

# Chess Engine: Technical Report

## Overview

This project presents a fully functional chess engine capable of evaluating positions, exploring move sequences, and selecting optimal continuations through classical search algorithms.

The implementation builds on the `python-chess` library, which provides data structures for representing the chessboard, moves, and game state. It also handles rule enforcement (e.g., legality of moves, detection of checkmate and stalemate, move notation, and position hashing), allowing the project to focus on the higher-level algorithmic components of a chess engine.

## Board evaluation

The board evaluation function is the core analytical component of the engine. Its purpose is to assign a numerical score to a given (static) chess position, indicating which side is favored and by how much. The score is expressed in centipawns, where one pawn corresponds to a value of 100. Following chess conventions, a positive score indicates an advantage for White, while a negative score indicates an advantage for Black.

The evaluation is based on two components: material balance and positional advantage. The former is given by a simple sum of the values of the pieces that each side still has on the board. These base values, drawn from classical chess programming, are intended to reflect the average relative strength of each piece:

$$\text{Pawn} = 100, \quad \text{Knight} = 320, \quad \text{Bishop} = 330, \quad \text{Rook} = 500, \quad \text{Queen} = 900.$$

The positional component refines this raw evaluation by introducing context-dependent adjustments based on piece placement and structural features of the position. It combines information from two main sources: PSTs (Piece–Square Tables) and heuristic bonuses/penalties.

PSTs are 8x8 numerical tables that encode the desirability of each square for each piece type. During evaluation, every piece contributes its base value plus the bonus (or penalty) corresponding to its square. For Black, the tables are mirrored vertically to maintain symmetry.

For example, knights are rewarded for occupying central squares (values in centipawns):

$$\text{KNIGHT\_TABLE} = \begin{bmatrix} -50 & -40 & -30 & -30 & -30 & -30 & -40 & -50 \\ -40 & -20 & 0 & 5 & 5 & 0 & -20 & -40 \\ -30 & 5 & 10 & 15 & 15 & 10 & 5 & -30 \\ -30 & 0 & 15 & 20 & 20 & 15 & 0 & -30 \\ -30 & 5 & 15 & 20 & 20 & 15 & 5 & -30 \\ -30 & 0 & 10 & 15 & 15 & 10 & 0 & -30 \\ -40 & -20 & 0 & 0 & 0 & 0 & -20 & -40 \\ -50 & -40 & -30 & -30 & -30 & -30 & -40 & -50 \end{bmatrix}$$

Heuristic features further refine this evaluation by applying bonuses and penalties for factors such as bishop pairs, rook coordination, and pawn structure (passed/doubled/isolated pawns).

## Move ordering

Before analyzing possible continuations, the engine assigns a priority score to each legal move and sorts them from most to least promising. Each move's score is based on simple heuristics (captures, promotions, checks).

The resulting ordered list does not determine which move is best, but rather establishes a ranking of candidate moves for deeper analysis in the search phase. This becomes vital when the engine begins to search through possible continuations: the algorithm can often skip large parts of its search if it encounters strong moves early. By examining the most promising moves first, the engine is more likely to find good positions quickly.

## Move search and Alpha-Beta pruning

While the evaluation function measures the quality of a single position, a chess engine must also reason about how the position might evolve after a sequence of moves. This is achieved through a search algorithm which simulates future play by exploring possible move sequences for both sides.

At a conceptual level, the program constructs a search tree, where each node represents a board position and each branch represents a possible move. White tries to maximize the score (find the best position for itself), while Black tries to minimize it (reduce White's advantage). This idea forms the basis of the minimax algorithm.

To make the search more efficient, the engine uses alpha–beta pruning, a technique that avoids exploring branches that cannot influence the final decision. It works by keeping track of two limits (alpha and beta) as the engine explores the tree of moves, where alpha represents the best score that the player to move can guarantee so far, and beta represents the best score that the opponent can force. As the search progresses, alpha and beta define a range of outcomes that are still worth considering: if the engine realizes that a line of play cannot possibly lead to a better result than one already found, it can safely ignore the rest of that branch.

The `search` function returns only a numerical score, not the actual move. This design reflects its role as a recursive evaluation tool: at every level of the tree, the function represents a player choosing the move that leads to the highest resulting score. A separate function (`best_move`) handles the top-level call: it performs a full search for each candidate move, compares their resulting scores, and then selects the one with the highest evaluation.

## Quiescence search

When the main search function reaches its maximum depth, it stops looking further ahead and asks for an evaluation of the current position. However, this approach can be unreliable if the position is tactically unstable (for example, if there are captures or checks that could immediately change the material balance). To handle this, instead of evaluating the position directly, the engine calls the quiescence search function (`qsearch`), which continues exploring only "noisy" moves.

This allows the engine to "tidy up" the evaluation at the edges of the search tree. Once the tactical noise has settled, the resulting score is passed back to the main search, giving a much more stable and realistic assessment of the position and avoiding the so-called horizon effect.

## Transposition Table caching

As the engine analyzes possible continuations, it often encounters the same position multiple times through different sequences of moves. Re-evaluating such positions repeatedly would be redundant and inefficient.

To prevent this, the engine maintains a Transposition Table (TT), a form of memory cache that stores information about previously analyzed positions. When the engine reaches a new position, it first checks whether this position already exists in the table. If so, it can immediately reuse the stored evaluation, rather than recomputing it from scratch.

Each board position is identified by a Zobrist hash, a 64-bit number that acts as a unique finger-print for that specific arrangement of pieces. The process is managed by two helper functions: `tt_store` and `tt_probe`.

After finishing the analysis of a position, the function `tt_store` saves the result into the table. It records the position's hash, the search depth, the best move found, and a flag describing how the stored score should be interpreted. The flag distinguishes whether the stored score is an exact evaluation, a lower bound ( $\text{score} \geq \beta$ ), or an upper bound ( $\text{score} \leq \alpha$ ), allowing the alpha–beta search to reuse information safely.

The function `tt_probe` checks whether the current position is already present in the transposition table. If an entry exists, the cached score is immediately returned to avoid redundant searching.

## Weaknesses and potential future upgrades

While the engine demonstrates reliable functionality and can autonomously play complete games without external input, it remains a simplified implementation designed for transparency and experimentation. The current design prioritizes readability and conceptual clarity over raw playing strength or computational efficiency, making it an effective tool for illustrating classical search and evaluation techniques.

Several areas for improvement can be identified:

- The engine uses a fixed search depth and lacks iterative deepening (a technique that searches progressively deeper while reusing previous results). Integrating this feature and combining it with a time management system would make the engine stronger and more adaptive under time constraints, allowing it to adjust its depth automatically.
- The evaluation model is purely linear and deterministic. Future work could include a more dynamic evaluation that depends on game phase (opening, middlegame, endgame), or even the integration of a simple neural network trained on high-quality chess data to provide more nuanced positional assessments.
- From an engineering perspective, performance could be improved through more efficient data representations and optimized computation. Implementing bitboards, enhancing memory caching, and introducing parallel or time-controlled search would increase both speed and scalability without altering the engine's overall logic.
- Additional features such as an opening book or endgame tablebases could further improve realism and performance.

See the project's [changelog](#) to view version history, updates, and WIP / planned features.