;
        let our_pawns = pos.pieces(us, PieceType::Pawn);
        let their_pawns = pos.pieces(them, PieceType::Pawn);

        if p.use_mobility {
            // Safe squares: not our own pieces, not covered by an enemy pawn.
            let area = !pos.pieces_of_color(us) & !pawn_attack_span(pos, them);
            for pt in PIECES {
                for square in pos.pieces(us, pt) {
                    let count = (piece_attacks(pt, square, occupied) & area).count()
as i32;
                    sink.term(
                        us,
                        Param::Mobility(pt),
                        count - p.mobility_typical[pt.index()],
                    );
                }
            }
        }

        if p.use_pawns {
            for square in our_pawns {
                let file = Bitboard::file(square.file());
                let neighbours = adjacent_files(square.file());
                // Doubled: another of our pawns further up the same file.
                if !(our_pawns & file & ranks_ahead(us, square.rank())).is_empty() {
                    sink.term(us, Param::Doubled, 1);
                }
                if (our_pawns & neighbours).is_empty() {
                    sink.term(us, Param::Isolated, 1);
                }
                // Passed: no enemy pawn ahead on this file or a neighbouring one.
                let ahead = (file | neighbours) & ranks_ahead(us, square.rank());
                if (their_pawns & ahead).is_empty() {
                    sink.term(us, Param::Passed(relative_rank(us, square)), 1);
                }
            }
        }

        if p.use_pieces {
            if pos.pieces(us, PieceType::Bishop).count() >= 2 {
                sink.term(us, Param::BishopPair, 1);
            }
            for square in pos.pieces(us, PieceType::Rook) {
                let file = Bitboard::file(square.file());
                if (file & (our_pawns | their_pawns)).is_empty() {
                    sink.term(us, Param::RookOpenFile, 1);
                } else if (file & our_pawns).is_empty() {
                    sink.term(us, Param::RookSemiOpenFile, 1);
                }
```

```
            }
        }

        if p.use_king_safety {
            let king = pos.king_square(us);
            // Pawn shield: our pawns on the king's file and its neighbours, one or
two ranks
            // ahead of it.
            let in_front = ranks_ahead(us, king.rank()) & !ranks_ahead(us,
two_ahead(us, king));
            let shield = our_pawns & (Bitboard::file(king.file()) |
adjacent_files(king.file()));
            let shield_count = (shield & in_front).count().min(3) as i32;
            sink.term(us, Param::PawnShield, shield_count);

            // Attacks on the king zone, weighted by attacker; the penalty grows
quadratically.
            let zone = king_attacks(king).with(king);
            let mut units = 0;
            for pt in PIECES {
                for square in pos.pieces(them, pt) {
                    let hits = (piece_attacks(pt, square, occupied) & zone).count() as
i32;
                    units += hits * p.king_attack_weights[pt.index()];
                }
            }
            let danger = (units * units * p.king_danger_scale /
100).min(p.king_danger_max);
            sink.king_danger(us, danger);
        }
    }

    if p.use_tempo {
        sink.term(pos.side_to_move(), Param::Tempo, 1);
    }
}
```

The trace

Every term is recorded for each side separately, so the shell's `eval` command can show exactly where a score comes from. Here is the Italian Game again, with White's king walked to e2:

```
> fen r1bq1rk1/pppp1ppp/2n2n2/2b1p3/2B1P3/2N2N2/PPPPKPPP/R1BQ3R w - - 0 1
> eval
Evaluation in pawns, each side from its own point of view:
                           White           Black      Net
                    Middle     End  Middle     End    (W-B)
  Material         +39.40  +42.00  +39.40  +42.00    +0.00
  Piece-squares     +0.20   -0.15   +0.75   -0.50    -0.55
  Mobility          -0.61   -0.91   -0.61   -0.96    +0.00
  Pawn structure    +0.00   +0.00   +0.00   +0.00    +0.00
  Passed pawns      +0.00   +0.00   +0.00   +0.00    +0.00
  Pieces            +0.30   +0.50   +0.30   +0.50    +0.00
  King safety       +0.10   +0.00   +0.44   +0.00    -0.34
  Tempo             +0.10   +0.05   +0.00   +0.00    +0.10
Phase 24/24: 100% middlegame, 0% endgame
Total: -0.79 for White, so -0.79 for White to move
```

Black's castled king has three shield pawns. White's king has one shield pawn (e4) and is worse placed on the piece-square table. That costs White most of a pawn, which is about what a strong player would say. Mobility is negative for both sides only because it is measured against the typical counts of a more open position. Only the difference between the sides matters.

`set mobility off`, `set pawns off`, `set pieces off`, `set kingsafety off` and `set tempo off` switch the groups off one at a time.

Getting it right

An evaluation bug doesn't crash anything; it just makes the engine play worse. Two kinds of test catch most of them:

- **Symmetry.** Every bench position and its mirror image must get the same score for the side to move. A term that forgets to flip a rank for Black fails this at once.
- **Hand-checked positions.** Small positions where each term's value can be worked out by hand: doubled and isolated pawns, a

passed pawn on the seventh, the bishop pair with a rook on an open file, a sheltered king against an exposed one.

Pitfalls

- **Double counting.** The piece-square tables already reward advanced pawns and central knights. The passed-pawn and mobility bonuses add to that, so the values have to be set with both in mind. Stage 16's tuning sets them together.
- **Speed.** Mobility and king attacks generate attacks for every piece at every leaf. On the bench, the engine searches about a quarter fewer nodes per second than with material and tables alone. The extra knowledge has to win more than the depth it costs. The SPRT is the judge.
- **Hand-set numbers.** Every value here is a reasonable guess. The next chapter replaces guessing with fitting them to game results.

What it's worth

Test	Result
Stage 15 vs stage 14, 8+0.08, SPRT [0, 5]	+163 −41 =39 over 243 games: passed (about +192 ± 45 Elo)

Further reading

- Chess Programming Wiki: Evaluation, Mobility, Pawn Structure, Passed Pawn, Bishop Pair, Rook on Open File, King Safety.

Texel tuning on self-play games

Stage 16 · tag `stage-16-tuning` *· code in* `crates/engine/src/tuning.rs`, `crates/tools/src/selfplay.rs`, `crates/tools/src/texel.rs`

Chapters 9 and 15 gave cb2026 an evaluation with 411 pairs of numbers: piece values, 384 piece-square table entries, and the positional terms. Every one was set by hand, from chess principles and a feel for what's reasonable. Hand-setting gets the signs right and the magnitudes roughly right. It can't get hundreds of numbers right *together*, since they interact (a passed-pawn bonus and the pawn table both reward advanced pawns). This chapter lets the games decide.

The idea

An evaluation predicts the outcome of a game. Positions it scores +2 pawns should usually be won, positions near 0 should often be drawn. Given a large set of positions from real games, each labelled with the game's result, we can measure how well the evaluation predicts, and adjust the numbers to predict better. This is **Texel tuning**, named after Peter Österlund's engine Texel, which made it popular around 2014.

1. Turn a score into an expected result with a sigmoid: `σ(s) = 1 / (1 + 10^(−K·s/400))`. A score of 0 expects ½, large scores expect

1 or 0.

2. The **error** is the mean squared difference between expected and actual results.
3. Fit `K` once, with the starting values. It only sets the scale between centipawns and results.
4. Adjust the parameters to minimise the error.

```
/// The mean squared error between expected and actual results.
pub fn error(positions: &[TrainingPosition], values: &[(f64, f64)], k: f64) -> f64 {
    let total: f64 = positions
        .iter()
        .map(|p| {
            let expected = sigmoid(tuning::evaluate(&p.features, values), k);
            (p.result - expected).powi(2)
        })
        .sum();
    total / positions.len().max(1) as f64
}
```

The data: cb2026's own games

The positions come from cb2026 playing itself, so no other engine's games or data are used. The `datagen` tool plays thousands of short games:

- each game starts from an opening position plus 8 random moves, so no two games are alike;
- each move gets a fixed 6,000-node search, which is fast and gives the same games on any machine;
- a game ends by the rules, or is **adjudicated** once the search sees a forced mate, or as a draw when the score stays near zero for long enough.

From each game it keeps the **quiet** positions: not in check, best move not a capture or promotion, no mate in sight. The evaluation is only

ever asked about quiet positions, because the quiescence search resolves captures first, so those are the ones it should be fitted to.

```
/// Plays one self-play game from `start` and returns its quiet positions with the
result.
pub fn play_game(engine: &mut Engine, start: &Position, nodes: u64) -> PlayedGame {
    engine.new_game();
    let stop = AtomicBool::new(false);
    let limits = Limits {
        nodes: Some(nodes),
        ..Limits::default()
    };
    let mut game = Game::from_position(start.clone());
    let mut recorded = Vec::new();
    let mut quiet_plies = 0;

    let result = loop {
        // The rules: is the game over?
        if let Some(outcome) = game.outcome() {
            break outcome.winner().map_or(0.5, winner);
        }
        if game.moves().len() >= MAX_PLIES {
            break 0.5;
        }

        let position = game.position();
        let mut score = None;
        let history = game.history_hashes();
        let reply = engine.search(position, &history, &limits, &stop, &mut |info| {
            score = Some(info.score)
        });
        let Some(mv) = reply.best_move else {
            break 0.5;
        };
        match score {
            // A forced mate: adjudicate.
            Some(Score::Mate(moves)) => {
                let mover = position.side_to_move();
                break winner(if moves > 0 { mover } else { !mover });
            }
            Some(Score::Centipawns(cp)) => {
                if is_quiet(position, mv) && cp.abs() < RECORD_SCORE_LIMIT {
                    // The search scores for the side to move; records keep White's
side.
                    let score = match position.side_to_move() {
                        Color::White => cp,
                        Color::Black => -cp,
                    };
                    recorded.push(Recorded {
                        position: position.clone(),
                        score,
                        best_move: mv,
                    });
                }
                quiet_plies = if cp.abs() <= DRAW_ADJUDICATION_SCORE {
                    quiet_plies + 1
                } else {
                    0
                };
```

```
                if game.moves().len() >= DRAW_ADJUDICATION_PLY
                    && quiet_plies >= DRAW_ADJUDICATION_LENGTH
                {
                    break 0.5;
                }
            }
            None => {}
        }
        game.play(mv).expect("the engine plays legal moves");
    };

    PlayedGame {
        positions: recorded,
        result,
    }
}
```

The evaluation as a dot product

Tuning hundreds of parameters on hundreds of thousands of positions needs the error's **gradient**: for each parameter, which direction reduces the error. Computing that by trying each parameter one step up and down (the original Texel method) means hundreds of full passes over the data per step. There is a much faster way.

Every term of cb2026's evaluation has the form "this parameter, this many times, for this side". The pawn on e4 is "pawn piece-square entry e4, once, for White". A knight with 6 safe squares is "knight mobility, 6 − typical times". So the evaluation can report its terms to any **sink** instead of adding them up:

```rust
/// One evaluation parameter: a `(middlegame, endgame)` pair in [`EvalParams`]. Every
term
/// of the evaluation is "this parameter, this many times, for this side".
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
pub enum Param {
    /// A piece's value.
    Material(PieceType),
    /// A piece-square table entry (table index, see [`pst_index`]).
    Pst(PieceType, usize),
    /// Mobility per safe square.
    Mobility(PieceType),
    /// A doubled pawn.
    Doubled,
    /// An isolated pawn.
    Isolated,
    /// A passed pawn on this relative rank.
    Passed(usize),
    /// The bishop pair.
    BishopPair,
    /// A rook on an open file.
    RookOpenFile,
    /// A rook on a semi-open file.
    RookSemiOpenFile,
    /// A pawn in the king's shield.
    PawnShield,
    /// The side to move.
    Tempo,
}

impl Param {
    /// How many parameters there are.
    pub const COUNT: usize = 411;

    /// The parameter's position in a flat list of all parameters, `0..COUNT`.
    pub fn index(self) -> usize {
        match self {
            Param::Material(pt) => pt.index(),
            Param::Pst(pt, square) => 6 + pt.index() * 64 + square,
            Param::Mobility(pt) => 390 + pt.index(),
            Param::Doubled => 396,
            Param::Isolated => 397,
            Param::Passed(rank) => 398 + rank,
            Param::BishopPair => 406,
            Param::RookOpenFile => 407,
            Param::RookSemiOpenFile => 408,
            Param::PawnShield => 409,
            Param::Tempo => 410,
        }
    }

    /// Every parameter, in [`Param::index`] order.
    pub fn all() -> Vec<Param> {
        let mut all = Vec::with_capacity(Param::COUNT);
        all.extend(PieceType::ALL.map(Param::Material));
        for pt in PieceType::ALL {
            all.extend((0..64).map(|square| Param::Pst(pt, square)));
```

```
        }
        all.extend(PieceType::ALL.map(Param::Mobility));
        all.extend([Param::Doubled, Param::Isolated]);
        all.extend((0..8).map(Param::Passed));
        all.extend([
            Param::BishopPair,
            Param::RookOpenFile,
            Param::RookSemiOpenFile,
            Param::PawnShield,
            Param::Tempo,
        ]);
        all
    }
}

impl EvalParams {
    /// The current value of `param`.
    pub fn value(&self, param: Param) -> MgEg {
        let t = &self.terms;
        match param {
            Param::Material(pt) => (self.mg_values[pt.index()],
self.eg_values[pt.index()]),
            Param::Pst(pt, sq) => (self.mg_pst[pt.index()][sq],
self.eg_pst[pt.index()][sq]),
            Param::Mobility(pt) => t.mobility[pt.index()],
            Param::Doubled => t.doubled,
            Param::Isolated => t.isolated,
            Param::Passed(rank) => t.passed[rank],
            Param::BishopPair => t.bishop_pair,
            Param::RookOpenFile => t.rook_open_file,
            Param::RookSemiOpenFile => t.rook_semi_open_file,
            Param::PawnShield => t.pawn_shield,
            Param::Tempo => t.tempo,
        }
    }

    /// Sets `param` to `value`.
    pub fn set_value(&mut self, param: Param, value: MgEg) {
        let t = &mut self.terms;
        match param {
            Param::Material(pt) => {
                self.mg_values[pt.index()] = value.0;
                self.eg_values[pt.index()] = value.1;
            }
            Param::Pst(pt, sq) => {
                self.mg_pst[pt.index()][sq] = value.0;
                self.eg_pst[pt.index()][sq] = value.1;
            }
            Param::Mobility(pt) => t.mobility[pt.index()] = value,
            Param::Doubled => t.doubled = value,
            Param::Isolated => t.isolated = value,
            Param::Passed(rank) => t.passed[rank] = value,
            Param::BishopPair => t.bishop_pair = value,
            Param::RookOpenFile => t.rook_open_file = value,
            Param::RookSemiOpenFile => t.rook_semi_open_file = value,
            Param::PawnShield => t.pawn_shield = value,
            Param::Tempo => t.tempo = value,
```

```
        }
    }
}

/// Receives the evaluation's terms as [`collect`] finds them. The evaluation itself
adds up
/// their values ([`trace`]); the tuner records how often each parameter applies.
pub trait TermSink {
    /// `side` gets parameter `param`, `count` times (`count` may be negative).
    fn term(&mut self, side: Color, param: Param, count: i32);
    /// `side`'s king is in danger: a middlegame penalty that isn't a parameter times
a count
    /// (it's quadratic in the attacks), so it is passed on as a value.
    fn king_danger(&mut self, side: Color, penalty: i32);
}
```

The engine's own evaluation is one sink: it multiplies each term by its value and adds it up. The tuner's sink records **coefficients** instead: for each parameter, how many times White gets it minus how many times Black does. After that, the middlegame and endgame scores are simple sums of coefficient × value, and the derivative of the score with respect to a parameter is just its coefficient (times its share of the phase blend). The engine and the tuner share one implementation of the terms, so they can't drift apart, and a test checks that the coefficients reproduce the evaluation exactly.

King danger is the one term that isn't linear: it is the *square* of the attack count. The tuner treats it as a constant computed with the current values and doesn't tune it.

Fitting: gradient descent with Adam

```
/// The error's gradient with respect to every parameter's middlegame and endgame
value.
pub fn gradient(positions: &[TrainingPosition], values: &[(f64, f64)], k: f64) ->
Vec<(f64, f64)> {
    let mut grad = vec![(0.0, 0.0); values.len()];
    let scale = k * std::f64::consts::LN_10 / 400.0;
    let max_phase = f64::from(MAX_PHASE);
    for p in positions {
        let s = sigmoid(tuning::evaluate(&p.features, values), k);
        // d(error)/d(score) for this position (the mean's 1/N is applied at the end).
        let d_score = -2.0 * (p.result - s) * s * (1.0 - s) * scale;
        let mg_share = f64::from(p.features.phase) / max_phase;
        for &(index, coefficient) in &p.features.coefficients {
            let c = f64::from(coefficient) * d_score;
            let g = &mut grad[index as usize];
            g.0 += c * mg_share;
            g.1 += c * (1.0 - mg_share);
        }
    }
    let n = positions.len().max(1) as f64;
    for g in &mut grad {
        g.0 /= n;
        g.1 /= n;
    }
    grad
}
```

Plain gradient descent struggles here. Material appears in every position, while "knight on a8" appears in a handful, so their gradients differ by orders of magnitude, and one step size can't suit both. **Adam** keeps running averages of each parameter's gradient and of its square, and scales each parameter's step by them. Rare and common parameters then move at similar rates.

```
/// Moves `values` one step against `grad`.
pub fn step(&mut self, values: &mut [(f64, f64)], grad: &[(f64, f64)]) {
    self.steps += 1;
    let correction1 = 1.0 - Self::BETA1.powi(self.steps);
    let correction2 = 1.0 - Self::BETA2.powi(self.steps);
    let update = |value: &mut f64, g: f64, m: &mut f64, v: &mut f64| {
        *m = Self::BETA1 * *m + (1.0 - Self::BETA1) * g;
        *v = Self::BETA2 * *v + (1.0 - Self::BETA2) * g * g;
        let m_hat = *m / correction1;
        let v_hat = *v / correction2;
        *value -= self.learning_rate * m_hat / (v_hat.sqrt() + Self::EPSILON);
    };
    for i in 0..values.len() {
        let (m, v) = (&mut self.first[i], &mut self.second[i]);
        update(&mut values[i].0, grad[i].0, &mut m.0, &mut v.0);
        update(&mut values[i].1, grad[i].1, &mut m.1, &mut v.1);
    }
}
```

A unit test checks the gradient against finite differences, which is the standard guard against a sign or factor error in hand-derived gradients.

Running it

The games were the hand-crafted evaluation's. Since stage 21 `datagen` plays with the built-in network unless told otherwise, so to repeat them, ask for no network (`--evalfile none`):

```
cargo run --release -p cb2026-tools --bin datagen -- --out data.txt --games 8000 --
seed 2026 --evalfile none
cargo run --release -p cb2026-tools --bin datagen -- --out data2.txt --games 16000 --
seed 3033 --evalfile none
cat data.txt data2.txt > data-all.txt
cargo run --release -p cb2026-tools --bin tune -- --data data-all.txt --from hand \
    --epochs 1500 --lambda 3e-9 --min-count 500
```

`tune` writes `crates/engine/src/tuned.rs`, a generated table of all 411 values, available as `EvalParams::tuned()` (and in the shell with `set eval tuned`). The engine's default is still the hand-written set, `EvalParams::hand_set()`: the tuned values didn't measure stronger

(see "What it's worth" below). The data file isn't committed (it's large, and anyone can regenerate it).

What happened: overfitting, then regularisation

The first run used 498,522 positions from 8,000 games and tuned all parameters freely for 3,000 epochs. The error dropped from 0.0845 to 0.0798, which looked like a success. The values said otherwise:

Parameter	Hand-set (mg, eg)	First run (mg, eg)
Knight value	320, 300	346, 321
Knight on g8	−35, −35	**−530, +321**
White king on c8	−50, −25	**+1246, −221**

Material moved sensibly. The piece-square entries for squares a piece almost never visits went wild, because Adam moves every parameter at a similar rate whether a hundred positions support it or a hundred thousand. Those numbers fit the training positions and mean nothing anywhere else. This is **overfitting**.

Two standard remedies fixed it:

- **A minimum number of occurrences.** A parameter seen in fewer than 500 positions keeps its starting value.
- **L2 regularisation.** Each parameter is pulled back toward its starting value with strength λ, so it only moves as far as the evidence pushes it.

```
/// Adds L2 regularisation to `grad`: a pull of strength `lambda` back toward `start`
for every
/// parameter, so a parameter only moves as far as the data can justify. Parameters
seen in
/// fewer than `min_count` positions get no gradient at all and keep their starting
values.
pub fn regularize(
    grad: &mut [(f64, f64)],
    values: &[(f64, f64)],
    start: &[(f64, f64)],
    seen: &[usize],
    lambda: f64,
    min_count: usize,
) {
    for i in 0..grad.len() {
        if seen[i] < min_count {
            grad[i] = (0.0, 0.0);
        } else {
            grad[i].0 += 2.0 * lambda * (values[i].0 - start[i].0);
            grad[i].1 += 2.0 * lambda * (values[i].1 - start[i].1);
        }
    }
}
```

To choose λ honestly, the last 10% of the data (about 50,000 positions, mostly from whole games not used for training) is held out as a **validation set**. Its error shows whether a fit generalises:

λ	Training error	Validation error
(hand-set values)	0.08432	0.08570
0 (no regularisation)	0.07937	0.08704
2×10^{-9}	0.08014	0.08546
1×10^{-8}	0.08099	0.08512
3×10^{-8}	0.08166	**0.08508**
1×10^{-7}	0.08237	0.08514

Without regularisation, the training error fell the most but the validation error *rose*: the fit made predictions on new games worse. The best λ improved validation by only 0.7%. 500,000 positions from 8,000 games are not much evidence for 822 numbers.

So a second batch of 16,000 games was generated (a different random seed), for 1.5 million positions in all, and the comparison repeated:

λ	Training error	Validation error
(hand-set values)	0.08522	0.08467
0	0.08212	0.08368
1×10^{-9}	0.08238	0.08333
3×10^{-9}	0.08260	**0.08328**
1×10^{-8}	0.08294	0.08335
3×10^{-8}	0.08330	0.08351

With three times the data, even the unregularised fit now generalises. The best validation gain doubles to 1.7%, and the best λ is ten times weaker, because the data can now justify larger moves on its own. That's the usual pattern: regularisation matters most when data is scarce. The values in `tuned.rs` come from the $\lambda = 3 \times 10^{-9}$ run.

Pitfalls

- **Overfitting.** With few positions, rare parameters get fitted to noise, as the first run above shows. Hold out a validation set, regularise, and get more data.
- **Correlated data.** Positions from the same game share its result, so 500,000 positions from 8,000 games carry less information than 500,000 independent ones.
- **The engine that made the data.** The games were played by the engine being tuned. Repeating the cycle (generate with the tuned engine, tune again) usually helps a little more.

- **Error isn't Elo.** A lower error is a good sign, not proof. Only games decide, so the tuned values must pass an SPRT like any other change.

What it's worth

Test	Result
Tuned vs hand-set values, 8+0.08, SPRT [0, 5], cloud	+310 −313 =187 over 810 games, −1.3 ± 21.0 Elo: no verdict (LLR −0.17) when the run hit the session's two-hour limit

No measurable gain. The 95% interval runs from about −22 to +20 Elo, so the tuned values might be a little better or a little worse, but they aren't the clear win the lower error promised. That is the "error isn't Elo" pitfall above, met for real. Likely causes:

- **Too little data.** About 1.5 million positions from 24,000 short games is small for 411 parameters; strong engines tune on hundreds of millions.
- **Strong regularisation.** The settings that stopped the overfitting also kept most values close to the hand-set ones, so the evaluation barely changed.
- **The hand-set values were reasonable.** Tuning helps most when the starting point is poor.

So the engine keeps the hand-set values, and this stage contributes the tools: self-play data generation, the tuner, and a tuned set anyone can try (`set eval tuned`, or an SPRT of their own). A later round, with more data from the stronger engine of later stages, can try again. With the hand-set values as default the stage changes nothing in play: the bench is the same as stage 15's.

Further reading

- Chess Programming Wiki: Texel's Tuning Method, Automated Tuning.
- Peter Österlund, the original description of the method on the TalkChess forum (2014).
- Diederik Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015.

Lazy SMP

Stage 17 · tag `stage-17-smp` *· code in* `crates/engine/src/lib.rs` *and* `tt.rs`

Every computer that runs cb2026 has several cores, and until now it used one. Splitting an alpha-beta search across threads is famously hard: the algorithm is sequential at heart, since what one move's search learns (alpha) decides how much of the next move's tree can be skipped. Decades of clever schemes divided the tree between threads and synchronised them carefully. The method nearly every engine uses today is the opposite of clever. It is called **Lazy SMP**.

The idea

Start several threads, and let every one of them search the *same* position with the same iterative deepening. They share exactly one thing: the transposition table.

This sounds like it should waste all but one thread's work, and it would without the shared table. But the threads never stay in step. Each finds results the others haven't reached yet and writes them to the table, and the others then read them and get cutoffs and good moves for free. The threads end up exploring slightly different parts of the tree, filling the table for each other. The main thread decides

when to stop and which move to play. The helpers' work only reaches it through the table.

```
/// A helper thread's search: iterative deepening on the same position until told to
stop.
/// Its results are only used through the transposition table. Odd-numbered helpers
start
/// one ply deeper, so the threads are less often searching the same depth at the same
time.
fn run_helper(
    searcher: &mut Searcher<'_>,
    position: &Position,
    mut root_moves: Vec<Move>,
    max_depth: u32,
    helper: usize,
) {
    let mut previous = None;
    let first = 1 + (helper % 2) as u32;
    for depth in first..=max_depth {
        let result = searcher.search_depth(position, &mut root_moves, depth,
previous);
        if !result.complete {
            break;
        }
        previous = Some(result.score);
        if let Some(best_move) = result.best_move {
            root_moves.retain(|&mv| mv != best_move);
            root_moves.insert(0, best_move);
        }
    }
}
```

Half the helpers start one ply deeper than the others. That small push keeps the threads from searching the same depth at the same moment, which increases how much they help each other.

A table for many threads

The transposition table now has several writers at once, so it has to cope with two threads writing the same slot at the same time. A lock per slot would be slow. cb2026 uses the **lockless hashing** trick of Hyatt and Mann instead:

```
/// One slot of the table: an entry packed into two 64-bit words that threads can read
and
/// write without locks.
///
/// With several search threads (Lazy SMP, stage 17), two threads can write the same
slot at
/// once, and a reader can see half of one write and half of the other: a "torn"
entry. The
/// trick, from Robert Hyatt and Timothy Mann's "A lock-less transposition table
/// implementation for parallel search chess engines" (ICGA Journal, 2002): store the
key
/// XORed with the data. A reader XORs the two words it loaded; it only gets the right
key back
/// if both words come from the same write, so a torn entry simply looks like a miss.
#[derive(Debug, Default)]
struct Slot {
    key_xor_data: AtomicU64,
    data: AtomicU64,
}
```

Each entry is packed into one 64-bit word, and the slot stores that word alongside `key XOR word`:

```
/// Packs everything but the key into one word:
/// move (16 bits) | score (16) | depth (8) | bound (2) | generation (8) | pv (1).
/// A bound of 0 marks an empty slot.
fn pack(
    best_move: Option<Move>,
    score: i32,
    depth: u8,
    bound: Bound,
    generation: u8,
    pv: bool,
) -> u64 {
    let bound_bits: u64 = match bound {
        Bound::Exact => 1,
        Bound::Lower => 2,
        Bound::Upper => 3,
    };
    u64::from(best_move.map_or(0, Move::to_bits))
        | u64::from(score as i16 as u16) << 16
        | u64::from(depth) << 32
        | bound_bits << 40
        | u64::from(generation) << 42
        | u64::from(pv) << 50
}

/// Unpacks a word made by [`pack`] (`None` for an empty slot).
fn unpack(key: u64, data: u64) -> Option<Entry> {
    let bound = match (data >> 40) & 3 {
        1 => Bound::Exact,
        2 => Bound::Lower,
        3 => Bound::Upper,
        _ => return None,
    };
    let move_bits = data as u16;
    Some(Entry {
        key,
        // Move 0 (a1 to a1) is never legal, so it stands for "no move".
        best_move: (move_bits != 0).then(|| Move::from_bits(move_bits)),
        score: i32::from((data >> 16) as u16 as i16),
        depth: (data >> 32) as u8,
        bound,
        generation: (data >> 42) as u8,
        pv: (data >> 50) & 1 == 1,
    })
}
```

A reader loads both words and XORs them back. If another thread was halfway through writing the slot, the reader gets one word from each write, the XOR doesn't give back the key, and the entry is treated as a miss. A torn entry can never be mistaken for a real one, and no thread ever waits. The words are Rust atomics, so all of this is safe

code: no `unsafe` is needed. The stored move is still checked for legality before it is played, as before.

In practice

`setoption name Threads value 4` in UCI, or `set threads 4` in the shell. With one thread (the default) the search is exactly the single-threaded one from the previous chapters.

A rough check on the cloud machine, with Kiwipete and 3 seconds per search (another test was using the other cores, so only the ratio means anything):

Threads	Depth	Nodes per second
1	15	1.18 million
2	16	2.19 million

Twice the threads gives almost twice the nodes per second, but not twice the useful work: much of the helpers' effort overlaps with the main thread's. Lazy SMP is usually measured in Elo against a one-thread copy of the same engine, playing on otherwise idle cores. That needs a machine with more cores than the cloud sessions have, so it runs on the test Mac (`scripts/mac-smp.sh`).

Pitfalls

- **Non-determinism.** With more than one thread, two runs of the same search can differ, since thread timing decides who writes the table first. The bench and every test that compares node counts use one thread.

- **Node counts.** Each helper adds its nodes to a shared counter, so `info nodes` shows the total over all threads. A `go nodes N` limit applies to the main thread only.
- **Torn entries.** Without the XOR check, a reader could combine one thread's key with another's data and use a score for the wrong position.
- **Shared state.** Only the transposition table is shared. Each helper has its own killers, history and search stack, so the threads never contend for anything else.

What it's worth

Test	Result
Stage 17 vs stage 16, 1 thread each, 8+0.08, test Mac	**+0.1 ± 4.1 Elo** over 15,188 games (+4196 −4192 =6800): no regression, SPRT [−5, 0] passed
4 threads vs 1 thread, 8+0.08, test Mac	**+191 ± 99 Elo** over 38 games (+22 −3 =13): a clear gain, SPRT [0, 25] passed

With one thread the search is unchanged except for the table's packing, so about 0 Elo is the expected result. Confirming it took 15,188 games, about 15 hours on the test Mac: the test has to tell 0 from −5 Elo, and when the truth sits right at one of the two hypotheses, the SPRT's evidence grows very slowly. Changes worth +100 Elo or more are decided in a few hundred games.

The thread-scaling test is the opposite case. It only had to show that four threads clearly gain (bounds [0, 25]), and 38 games were enough. The size of the gain is rough, though: the 95% interval runs from about +90 to +290 Elo, and pinning it down would take many more games.

Further reading

- Chess Programming Wiki: Lazy SMP, Parallel Search, Shared Hash Table.
- Robert Hyatt and Timothy Mann, "A lock-less transposition table implementation for parallel search chess engines", *ICGA Journal* 25 (2002).

Tablebases, opening books and Chess960

Stage 18 · tag `stage-18-tablebases` · code in `crates/engine/src/tablebase.rs`, `crates/syzygy` and `crates/chess/src/position.rs`

The search and evaluation of the previous chapters are cb2026's own judgement. This chapter adds knowledge from outside: perfect endgame play from tablebases. It also adds the last part of Chess960 support: the 960 starting positions themselves. And it adds opening books, for playing well-known opening moves without searching.

Endgame tablebases

With seven pieces or fewer on the board, chess is solved. Every such position has been searched to the end, backwards from every checkmate. The results are stored in **tablebases**. The **Syzygy** format, by Ronald de Man, is what nearly every engine uses. It comes in two kinds of file:

- **WDL** tables (win/draw/loss) answer "who wins?". They are compact, and engines probe them during the search.
- **DTZ** tables (distance to zeroing) give the number of moves to the next capture or pawn move on the best path. That is what a won

position needs to make progress under the fifty-move rule. Engines use them at the root to choose the move.

Syzygy results take the fifty-move rule into account, which adds two outcomes. A **cursed win** is a forced mate that takes too long: the fifty-move rule draws it first. A **blessed loss** is the same thing seen from the other side. cb2026 scores both as draws.

The 3- to 5-piece tables take about 1 GB; all 6-piece tables, about 150 GB. The repository doesn't include them. `scripts/download-syzygy.sh <directory> 3-4-5` fetches the 5-piece set, then `setoption name SyzygyPath value <directory>` (or `set syzygy <directory>` in the shell) loads it.

Who reads the files

The Syzygy format is an intricate compressed encoding, and a prober is thousands of lines of careful code with no strength to gain from writing it again. So this is the one place where cb2026 uses someone else's chess code: the `shakmaty-syzygy` library (GPL-3.0, the prober behind Lichess's tablebase server). The `docs/PRD.md` section "Tablebase probing" records that decision. cb2026 itself is `MIT OR Apache-2.0`, so the prober also sits behind a cargo feature, `syzygy`: a binary built with it is GPL-3.0 as a whole, and one built without it has no GPL code (README, "License").

The search never sees it. The engine crate defines a small trait, and the separate `cb2026-syzygy` crate implements it:

```
/// When to probe.
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub struct TablebaseParams {
    /// Probe in the search only with at least this much depth left (the UCI
    /// `SyzygyProbeDepth`; 1 probes at every node).
    pub probe_depth: u32,
    /// Probe only positions with at most this many pieces (`SyzygyProbeLimit`).
    pub probe_limit: u32,
}

impl Default for TablebaseParams {
    fn default() -> TablebaseParams {
        TablebaseParams {
            probe_depth: 1,
            probe_limit: 7,
        }
    }
}
```

The engine crate does no I/O, so it can't open files itself. The program that embeds it (the `cb2026` binary) hands it a loader function. Tests use a tiny in-memory "tablebase" instead, which says that king and queen against king is won, and that's enough to test the search integration without any files.

Probing in the search

```
// Endgame tablebases: right after a capture or pawn move, a covered position's result
// is known exactly.
if let Some((tb, tb_params)) = self.tablebase
    && ply > 0
    && depth >= tb_params.probe_depth
    && pos.halfmove_clock() == 0
    && tablebase::covers(tb, &tb_params, pos)
    && let Some(wdl) = tb.probe_wdl(pos)
{
    self.tb_hits += 1;
    self.note(Note::Tablebase);
    let score = tablebase::score(wdl, ply);
    if self.techniques.tt {
        let stored = score_to_tt(score, ply);
        self.tt
            .store(pos.hash(), None, stored, MAX_PLY, Bound::Exact, tt_pv);
    }
    return score;
}
```

Three conditions decide when to probe:

- **Few enough pieces.** At most what the tables cover (and the `SyzygyProbeLimit` option).
- **No castling rights.** The tables don't include them.
- **Right after a capture or pawn move.** A WDL result assumes the fifty-move counter has just been reset. Probing then also catches the moment the search crosses into tablebase territory, which is the useful moment.

A win scores `TB_WIN - ply`: above every evaluation and below every real mate, so the engine still prefers an actual checkmate, and a nearer tablebase win over a farther one. The result goes into the transposition table, and `info ... tbhits` counts the probes that answered.

At the root

```
// In a tablebase position, keep only the move the tables recommend: it keeps the best
// result and makes progress under the fifty-move rule. The search still runs (and
// reports), but can only choose that move.
if let Some(tb) = &self.tablebase
    && tablebase::covers(tb.as_ref(), &self.tablebase_params, position)
    && let Some((_, mv)) = tb.probe_root(position)
    && root_moves.contains(&mv)
{
    root_moves = vec![mv];
}
```

When the game itself is in the tables, the DTZ tables pick the move. Searching wouldn't help: a search sees "won" everywhere and has no reason to make progress, so it can shuffle until the fifty-move rule draws the game. The tables' move always keeps the best result and heads for the next zeroing move.

Chess960

Chess960 (Fischer Random Chess) shuffles the back rank: the bishops on opposite colours, the king somewhere between the rooks, the black pieces mirroring the white ones. That gives 960 possible starting positions. cb2026 handled Chess960 castling from stage 2 (castling is "king takes own rook", and the rights are stored as rook files) and Shredder-FEN from stage 1. What was missing was the starting positions themselves.

They are numbered 0 to 959 (Reinhard Scharnagl's numbering; number 518 is the standard position), and the number decodes into a back rank one piece at a time:

```
/// Chess960 starting position number `n` (0–959, Scharnagl's numbering; 518 is the
standard
/// starting position), or `None` if `n` is out of range.
///
/// The back rank is built from `n` in mixed-radix steps: the light-squared bishop (4
/// choices), the dark-squared bishop (4), the queen on one of the 6 empty squares,
the two
/// knights on 2 of the 5 left (10 ways), and the rooks and king, in that order, on
the last
/// three, so the king always stands between its rooks.
pub fn chess960(n: u32) -> Option<Position> {
    if n >= 960 {
        return None;
    }
    let mut rank = [None; 8];
    let mut n = n as usize;
    rank[2 * (n % 4) + 1] = Some('B');
    n /= 4;
    rank[2 * (n % 4)] = Some('B');
    n /= 4;
    let empty = |rank: &[Option<char>; 8]| -> Vec<usize> {
        (0..8).filter(|&i| rank[i].is_none()).collect()
    };
    rank[empty(&rank)[n % 6]] = Some('Q');
    n /= 6;
    const KNIGHTS: [(usize, usize); 10] = [
        (0, 1),
        (0, 2),
        (0, 3),
        (0, 4),
        (1, 2),
        (1, 3),
        (1, 4),
        (2, 3),
        (2, 4),
        (3, 4),
    ];
    let (first, second) = KNIGHTS[n];
    let free = empty(&rank);
    rank[free[first]] = Some('N');
    rank[free[second]] = Some('N');
    for (square, piece) in empty(&rank).into_iter().zip(['R', 'K', 'R']) {
        rank[square] = Some(piece);
    }
    let white: String = rank
        .iter()
        .map(|p| p.expect("every square filled"))
        .collect();
    // Shredder-FEN castling: the files of the rooks.
    let rook_files: Vec<char> = (0..8)
        .filter(|&i| rank[i] == Some('R'))
        .map(|i| (b'A' + i as u8) as char)
        .collect();
    let castling: String = rook_files
        .iter()
        .rev()
```

```
        .copied()
        .chain(rook_files.iter().rev().map(|c| c.to_ascii_lowercase()))
        .collect();
    let fen = format!(
        "{}/pppppppp/8/8/8/8/PPPPPPPP/{white} w {castling} - 0 1",
        white.to_ascii_lowercase()
    );
    Position::from_fen(&fen).ok()
}
```

A test builds all 960 and checks that they are distinct and legal, with the bishops on opposite colours and the king between the rooks.

- In the shell, `new960 [n]` starts from position `n`, or a random one.
- `match --chess960` plays test matches from the 960 positions and sets `UCI_Chess960` in both engines. A first run played Chess960 games with castling to completion, with no illegal moves.

Opening books

At the start of a game the search spends its time rediscovering well-known opening theory, and it plays the same move every time from the same position. An **opening book** maps positions to good moves, each with a weight. With `OwnBook` on, the engine looks the position up before searching and plays a book move at once, chosen at random in proportion to the weights:

```
// The opening book: play its move at once (unless asked to analyse).
if self.own_book
    && !limits.infinite
    && limits.search_moves.is_empty()
    && let Some(book) = &self.book
{
    // The dice: the clock's nanoseconds, mixed with the position.
    let nanos = std::time::SystemTime::now()
        .duration_since(std::time::UNIX_EPOCH)
        .map_or(0, |d| d.subsec_nanos());
    let random = u64::from(nanos) ^ position.hash();
    if let Some(mv) = book.pick(position, random)
        && root_moves.contains(&mv)
    {
        return SearchResult {
            best_move: Some(mv),
            ..SearchResult::default()
        };
    }
}
```

cb2026 reads the **Polyglot** format (`.bin` files), which nearly every engine and GUI supports. A book is a sorted list of 16-byte entries: position key, move, weight and some learning data the format reserves, all big-endian. Looking a position up is a binary search for its key:

```
/// The book's moves for `pos`, heaviest first (illegal ones, from a corrupt book or a
key
/// collision, are skipped).
pub fn moves(&self, pos: &Position) -> Vec<BookMove> {
    let key = polyglot_key(pos);
    let start = self.entries.partition_point(|e| e.0 < key);
    let mut moves: Vec<BookMove> = self.entries[start..]
        .iter()
        .take_while(|e| e.0 == key)
        .filter_map(|&(_, raw, weight)| decode_move(pos, raw).map(|mv| BookMove { mv,
weight }))
        .collect();
    moves.sort_by_key(|m| std::cmp::Reverse(m.weight));
    moves
}
```

The key is a Zobrist hash (chapter 4), with one difference that matters: the random numbers are fixed by the format. Any program can then compute the same key for the same position and read any book. The 781 numbers (768 for pieces on squares, 4 for castling rights, 8 for en

passant files, 1 for the side to move) were published with Fabien Letouzey's PolyGlot program. cb2026 takes them from the format's published description and checks them against its test hashes:

```
/// The Polyglot key of `pos`: the XOR of the format's random numbers for every piece
on its
/// square, each castling right, the en passant file (only when a pawn could actually
capture
/// there) and White to move.
pub fn polyglot_key(pos: &Position) -> u64 {
    let mut key = 0;
    for square in pos.occupied() {
        let piece = pos.piece_at(square).expect("an occupied square");
        // The format's piece kinds: black pawn, white pawn, black knight, white
knight, ...
        let kind = 2 * piece.piece_type.index() + usize::from(piece.color ==
Color::White);
        key ^= POLYGLOT_KEYS[64 * kind + square.index()];
    }
    let rights = [
        (Color::White, CastlingSide::KingSide),
        (Color::White, CastlingSide::QueenSide),
        (Color::Black, CastlingSide::KingSide),
        (Color::Black, CastlingSide::QueenSide),
    ];
    for (i, (color, side)) in rights.into_iter().enumerate() {
        if pos.castling().has(color, side) {
            key ^= POLYGLOT_KEYS[768 + i];
        }
    }
    // cb2026 only records an en passant square when an enemy pawn can capture there,
which is
    // exactly the Polyglot rule.
    if let Some(square) = pos.en_passant() {
        key ^= POLYGLOT_KEYS[772 + square.file().index()];
    }
    if pos.side_to_move() == Color::White {
        key ^= POLYGLOT_KEYS[780];
    }
    key
}
```

One detail is easy to get wrong: the en passant file only counts when a pawn of the side to move can actually make the capture. cb2026's positions already record en passant squares that way (chapter 4), for the same reason: equal positions must hash equally.

Polyglot encodes castling as "king takes rook" (`e1h1`), the same convention cb2026 uses internally, so castling moves need no special treatment.

The book is **off by default**. Tests need to see the engine's own moves, a book would make every test game start alike, and GUIs and tournament managers usually supply their own books. In the shell:

```
> set book /path/to/book.bin
Opening book with 6 entries.
> book
Book moves:
  e4       weight    50  (50%)
  d4       weight    30  (30%)
  Nf3      weight    20  (20%)
The engine ignores the book (set ownbook on to use it).
```

The repository has no book, but the v1.0.0 release comes with one: September 2026's, made from Lichess's games as below (`cb2026-lichess-2026-09.bin`). Any Polyglot book works with cb2026, and it can make its own.

Making a book from people's games

Lichess publishes every rated game played on it, a month to a file, under the Creative Commons CC0 licence: about 90 million games and 30 GB a month. The `make-book` tool turns games like these into a Polyglot book. It reads them from standard input, so the month is decompressed on the way in and never stored:

```
scripts/lichess-book.sh 2026-09 ~/books/lichess-2026-09.bin
```

It keeps the games where both players are rated 2200 or more, at blitz or slower, with no bots. Lichess's files include engines playing each other on BOT accounts, at ratings well above any person's: in the first 150,000 games of September 2026, bots played most of the games

between two players rated 2600 or more. Of the whole month, 1.7% of the games passed: 1,499,615 of 89,616,462. For each position in their first 20 plies, the tool counts the move played and what it scored for the side that played it. Positions are found by their Polyglot key, so moves reached by different move orders add up:

```
/// Counts the first `plies` plies of `game`, which ended `result` ("1-0", "0-1" or
/// "1/2-1/2").
pub fn add_game(&mut self, game: &PgnGame, result: &str, plies: usize) {
    // Points for White and for Black: 2 for a win, 1 for a draw, 0 for a loss.
    let points = match result {
        "1-0" => [2, 0],
        "0-1" => [0, 2],
        _ => [1, 1],
    };
    let mut pos = game.start.clone();
    for &mv in game.moves.iter().take(plies) {
        let stats = self
            .moves
            .entry((polyglot_key(&pos), encode_move(mv)))
            .or_default();
        match points[pos.side_to_move().index()] {
            2 => stats.wins += 1,
            1 => stats.draws += 1,
            _ => stats.losses += 1,
        }
        pos.make_move(mv);
    }
    self.games_used += 1;
}
```

A move goes into the book when at least 20 games played it and it scored at least 35%. Its weight is its points, two for a win and one for a draw, as in PolyGlot's own `make-book`. The engine then plays popular moves that do well most often, and a move nobody wins with not at all:

```
/// The book: the moves played in at least `min_games` games that scored at least
/// `min_score` and won some points, each weighted by its points (scaled down where a
/// position's points don't fit the format's 16 bits).
pub fn book(&self, settings: &BookSettings) -> OpeningBook {
    let mut by_position: HashMap<u64, Vec<(u16, u32)>> = HashMap::new();
    for (&(key, mv), stats) in &self.moves {
        if stats.games() >= settings.min_games
            && stats.score() >= settings.min_score
            && stats.points() > 0
        {
            by_position
                .entry(key)
                .or_default()
                .push((mv, stats.points()));
        }
    }
    let mut entries = Vec::new();
    for (key, moves) in by_position {
        let most = moves.iter().map(|&(_, points)| points).max().unwrap_or(0);
        let scale = f64::from(u16::MAX) / f64::from(most.max(u32::from(u16::MAX)));
        for (mv, points) in moves {
            let weight = (f64::from(points) * scale).round().max(1.0) as u16;
            entries.push((key, mv, weight));
        }
    }
    OpeningBook::from_entries(entries)
}
```

The minimum number of games matters more than it looks. A blitz game is usually decided long after the opening, so one game says almost nothing about its first moves. Twenty games of the same move begin to.

September 2026's book took 58 minutes to make, nearly all of it downloading the 29 GB file. Of the 8,036,080 different moves the tool counted, in all the positions it saw, 97,432 passed, a book of 1.6 MB: most of the rest were played in fewer than 20 games. From the starting position the engine plays 1.e4 47% of the time, 1.d4 33%, 1.Nf3 9% and 1.c4 6%; 1.b3, 1.g3 and 1.f4 about 1% each; and 13 more moves, 1.Na3 among them, less than half a percent each. Those passed the test, played often enough and scoring well enough, and their small weight keeps them rare.

Pitfalls

- **Probing too often.** Every probe reads from disk (or the operating system's file cache). `SyzygyProbeDepth` stops probes near the leaves, where they cost more than they save.
- **Shuffling in won endings.** A search that knows "won" but not "how" can draw a won game under the fifty-move rule. That's why the root uses the DTZ tables.
- **Castling rights.** A position with castling rights is never in the tables, even with few pieces.
- **Chess960 notation.** With `UCI_Chess960` on, castling is sent as king-takes-rook (`e1h1`), and GUIs expect it. cb2026 accepts both notations on input.

What it's worth

Test	Result
Without tables or book loaded	No functional change (bench unchanged)
With the 3-4-5 piece tables	Measured on the test Mac (`SyzygyPath` set for one engine)

Further reading

- Chess Programming Wiki: Syzygy Bases, Endgame Tablebases, Chess960, Opening Book, PolyGlot.
- syzygy-tables.info, to look up any tablebase position.

Strength levels

Stage 19 · tag `stage-19-levels` *· code in* `crates/engine/src/levels.rs` *and* `lib.rs`*. A person's horizon and the human scale came later (cjus/chessengine#151, for cjus/chessgui#113), with the* `mistakes` *tool in* `crates/tools/src/mistakes.rs`*.*

Everything so far has made cb2026 stronger. This chapter makes it *weaker*, on purpose, for the many players who want an opponent they can beat sometimes. Done badly, this is easy and unpleasant to play against. Done well, the engine feels like a human of the chosen strength.

Why "search less" isn't enough

The obvious knob is depth or nodes: give the engine 500 nodes instead of 5 million. It does get weaker, but in an odd way. Even a tiny search, with the quiescence search behind it, sees every hanging piece and every one-move tactic. Its mistakes are the subtle long-term ones a strong player makes, while a club player's are the tactical oversights the engine never makes. Its moves look like a strong player's whose judgement has gone strange.

Two knobs, combined

cb2026's levels combine a smaller search with **choosing among the top moves**:

```
/// The settings at a strength `level` from 0 to [`FULL_STRENGTH`], which can fall
between two
/// levels: `UCI_Elo` maps onto in-between strengths.
pub fn settings_at(level: f64) -> LevelSettings {
    if level >= f64::from(FULL_STRENGTH) {
        return LevelSettings {
            nodes: None,
            candidates: 1,
            margin: 0,
            temperature: 0.0,
            horizon_chance: 0.0,
            horizon_depth: 0,
        };
    }
    let level = level.max(0.0);
    // 1.5 to the whole levels, then to the fraction, so that a whole level's node
count is exact.
    let growth = 1.5f64.powi(level.floor() as i32) * 1.5f64.powf(level.fract());
    LevelSettings {
        // 400 nodes at level 0, growing by half each level: about 890,000 at level
19.
        nodes: Some((400.0 * growth) as u64),
        candidates: 4,
        margin: (300.0 - 15.0 * level).round() as i32,
        temperature: 120.0 - 6.0 * level,
        // A whole level's: the depth can't be a fraction, and the chance jumps where
the depth
        // changes, so between two levels the lower one's.
        horizon_chance: HORIZON_CHANCES[(level.floor() as
usize).min(HORIZON_CHANCES.len() - 1)],
        horizon_depth: if level.floor() < TWO_PLY_HORIZON_FROM {
            1
        } else {
            2
        },
    }
}
```

- **Nodes per move** grow by half with each level: 400 at level 0, about 23,000 at level 10, about 890,000 at level 19. Level 20 is full strength, with no limit.
- **Candidates.** The engine searches the best 4 moves (using MultiPV, below), and then picks one at random, weighting each by how

close it is to the best:

```
/// Chooses a move from `candidates` (move and score, best first) for a level, using
`random`
/// (any 64-bit number) as the dice. Each move within the margin is weighted by
/// `exp(-(best - score) / temperature)`.
pub fn choose(candidates: &[(Move, i32)], settings: &LevelSettings, random: u64) ->
Option<Move> {
    let &(first, best) = candidates.first()?;
    if settings.temperature <= 0.0 {
        return Some(first);
    }
    let weights: Vec<(Move, f64)> = candidates
        .iter()
        .filter(|&&(_, score)| best - score <= settings.margin)
        .map(|&(mv, score)| {
            (
                mv,
                (-(f64::from(best - score)) / settings.temperature).exp(),
            )
        })
        .collect();
    let total: f64 = weights.iter().map(|(_, w)| w).sum();
    // A number in [0, total).
    let mut roll = (random >> 11) as f64 / (1u64 << 53) as f64 * total;
    for &(mv, weight) in &weights {
        if roll < weight {
            return Some(mv);
        }
        roll -= weight;
    }
    Some(first)
}
```

A move scoring the same as the best is as likely as the best. One a few dozen centipawns worse is still common at low levels. One that throws away a piece is outside the **margin** and is never played. The weak levels make inaccuracies, and more of them the lower the level, instead of random blunders.

That turned out to be too tidy. People don't only make inaccuracies: they blunder, often (below). Every candidate above comes from a search that sees the short tactics, so a level seldom left a piece where it could be taken, and seldom missed one left for it. Carlos played the level then rated 1050 and found it "much stronger than a 1050 level

player" (cjus/chessgui#113). Counted the way Lichess counts, it blundered 0.7 times in 40 moves; people rated about 1000 on Lichess blunder 3.4 times.

A person's horizon

People's worst mistakes come from how far they look. A beginner moves a piece without seeing the reply, and it is taken. A stronger player sees the reply but not the move after it, and walks into a fork or a pin. So a weaker level chooses many of its moves **with a person's horizon**: a search of exactly one ply below level 8, the move alone, or two from level 8, the move and the reply, with no quiescence search, no transposition table and nothing pruned (`Techniques::horizon`). The positions at the end are evaluated as they stand, in the middle of any exchange, as a hurried eye sees them. Then the level chooses among that search's best moves as usual.

```
/// Whether this move is chosen with a person's horizon, using `random` (any 64-bit
number) as
/// the dice: the plies to search if so, `None` for the level's usual search.
pub fn horizon(settings: &LevelSettings, random: u64) -> Option<u32> {
    // A number in [0, 1).
    let roll = (random >> 11) as f64 / (1u64 << 53) as f64;
    (roll < settings.horizon_chance).then_some(settings.horizon_depth)
}
```

Such a move is often fine: it is a blunder only when there was a tactic to miss, which is when people blunder too. How often a level looks that way is fitted to people's blunders, level by level (below):

```
/// The chance at each level, 0 to 19, that a move is chosen with a person's horizon:
fitted with
/// the `mistakes` tool so that each level blunders as often as [`PEOPLE_BLUNDERS`]
says (see the
/// module docs). It rises at [`TWO_PLY_HORIZON_FROM`], where a horizon that sees the
reply makes
/// fewer blunders per move. Full strength: never.
///
/// The fit, in three rounds of 40 self-play games a level, each move judged at
100,000 nodes:
/// a level blunders about as often as its usual search does (2.8 times in 40 moves at
level 0
/// with the horizon off, none by level 18), plus the chance times what a horizon move
adds
/// (about 5 blunders in 40 with one ply, 3 to 4 with two). Each round set each
level's chance
/// from how far the last missed, and the last is smoothed into a straight line
through levels 0
/// to 7 and another through 8 to 19, since one level's 40 games carry about ± 0.25
blunders of
/// noise (chapter 19 has the rounds).
pub const HORIZON_CHANCES: [f64; FULL_STRENGTH as usize] = [
    0.78, 0.73, 0.68, 0.64, 0.59, 0.55, 0.50, 0.46, 0.70, 0.70, 0.69, 0.65, 0.62,
0.58, 0.55, 0.51,
    0.48, 0.44, 0.41, 0.37,
];
```

Into the search

The dice are rolled first, beside the level's node limit. A move with a person's horizon gets a depth limit of its one or two plies, and never in analysis: an infinite search is someone asking what the position holds, not a level playing a game.

```
// A weaker level searches fewer nodes and several lines to choose from.
let level = levels::settings_at(self.level());
let mut level_limits = limits.clone();
if let Some(nodes) = level.nodes {
    level_limits.nodes = Some(limits.nodes.map_or(nodes, |n| n.min(nodes)));
}
// Now and then, only as far ahead as a person would look (see `levels`), on one
// thread. Never when analysing.
let horizon = (!limits.infinite)
    .then(|| levels::horizon(&level, dice(position)))
    .flatten();
if let Some(plies) = horizon {
    level_limits.depth = Some(plies);
}
let limits = &level_limits;
let lines = self.multipv.max(level.candidates).max(1);
// The levels were calibrated with the hand-crafted evaluation (chapter 19), so below
// full strength the search keeps it: their ratings stay true. Calibrating them again
// with a network is for later.
let full_strength = self.level() >= f64::from(levels::FULL_STRENGTH);
```

The dice are the clock's nanoseconds mixed with the position's hash, the same as for the choice among the best moves, so a level goes its own way in each game:

```
/// Dice for a weaker level's choices: the clock's nanoseconds mixed with the
position, so two
/// games from the same position go their own ways.
fn dice(position: &Position) -> u64 {
    let nanos = std::time::SystemTime::now()
        .duration_since(std::time::UNIX_EPOCH)
        .map_or(0, |d| d.subsec_nanos());
    (u64::from(nanos) ^ position.hash()).wrapping_mul(0x9E37_79B9_7F4A_7C15)
}
```

Then the search runs with another set of techniques, on one thread:

```
let techniques = match horizon {
    Some(_) => Techniques::horizon(),
    None => *techniques,
};
let threads = if horizon.is_some() { 1 } else { *threads };
```

`Techniques::horizon` starts from `Techniques::sound`, the set that only makes the search faster and never changes its result: alpha-beta, move ordering and PVS, with null move, reductions, extensions and every kind of pruning off. So no move is skipped and none is

searched deeper for being interesting. It takes away three more, the first of which would look past the horizon:

```
/// A person's look at a position, for a weaker level's move with a person's horizon
/// ([`crate::levels::horizon`]): exactly the plies the search is given, every move at
each,
/// and the positions at the end evaluated as they stand. [`sound`](Techniques::sound)
/// without the quiescence search, aspiration windows or the transposition table,
whose
/// shallow, unresolved scores would mislead the searches after it.
pub fn horizon() -> Techniques {
    Techniques {
        quiescence: false,
        aspiration: false,
        tt: false,
        ..Techniques::sound()
    }
}
```

- **The quiescence search** (chapter 7), which would otherwise play out the captures at the horizon. Without it, the position after a queen takes a defended pawn is evaluated a pawn up, whatever recaptures: that is what a hurried eye sees, and the whole point.
- **Aspiration windows** (chapter 11), which start each iteration around the last one's score. At one or two plies there is no last score worth trusting.
- **The transposition table** (chapter 8). A horizon search's scores are unresolved and shallow; written to the table, they would be read as real results by the full searches of the moves after it. So it neither reads nor writes the table.

One thread, because Lazy SMP's helpers (chapter 17) share their work only through the transposition table, which this search doesn't use: they would search for nothing. Everything after the search is as before: it returns its best lines, and `choose` picks one with the level's margin and temperature.

MultiPV

To choose among the best moves, the engine has to know the best *several* moves, not just the best one. **MultiPV** (multiple principal variations) searches the root once per line: line 1 normally, line 2 with line 1's move left out, line 3 with both left out, and so on.

```
/// The main thread's iterative deepening: it decides when to stop, reports progress,
and
/// returns the best lines of the last depth it searched, best first.
///
/// With several lines (MultiPV), each depth searches the root once per line, leaving
out the
/// moves already chosen as better lines at that depth.
fn main_search(
    searcher: &mut Searcher<'_>,
    job: MainSearch<'_>,
    on_info: &mut dyn FnMut(&SearchInfo),
) -> Vec<RootResult> {
    let MainSearch {
        position,
        mut root_moves,
        max_depth,
        start,
        budget,
        time,
        helper_nodes,
        lines: wanted,
        reported,
    } = job;
    let wanted = wanted.min(root_moves.len()).max(1);
    let mut lines: Vec<RootResult> = Vec::new();
    // For the time manager: how many depths in a row the best move has stayed the
same.
    let mut stable = 0;
    for depth in 1..=max_depth {
        let mut this_depth: Vec<RootResult> = Vec::new();
        let mut complete = true;
        for k in 0..wanted {
            let mut moves: Vec<Move> = root_moves
                .iter()
                .copied()
                .filter(|&mv| !this_depth.iter().any(|line| line.best_move ==
Some(mv)))
                .collect();
            let previous = lines.get(k).map(|line| line.score);
            let result = searcher.search_depth(position, &mut moves, depth, previous);
            // A stopped iteration still counts if it finished at least one move: the
first
            // move searched is the previous best, so its result is at least as well-
founded.
            if result.complete || result.score > -INFINITY {
                this_depth.push(result.clone());
            }
            if !result.complete {
                complete = false;
                break;
            }
        }
        // A later line can score higher than an earlier one (it is searched with its
own
        // window, and the transposition table has learned more by then), so the lines
are
```

```
        // sorted, best first, before they are reported. The sort is stable: equal
scores keep
        // the order they were found in.
        this_depth.sort_by_key(|line| std::cmp::Reverse(line.score));
        for (k, line) in this_depth.iter().enumerate().take(reported) {
            on_info(&SearchInfo {
                depth,
                seldepth: line.seldepth,
                score: to_score(line.score),
                nodes: line.nodes + helper_nodes.load(Ordering::Relaxed),
                time: start.elapsed(),
                pv: line.pv.clone(),
                hashfull: searcher.hashfull(),
                tbhits: searcher.tb_hits(),
                multipv: k as u32 + 1,
            });
        }
        // The best move and score before this depth, for the time manager.
        let before = lines.first().map(|line| (line.best_move, line.score));
        if !this_depth.is_empty() {
            lines = this_depth;
        }
        if !complete {
            break;
        }
        // Next depth: the lines' moves first, best first.
        for line in lines.iter().rev() {
            if let Some(mv) = line.best_move {
                root_moves.retain(|&m| m != mv);
                root_moves.insert(0, mv);
            }
        }
        // A mate found at this depth can't be beaten by searching deeper.
        if wanted == 1 && lines[0].score.abs() >= MATE - depth as i32 {
            break;
        }
        // Starting another depth now would probably not finish in time. How soon
"now" is
        // depends on how settled the search is (see `time::adapted_soft`).
        let (best_move, score) = (lines[0].best_move, lines[0].score);
        let fall = match before {
            Some((previous_move, previous_score)) => {
                stable = if previous_move == best_move {
                    stable + 1
                } else {
                    0
                };
                previous_score - score
            }
            None => 0,
        };
        let soft = time::adapted_soft(&budget, stable, fall, time);
        if soft.is_some_and(|soft| start.elapsed() >= soft) {
            break;
        }
        if searcher.aborted() {
            break;
```

```
        }
    }
    lines
}
```

MultiPV is also useful in its own right: in analysis, GUIs show the top lines side by side (`setoption name MultiPV value 3`, or `set multipv 3` in the shell). Each line costs nearly a full search, so three lines take about three times as long.

Options

Option	What it does
`Skill Level` 0–20	The level; 20 is full strength
`UCI_LimitStrength` + `UCI_Elo` 600–2500	Limit to a rating on the levels' human scale (below)
`MultiPV` 1–256	Lines searched and reported

In the shell, `level 5`, `elo 1300` and `elo off` do the same, and say what the engine now plays at:

```
> level 5
Strength: level 5, about 1100 on Lichess (rapid, by blunders): 3037 nodes per move,
choosing among the best 4 moves; a person's horizon (1 ply) on 55% of its moves
> level 20
Strength: full
> elo 1350
Strength: level 7.5, about 1350 on Lichess (rapid, by blunders): 8370 nodes per move,
choosing among the best 4 moves; a person's horizon (1 ply) on 46% of its moves
```

`Skill Level` caps `UCI_Elo`: with `level 5` set, `elo 1350` still plays at level 5.

A human scale

What rating is level 5? Until cjus/chessgui#113 the answer was a rating against other engines, measured in games (the next section).

But a rating against engines says little about what a person meets: engines punish every blunder at once, people often don't, so a level that blunders like a person rated 1000 rates far lower against engines, and a level rated 1000 against engines plays far more cleanly than a person rated 1000. Carlos chose a human scale.

People's blunders. Lichess publishes every rated game it hosts (https://database.lichess.org/), and about one in twelve carries the evaluation its analysis gave each move. Lichess judges a move by how much it lowers the mover's winning chances, from −1 to 1: a drop of 0.1 is an inaccuracy, 0.2 a mistake, 0.3 a blunder, with the chances from centipawns through a curve fitted to its games (https://lichess.org/page/accuracy). The `mistakes` tool counts the same way (`cb2026_tools::mistakes`). Its `--people` mode, on 60,000 rated rapid games of January 2025 with evaluations, gave, per 40 moves of one player:

Lichess rating	600	800	1000	1200	1400	1600	1800	2000	2200	2400
Blunders	4.6	3.9	3.4	3.0	2.8	2.4	2.1	1.9	1.5	0.9
Mistakes	2.0	1.8	1.7	1.7	1.7	1.6	1.5	1.3	1.2	0.9
Inaccuracies	4.2	3.9	3.9	3.8	3.9	3.6	3.7	3.4	3.0	2.5

People blunder two to four times a game across nearly the whole range, and the rate falls slowly with rating. The levels did far less: 2.8 at level 0, 0.7 at level 6, 0.2 at level 12 and none by level 18, judged by a full-strength search of 100,000 nodes in 32 games each.

Counting as Lichess counts

Every count above, people's and the levels', comes from the same two functions. An evaluation becomes the mover's winning chances

through Lichess's curve, with centipawns capped at ±1000 and a mate in n counted as (21 − min(10, n)) × 100 centipawns, so a mate soon is a little surer than a mate later. A move is judged by how far it lowers them:

```
/// The mover's winning chances, -1 to 1, for an evaluation from its side.
pub fn winning_chances(score: Score) -> f64 {
    let cp = match score {
        Score::Centipawns(cp) => cp.clamp(-CENTIPAWN_CAP, CENTIPAWN_CAP),
        Score::Mate(moves) => (21 - moves.abs().min(10)) * 100 * moves.signum(),
    };
    2.0 / (1.0 + (-WINNING_CHANCES_SLOPE * f64::from(cp)).exp()) - 1.0
}

/// How bad a move was: 0 for none of these, 1 an inaccuracy, 2 a mistake, 3 a
blunder, by the
/// drop in the mover's winning chances.
pub fn classify(drop: f64) -> usize {
    THRESHOLDS.iter().filter(|&&at| drop >= at).count()
}
```

The curve's slope, the cap and the thresholds are Lichess's (its accuracy page and lila's source, AGPL-3.0: the numbers only, none of its code). The curve is steep near equality and flat in decided positions, so throwing away a pawn in a level game counts, and throwing away a rook in a game already lost barely does, as on Lichess.

People are counted in Lichess's own games. Each game's moves are one line of the PGN file, with the evaluation Lichess's analysis gave after each move in a comment, `{ [%eval 0.17] }` in pawns from White's side, or `[%eval #-3]` for a mate. Each evaluation is White's chances after one move and before the next; White's fall when White errs, and Black's (the same, negated) when Black does. A move is counted for its player's rating band:

```
let mut before = winning_chances(Score::Centipawns(START_EVALUATION));
for (ply, text) in line.split("[%eval ").skip(1).enumerate() {
    let Some(after) = text
        .split(']')
        .next()
        .and_then(white_score)
        .map(winning_chances)
    else {
        break;
    };
    // White's chances: Black's are the same, negated.
    let (player, drop) = if ply % 2 == 0 {
        (white, before - after)
    } else {
        (black, after - before)
    };
    tallies
        .entry((speed, player / band * band))
        .or_insert_with(Tally::default)
        .add(classify(drop));
    before = after;
}
```

A level plays itself from the same openings and random first moves as the self-play tools, and a full-strength search of a fixed number of nodes judges every move. Each position is searched once: its score is the "after" of one move and, from the other side, the "before" of the next. A game's end is scored as it stands, a mate as lost, a draw as even:

```
let mut game = Game::from_position(start.clone());
let mut tally = Tally::default();
let mut before = evaluate(judge, &game);
while game.outcome().is_none() && game.moves().len() < MAX_PLIES {
    let history = game.history_hashes();
    let reply = player.search(
        game.position(),
        &history,
        &Limits::default(),
        &stop,
        &mut |_| {},
    );
    let Some(mv) = reply.best_move else {
        break;
    };
    game.play(mv).expect("the engine plays legal moves");
    // The position after, from the opponent's side: its chances are the mover's,
negated.
    let after = if let Some(outcome) = game.outcome() {
        match outcome.winner() {
            // The mover just won: the opponent is mated.
            Some(_) => Score::Mate(-1),
            None => Score::Centipawns(0),
        }
    } else {
        evaluate(judge, &game)
    };
    let drop = winning_chances(before) + winning_chances(after);
    tally.add(classify(drop));
    before = after;
}
```

The judge is weaker than Lichess's analysis: it misses a blunder now and then, and now and then calls a sound move one (full strength, which makes none, measured 0.3 in 40 moves). It errs alike for every level, which is what matters for fitting them to each other; beside people's counts, from Lichess's own analysis, the levels' are close rather than exact.

The levels on that scale. Level 0 is now a Lichess rapid rating of 600 and each level 100 more, to 2500 at level 19, and each level's chance of a person's horizon is fitted so that it blunders as often as people of its rating:

```
/// Each level's rating, 0 to [`FULL_STRENGTH`], on [`RATING_SCALE`]: the Lichess
rapid rating
/// of the people who blunder as often as the level does ([`PEOPLE_BLUNDERS`]), 600 at
level 0
/// and 100 more for each level, to 2500 at level 19. Full strength makes no blunders
to compare,
/// and plays far above any person: its 3000 only stands above the scale, and is never
asked for
/// ([`ELO_RANGE`] stops at level 19's).
pub const RATINGS: [u32; FULL_STRENGTH as usize + 1] = [
    600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900,
2000, 2100,
    2200, 2300, 2400, 2500, 3000,
];
```

The fit took three rounds of 40 games a level with the `mistakes` tool. A level blunders about as often as its usual search does, plus its chance of a person's horizon times what such a move adds: about 5 blunders in 40 moves with one ply, 3 to 4 with two. The first round put that at 4.1 for two plies, measured with every move a horizon move; played among a level's other moves, a two-ply horizon move meets fewer tactics, and the upper levels fell short. Each round set each level's chance from how far the last missed, and the last is smoothed into one straight line through levels 0 to 7 and another through 8 to 19, since 40 games of one level carry about $\pm$ 0.5 blunders of noise.

Level 6, worked through. With its usual search, level 6 blundered 0.7 times in 40 moves. A one-ply horizon move, played among its other moves, blunders about 5 times in 40. People rated 1200 blunder 3.0 times. So level 6 needs the horizon on about $(3.0 - 0.7) / (5 - 0.7) \approx 0.53$ of its moves. The rounds measured 3.3 blunders at a chance of 0.48 and 2.7 at 0.50, both within the noise of 3.0, and the straight line through levels 0 to 7 gives it 0.50. From level 8 the same sum uses a two-ply horizon move's blunders, about 3 in 40 instead of 5, so the

chance jumps up there: a horizon that sees the reply errs less often, and must be used more often to blunder as much.

The check, with openings none of the rounds used:

Level	Rating	People's blunders	The level's
0	600	4.5	3.8
2	800	4.1	3.9
4	1000	3.4	2.5
6	1200	3.0	3.7
8	1400	2.7	2.4
10	1600	2.4	2.8
12	1800	2.1	2.3
14	2000	1.8	1.5
16	2200	1.5	1.7
18	2400	1.1	1.3
19	2500	0.9	1.0

Over all twenty levels the level blunders 2.5 times in 40 moves on average, people of the same ratings 2.6. Full strength measured 0.3, which is about the judge's own floor: a search of 100,000 nodes now and then disagrees with a stronger one.

`UCI_Elo` maps onto the table. Between two levels the strength is in between too: the node count, margin and temperature are smooth in the level, so `settings_at` takes a fractional one, and an Elo of 1350, between level 7 (1300) and level 8 (1400), plays at level 7.5. Its horizon is level 7's: a horizon's depth can't be a fraction, and the chance jumps where the depth changes.

```
/// The strength for a `UCI_Elo` setting (clamped to [`ELO_RANGE`]): the level with
that measured
/// rating in [`RATINGS`], interpolated linearly between the two levels it falls
between.
pub fn level_for_elo(elo: u32) -> f64 {
    let (low, high) = ELO_RANGE;
    let elo = elo.clamp(low, high);
    // The first level rated at least `elo`; there is one, since full strength is
above the range.
    let above = RATINGS
        .iter()
        .position(|&r| r >= elo)
        .unwrap_or(RATINGS.len() - 1);
    if above == 0 {
        return 0.0;
    }
    let (below_rating, above_rating) = (RATINGS[above - 1], RATINGS[above]);
    (above - 1) as f64 + f64::from(elo - below_rating) / f64::from(above_rating -
below_rating)
}
```

The range runs from level 0's rating to level 19's: **600 to 2500**. CNP's `engine/levels` names the scale "Lichess rapid, estimated from how often people blunder", so a client can say what the numbers mean.

It is an estimate from blunders alone. A level's other moves are an engine's, its inaccuracies and mistakes fewer than people's at the low end, and the counts come from Lichess's own analysis for people but from a search of 100,000 nodes for the levels. Only games against people would rate the levels for sure: as a Lichess bot, say (`docs/lichess.md`).

Calibration against engines (until cjus/chessgui#113)

Before the human scale, the levels were **calibrated** against engines, as the PRD then required: each played games against reference engines with known ratings on the CCRL list, and `UCI_Elo` was fitted to the results. Those ratings were for the levels without a person's horizon; they no longer hold, and are kept here as the record.

`scripts/mac-levels.sh` does that on the test Mac. Each level plays 200 games against each of the four reference engines nearest its expected strength, chosen from twelve open-source engines rated 1058 to 3049 on CCRL Blitz (the PRD lists them). The gauntlet tool fits the level's rating on their scale, holding theirs fixed.

It ran twice. The first run, on 2026-10-18 (cjus/chessengine#103), rated every level from 0 to 19 at 8+0.08: 16,000 games in about six and a half hours. A check of levels 4, 10 and 16 at 40+0.4 then rated them 111 to 272 lower (cjus/chessengine#106; "Pitfalls" says why). So the second run rated every other level at 40+0.4: levels 0, 2, 6, 8, 12, 14, 18 and 19, with 4, 10 and 16 from the check, 8,800 games in about fourteen and a half hours. Each odd level below 19 is put halfway between its neighbours. In the first run's table, that lands within 25 of the odd levels' measured ratings, less than each level's own margin.

Level	Nodes per move	Rating at 40+0.4	At 8+0.08
0	400	629 ± 55	752 ± 41
1	600	*703*	823 ± 35
2	900	777 ± 36	851 ± 34
3	1,350	*833*	927 ± 30
4	2,025	888 ± 32	999 ± 30
5	3,037	*957*	1052 ± 27
6	4,556	1025 ± 27	1131 ± 25
7	6,834	*1112*	1242 ± 24
8	10,251	1198 ± 24	1307 ± 24
9	15,377	*1295*	1414 ± 24
10	23,066	1391 ± 25	1534 ± 25
11	34,599	*1501*	1642 ± 26
12	51,898	1610 ± 26	1783 ± 28
13	77,847	*1731*	1924 ± 30
14	116,771	1852 ± 26	2106 ± 27
15	175,157	*1974*	2240 ± 26
16	262,736	2096 ± 27	2368 ± 24
17	394,104	*2217*	2475 ± 24
18	591,156	2337 ± 25	2533 ± 25
19	886,735	2483 ± 23	2599 ± 22

The ratings in italics are halfway between their neighbours, not measured.

Full strength rated 2782 ± 11 in the gauntlet of 2026-10-21, at 8+0.08 (2714 ± 10 on 2026-10-18, before correction history and double extensions). With the network as the default (chapter 21) it rated 3236 ± 16 on 2026-10-26, and 3286 ± 17 with the second network on 2026-10-27. The levels keep the hand-crafted evaluation, so their ratings stand. It uses the clock, so unlike the levels it gains from more time much as its opponents do. Level 19 played about 300 below it. The 40+0.4 column was the code's table, and `UCI_Elo` ran from level 0's rating to level 19's, rounded inward to 50: 650 to 2450 (1000 to 2400 before any calibration, 800 to 2600 after the first).

Before any calibration, the levels followed a guessed straight line from 1000 at level 0 to 2400 at level 19. That was about 370 too high at the bottom and close at the top. A level gained 55 to 75 points up to level 6, where the randomness dominates, then more with each level: about 120 from level 12 up, and 146 from 18 to 19.

With a person's horizon the levels lose far more often to engines, so their ratings against engines would now be much lower, the lower levels below every reference engine (the weakest is rated 1058). Rating them on CCRL's scale again would need weaker engines.

Levels and the network

From stage 21 the engine evaluates with an NNUE network (chapter 21), but the levels keep the hand-crafted evaluation they were calibrated with: below full strength the search doesn't use the network. With it, the same node limits would play stronger and every rating above would be wrong. Calibrating the levels again with the network is for later.

Pitfalls

- **A rating against engines isn't a rating against people.** Engines punish every blunder; people miss many. So a level's rating against engines and the rating of the people it plays like can be hundreds of points apart, and a level can't be fitted to both (the human scale, above).
- **A horizon search must not teach the table.** Its scores are unresolved, evaluated in the middle of exchanges; written to the transposition table they would mislead the next full search. `Techniques::horizon` leaves the table alone, and runs on one thread.
- **Blunders come in bunches.** One bad game can hold several, so 40 games of a level carry about ± 0.5 blunders in 40 moves of noise. The chances are smoothed across levels rather than set level by level from one run.
- **Node limits checked too rarely.** The search used to check its limits every 1024 nodes. At 400 nodes a move that's far too coarse, so the node count is now checked at every node (it costs nothing, unlike reading the clock).
- **Randomness at full strength.** Level 20 must always play the best move, or test results would be noise. With `temperature` 0, the choice is the first line.
- **Repeating the same game.** The dice come from the clock and the position, so a weak level varies its play from game to game.
- **A level doesn't get stronger with more time.** Its node limit is the same at any time control, while its opponents use the time they're given. The first calibration ran at 8+0.08, and the check at 40+0.4

rated levels 4, 10 and 16 lower by 111, 143 and 272. So the table now comes from 40+0.4. That is still faster than CCRL's own 2'+1", where the levels would rate a little lower again. The top levels lost less than the middle ones (116 at level 19, 196 at 18), probably because the search also stops when the clock runs out. At 8+0.08 their node counts didn't always fit in the time, so they played weaker than their limits there; at 40+0.4 they search them in full.

- **Level 0 is rated from above.** It is weaker than every reference engine: at 40+0.4 it scored 6% against the weakest (1058) and almost nothing against the rest, so its rating has the widest margin (± 55). No engine below 1000 passed the checks for the reference list.
- **The reference engines have bugs too.** Walleye 1.3.0 ignores an en passant capture in the moves it is sent, so its board goes wrong and it later plays an illegal move: about 1% of its games, which it loses. CCRL's games saw the same bug, so its rating includes it. Arche 0.3.8 twice answered `bestmove 0000` (no move) in a position with legal moves, at 40+0.4 only, and lost those games too.

What it's worth

At full strength nothing changes (the bench is unchanged). The levels blunder as people of Lichess rapid ratings 600 to 2500 do, within the noise of the measurement, where before even level 0 blundered less than people rated 600 and level 18 not at all. `UCI_Elo` means a rating on that human scale, estimated from blunders; against engines the levels' old CCRL ratings no longer hold.

Further reading

- Chess Programming Wiki: Strength, Multi-PV.
- Lichess, accuracy and win percentage, and its open database of games.

XBoard, pondering and playing on Lichess

Stage 20 · tag `stage-20-xboard` · code in `crates/protocol/src/xboard.rs`, `uci.rs` and `crates/cli/tests/conformance.rs`

This is the last stage of the hand-crafted engine. Nothing here makes cb2026 search better; it makes the engine work in more places. It adds a second GUI protocol, thinking on the opponent's time, a test suite that plays the part of the GUIs, and a setup for playing people online.

Two protocols, two ideas of who keeps the game

Chapter 4 introduced UCI, where the GUI owns the game. Before every move it sends the whole position (`position startpos moves e2e4 e7e5 ...`) and says `go`. The engine remembers nothing it needs between moves; it's a function from position to move.

XBoard, also called **CECP** (the Chess Engine Communication Protocol), is older (it grew out of the GNU Chess interface in the early 1990s) and works the other way round. **The engine keeps the game.** The GUI sends one move at a time, and the engine decides by itself when it's its turn:

```
> new                 start a game; the engine plays Black
> level 40 5 0        40 moves in 5 minutes
> post                show thinking
> time 30000          the engine's clock: 300.00 seconds (centiseconds)
> otim 30000          the opponent's clock
> usermove e2e4       the opponent's move: now it's the engine's turn
< 1 -15 0 21 e7e5     thinking: depth, score, time, nodes, PV
< ...
< move e7e5
```

So an XBoard engine is a small state machine. It holds the position and the moves that led to it (for `undo` and `remove`, which take back one or two plies). It also tracks which side it plays: `new` makes it Black, `go` makes it play the side to move and start thinking, and `force` makes it play neither, so the GUI can enter moves from a game. Finally it holds the clock settings: `level`, `st` (seconds per move), `sd` (maximum depth), and the clock readings `time` / `otim` sent before each move.

cb2026 keeps all of that in one `Session`, and turns it into the same `Limits` the UCI side builds, so both protocols drive exactly the same search through the Engine API.

Thinking output

With `post`, every completed iteration prints a line in the format XBoard expects:

```
/// A thinking line: `depth score time nodes pv`, score in centipawns (mates as 100000
+ moves),
/// time in centiseconds, the PV in coordinate notation.
fn thinking_line(info: &SearchInfo, chess960: bool) -> String {
    let score = match info.score {
        Score::Centipawns(cp) => cp,
        Score::Mate(moves) if moves > 0 => 100_000 + moves,
        Score::Mate(moves) => -100_000 + moves,
    };
    let pv: Vec<String> = info.pv.iter().map(|&mv| move_text(mv, chess960)).collect();
    format!(
        "{} {score} {} {} {}",
        info.depth,
        info.time.as_millis() / 10,
        info.nodes,
        pv.join(" ")
    )
}
```

Scores are in centipawns from the engine's side. Mates use a convention: 100000 plus the number of moves to mate, negative when being mated, so a GUI can show "#3".

With MultiPV, each depth has several lines, and the order matters. XBoard sorts a depth's lines by score to show them. But it keeps the last line it reads as the engine's score and PV, which go into the saved game, the evaluation graph and adjudication. So cb2026 holds a depth's lines until the last one arrives, then sends them best last: that's the `expected` count and `flush` in the search thread below. Before 1.0.0 it sent only the best line, which the GUI matrix found (#170).

Feature negotiation

XBoard learns what an engine supports from `feature` lines sent in answer to `protover 2`. Some features change the protocol itself:

- `usermove=1`: moves arrive as `usermove e2e4`, not as bare `e2e4`, so they can't be confused with commands.

- `setboard=1`: positions arrive as FEN, not through the old `edit` mode.
- `san=0`: moves are in coordinate notation.
- `ping=1`: `ping N` is answered with `pong N`, so the GUI can tell when the engine has caught up.

Others announce abilities: `analyze`, `memory` (hash size), `smp` (threads), `egt="syzygy"` (tablebases) and `variants="normal,fischerandom"` (Chess960). In Chess960, XBoard writes castling as `O-O` and `O-O-O`, because "king to g1" is ambiguous when the king may start on g1.

Pitfalls

- **The race after `move`.** As soon as the engine prints `move e7e5`, the GUI may answer with the next move. If the engine's own board hasn't played e7e5 yet, the reply looks illegal. The search thread records the move where the session can see it *before* printing it, and the session plays any recorded moves before it handles the next command.
- **Commands while thinking.** `force`, `new`, `undo` or `result` can arrive mid-search. The search is stopped without moving: the GUI no longer wants a move. `?` is different. It means "move now", so the search stops and plays its best move so far.
- **Clocks in centiseconds**, `level` times in minutes or `minutes:seconds`, and `st` in seconds. Three units for one idea.

Pondering

A chess clock only runs on your own turn, but nothing stops a program from thinking during the opponent's. **Pondering** guesses the opponent's reply (the second move of the principal variation) and searches the position after it while the opponent thinks. If the guess is right (a **ponder hit**), that search has a head start, often of several plies. If it's wrong, it's thrown away, and nothing is lost but electricity.

Strong engines guess right about half the time, so pondering is worth real strength in games against opponents who think. It's no use in self-play tests (both engines would share the cores), which is why the `match` tool leaves it off.

In UCI

The GUI manages pondering. After `bestmove e7e5 ponder g1f3` it sends `position ... e7e5 g1f3` and `go ponder wtime ...`. The engine searches as if with no limit until either `ponderhit` (the opponent played g1f3: switch to the real clock) or `stop` (they didn't: report a move, which the GUI ignores, and wait for a new `position`):

```
// Pondering: think on the opponent's time, with no limit, until the GUI says
// `ponderhit` (the opponent played the expected move: switch to the real clock) or
// `stop` (it didn't). The move must not be reported before either.
let pondering = args.contains(&"ponder");
let ponder_flag = Arc::new(AtomicBool::new(pondering));
let ponder = pondering.then(|| {
    (
        limits.clone(),
        engine.time.clone(),
        Arc::clone(&ponder_flag),
    )
});
let limits = if pondering {
    Limits {
        infinite: true,
        ..limits
    }
} else {
    limits
};
let infinite = limits.infinite && !pondering;
```

On `ponderhit` a timer starts with the time the search would have been given for this move. The pondering time is a bonus on top, not counted against it.

In XBoard

XBoard leaves pondering to the engine: `hard` turns it on, `easy` off. After printing its move, cb2026's search thread carries on by itself. It plays the expected reply on its own board and searches with no limit:

```
loop {
    expected.set(multipv.min(position.legal_moves().len()).max(1));
    let stop = Arc::clone(&lock(&thread_shared).stop);
    let result = engine.search(&position, &keys, &limits, &stop, &mut on_info);
    flush();
    // Analysis and pondering wait to be stopped, or for the ponder hit.
    while (analysis || lock(&thread_shared).pondering.is_some())
        && !stop.load(Ordering::Relaxed)
    {
        thread::sleep(Duration::from_millis(1));
    }
    let mut shared = lock(&thread_shared);
    // Stopped by the session, during analysis, or a ponder miss: no move.
    if analysis || shared.pondering.is_some() || thread_quit.load(Ordering::SeqCst) {
        break;
    }
    let Some(mv) = result.best_move else {
        break;
    };
    // Record the move before announcing it (see `Shared::played`).
    shared.played.push(mv);
    send(&output, &format!("move {}", move_text(mv, chess960)));
    // Ponder on the expected reply, if there is one and the game goes on.
    let Some(reply) = result.ponder_move.filter(|_| ponder) else {
        break;
    };
    keys.push(position.hash());
    position.make_move(mv);
    keys.push(position.hash());
    position.make_move(reply);
    if position.legal_moves().is_empty() {
        break;
    }
    shared.pondering = Some(reply);
    shared.stop = Arc::new(AtomicBool::new(false));
    limits = Limits {
        infinite: true,
        ..Limits::default()
    };
}
```

When the opponent's `usermove` arrives, the session compares it with the expected move. A hit clears the ponder flag and starts the timer, as in UCI, and the same search goes on to produce the next move. A miss stops the thread without a move and starts a fresh search. Each search gets its own stop flag, so a timer left over from one search can never stop the next.

Conformance tests: standing in for the GUIs

Every GUI drives engines a little differently, and the differences are where engines break. GUIs send Windows line endings, change options mid-search, skip `ucinewgame`, give a few milliseconds on the clock, or send `stop` a microsecond after `go`. Claude, who writes most of this code, can't operate a GUI. So `crates/cli/tests/conformance.rs` starts the real `cb2026` binary, plays scripted sessions with exactly those habits, and checks every answer. It runs with the other tests on every change. The manual checklist in `docs/gui-compat.md` covers what a script can't: how the engine looks and behaves inside each GUI.

Playing on Lichess

Lichess lets programs play as BOT accounts through its Bot API. lichess-bot is the usual bridge: it accepts challenges and talks UCI to the engine. `docs/lichess.md` has the setup: the BOT account, the config, and the settings that matter. Two of those settings are worth explaining here:

- **Move overhead.** The clock runs on Lichess's server. A move that the engine makes with 50 ms to spare can arrive after its time has run out, so the engine keeps a margin per move (the `Move Overhead` option, 300 ms suggested online).
- **One game at a time,** with all threads. Two games at once would split the cores and make both weaker.

What it's worth

At full strength the search is unchanged (the bench is unchanged). Pondering's value depends on the opponent and the time control; it isn't measured in the self-play tests, which run without it. The GUI compatibility matrix is checked on the test Mac before the 1.0 release.

Further reading

- Tim Mann and H.G. Muller, Chess Engine Communication Protocol, the XBoard specification.
- Chess Programming Wiki: Chess Engine Communication Protocol, Pondering.
- lichess-bot and the Lichess Bot API.

NNUE

Stage 21 · in progress · code in `crates/nnue`, `crates/tools/src/selfplay.rs`, `crates/trainer`, `crates/engine/src/search.rs`

> This chapter is being written as stage 21 is built. So far it covers the network's definition (its inputs, its layers and its whole-number weights), its training data, the trainer and the network in the search. What it's worth comes once a network has been trained on the full data and measured.

Chapters 15 and 16 built an evaluation from terms a person designed and tuned their weights on cb2026's own games. An **NNUE** ("efficiently updatable neural network") replaces those terms with a small neural network that learns what matters from millions of positions. Since 2020 it has been the evaluation of almost every top engine. This chapter builds one from scratch, with a trainer of cb2026's own, so every step can be followed.

The idea

A neural network is layers of numbers. Each value in a layer is a weighted sum of the values in the layer before, passed through a simple function (the **activation**). Training adjusts the weights so the last layer predicts something: here, how good the position is.

Two things make this work for chess at engine speed:

1. **Inputs that barely change.** The network's inputs say which piece stands on which square. A move changes only a few of them, so the first layer, which does almost all of the work, can be updated by adding and subtracting a few rows of weights instead of being recomputed. That running first layer is the **accumulator**.
2. **Whole numbers.** The weights are rounded to integers, so the arithmetic is exact and the CPU can do many small integer operations at once.

The inputs

An input is one fact, such as "a white knight stands on f3". There are 2 colours × 6 piece types × 64 squares = 768, one on for each piece. The network reads them from both sides' points of view: from a side's view its own pieces are "ours", and Black sees the board flipped top to bottom. So the same weights describe White's knight on f3 for White and Black's knight on f6 for Black.

```
/// The input for `piece` standing on `square`, seen from `view`'s side of the board.
pub fn index(view: Color, piece: Piece, square: Square) -> usize {
    let side = if piece.color == view { 0 } else { 1 };
    let square = match view {
        Color::White => square,
        Color::Black => square.flip_rank(),
    };
    side * PER_SIDE + piece.piece_type.index() * SQUARES + square.index()
}
```

None of these inputs depends on where a king stands, which keeps the network small and its updates cheap: no move ever forces a full recomputation. Stronger networks add the king's square ("king buckets"); they need much more training data, and cb2026 will try them later, measured like everything else.

A move switches off the inputs of the pieces it lifts and switches on those it puts down. A capture also lifts the captured piece, en passant lifts a pawn beside the destination, and castling lifts and puts down both the king and the rook. That is at most two off and two on:

```
/// The pieces a move lifts and puts down, which are the inputs it switches off and
on. They are
/// pieces and squares, not input numbers, so they serve both views.
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub struct Changes {
    removed: [Option<(Piece, Square)>; 2],
    added: [Option<(Piece, Square)>; 2],
}

impl Changes {
    /// What playing `mv` changes in `pos`, the position before the move. `mv` must be
legal in
    /// `pos`.
    pub fn of(pos: &Position, mv: Move) -> Changes {
        let us = pos.side_to_move();
        let (from, to) = (mv.from(), mv.to());
        let piece = pos.piece_at(from).expect("a piece on the from-square");

        if mv.kind() == MoveKind::Castle {
            // "King takes own rook": `to` is the rook's square, as in
`Position::make_move`.
            let side = if to.file() > from.file() {
                CastlingSide::KingSide
            } else {
                CastlingSide::QueenSide
            };
            let (king_to, rook_to) = castling_destinations(us, side);
            let rook = pos.piece_at(to).expect("the castling rook");
            return Changes {
                removed: [Some((piece, from)), Some((rook, to))],
                added: [Some((piece, king_to)), Some((rook, rook_to))],
            };
        }

        let captured = mv.is_capture().then(|| {
            // En passant captures the pawn beside the destination, not on it.
            let square = match mv.kind() {
                MoveKind::EnPassant => Square::new(to.file(), from.rank()),
                _ => to,
            };
            (pos.piece_at(square).expect("a piece to capture"), square)
        });
        let landing = match mv.promotion() {
            Some(piece_type) => Piece::new(us, piece_type),
            None => piece,
        };
        Changes {
            removed: [Some((piece, from)), captured],
            added: [Some((landing, to)), None],
        }
    }

    /// The pieces lifted: their inputs go off.
    pub fn removed(&self) -> impl Iterator<Item = (Piece, Square)> + '_ {
        self.removed.iter().flatten().copied()
    }
```

```
    /// The pieces put down: their inputs come on.
    pub fn added(&self) -> impl Iterator<Item = (Piece, Square)> + '_ {
        self.added.iter().flatten().copied()
    }
}
```

The layers

```
  768 inputs ──► 256 hidden ─┐                (from the side to move's view)
                             ├──► 1 output ──► centipawns
  768 inputs ──► 256 hidden ─┘                (from the other side's view)
```

Each view has a hidden layer of 256 values. Both views use the same first-layer weights, which the flipped board makes possible. The output layer takes the side to move's 256 values first, then the other side's. A position and its mirror image (colours swapped, board flipped) give exactly the same inputs in the same order, so they get exactly the same score, which a test checks.

The activation clips each hidden value to the range 0 to 1 and squares it (**SCReLU**, the squared clipped ReLU). Squaring lets the network weigh a large hidden value more than a middling one. Plain clipping (**CReLU**) is also supported, so the two can be compared.

```
/// The network's output for `pos`, from the side to move's view, before the eval
scale.
/// The trainer compares `σ(output)` with its targets.
pub fn output(&self, pos: &Position) -> f32 {
    let us = pos.side_to_move();
    let (our_weights, their_weights) = self.output_weights.split_at(self.hidden);
    let half = |view: Color, weights: &[f32]| -> f32 {
        self.hidden_layer(pos, view)
            .iter()
            .zip(weights)
            .map(|(&value, weight)| self.activate(value) * weight)
            .sum()
    };
    half(us, our_weights) + half(!us, their_weights) + self.output_bias
}
```

That is the network as the trainer sees it, in floating point. A hidden layer of 256 means about 200,000 weights, a good size for the 100 million positions a first network trains on. The size is stored in the network's file, so networks of 128 or 512 can be tried too.

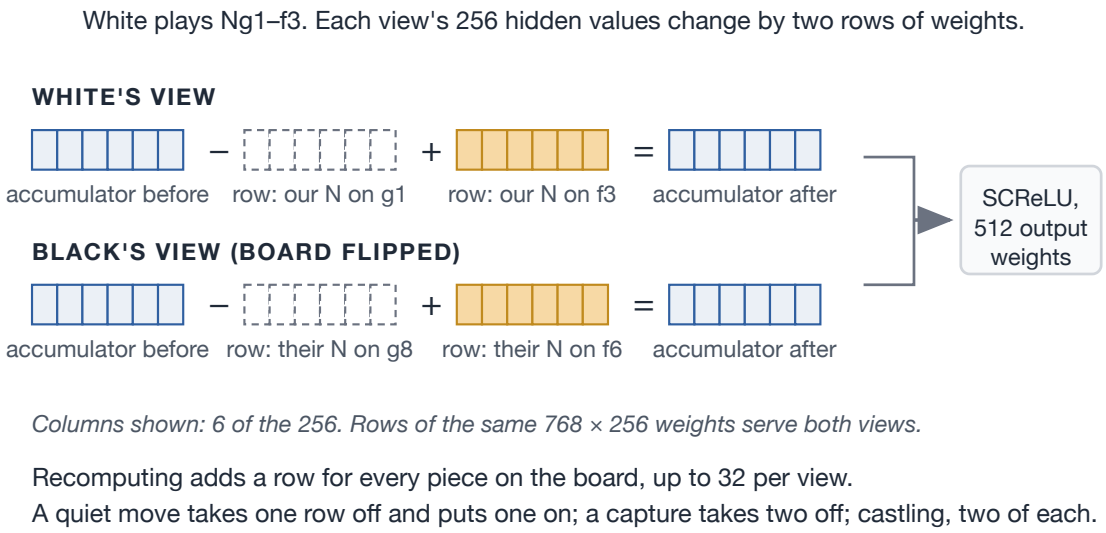

Figure 21.1 A move changes the accumulator by a few rows of weights instead of recomputing it

Whole numbers instead of floats

The engine runs the same network with whole-number weights: the first layer's weights times 255, the output weights times 64.

```
/// The scale of first-layer weights and biases: a hidden value of 1.0 is 255.
pub const QA: i32 = 255;
/// The scale of output weights.
pub const QB: i32 = 64;
/// The largest first-layer weight or bias: [`WEIGHT_LIMIT`]
(crate::float::WEIGHT_LIMIT) × [`QA`], rounded.
pub const MAX_INPUT_WEIGHT: i16 = 505;
/// The largest output weight: [`WEIGHT_LIMIT`](crate::float::WEIGHT_LIMIT) × [`QB`],
rounded.
pub const MAX_OUTPUT_WEIGHT: i16 = 127;
```

A hidden value of 1.0 becomes 255, so the clipping is a clamp to 0..255. The accumulator is the first layer kept up to date move by

move:

```
/// The accumulator for `pos`, computed from scratch: the biases plus the row of every
input
/// that is on.
pub fn accumulator(&self, pos: &Position) -> Accumulator {
    let views = Color::ALL.map(|view| {
        let mut values = self.input_biases.clone();
        for input in features::active(pos, view) {
            add(&mut values, self.row(input));
        }
        values
    });
    Accumulator { views }
}

/// Brings `acc` up to date after a move with these `changes`: subtract the rows of
the
/// inputs switched off, add the rows of those switched on. Same result as
/// [`Network::accumulator`] on the new position, for a fraction of the work.
pub fn update(&self, acc: &mut Accumulator, changes: &Changes) {
    for view in Color::ALL {
        let values = &mut acc.views[view.index()];
        for (piece, square) in changes.removed() {
            subtract(values, self.row(features::index(view, piece, square)));
        }
        for (piece, square) in changes.added() {
            add(values, self.row(features::index(view, piece, square)));
        }
    }
}
```

Integers can overflow, so the sizes are chosen to make that impossible. The trainer keeps every weight within ±1.98, so a first-layer weight is at most 505 and an output weight at most 127. A legal position has at most 32 pieces, so a hidden value is at most 33 × 505 = 16,665 (the 32 pieces plus the bias), inside an `i16`'s 32,767. An activation times an output weight is at most 255 × 127 = 32,385, which fits too.

```
/// A hidden value clipped to 0..1, at the scale QA: 0..255.
fn clipped(value: i16) -> i32 {
    i32::from(value).clamp(0, QA)
}

/// The output layer's sum with CReLU: each clipped hidden value times its weight.
fn crelu_dot(values: &[i16], weights: &[i16]) -> i64 {
    values
        .iter()
        .zip(weights)
        .map(|(&v, &w)| i64::from(clipped(v) * i32::from(w)))
        .sum()
}

/// The output layer's sum with SCReLU: each clipped hidden value squared, times its
weight.
/// Computed as `(a × w) × a`: `a × w` is at most 255 × 127 = 32,385 and fits in an
`i16`.
fn screlu_dot(values: &[i16], weights: &[i16]) -> i64 {
    values
        .iter()
        .zip(weights)
        .map(|(&v, &w)| {
            let a = clipped(v);
            i64::from(a * i32::from(w)) * i64::from(a)
        })
        .sum()
}
```

Rounding costs a little accuracy. With the random weights the tests use, the integer network is on average 2–4 centipawns from the floating-point one. Given the rounded weights, it computes the same thing to within a centipawn.

The output's scale

The output, times an **eval scale**, is in centipawns. The scale isn't arbitrary: it has to match the hand-crafted evaluation's centipawns, because the search's pruning margins were all tuned in those centipawns.

The trainer will fit the network to two things at once, the search score of each position and the result of its game, turning scores into expected results with a sigmoid, `σ(cp / scale)`. If the scale doesn't

match how cb2026's centipawns actually relate to results, the two targets pull against each other and the network's centipawns stretch. Chapter 16's Texel tuner fits exactly this relation as its `K`: on 40,304 self-play positions it came out at 1.19, which is a scale of 400 / (1.19 × ln 10) ≈ 146. Many engines use 400; cb2026's centipawns are steeper.

```
/// The position's value in centipawns, from `side_to_move`'s point of view, given its
/// accumulator. Nothing here keeps it below the search's mate scores: the engine
clamps it.
pub fn evaluate(&self, acc: &Accumulator, side_to_move: Color) -> i32 {
    let ours = acc.view(side_to_move);
    let theirs = acc.view(!side_to_move);
    let (our_weights, their_weights) = self.output_weights.split_at(self.hidden);
    // The sum is at the scale QA × QB. SCReLU's squares are at QA × QA × QB: divide
one QA out.
    let sum = match self.activation {
        Activation::ClippedRelu => {
            crelu_dot(ours, our_weights) + crelu_dot(theirs, their_weights)
        }
        Activation::SquaredClippedRelu => {
            (screlu_dot(ours, our_weights) + screlu_dot(theirs, their_weights)) /
i64::from(QA)
        }
    };
    let output = sum + i64::from(self.output_bias);
    // The sum is at most 2 × 4096 × 255 × 127 ≈ 2.7 × 10^8 and the bias an i32 (≈ 2.1
× 10^9);
    // times an eval scale of at most 10^4 that fits easily in an i64, and divided by
    // QA × QB it is at most about 1.5 × 10^9, inside an i32.
    (output * i64::from(self.eval_scale) / i64::from(QA * QB)) as i32
}
```

The network file

A network is stored as a 48-byte header followed by the numbers. The header says everything about the network's shape (inputs, hidden size, activation, scales) and carries a checksum, so a damaged or foreign file is refused before it is used. Each input's weights are one row, so switching an input on or off adds one row to the accumulator. The default network is part of the source, in `nets/`, and each release carries it as a download too; other networks aren't kept in the repository.

The training data

The network learns from positions of cb2026's own games, the same self-play games as chapter 16's tuner, played at a fixed number of nodes per move. The tuner only needed each position and its game's result. The trainer also wants the search's score, and since producing data takes the test Mac about 12 hours per 100 million positions, the format should keep anything a later network might want. Each position becomes a **record** of 32 bytes:

Bytes	Field
0–7	which squares are occupied (a bitboard)
8–23	a 4-bit code for each occupied square's piece, in square order
24	the side to move and the en passant square
25	the halfmove clock
26–27	the search's score, in centipawns from White's side
28	the game's result
29–30	the best move
31	the format's version

A position has at most 32 pieces, so 32 four-bit codes, 16 bytes, hold them all. The codes are the piece types plus 8 for Black, and one more: code 6 is a rook that can still castle. That is how a record keeps castling rights, and it works for Chess960, where a right belongs to one particular rook. The layout follows marlinformat, the format of the trainer of the engine Marlin.

```
/// The record of `pos`, where the search chose `best_move` with a score of `score`
/// centipawns from White's side, in a game that ended in `result`. A score beyond an
`i16`
/// is clamped.
pub fn new(
    pos: &Position,
    score: i32,
    result: GameResult,
    best_move: Move,
) -> Result<Record, RecordError> {
    let occupied = pos.occupied();
    if occupied.count() > MAX_PIECES {
        return Err(RecordError::TooManyPieces(occupied.count()));
    }
    let castling_rooks = castling_rooks(pos);
    let mut codes = 0u128;
    for (i, square) in occupied.squares().enumerate() {
        let piece = pos
            .piece_at(square)
            .expect("an occupied square has a piece");
        let kind = if castling_rooks.contains(square) {
            CASTLING_ROOK
        } else {
            piece.piece_type.index() as u8
        };
        let code = kind
            | if piece.color == Color::Black {
                BLACK
            } else {
                0
            };
        codes |= u128::from(code) << (4 * i);
    }
    let side = if pos.side_to_move() == Color::Black {
        0x80
    } else {
        0
    };
    let en_passant = pos.en_passant().map_or(NO_EN_PASSANT, |s| s.index() as u8);
    Ok(Record {
        occupied: occupied.0,
        codes,
        turn: side | en_passant,
        halfmove_clock: pos.halfmove_clock().min(255) as u8,
        score: score.clamp(i32::from(i16::MIN), i32::from(i16::MAX)) as i16,
        result,
        best_move: best_move.to_bits(),
    })
}
```

The record keeps the position itself, not the network's inputs. The trainer turns the pieces into inputs as it reads them, so a later network

with different inputs, such as king buckets, can learn from the same records without generating them again:

```
/// Every piece and its square, in square order from a1. This is all the trainer needs
to
/// compute the network's inputs.
pub fn pieces(&self) -> impl Iterator<Item = (Piece, Square)> + '_ {
    Bitboard(self.occupied)
        .squares()
        .enumerate()
        .map(|(i, square)| (piece_of(self.code(i)), square))
}
```

Balanced starts

Each game starts from one of 50 balanced openings plus 8 random moves, so that no two games are alike. Random moves are often bad ones, though, and many games began with one side already far ahead. In 600 games, only 18.5% of the recorded positions were within a pawn of level and 37.5% were more than five pawns from it; White won 54% of the games.

So when writing records, `datagen` searches each start like a move of the game and draws the random moves again while the start scores beyond 200 centipawns:

```
/// The start for game `index`, roughly balanced if `settings.max_start_score` is set: each try
/// is searched like a move of the game, and the first that scores within the limit is used. If
/// none of the `START_ATTEMPTS` (20) tries does, the most balanced is.
pub fn balanced_start(engine: &mut Engine, settings: &DatagenSettings, index: usize) -> Position {
    let Some(limit) = settings.max_start_score else {
        return starting_position(settings, index);
    };
    let mut best: Option<(i32, Position)> = None;
    for attempt in 0..START_ATTEMPTS {
        let start = random_start(settings, index, attempt);
        let imbalance = start_score(engine, &start, settings.nodes);
        if imbalance <= limit {
            return start;
        }
        if best.as_ref().is_none_or(|(b, _)| imbalance < *b) {
            best = Some((imbalance, start));
        }
    }
    best.expect("at least one try").1
}
```

Largest start score	Within ±100 cp	Beyond ±500 cp	White / draw / Black
no limit	18.5%	37.5%	54 / 17 / 29%
300 cp	24.4%	27.9%	45 / 21 / 34%
200 cp	30.0%	25.3%	41 / 23 / 37%

A quarter of the positions are still far from level, because games at a few thousand nodes a move do get decided, and a network has to judge won positions too. What the limit removes is games that were lopsided from their first move. It costs nothing: balanced games last longer and give more positions each, which pays for the extra search. The tuner's text output keeps every start, so chapter 16's runs can still be repeated.

`datagen --format records` writes records, and `records` checks a file and shows what is in it. Sixty games make about 4,100 records,

132 KB; 100 million records take 3.2 GB. Records have no file header, so files from separate runs can simply be joined.

That makes a 12-hour run safe to interrupt. `scripts/mac-nnue-data.sh` makes the data in chunks of 56,000 games, each its own file, and the trainer reads them all (`--data` once per file). A run that stops loses only the chunk it was on: started again, it keeps the finished files and makes the rest. Each game's random opening moves come from its number, and chunk k's numbers start where chunk k − 1's ended, so the chunks never repeat a game, and a chunk made twice holds the same games.

The trainer

The trainer (`crates/trainer`) learns the weights. It uses no machine-learning library: every gradient is worked out by hand, so each step can be read.

What it learns to predict

The network's output `y` means `y × eval scale` centipawns, and `σ(y)` the expected result for the side to move. Each record gives two estimates of what `σ(y)` should be: the search's score, `σ(score / eval scale)`, which is precise but sometimes wrong, and the game's result, which is right about the game but noisy about any one position in it. The **target** blends them, a quarter result by default:

```
/// The example for `record`.
pub fn from_record(record: &Record, targets: Targets) -> Example {
    let us = record.side_to_move();
    let mut inputs = [[0u16; MAX_ACTIVE]; 2];
    let mut count = 0;
    for (piece, square) in record.pieces() {
        inputs[0][count] = features::index(us, piece, square) as u16;
        inputs[1][count] = features::index(!us, piece, square) as u16;
        count += 1;
    }
    let score = f32::from(record.score_for_side_to_move());
    let result = match us {
        Color::White => record.result().for_white(),
        Color::Black => 1.0 - record.result().for_white(),
    };
    let w = targets.result_weight;
    let target = (1.0 - w) * sigmoid(score / targets.eval_scale) + w * result;
    Example {
        inputs,
        count,
        target,
    }
}
```

The eval scale comes from the data, fitted as chapter 16 fitted `K`: on 134,000 records with balanced starts it came out at 159, close to the 146 found earlier.

Forward, loss, backward

The forward pass is the network itself, run on the example's inputs:

```
/// The network's output for `example`, before the sigmoid. Leaves the hidden layer in
/// `scratch` for [`backward`].
pub fn forward(net: &FloatNetwork, example: &Example, scratch: &mut Scratch) -> f32 {
    let h = net.hidden;
    let mut output = net.output_bias;
    for view in 0..2 {
        let values = &mut scratch.hidden[view * h..(view + 1) * h];
        values.copy_from_slice(&net.input_biases);
        for &input in example.inputs(view) {
            let row = &net.input_weights[usize::from(input) * h..(usize::from(input) +
1) * h];
            values.iter_mut().zip(row).for_each(|(v, w)| *v += w);
        }
        let weights = &net.output_weights[view * h..(view + 1) * h];
        output += values
            .iter()
            .zip(weights)
            .map(|(&v, w)| net.activate(v) * w)
            .sum::<f32>();
    }
    output
}
```

The loss is the squared difference between `σ(y)` and the target. Training needs its **gradient**: for every weight, how much the loss changes as that weight changes. The chain rule gives it, working back from the output. The loss's slope with respect to `y` is `2 (σ(y) − target) σ(y) (1 − σ(y))`:

```
/// The loss for an output and a target, `(σ(output) - target)²`, and its slope with
respect to
/// the output.
pub fn loss(output: f32, target: f32) -> (f32, f32) {
    let p = sigmoid(output);
    let error = p - target;
    (error * error, 2.0 * error * p * (1.0 - p))
}
```

From there, an output weight's gradient is that slope times its hidden value's activation, and a hidden value's is the slope times its output weight times the activation's own slope, which is zero wherever the clipping holds the value at 0 or 1. A first-layer weight collects its hidden value's gradient once for each view in which its input is on: only about 32 of the 768 rows change for any one position.

```
/// Adds `example`'s gradients to `grads`, given the hidden layer [`forward`] left in
`scratch`
/// and `d_output`, the loss's slope with respect to the output.
pub fn backward(
    net: &FloatNetwork,
    example: &Example,
    scratch: &mut Scratch,
    d_output: f32,
    grads: &mut Gradients,
) {
    let h = net.hidden;
    grads.output_bias += d_output;
    for view in 0..2 {
        let values = &scratch.hidden[view * h..(view + 1) * h];
        let weights = &net.output_weights[view * h..(view + 1) * h];
        let output_grads = &mut grads.output_weights[view * h..(view + 1) * h];
        for i in 0..h {
            output_grads[i] += d_output * net.activate(values[i]);
            scratch.d_hidden[i] = d_output * weights[i] * slope(net.activation,
values[i]);
        }
        let d_hidden = &scratch.d_hidden;
        grads
            .input_biases
            .iter_mut()
            .zip(d_hidden)
            .for_each(|(g, d)| *g += d);
        for &input in example.inputs(view) {
            let row =
                &mut grads.input_weights[usize::from(input) * h..(usize::from(input) +
1) * h];
            row.iter_mut().zip(d_hidden).for_each(|(g, d)| *g += d);
        }
    }
}
```

Hand-written gradients need checking. The test nudges weights a little each way, measures how the loss moves, and compares that with the formulas: they agree to 0.1%. The check itself taught two lessons. Near the clipping's corners a nudge measures the wrong slope, so those hidden values are left out. And in `f32`, $\sigma(y)$ near ½ is only exact to about 6×10^{-8}, a few percent of the change a small nudge makes, so the test measures the loss in `f64`.

Stepping: AdamW

The gradients of a batch of records are averaged, and **AdamW** moves each weight against its gradient, as chapter 16's tuner did. It keeps a running average of each weight's gradient and of its square, and steps by about their ratio, so rarely used inputs learn as fast as common ones. It also pulls every weight a little towards zero (weight decay), and afterwards clips it to ±1.98, the limit the whole-number network relies on:

```
/// Moves every weight of `net` one step against `grads`, then clips it to the weight
limit.
pub fn step(&mut self, net: &mut FloatNetwork, grads: &Gradients) {
    self.steps += 1;
    let correction1 = (1.0 - f64::from(Self::BETA1).powf(self.steps as f64)) as f32;
    let correction2 = (1.0 - f64::from(Self::BETA2).powf(self.steps as f64)) as f32;
    let rate = self.learning_rate;
    let update = |weight: &mut f32, g: f32, m: &mut f32, v: &mut f32, decay: f32| {
        *m = Self::BETA1 * *m + (1.0 - Self::BETA1) * g;
        *v = Self::BETA2 * *v + (1.0 - Self::BETA2) * g * g;
        let (m_hat, v_hat) = (*m / correction1, *v / correction2);
        *weight -= rate * (m_hat / (v_hat.sqrt() + Self::EPSILON) + decay * *weight);
    };
    let (first, second) = (&mut self.first, &mut self.second);
    let groups = [
        (
            &mut net.input_weights,
            &grads.input_weights,
            &mut first.input_weights,
            &mut second.input_weights,
        ),
        (
            &mut net.input_biases,
            &grads.input_biases,
            &mut first.input_biases,
            &mut second.input_biases,
        ),
        (
            &mut net.output_weights,
            &grads.output_weights,
            &mut first.output_weights,
            &mut second.output_weights,
        ),
    ];
    for (weights, g, m, v) in groups {
        for i in 0..weights.len() {
            update(
                &mut weights[i],
                g[i],
                &mut m[i],
                &mut v[i],
                self.weight_decay,
            );
            weights[i] = weights[i].clamp(-WEIGHT_LIMIT, WEIGHT_LIMIT);
        }
    }
    // The output bias: no decay and no clipping.
    let (m, v) = (&mut first.output_bias, &mut second.output_bias);
    update(&mut net.output_bias, grads.output_bias, m, v, 0.0);
}
```

The loop

Each **epoch** visits every training record once in a new random order, in batches of 16,384. A batch is shared out among the CPU's threads, and their gradients are added up in a fixed order, so the same settings always give the same network. The learning rate drops to a tenth for the last quarter of the epochs. After each epoch the trainer writes the network, with whole-number weights, as a file the engine can read, and a checkpoint to resume from.

The last 1% of the records, whole games the training never sees, are the **validation set**. The training loss shows how well the network fits the positions it learns from; the validation loss shows how well it has learnt chess. A test makes the difference plain. A student network learns the scores of a random "teacher" network of the same shape: on 1,920 positions its training loss falls 200-fold, but on unseen positions it is only 38% better than guessing the average, and with 576 positions no better at all. It has learnt the positions by heart. Only many positions teach a network chess, which is why a first one needs about 100 million.

On the M1 Max, with its 8 performance cores, the trainer handles about 2 to 2.4 million records a second, so an epoch of 100 million records takes under a minute and 20 epochs under 20 minutes. Generating those records takes about 12 hours: data, not training, is what takes the time. Every thread gets an equal share of each batch and the batch waits for the slowest, so the trainer leaves the efficiency cores out: with them, 10 threads were 17% slower than 8.

On the GPU

The same training runs on the Mac's GPU with `--backend gpu` (in a trainer built with `--features gpu`). It goes through wgpu, a Rust library for WebGPU, which runs on Metal on the Mac and on Vulkan or DirectX 12 elsewhere, with an API that is safe Rust. The code that runs on the GPU, a **kernel**, is written in WGSL, WebGPU's shading language. Thousands of copies of a kernel, its **invocations**, run at once, each computing one number, such as one hidden value of one example, in **workgroups** of 64 that the GPU schedules together. A training step is a row of kernels, each one step of the formulas above: the hidden layer, the output and loss, the hidden values' slopes, the gradients, AdamW, the loss's running total. The network and AdamW's averages stay on the GPU for the whole run; only each batch's inputs and targets go up, and the network comes back once an epoch, for the file and the checkpoint. The CPU and GPU versions sit behind one interface, `Backend`, with one call per batch, so the loop above drives both.

A GPU is fast at work that splits into many equal, independent pieces, and slow at anything else. Two parts of the backward pass weren't like that at first.

Sums over the batch. An output weight's gradient is a sum over all 16,384 examples. With one invocation per weight looping over the whole batch, a network of 256 had only 513 invocations for a GPU that runs tens of thousands, and those sums took 25 of a batch's 35 milliseconds. So the batch is cut into chunks of 128 examples: one kernel sums each weight over each chunk, and a second adds up the chunks.

The sparse gradient. A first-layer weight collects a term from every example in which its input is on. On the CPU, each thread adds its examples' terms into its own copy of the gradients. On the GPU, thousands of invocations would be adding into the same weights at once, which needs atomic addition, and WGSL has none for floats. So the CPU sorts each batch's rows by input first, and each weight adds up its own list with nobody else writing to it. The sort is a counting sort: count each input's rows, turn the counts into starting places, then put each row in its place:

```
// A counting sort: input i's rows will be entries[offsets[i]..offsets[i + 1]], in row
// order.
for i in 1..=INPUTS {
    offsets[i] += offsets[i - 1];
}
let mut next = offsets.clone();
let mut entries = vec![0u32; offsets[INPUTS] as usize];
for (row, inputs) in inputs.chunks_exact(MAX_ACTIVE).enumerate() {
    for &input in inputs.iter().take_while(|&&input| input != NONE) {
        entries[next[input as usize] as usize] = (first_row + row) as u32;
        next[input as usize] += 1;
    }
}
```

The lists turned out to be very uneven: in a batch the average input is on in about 700 of the 32,768 rows, but a king on its first square is on in over 11,000. A kernel is as slow as its slowest invocation, so one invocation per weight, looping over its list, made this kernel as slow as its longest list, whatever the network's size: a network of 512 trained no faster than one of 1,024. So the lists are also cut into segments of up to 256 rows. One kernel sums each segment for every hidden value, and a second adds up each input's segments:

```
// The first layer's weight gradients, first stage. A weight's gradient is its hidden
value's
// slope, summed over every example and view in which its input is on. WGSL has no
atomic
// addition for floats, so instead of each row adding into the weights it touches, the
rows are
// grouped by input beforehand (`entries`, built on the CPU) and each weight sums its
own group.
// The groups are very uneven: in a batch of 16,384 the average input is on in about
700 rows,
// but a king on its first square in over 11,000, and the invocation adding those up
would keep
// the rest of the GPU waiting. So each group is cut into segments of up to SEGMENT
rows, one
// invocation sums one hidden value over one segment, and input_gradients.wgsl adds up
each
// input's segments. The order is fixed, so every run gives the same result.

@group(0) @binding(1) var<storage, read> d_hidden: array<f32>;
// The rows, grouped by input.
@group(0) @binding(2) var<storage, read> entries: array<u32>;
// Segment s is entries[segments[2s]..segments[2s + 1]].
@group(0) @binding(3) var<storage, read> segments: array<u32>;
// Per segment: its sum for each hidden value.
@group(0) @binding(4) var<storage, read_write> segment_sums: array<f32>;

@compute @workgroup_size(64)
fn main(@builtin(global_invocation_id) id: vec3<u32>) {
    let unit = id.x;
    let segment = id.y;
    if (unit >= sizes.hidden || segment >= sizes.segments) {
        return;
    }
    var sum = 0.0;
    for (var n = segments[2u * segment]; n < segments[2u * segment + 1u]; n++) {
        sum += d_hidden[entries[n] * sizes.hidden + unit];
    }
    segment_sums[segment * sizes.hidden + unit] = sum;
}
```

That doubled the speed of networks of 512. Every sum runs in a fixed order, so the same GPU gives the same network every time, but not quite the CPU's: floats added in another order round differently. The tests run both on the same batch and compare the gradients, an AdamW step and a short training run; breaking the segments on purpose makes the gradients' test fail.

Measured on the M1 Max (32 GPU cores) on 2.8 million records, in records a second:

Hidden layer	CPU, 8 threads	GPU
256	2.3 million	5.3 million
512	1.1 million	3.0 million
1,024	0.54 million	1.4 million

The training losses were the same to six decimals. The CPU's time grows with the network, since it does every multiplication itself. The GPU's grows more slowly. At first it didn't even set the pace at 256: the CPU prepared each batch, decoding the records and sorting the rows, on one thread in about 4 milliseconds, while the GPU's work took 3.5, so the GPU waited. Now each of the trainer's threads prepares a share of the batch, and the shares are joined input by input, in order, so the lists come out exactly as one thread would make them:

```
// Input i's rows are each share's rows for input i, the shares in order: the same
lists, in
// the same row order, whatever the number of threads.
let mut offsets = Vec::with_capacity(INPUTS + 1);
let mut entries = Vec::with_capacity(shares.iter().map(|share|
share.entries.len()).sum());
for i in 0..INPUTS {
    offsets.push(entries.len() as u32);
    for share in &shares {
        let rows = share.offsets[i] as usize..share.offsets[i + 1] as usize;
        entries.extend_from_slice(&share.entries[rows]);
    }
}
offsets.push(entries.len() as u32);
// Each input's list cut into segments of up to SEGMENT rows.
let mut segments = Vec::new();
let mut first_segment = Vec::with_capacity(INPUTS + 1);
for i in 0..INPUTS {
    first_segment.push((segments.len() / 2) as u32);
    for start in (offsets[i]..offsets[i + 1]).step_by(SEGMENT) {
        segments.extend([start, offsets[i + 1].min(start + SEGMENT as u32)]);
    }
}
first_segment.push((segments.len() / 2) as u32);
```

On 8 threads the preparation takes 0.8 milliseconds, and training at 256 went from 2.7 to 5.3 million records a second, with the same losses. The GPU's own work is now the limit at every size.

Measuring had a trap of its own. On a file of 141,000 records the GPU seemed no faster than the CPU at 256. An epoch was only nine batches there, and the trainer reads the loss back 20 times an epoch for its progress line, so after every batch the CPU waited for the GPU to finish, and the two never worked at the same time. A real run has thousands of batches an epoch. A benchmark must look like the run it stands for.

Other Macs

The numbers belong to one Mac. The M3 Max, where cb2026's long runs go, differs from the M1 Max in each of the things that set them. Its performance cores are faster, and it has 10 or 12 of them rather than 8, so the CPU trainer speeds up. Its GPU is a newer design, with 30 or 40 cores rather than 24 or 32, that gives each running kernel only the on-chip memory it uses rather than the most it might need, so more invocations fit at once (Apple calls this Dynamic Caching). Its memory bandwidth is 300 or 400 GB/s, depending on the model, against the M1 Max's 400, and the sparse gradient's kernels do little arithmetic for each number they read, so memory is likely to bound them. So the GPU's lead, 2.3 to 2.8 times on the M1 Max, could grow or shrink there. `scripts/mac-trainer-speed.sh` measures both backends on the Mac it runs on. On the M3 Max with 10 performance cores and 30 GPU cores, on 2.9 million records (cjus/chessengine#187):

Hidden layer	CPU, 10 threads	GPU
256	3.3 million	5.1 million
512	1.7 million	2.9 million
1,024	0.78 million	1.5 million

Its CPU trainer is about 45% faster than the M1 Max's, and its GPU about as fast, so the GPU's lead shrank to 1.5 to 1.9 times. It still leads at every size, so the long runs train on the GPU. With all 14 cores, the efficiency cores too, the CPU trained more slowly at 256, 3.1 million records a second against 3.3: each thread gets an equal share of a batch, and the batch waits for the slowest, which is why the trainer uses only the performance cores unless told otherwise. A GPU of another design may also round a little differently, so a network is only repeated exactly on the same machine, backend and thread count.

The network in the search

The search evaluates millions of positions a second, each a move away from one it has just seen. So each search thread keeps a stack of accumulators, one per ply: the one at ply `p` belongs to the position `p` moves from the root. Wherever the search is about to look at a child, it builds the child's accumulator from its own with the move's changes; for a null move, which moves no piece, it copies it. The root's is computed from scratch at the start of each search.

```
/// Writes into `child` the accumulator after a move with these `changes`, given
`parent`,
/// the accumulator before it: what the search does at every move, without allocating.
pub fn update_into(&self, parent: &Accumulator, child: &mut Accumulator, changes:
&Changes) {
    child.copy_from(parent);
    self.update(child, changes);
}
```

```
/// Computes the root's accumulator from scratch: every search starts here.
fn refresh_accumulator(&mut self, root: &Position) {
    if let Some(nnue) = &mut self.nnue {
        nnue.net.refresh(root, &mut nnue.accumulators[0]);
    }
}

/// Makes the accumulator for the child at `ply + 1` from the one at `ply`, for `mv`
played
/// in `pos` (`None`: a null move, which changes no piece). Called wherever a child is
/// about to be searched.
fn update_accumulator(&mut self, pos: &Position, mv: Option<Move>, ply: u32) {
    if let Some(nnue) = &mut self.nnue {
        let (parents, children) = nnue.accumulators.split_at_mut(ply as usize + 1);
        let (parent, child) = (&parents[ply as usize], &mut children[0]);
        match mv {
            Some(mv) => nnue.net.update_into(parent, child, &Changes::of(pos, mv)),
            None => child.copy_from(parent),
        }
    }
}

/// Whether `pos` is an ending the hand-crafted evaluation plays better than the
network:
/// a lone king ([`Techniques::lone_king`]) or a small pawnless ending
/// ([`Techniques::pawnless_endings`]).
fn hand_crafted_ending(&self, pos: &Position) -> bool {
    (self.techniques.lone_king && lone_king(pos))
        || (self.techniques.pawnless_endings && pawnless_ending(pos))
}

/// The static evaluation of `pos`, at `ply`: the network's, from the accumulator the
/// search has kept up to date, or the hand-crafted evaluation's (also with a network
in
/// the endings it plays better, see [`Searcher::hand_crafted_ending`]).
fn evaluate(&self, pos: &Position, ply: u32) -> i32 {
    match &self.nnue {
        Some(nnue) if !self.hand_crafted_ending(pos) => {
            let acc = &nnue.accumulators[ply as usize];
            // Debug builds check every incremental update against a fresh
computation.
            debug_assert!(
                *acc == nnue.net.accumulator(pos),
                "accumulator out of step at ply {ply}"
            );
            let value = nnue.net.evaluate(acc, pos.side_to_move());
            value.clamp(-NNUE_EVAL_LIMIT, NNUE_EVAL_LIMIT)
        }
        _ => evaluate(pos, self.params),
    }
}
```

Getting this right everywhere is the hard part. The search makes moves in five places: the main move loop, null-move pruning,

ProbCut, the quiescence search and the root. Miss one and the network quietly evaluates the wrong position. So in a debug build every evaluation compares the accumulator it was handed with one computed from scratch, and a test searches positions with castling, en passant, promotions, Chess960 castling and checks. Leaving out the null move's copy, or the quiescence search's update, fails it at once.

The network's score is clamped to ±20,000 centipawns, so that nothing it computes could be taken for a tablebase win or a mate. Correction history (chapter 14.2) corrects it just as it corrects the hand-crafted evaluation.

Since the first network passed its test (below), it is built in and used by default. The UCI option `EvalFile`, the shell's `set evalfile` and `cb2026 bench NET` load another; `none`, or the `nnue` switch, goes back to the hand-crafted evaluation, and then the bench is what it was before networks, to the node.

A lesson from a random network

The first test searched with a network of random weights, and on one position, Kiwipete, it took 11 million nodes to reach depth 4, where the hand-crafted evaluation needs 5,000. The accumulators were right. The evaluation was the problem: random scores tell the quiescence search nothing, so almost no capture sequence is cut off early, and Kiwipete is full of captures. A network built by hand to count material, a sensible evaluation if a simple one, searched 26,000. A network trained on only 134,000 positions searches the bench in 4.6 times as many nodes as the hand-crafted evaluation. Pruning is only as good as the evaluation it trusts.

Each node costs about 5% more with a 256-wide network than with the hand-crafted evaluation: 1.50 million nodes a second against 1.57 million on the M1 Max, in plain Rust that the compiler vectorises.

The first network

The first network's data took the M3 Max 12.4 hours: 1,344,000 games of the hand-crafted evaluation against itself, on 9 threads, gave 104 million positions, 3.3 GB. White won 40.5% of the games, 23.8% were drawn and Black won 35.6%, and 30% of the scores were within a pawn of level, thanks to the balanced starts. Training took 7 minutes on the M3 Max's GPU for a network of 256 and 13 minutes for one of 512.

The losses tell how it went. The 256 network's validation loss fell from 0.0103 after the first epoch to 0.0094 after the twentieth, with a visible step at the sixteenth, where the learning rate drops. Its training loss ended at 0.0090, close to the validation loss: it learnt chess, not the positions. (The 134,000-position sample above ended at 0.047.) The 512 network did a little better on unseen games, 0.0093, but its training loss pulled further away, 0.0086: with 100 million positions, a wider network starts to learn the positions by heart.

Then the checks that a network is sane before it plays a game:

Position (side to move)	Hand-crafted	256	512
The starting position	+10	+24	+15
White without its queen	−931	−819	−785
White without a knight	−269	−227	−236
White without a pawn	−52	−19	−44
A pawn on the 7th, kings far away	+290	+752	+745
King and bishop against king, a dead draw	+269	+148	+149

The material is roughly where the hand-crafted evaluation puts it, the pawn about to promote is worth far more to the networks than to the hand-crafted evaluation, and every position mirrored (the colours swapped and the board turned round) scores exactly the same. The bench, the other side of the lesson above: with the 256 network the search needs 716,942 nodes, fewer than the hand-crafted evaluation's 749,181, where the weak networks needed several times as many. A good evaluation prunes well.

The first games: 200 at one second each plus 0.01 a move, the 256 network against the hand-crafted evaluation, ended 172 wins, 4 losses and 24 draws, about +424 Elo ± 71. Games that fast exaggerate any difference in evaluation, so that is a first sign rather than a measurement; the SPRT at the usual 8 seconds plus 0.08 is what counts. It passed after 14 games, all of them won: the log-likelihood ratio reached 3.21, beyond the 2.94 that accepts the gain. A result that lopsided proves the network stronger but not by how much: with no losses or draws there is nothing to measure a difference from, and that takes a fixed number of games. Stage 9's piece-square tables passed the same way, 14 to 0.

The network by default

Passing made the network the default. Its file is part of the source (395 KB, in `nets/`, with a note of how it was made), and the build puts its bytes into the binary, where the engine reads them the first time it needs them:

```
/// The default NNUE network, built into the binary: gen3, 256 wide, trained on 201
million
/// positions of gen2's self-play; gen2 learnt from gen1's, and gen1 from the hand-
crafted
/// evaluation's. It beat gen2 by SPRT, about +53 Elo (cjus/chessengine#222), as gen2
had beaten
/// gen1 (cjus/chessengine#214) and gen1 the hand-crafted evaluation
(cjus/chessengine#195).
/// `nets/README.md` says how each was made.
const BUILTIN_NETWORK: &[u8] = include_bytes!("../../../nets/cb2026-gen3-256.nnue");

/// What `EvalFile` shows when the built-in network is in use, and accepts to choose
it.
pub const BUILTIN_NETWORK_NAME: &str = "<built-in>";

/// The built-in network, read the first time it is asked for and shared from then on.
pub fn builtin_network() -> Arc<Network> {
    static NETWORK: OnceLock<Arc<Network>> = OnceLock::new();
    NETWORK
        .get_or_init(|| {
            Arc::new(
                Network::from_bytes(BUILTIN_NETWORK)
                    .expect("the built-in network is a valid network file (tested)"),
            )
        })
        .clone()
}
```

`EvalFile` now shows `<built-in>`. Empty, or `none`, means the hand-crafted evaluation, and a path another network; the shell's `set evalfile`, `cb2026 bench` and `datagen --evalfile` read the same words, decided in one place (`EvalChoice`). Two kinds of position keep the hand-crafted evaluation even so.

The strength levels. Chapter 19 calibrated them with the hand-crafted evaluation. With the network the same node limits would play stronger, and the ratings would be wrong, so below full strength

the search keeps the evaluation they were measured with, until they are calibrated again.

A lone king. Making the network the default broke one test: king and queen against king, played at 3,000 nodes a move, with which the self-play tools check that they record a won game. The network drew it. Five such endings, each played out by both evaluations:

Nodes a move	Hand-crafted wins	Network wins
3,000	5 of 5	0 of 5
20,000	5 of 5	2 of 5
100,000	5 of 5	4 of 5
500,000	5 of 5	5 of 5, more slowly

To mate a lone king, the side ahead drives it to the edge, and the hand-crafted evaluation rewards that, through its king tables and mobility. The network was trained on balanced games, where a bare king is rare, and it scores those positions much alike: a big lead, whatever the kings do. So the search shuffles until a repetition or the fifty-move rule ends the game. A deep search finds the mate anyway, which is why the games at 8 seconds never showed it, but fast games and shallow analysis don't. Stockfish kept special evaluations for such endings beside its networks for years. Here, when exactly one side has only its king, the search uses the hand-crafted evaluation (the `loneking` switch), and the network wins all five at 3,000 nodes, as quickly as the hand-crafted evaluation.

Small pawnless endings. The lone king wasn't the only gap. Before v1.0.0 was tagged, a test of endgames played out won positions, each confirmed by the Lichess tablebases, with cb2026 on both sides and

no tablebases of its own, once with each evaluation. Where the defender keeps a pawn, or a rook against a rook and pawn, the network converted as often as the hand-crafted evaluation or more. With a queen against a single piece it failed, at 200,000 nodes a move, about what a game at 8+0.08 gives a move:

Ending	Network	Hand-crafted
K+Q against K+R (45 positions)	15	32
K+Q against K+B (15)	12	15
K+Q against K+N (15)	9	15

The cause is the lone king's again. A queen ahead is a big lead to the network wherever the kings stand, and nothing in its few such training positions showed it how to make progress, so nearly every failure was a draw by the fifty-move rule. A queen against a rook is hard for the hand-crafted evaluation too, since the rook can harass the king for many moves, but its king terms push the defending king towards the edge, and that is enough most of the time. So the `pawnless` switch widens the rule: in a pawnless ending of at most five men (three pieces besides the kings, `PAWNLESS_ENDING_PIECES`), the search uses the hand-crafted evaluation. Those are the endings the common Syzygy tables solve, so an engine with tablebases plays them perfectly with either evaluation. With the switch, the network converts exactly what the hand-crafted evaluation does, 32, 15 and 15, since the search is the same. Its SPRT, from 100 such endings, wins and draws (`tests/data/pawnless-endings.epd`), passed at +39 ± 16 Elo in 1,195 games, and 1,000 games from the usual openings showed nothing else changed, +1.7 ± 15.3 (cjus/chessengine#208).

The bench now searches with the network: 715,827 nodes, against the hand-crafted evaluation's 749,181. A few of its positions reach a bare king or a small pawnless ending deep in the search, where the hand-crafted evaluation takes over. Every commit message's `Bench:` line changes with it.

A new evaluation is a new way for the engine to fail: an accumulator updated at every move, and nodes about 6% slower, so less time to spare. Before v1.0.0 was tagged, the release's robustness run was played again with the network: 10,000 games against itself, 6,000 of them at 1+0.01, with no time losses, illegal moves, failures or aborted games. So were 2,400 more with four search threads each, which start and stop on every move.

Against other engines the network counts for even more than the SPRT could say. In the gauntlet of chapter 19, seven rated engines at 8+0.08, the engine with the network rated 3236 ± 16, where the hand-crafted evaluation had rated 2782 ± 11: about 450 Elo. The reference engines are now too weak for a sharp figure. The engine won at least 96.6% of its games against five of the seven, and such scores say little, so the rating rests on the two strongest, against which it performed at 3223 and 3245.

The second network

The first network learnt from the hand-crafted evaluation's games: that evaluation's searches gave the scores, and its play chose the positions and decided how the games ended. The usual next step is for a network to teach its successor. Its searches judge positions better, and its games go where a stronger player goes. So the second

network, gen2, learnt from the first network's own games, with everything else as before: the same balanced starts, 6,000 nodes a move, about 100 million positions and 20 epochs. Whatever changed came from who played.

The data took the M3 Max 15.1 hours, against 12.4 for the first: 1,288,000 games gave 102.9 million positions, each taking about 23% longer to make than with the hand-crafted evaluation. White won 40.9% of the games, 25.9% were drawn (23.8% before) and Black won 33.2%. Training took 7 minutes for a network of 256 and 13 for one of 512, as before.

The validation losses ended at 0.0097 for 256 and 0.0096 for 512, above the first networks' 0.0094 and 0.0093. That says nothing about which is better: each loss is measured on its own generation's games, whose targets came from a different player, so losses compare only networks trained on the same data. The gap between the training and validation losses tells the same story as before: 0.0003 for 256 and 0.0007 for 512, which again learnt more of its positions by heart.

The first network's checks, again:

Position (side to move)	Hand-crafted	gen1, 256	gen2, 256	gen2, 512
The starting position	+10	+24	+31	+29
White without its queen	−931	−819	−844	−797
White without a knight	−269	−227	−236	−241
White without a pawn	−52	−19	−23	−15
A pawn on the 7th, kings far away	+290	+752	+628	+606
King and bishop against king, a dead draw	+269	+148	+118	+71

The material barely moved, and the dead draw came closer to zero, most of all in the wider network. All 285 positions of the test openings and the endgame tests score the same mirrored, in all three networks. The 256 network's bench is 701,720 nodes, against gen1's 715,827, at the same speed. The 512 network's is 732,931, and it searches 25% fewer nodes a second.

In the endgame test above, with the rules for a bare king and small pawnless endings as shipped, gen2 converts exactly what gen1 does, 98 and 108 of 130 positions at 3,000 and 20,000 nodes a move: those endings get the hand-crafted evaluation either way. Where the network does play, it was quicker: a queen against a pawn took 19 plies on average at 20,000 nodes, against gen1's 46. Without the rules, at 200,000 nodes, gen2 had learnt a little of what the queen endings need:

Ending	gen1	gen2	Hand-crafted
K+Q against K+R (30 positions)	10	13	20
K+Q against K+B (15)	12	13	15
K+Q against K+N (15)	9	12	15

Better, but short of the hand-crafted evaluation, so the `pawnless` rule stays. The network's own games hold few such endings to learn from; data with more of them is a job for the next generation.

None of these checks says whether gen2 plays better. That takes games. In the SPRT against gen1 (cjus/chessengine#214), gen2 given by `EvalFile` against the built-in gen1 at 8+0.08, gen2 won 98 games, lost 16 and drew 95, about +144 ± 35 Elo, and passed after 209 games. That is a big step for a network of the same size trained the same way on as many positions, and it came from the data alone: the first network's searches gave better scores, and its games went where a stronger player goes. The figure is likely too high. An SPRT that stops this early stops because the games went well, and self-play overstates gains against other engines. gen2 is clearly stronger; how much stronger against other engines is a gauntlet's question.

v1.0.0 wasn't tagged yet, so gen2 replaced gen1 as the built-in network before it was (cjus/chessengine#216). Only the weights changed, so the code the release's tests had run stayed the same, and the bench went from 715,827 nodes to 701,720. The gauntlet and the robustness run were played again with gen2 before the tag (cjus/chessengine#215). In the gauntlet it rated 3286 ± 17, 50 above gen1's 3236 ± 16: about a third of what the SPRT measured, since self-play overstates gains against other engines. Against the two engines

that carry the rating it scored 91.6% and 78.4%, where gen1 had scored 86.8% and 75.6%. The robustness run's 10,000 games were clean again.

The third network

gen3 changed two things at once: it learnt from gen2's games, and from twice as many of them, 200 million positions. More positions are the cheapest gain a network can get, since they need no new code, only time on the test Mac: 29.1 hours for these, 45 chunks of 56,000 games (cjus/chessengine#219). The games came out much as gen2's had: White won 40.8%, 26.2% were drawn and Black won 33.0%. Training took 13 minutes for a network of 256 and 24 for one of 512, twice as long as before, since each epoch held twice the records.

The extra data showed first in the losses. The gap between training and validation loss halved, to 0.00018 for 256 and 0.00038 for 512: with twice the positions, a network learns fewer of them by heart. The wider network gained more. Its validation loss pulled ahead of the 256 network's by 0.00029, against 0.00011 with gen2's data, since a network with more weights needs more positions before its size pays.

The same checks again, now against gen2:

Position (side to move)	Hand-crafted	gen2, 256	gen3, 256	gen3, 512
The starting position	+10	+31	+38	+38
White without its queen	−931	−844	−849	−766
White without a knight	−269	−236	−247	−233
White without a pawn	−52	−23	−35	−41
A pawn on the 7th, kings far away	+290	+628	+592	+603
King and bishop against king, a dead draw	+269	+118	+83	+89

The dead draw came closer to zero again, and all 285 test positions score the same mirrored. The networks are settling: across those positions gen3's 256 network differs from gen2's by 37 centipawns on average, where gen2 had differed from gen1 by 78. Its bench is 767,812 nodes, more than gen2's 701,720, at the same speed; the 512 network's is 868,798.

In the endgame test, with the rules as shipped, gen3 converts what gen2 does at 3,000 nodes a move, 98 of 130, and one fewer at 20,000, 107, a rook and pawn against a rook that it drew. Without the rules it won a few more queen endings at 200,000 nodes: 41 of 60, against gen2's 38 and the hand-crafted evaluation's 50. Each generation learns a little more of them, but slowly, so the `pawnless` rule stays.

Whether gen3 plays better is again the SPRT's question. Against gen2 (cjus/chessengine#222), at 8+0.08, gen3 won 176 games, lost 86 and drew 333, about +53 ± 19 Elo, and passed after 595 games. Each generation gains less than the one before, +144 for gen2 and +53 for gen3, as the networks' evaluations close in on what their games can

teach them. The bench had pointed the other way, 767,812 nodes against gen2's 701,720: fewer nodes often means a sharper evaluation, but not always, and only games decide. v1.0.0 ships gen2 even so: it had been through the release's tests, and a gain this size waits for the next release.

Twice the data had helped the 512 network most, so it had its own SPRT, against gen3's 256 (cjus/chessengine#223). After 8,180 games at 8+0.08 it stood at 1,653 wins, 1,627 losses and 4,900 draws, about +1 ± 5 Elo, with no verdict, and was stopped. The wider network knows more but searches about a quarter fewer nodes a second, and at this time control the two cancel. So the 256 stays the candidate: as strong, and faster. Wider networks tend to do relatively better at longer time controls, where each extra ply is worth less and so a quarter fewer nodes costs less; that test is for a later network.

v1.0.0's tag went on the commit its tests had run on, so `main` could move on: gen3 became the built-in network for v1.1 (cjus/chessengine#228), and the bench went from 701,720 nodes to 767,812. Its gauntlet (cjus/chessengine#227) rated it 3379 ± 21, 93 above gen2: more than the SPRT's +53, where gen2's gauntlet had shown only a third of its SPRT. Against Stash v27.0, the strongest reference engine, it scored 88.4%, where gen2 had scored 78.4%. One explanation, untested: self-play measures a network against a near-copy of itself, which shares its blind spots, while twice the data mostly taught gen3 to generalise (the smaller gap between its training and validation losses), which may count for more against engines unlike it. The figure is still blunt: five of the seven engines lost nearly every game, so it rests on two.

Pitfalls

- **Trainer and engine must agree** on the numbering, the layers and the scales. They share `crates/nnue`, and tests pin it down: every input has its own number, a move's changes turn one position's inputs into the next one's, updates match recomputing, and mirrored positions score the same.
- **An eval scale that doesn't match** quietly changes every margin in the search.
- **Overflow** in the integer arithmetic gives silently wrong scores. The weight limit and the bounds above rule it out, and a test runs the largest possible weights.
- **Endings the data never showed.** A network learns only what its positions show. Balanced self-play has few bare kings, so the first network couldn't drive one to the edge. Test the endings every beginner must win, king and queen or rook against king, at low node counts, where the search can't cover the gap.
- **A network that quietly doesn't load.** An engine that can't read its `EvalFile` says so in an `info string` and plays on with the hand-crafted evaluation, and a match tool ignores info strings, so a test of the network would measure the old evaluation against itself. `mac-sprt.sh` loads each network through UCI before the match and stops if it doesn't load.
- **A move the accumulators miss.** Every place the search makes a move must update the child's accumulator; a debug build checks each evaluation.
- **A GPU kernel is as slow as its slowest invocation.** One invocation per weight looks natural, but if some weights have far

more work than others (the kings' inputs in a batch), the GPU sits idle while they finish. Cut the work into equal pieces and add them up after.

- **A benchmark that doesn't look like the real run.** A small file made the GPU wait at every progress report and look slower than it is.
- **Scores from the wrong side.** The search scores for the side to move; records keep White's side, like the result, and the trainer turns both to the side to move's view. Mixing the two up teaches the network nonsense for every position with Black to move.

Further reading

- Chess Programming Wiki: NNUE.
- WGSL, WebGPU's shading language, and wgpu, the Rust library the trainer's GPU backend runs on.
- Yu Nasu, *Efficiently Updatable Neural-Network-based Evaluation Functions for Computer Shogi* (2018), which introduced NNUE.

Index

Page numbers run in order, so a term is usually introduced at the first page listed. For a term the book uses on most pages, only the pages that discuss it are listed.

A

accumulator 315, 319, 320, 322, 339, 340, 341, 342, 348, 355

Adam 254, 256, 260, 331, 334, 336

all-node 57, 146, 169, 202, 205, 208, 209, 212, 213, 215, 223

alpha-beta 47, 62, 87

aspiration window 79, 139, 142, 144, 146, 150, 151, 152, 287

B

balanced starts 325, 328, 343, 349

bench 61, 103, 134, 192, 342

best-move stability 76

beta cutoff 8, 47, 58, 115, 124, 125, 132, 133

bishop pair 231, 238, 240, 245, 251

bitboard 10, 11, 12, 14, 15, 16, 17, 19, 28, 29, 31, 226, 242

black magics 31

blessed loss 269

blunder 283, 284, 285, 291, 292, 293, 295, 296, 297, 298, 302, 303

bucket 94, 95, 97, 98, 99, 100, 101, 102, 103, 104, 315, 325

bulk counting 25

C

cache line 97, 98, 99, 105

Caissa 236

capture history 128, 130, 131, 132, 134, 137, 138

castling rights 9, 12, 13, 21, 23, 32, 33, 271, 275, 276, 280, 323

CECP 305

centipawn 71, 77, 79, 148, 227, 230, 293, 307, 321, 322

check extension 58, 186, 187, 188, 191, 217, 218, 221, 224

check mask 17, 18

Chess Programming Wiki 5, 10, 14, 25, 26, 31, 42, 62, 75, 80, 88, 95, 96, 104

Chess960 12, 20, 21, 268, 272, 273, 274, 280, 307, 308

conformance tests 312

continuation history 78, 115, 121, 122, 124, 125, 126, 127, 174, 203, 204, 221

copy-make 24

correction history 60, 148, 193, 195, 196, 225, 228, 234, 235, 301, 342

countermove 114, 115, 116, 117, 119, 120, 121, 122, 123, 124, 127, 201

counting sort 335

Crafty 162, 164

CReLU 318, 321

cursed win 269

cut node 141, 167, 205, 206, 207, 215, 223

cutoff count 146, 208, 211, 216, 223

D

Deep Thought 197
delta pruning 84, 147, 184, 185
divide 25, 26, 322
double extension 54, 134, 148, 189, 193, 194, 195, 196, 198, 220, 222, 223, 301
doubled pawn 240, 251
DTZ 268, 271, 280

E

effective branching factor 63
Elo 127, 137, 208, 221, 222, 223, 235, 259, 266
en passant 13, 276
Engine API 35, 306
epoch 255, 256, 333, 334, 338, 343, 349, 352
eval scale 318, 321, 322, 327, 328, 355

F

fail high 91, 151
fail low 59, 91, 208, 210
feature negotiation 307
FEN 9, 12, 13, 14, 41, 272, 273, 308
fifty-move rule 9, 37, 61, 66, 269, 271, 280, 346, 347
Fruit 176
futility pruning 147, 182, 185, 199, 225, 230

G

game phase 44, 45, 108, 109, 112
gauntlet 299, 301, 348, 351, 354
gradient 250, 254, 255, 257, 327, 329, 330, 331, 332, 333, 334, 335, 336, 338
graph history interaction 95, 96
gravity 115, 116, 119, 123, 124, 133, 134

H

hand-crafted evaluation 237, 301, 341, 342, 343, 344, 345, 346, 347, 348
hard limit 70, 71, 72, 73, 77, 78, 79
hashfull 95, 111, 290
helper thread 38, 67, 234, 262
history heuristic 87, 114, 120, 127, 138, 178
history pruning 55, 147, 173, 174, 175, 176, 177
history table 114, 121, 122, 125, 126, 127, 128, 129, 173, 175, 176, 178, 201
horizon effect 73, 75, 81, 88

I

IIR 177, 191, 197, 198, 199, 217, 219, 221
improving 50, 147, 182, 186, 199, 200, 201, 217, 219, 221
internal iterative deepening 198, 224
internal iterative reductions 49, 147, 198, 224
isolated pawn 240, 244, 251
iterative deepening 63, 75, 198

K

kernel 334, 335, 338, 355
killer move 87, 113, 114, 115, 116, 117, 119, 120, 123, 124, 134, 201, 266
king buckets 315, 325
king safety 238, 240, 241, 244, 245
king zone 240, 243
Kiwipete 25, 60, 86, 87, 95, 118, 163, 177, 265, 342

L

late move pruning 8, 55, 125, 147, 170, 173, 174, 175, 176, 199, 200, 217

late move reductions 125, 146, 165, 172, 186, 201

Lazy SMP 38, 67, 261, 263, 265, 267, 287

legal move generation 17, 25

Lichess 277, 284, 291, 292, 293, 295, 298, 312, 313

lichess-bot 312, 313

LLR 80, 104, 127, 137, 151, 164, 177, 221, 222, 223, 224, 235, 259

LMP 170, 171

LMR 171, 203, 217, 218, 221

lockless hashing 262

lower bound 53, 83, 85, 90, 160, 161, 163, 190, 230

M

magic bitboards 16, 27, 30, 31

magic number 27, 29, 31

mailbox 11, 12, 14

marlinformat 323

mate score 92, 151, 156, 161, 163, 227, 229, 230, 234, 322

minimax 46, 47, 60, 62, 87, 96, 113, 153

mobility 45, 109, 238, 239, 242, 244, 245, 250, 251, 252, 346

Move Overhead 71, 72, 312

move picker 116, 204

multi-cut 54, 189, 190, 191, 192, 193, 194, 198, 218, 221, 231

MultiPV 282, 288, 289, 291, 307

MVV-LVA 86, 130, 131, 132, 137

N

negamax 6, 43, 46, 47, 48, 51, 54, 62, 91, 142, 154, 168, 189

negative extension 54, 148, 190, 191, 192, 194, 218, 220, 222

NNUE 301, 314, 345, 356

null window 141, 167, 194

null-move pruning 7, 51, 135, 153, 154, 155, 156, 157, 158, 181, 225, 231, 341

O

open file 240, 245, 251, 270

opening book 36, 65, 268, 274, 275, 277, 280

overfitting 256, 258, 259

P

passed pawn 238, 240, 244, 245, 246, 251

pawn key 230, 236

pawn shield 240, 243

perft 15, 24, 25, 26, 30, 31, 33, 34, 118

person's horizon 281, 284, 285, 287, 291, 295, 296, 298, 301

PEXT 31

piece-square table 44, 106, 108, 112, 237, 244, 245, 246, 251, 344

pin 6, 17, 18, 19, 20, 26, 185, 284, 355

Polyglot 275, 276, 277, 278, 280

ponder hit 78, 309, 310, 311

pondering 305, 309, 310, 311, 313

principal variation 57, 139, 152, 169, 202, 209, 210, 215, 216

principal variation search 139, 140, 152

ProbCut 51, 52, 53, 148, 158, 159, 160, 161, 162, 163, 164, 178, 193, 342

pseudo-legal 17

Q

quiescence search 85, 86, 132, 137, 161, 163, 184, 287, 342

R

razoring 50, 51, 147, 179, 182, 184, 185, 225, 230

regularisation 256, 257, 258, 259

repetition 32, 40, 61, 95, 156, 346

replacement strategy 93, 96, 104

reverse futility pruning 50, 147, 181, 182, 185, 199, 200, 207, 211, 219, 225, 230

S

SAN 34

SCReLU 318, 319, 321, 322

search instability 151, 152

SEE 56, 84, 87, 131, 147, 179, 181, 183, 184, 185

seldepth 41, 48, 73, 83, 85, 111, 144, 290

self-play 78, 107, 148, 203, 208, 246, 249, 259, 285, 294, 309, 313, 322, 323

shakmaty-syzygy 269

Shredder-FEN 13, 272, 273

sigmoid 246, 247, 254, 321, 328, 329

singular extension 49, 92, 188, 191, 217, 221, 222

singular_beta 54, 148, 188, 189, 191, 192, 193, 194

soft limit 70, 71, 72, 73, 76, 77, 79, 80

SPRT 80, 127, 137, 221, 222, 223, 235, 259, 266, 351, 354

staged generation 118

stand pat 83, 85, 87, 228, 231

static exchange evaluation 7, 87, 129, 130, 160, 179, 185

Stockfish 148, 149, 208, 211, 212, 213, 223

strength level 281, 291, 345

Syzygy 268, 269, 280, 308, 347

T

tablebase 37, 38, 39, 49, 50, 66, 67, 68, 268, 269, 270, 271, 280, 347

tapered evaluation 106, 107

tempo 45, 109, 238, 239, 241, 243, 244, 251, 252

Texel tuning 246, 250, 260, 322

transposition table 89, 262, 287

TT cutoff 90, 91, 96

ttPv 209, 223

U

UCI 32, 42, 72

UCI_Elo 282, 291, 297, 298, 301, 303

upper bound 90, 127, 161

V

validation set 257, 258, 333

verification search 155

W

WDL 268, 271

wgpu 334, 356

WGSL 334, 335, 336, 356

workgroup 334

X

X-FEN 13, 14

x-ray 180, 181

XBoard 42, 305, 306, 307, 308, 310, 313

Z

Zobrist hashing 12, 32, 33, 42, 89, 90, 96, 99, 226, 275

zugzwang 155, 156, 157