ΔΙΕΘΝΕΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΗΣ ΕΛΛΑΔΟΣ

SCHOOL OF ENGINEERING
Department of Information and Electronic Engineering

DIPLOMA THESIS

Study and Implementation of a Chess Engine

Student
Konstantinos Despoinidis
Student ID: 2021035

Supervisor
Panagiotis Tzekis
Professor

May 12, 2026

Title of Dissertation: Study and Implementation of a Chess Engine
Code of Dissertation: 25331
Student's full name: Konstantinos Despoinidis
Supervisor's full name: Panagiotis Tzekis
Date of undertaking: 18-10-2025
Date of completion: 12-05-2026

I hereby affirm the authorship of this paper as well as the acknowledgement and credit of whichever assistance I received in its composition. I have, furthermore, noted the various sources from which I extracted data, ideas, visual or written material, in paraphrase or exact quotation. Moreover, I affirm the exclusive composition of this paper by myself only, for the purpose of it being a dissertation, in the Department of Information and Electronic Engineering of the I.H.U.

The approval of this dissertation by the Department of Information and Electronic Engineering of the International Hellenic University, does not necessarily entail the adoption of the author's views, on behalf of the Department.

«Στους γονείς μου»

Prologue

I chose this thesis topic because it allowed me to combine a personal passion for chess with my interest in low-level systems programming. Implementing a chess engine from scratch in `C` was a unique challenge that required me to think critically about performance and hardware efficiency.
Through this project, I gained hands-on experience with the fundamentals of Artificial Intelligence, specifically in search algorithms and heuristic design. I also learned the importance of code optimization; when dealing with millions of potential moves, every microsecond and byte of memory matters. While building a chess engine is a notoriously difficult task, the process has significantly improved my problem-solving skills and my confidence in handling complex, high-performance programming projects.

Abstract

This thesis presents *Irida*, a competitive chess engine written in C that adheres to the Universal Chess Interface (UCI) protocol. Its architecture separates board-state mechanics (*Castro*) from search logic, combining hybrid bitboard/grid representation, magic bitboards, Zobrist hashing, and full rule support (castling, en passant, and draw conditions). Correctness and efficiency are validated through *perft* verification and throughput microbenchmarks.

The search module is based on negamax with alpha–beta pruning and iterative deepening, extended with quiescence search, transposition tables, null move pruning, late move reductions (LMR), aspiration windows, and principal variation search (PVS). Evaluation supports both a handcrafted tapered function and NNUE integration. Experimental assessment uses SPRT, Elo estimation, and staged ablation under two controls (fixed movetime and fixed nominal depth). Results show that quiescence, transposition tables, and LMR provide the largest practical gains under short time budgets, while aspiration/PVS are time-control-sensitive and do not consistently improve strength. The final reference match against strength-limited Stockfish 18 yields an estimated rating around 2110 Elo. Overall, the thesis contributes a reproducible, evidence-driven workflow for developing and validating chess-engine search stacks.

Μελέτη και Υλοποίηση Μηχανής Σκακιού

Κωνσταντίνος Δεσποινίδης

Περίληψη

Η παρούσα διατριβή παρουσιάζει την *Irida*, μια ανταγωνιστική μηχανή σκακιού γραμμένη σε C η οποία είναι συμβατή με το πρωτόκολλο Universal Chess Interface (UCI). Η αρχιτεκτονική του συστήματος βασίζεται στο *Castro*, μια εξειδικευμένη βιβλιοθήκη που διασφαλίζει την ορθότητα της αναπαράστασης του σκακιού μέσω μιας υβριδικής προσέγγισης που συνδυάζει bitboard και grid-state. Η βάση αυτή ενσωματώνει magic bitboards για τις ολισθαίνουσες επιθέσεις, Zobrist hashing για αναγνώριση θέσεων και πλήρη υλοποίηση κανόνων (ροκέ, en passant και συνθήκες ισοπαλίας). Με τον διαχωρισμό της μηχανικής σκακιού από τη λογική αναζήτησης, επιτυγχάνεται υψηλή απόδοση παραγωγής κινήσεων, η οποία επαληθεύεται με *perft* και διαγνωστικά μικρο-benchmarks.
Ο πυρήνας αναζήτησης βασίζεται σε negamax με κλάδεμα άλφα-βήτα και επαναληπτική εμβάθυνση, εμπλουτισμένη με quiescence search, πίνακες μεταθέσεων, null move pruning, late move reductions (LMR), aspiration windows και principal variation search (PVS). Η αξιολόγηση υποστηρίζει τόσο χειροποίητη tapered συνάρτηση όσο και ενσωμάτωση NNUE. Η πειραματική αποτίμηση γίνεται με SPRT, εκτίμηση Elo και σταδιακές μελέτες αφαίρεσης σε δύο καθεστώτα ελέγχου (σταθερός χρόνος ανά κίνηση και σταθερό ονομαστικό βάθος). Τα αποτελέσματα δείχνουν ότι το quiescence, οι πίνακες μεταθέσεων και το LMR προσφέρουν τα μεγαλύτερα πρακτικά οφέλη σε μικρό χρόνο σκέψης, ενώ τα aspiration/PVS παρουσιάζουν εξάρτηση από το time control και δεν βελτιώνουν σταθερά τη δύναμη. Στον τελικό αγώνα αναφοράς έναντι περιορισμένου Stockfish 18, η *Irida* εκτιμάται περίπου στα 2110 Elo. Συνολικά, η διατριβή προτείνει μια αναπαραγώγιμη και τεκμηριωμένη μεθοδολογία για την ανάπτυξη και αξιολόγηση μεθόδων αναζήτησης σε μηχανές σκακιού.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Professor Panagiotis Tzekis, for their support throughout the duration of this project. I am especially grateful for the academic autonomy they afforded me. Their trust gave me the freedom to shape this work personally and develop the confidence to conduct research on my own terms. I am also profoundly indebted to my family for their unwavering love and belief in my potential. To my parents, thank you for being my constant foundation and for the countless ways you supported me, both seen and unseen, during my education. Finally, to my friends, thank you for sticking by me even when I became a person who only talks about their thesis. I am grateful for the breaks, the laughs, and for simply being there when I needed to get my head out of my work.

Table of Contents

Figure Catalogue

Table Catalogue

Abbreviations

PVS	Principal Variation Search
NMP	Null Move Pruning
LMR	Late Move Reductions
TT	Transposition Table
NNUE	Efficiently Updatable Neural Network
UCI	Universal Chess Interface
NPS	Nodes Per Second
LLR	Log Likelihood Ratio
SPRT	Sequential Probability Ratio Test
EBF	Effective Branching Factor
CCRL	Computer Chess Ranking Lists
TCEC	Top Chess Engine Championship
LC0	Leela Chess Zero
CPU	Central Processing Unit
GPU	Graphics Processing Unit
PST	Piece-Square Table
MCTS	Monte-Carlo Tree Search
MVV-LVA	Most Valuable Victim - Least Valuable Attacker
SMP	Symmetric Multiprocessing
ep	en-passant
CI	Confidence Interval
TC	Time Control
i.i.d.	independent and identically distributed

Chapter 1: Introduction

1.1 Motivation

Chess has long been regarded as a benchmark problem in computer science due to its immense combinatorial complexity. Despite having simple and well-defined rules, the number of possible positions grows exponentially, making exhaustive search infeasible in practice. While the game has been partially solved in restricted cases, most notably with endgame tablebases covering up to seven pieces [1], [2], the complete solution of chess remains computationally out of reach with current technology. This gap between theoretical solvability and practical limitations makes chess an ideal domain for exploring efficient algorithms and intelligent decision-making.

From a programming perspective, developing a chess engine presents a challenging and rewarding problem. Performance, memory management, and algorithmic efficiency, particularly when working in a low-level language such as C, must be taken seriously. Implementing core components such as move generation, evaluation functions, and search algorithms, force the developer to deep dive in fundamental concepts in systems programming and optimization.

Another motivating factor is the evolving landscape of computer chess, especially with the introduction of neural-network-based evaluation methods such as NNUE (Efficiently Updatable Neural Networks). These hybrid approaches combine traditional handcrafted evaluation techniques with machine-learned models, resulting in highly performant engines [3]. However, they also introduce some uncertainty, often referred to as the “black box” problem, where the reasoning behind certain evaluations becomes less interpretable.

Finally, building a chess engine provides lots of educational value. It serves as a practical application in search theory, including algorithms such as minimax, iterative deepening and alpha-beta pruning. At the same time, it showcases the importance of low-level programming, since efficiency in memory management and execution is critical for a successful chess engine.

1.2 Objectives and Scope

The primary objective of this thesis is to design and implement a competitive chess engine in C that is fully compliant with the Universal Chess Interface (UCI) protocol [4].

The scope of the project includes the following:

- **Move Generation:** Implementation of a complete and efficient move generator capable of handling all legal positions, including special moves such as castling, en passant, and promotions.
- **Search Algorithms and Optimizations:** Development of a high-performance search framework based on Negamax with Alpha-Beta pruning, enhanced with common optimizations such as iterative deepening, move ordering, and pruning techniques.
- **Evaluation Function:** Design of a handcrafted evaluation function to assess positional strength,

along with integration of NNUE (Efficiently Updatable Neural Networks) to improve evaluation quality.

- **UCI Protocol Implementation:** Full support for the UCI protocol to ensure compatibility with existing graphical user interfaces and benchmarking tools.
- **Ablation Studies and Analysis:** Systematic evaluation of individual components and optimizations to measure their impact on engine strength and performance.

The following aspects are explicitly out of the scope of this thesis:

- **Custom NNUE Training:** The project utilizes pre-trained NNUE networks and does not involve the training or generation of new neural models.
- **Evaluation Function Tuning:** The evaluation function is based on hand-crafted heuristics and fixed parameter values, rather than systematically tuned or automatically optimized weights.

This scope ensures a focused exploration of the fundamental techniques behind modern chess engines, while avoiding areas that would significantly expand the project beyond a manageable size.

1.3 Structure of the Thesis

This thesis is structured as follows. First, the necessary background on chess and computer chess programming is introduced to establish the required theoretical foundation. Next, the overall system design and architecture are presented, including the analysis of the UCI protocol. This is followed by a detailed examination of the implementation of *Castro* [5] and *Irida* [6], covering key components such as board representation, move generation, search algorithms, and evaluation techniques. Subsequently, the testing methodology and experimental environment are described, providing the basis for performance assessment. The results are then presented and analyzed, including ablation studies and measured Elo improvements. Finally, the thesis ends with a discussion of the main findings, the contributions of this work, and possible directions for future improvements.

Chapter 2: Background and Literature Review

2.1 Rules of Chess

Chess is a two-player board game played between White and Black on an 8×8 grid. The horizontal coordinates are labeled *files* (a–h), while the vertical coordinates are labeled *ranks* (1–8). Each player begins with sixteen pieces: eight pawns, two knights, two bishops, two rooks, one queen, and one king. The standard starting position is shown in Figure 2.1.

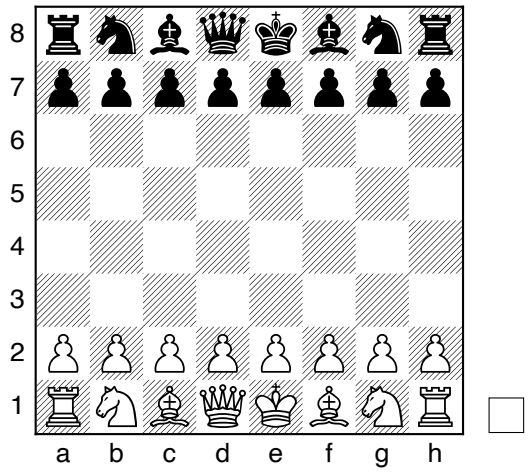


Figure 2.1: Starting position

2.1.1 Movement

Each piece type in chess follows distinct movement rules.

Players alternate turns, with White making the first move. A move consists of transferring a piece from one square to another according to its movement rules. A capture occurs when a piece moves to a square occupied by an opponent's piece, removing it from the board.

Pawns move forward one square at a time along the file on which they are placed. On their first move, they may advance two squares forward, provided that both squares in front of them are unoccupied. Pawns capture diagonally forward one square. When a pawn reaches the final rank (rank 8 for White or rank 1 for Black), it must be promoted to a queen, rook, bishop, or knight.

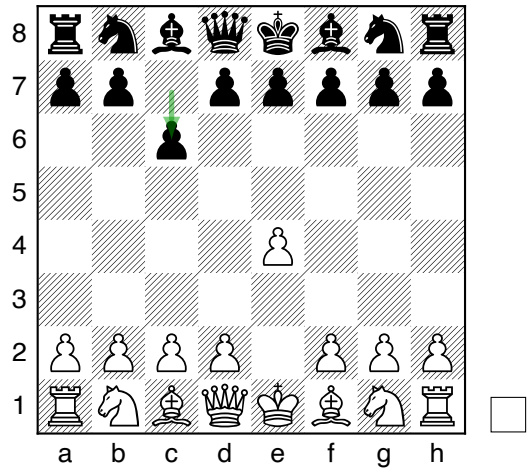


Figure 2.2: Caro-Kann Defence

A special pawn capture called *en passant* (French for "in passing") may occur when a pawn advances two squares from its starting position and lands adjacent to an opposing pawn. In this case, the opposing pawn may capture it as if it had moved only one square forward. This capture must be performed immediately on the following move or the opportunity is lost.

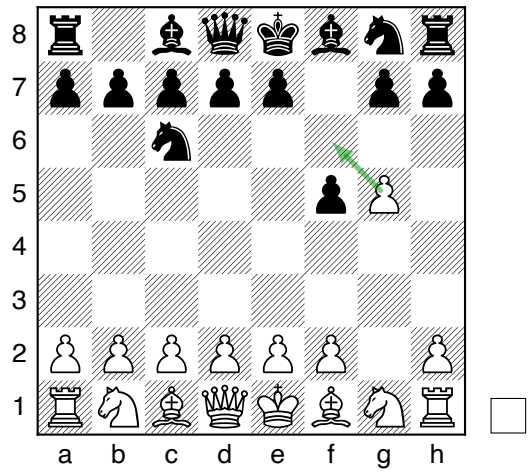


Figure 2.3: En passant example

Knights move in an L-shape: two squares in one direction (horizontal or vertical), followed by one square perpendicular to that direction. They are the only pieces that can jump over other pieces.

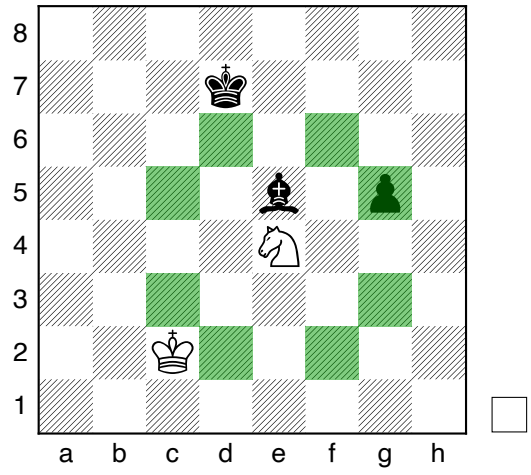


Figure 2.4: Knight movement

Bishops move diagonally any number of squares, provided that no pieces block their path. Each bishop is confined to squares of a single color throughout the game, since diagonal movement does not allow them to change color complexes.

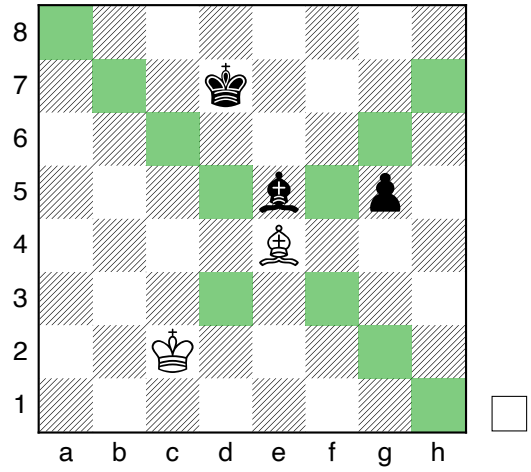


Figure 2.5: Bishop movement

Rooks move horizontally or vertically across the board for any number of squares, subject only to blocking pieces. They play a central role in controlling open files and ranks, particularly in the endgame.

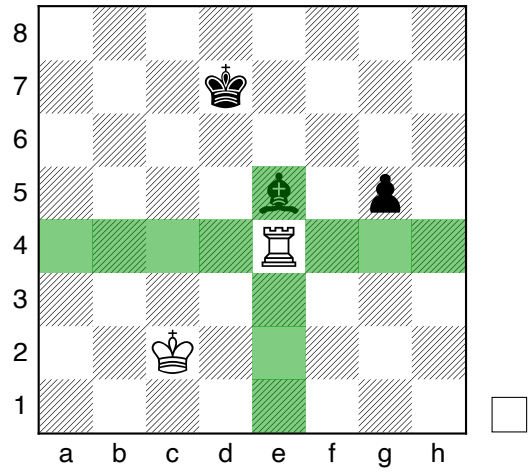


Figure 2.6: Rook movement

The queen combines the movement capabilities of both the rook and the bishop. It may move any number of squares horizontally, vertically, or diagonally, making it the most powerful piece in terms of mobility.

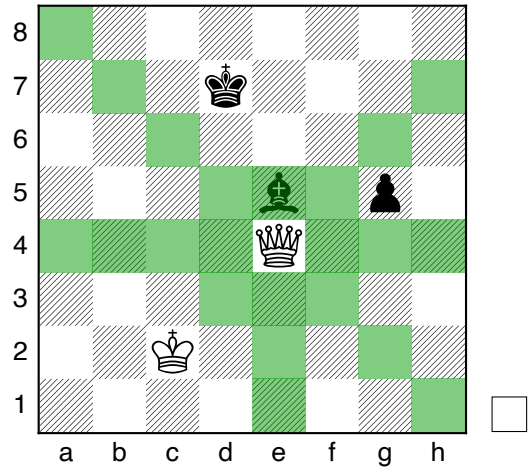


Figure 2.7: Queen movement

The king moves one square in any direction: horizontally, vertically, or diagonally. Although its mobility is limited, it is the most important piece, as the objective of the game is to checkmate the opponent's king.

A king is said to be in *check* when it is under attack by one or more opposing pieces. In such a situation, the player must make a move that removes the threat, either by moving the king, capturing the attacking piece, or blocking the attack. A fundamental rule of chess is that no move may leave or place one's own king in check; any such move is considered illegal.

Since a king attacks by controlling all adjacent squares, two kings may never occupy adjacent squares or directly attack each other.

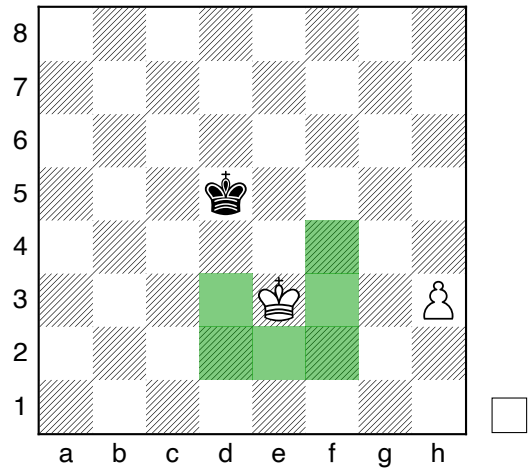


Figure 2.8: King movement

The king is also subject to special rules, one of the most important being castling. Castling is a move in which the king moves two squares towards a rook on the same rank, while the rook is simultaneously transferred to the square over which the king has crossed. Castling is the only move in chess that allows two pieces to move at the same time.

In order to castle, the following conditions must be satisfied:

- The king and the chosen rook must not have previously moved.
- There must be no pieces between the king and the rook.
- The king must not be in check, and it must not pass through or land on a square that is under attack.

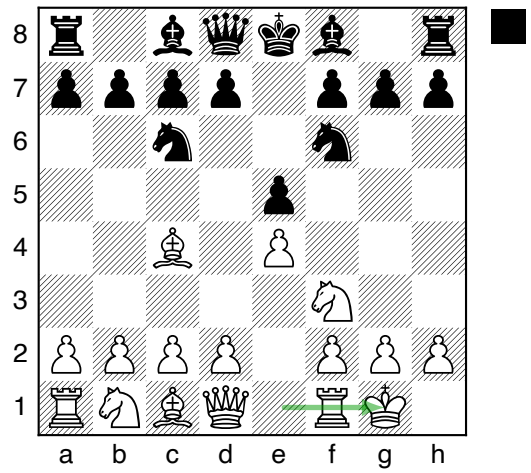


Figure 2.9: Castling example

2.1.2 Results

A chess game can end in several distinct ways.

A win is typically achieved by checkmate, although in practical play it may also occur through resignation or time forfeiture. Checkmate is a position in which the king is in check and no legal move exists that can remove this threat. This may occur either by moving the king to a safe square, capturing the attacking piece, or interposing another piece to block the attack. If none of these options are available, the position is checkmate.

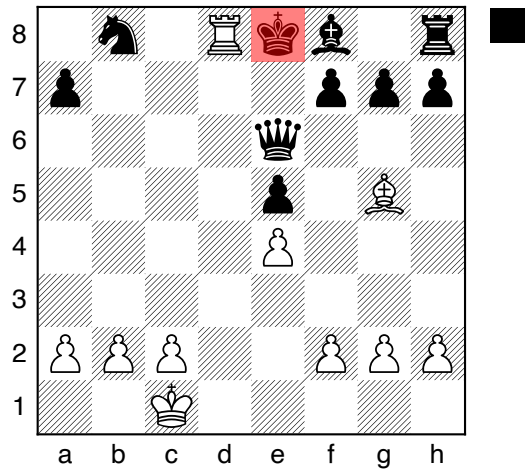


Figure 2.10: Paul Morphy's Opera Game checkmate

A draw, on the other hand, can be reached in multiple ways. The simplest form is stalemate, a position in which the side to move has no legal moves, yet its king is not in check.

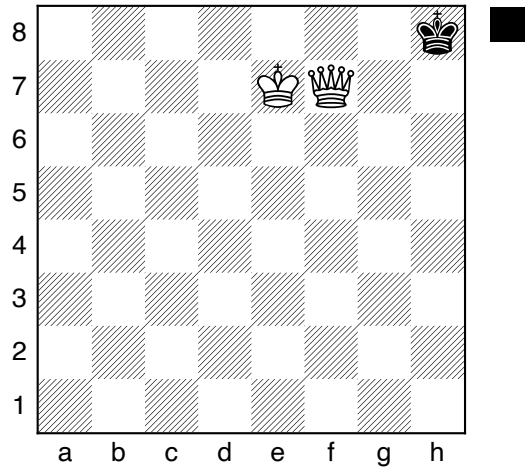


Figure 2.11: Stalemate example

In addition, a draw may occur under the following conditions:

- **Threefold repetition:** The same position occurs three times with the same player to move and identical legal possibilities
- **Fifty-move rule:** Fifty consecutive moves are played by both sides without any pawn movement or capture.
- **Insufficient material:** Neither side has sufficient material to force a checkmate, such as king versus king, king and bishop versus king, or king and knight versus king.
- **Agreement:** The players mutually agree to a draw.

A deep understanding of the rules described above is essential for implementing correct move generation and handling all edge cases. For a complete and authoritative specification, refer to the official FIDE Laws of Chess [7].

2.2 Fundamental Principles of Computer Chess

Computer chess models the game as a deterministic, perfect-information, zero-sum system. This means that both players have complete knowledge of the game state, and there is no element of chance. The zero-sum property implies that any advantage gained by one player corresponds to an equivalent disadvantage for the opponent. Formally, if a position s is assigned a value $V(s)$, then from the opposing player's perspective the value is $-V(s)$.

The decision-making process in computer chess is typically formulated using the minimax paradigm. Given a game tree rooted at position s, the optimal value of the position can be defined recursively as:

$$V(s) = \begin{cases} \max\limits_{a \in A(s)} V(s_a), & \text{if } s \text{ is a maximizing (White) node} \\ \min\limits_{a \in A(s)} V(s_a), & \text{if } s \text{ is a minimizing (Black) node} \end{cases} \tag{1}$$

where $A(s)$ denotes the set of legal moves in position s, and s_a is the resulting position after applying move a.

2.2.1 Negamax formulation

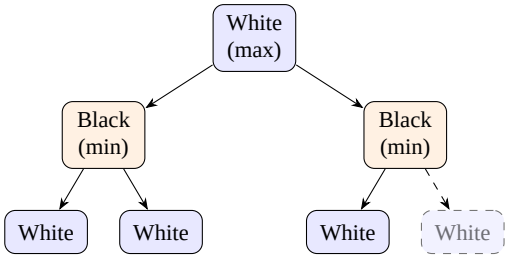
The alternating max and min in Equation (1) is often implemented as a single recursive routine *negamax*: one always evaluates positions from the side to move's perspective and *negates* scores when crossing the edge to the opponent. If s is a terminal or leaf-under-depth position with static score $f(s)$ from the side to move's viewpoint, then $V(s) = f(s)$; otherwise

$$V(s) = \max_{a \in A(s)} \big(-V(s_a) \big), \tag{2}$$

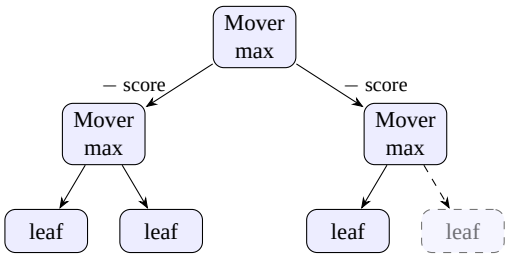
because $V(s_a)$ is defined in the child's frame and must be negated to express the parent's utility. This is algebraically the same minimax recursion as above, but every node is a "max for the mover" node and the implementation does not branch on player colour.

2.2.2 Game tree fragment and negamax view

Figure 2.12 shows a shallow fragment of a game tree. Panel (a) highlights the alternating maximizer (White-to-move) and minimizer (Black-to-move) layers in the classical view. Panel (b) redraws the same fragment in the negamax view: each internal node is a maximization for the side to move, and returning a score to the parent negates it, exactly as in Equation (2).



(a) Classical minimax layers; dashed edge indicates a branch that alpha-beta can skip once a refutation is known.



(b) Negamax view of the same fragment: every internal node is a max for the side to move; edges to children negate returned scores.

Figure 2.12: Shallow game-tree views: alternating minimax versus a single negamax routine.

2.2.3 Search windows and cutoff semantics

Depth-first alpha-beta maintains a *window* (α, β) of plausible scores (in negamax coordinates, both from the side to move's perspective at the current node). While searching a child, any return value $v \leq \alpha$ cannot improve the node's value and yields an *alpha cutoff* (sometimes called a *fail low* at the parent: the child establishes only an upper bound on how good that line was). Symmetrically, if $v \geq \beta$, the line is strong enough to force a *beta cutoff* (*fail high*: the child proves a lower bound at least β, so remaining siblings need not be searched). When neither bound bites, the search returns an *exact* value inside the window.

These terms also describe the *root* outcome under a *narrow* window, as used by aspiration search: if the true minimax value lies below the left end of the window, the iteration *fails low* (the search returns an upper-bound-style outcome relative to the window); if it lies above the right end, it *fails high*. The engine then widens the window and re-searches until the value lies inside or the window is widened to the full real line.

In practice, the full game tree cannot be explored due to its enormous size. Instead, the search is truncated at a fixed depth d, and a heuristic evaluation function $f(s)$ is used to approximate the value of non-terminal positions:

$$V(s) \approx f(s) \quad \text{for leaf nodes at depth } d$$

The evaluation function typically encodes domain-specific knowledge, such as material balance, piece activity, king safety, and pawn structure.

The complexity of chess arises from its large branching factor and depth. The average branching factor

b is approximately 35, meaning that each position has, on average, 35 legal moves. A naive minimax search to depth d therefore requires examining:

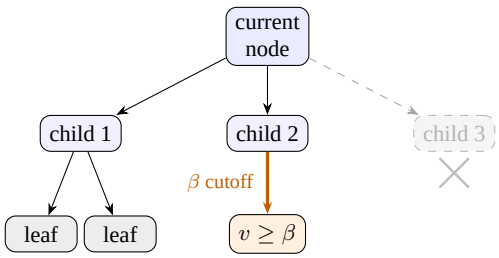
$$O(b^d)$$

nodes. For example, a depth-6 search would involve on the order of $35^6 \approx 1.8 \times 10^9$ positions, which is computationally expensive.

To address this, various optimizations are employed. The most fundamental is alpha-beta pruning, which reduces the number of nodes that must be evaluated by eliminating branches that cannot influence the final decision (Figure 2.13). In the best case, alpha-beta pruning reduces the effective complexity to:

$$O(b^{d/2})$$

effectively doubling the achievable search depth for the same computational cost, although in practice performance depends heavily on move ordering quality [8], [9].



Solid edges are explored; once a child proves $v \geq \beta$ (negamax coordinates), remaining siblings at the same parent need not be expanded.

Figure 2.13: Schematic *beta cutoff*: the third child is never visited because an earlier line already refutes the current window.

Historically, Shannon distinguished between two primary approaches to computer chess. Type A strategies rely on exhaustive search with minimal domain knowledge, exploring the game tree uniformly. In contrast, Type B strategies use selective search, focusing on promising moves while pruning less relevant branches based on heuristics [10]. Modern chess engines combine these approaches, performing deep search while incorporating selective extensions, pruning techniques, and sophisticated evaluation functions.

Despite these optimizations, the state space of chess is extraordinarily large, often estimated to be on the order of 10^{43} possible positions, with a game-tree complexity around 10^{120} (order-of-magnitude figures popularised in early computer-chess discussions of search limits [10]). As a result, computer chess fundamentally relies on approximations, heuristics, and efficient search strategies to make strong decisions within practical time constraints.

2.3 Classical Engine Architecture

A classical chess engine follows a clear sequence of steps to choose the best move in a position. In general, this process consists of move generation, move ordering, search, and evaluation [3].

The first step is to generate all legal moves. This step must be correct, as even a single missing legal move or an illegal one can lead to incorrect results. Move generation therefore has to fully respect the rules of chess, including checks, pins, castling, en passant, and promotions. It must also ensure that no move leaves the king in check.

Once all legal moves are generated, they are ordered. Some moves are more promising than others, so it makes sense to examine them first. Good move ordering improves search efficiency, especially when using alpha-beta pruning, where strong moves considered early can significantly reduce the number of positions that need to be explored. Common techniques include prioritizing captures, promotions, killer moves, and moves that have performed well in similar positions.

Next comes the search. The engine explores possible moves and their consequences by building a game tree, typically using a depth-limited minimax algorithm with alpha-beta pruning. In practice, this is often implemented using the single-routine *negamax* formulation of Equation (2); see Figure 2.12 for a comparison with the classical minimax view. The search is further improved with techniques such as iterative deepening, which increases depth gradually, and quiescence search, which extends the search in unstable positions to avoid missing important tactical sequences.

When the search reaches its limit, the engine evaluates the resulting positions using a static evaluation function. This function assigns a score based on factors such as material balance, piece activity, king safety, and pawn structure. The scores are then propagated back through the tree to determine the best move.

Modern engines include additional features to improve performance. Transposition tables store previously evaluated positions to avoid redundant work, while time management controls how long the engine searches based on the situation. Together, these components allow the engine to search efficiently and make strong decisions.

2.4 Modern Trends: Neural Networks and Alpha-Beta Refinements

Modern chess engines have increasingly adopted neural networks, most notably NNUE (Efficiently Updatable Neural Networks), as part of their evaluation pipeline [11]–[15]. These networks either replace traditional hand-crafted evaluation functions entirely or augment them to improve accuracy, particularly in complex or tactically sensitive positions.

At the same time, the search process has been heavily optimized through advanced pruning and reduction techniques. These methods aggressively limit the number of positions explored, allowing engines to search deeper while maintaining efficiency. Although not all such techniques are implemented in Irida, a subset is used to reduce the search space effectively.

In addition, modern engines take advantage of parallelism by distributing the search across multiple

threads. This significantly accelerates computation, enabling strong performance even on relatively modest hardware.

Finally, some engines adopt a fundamentally different approach based on reinforcement learning and Monte-Carlo Tree Search (MCTS) [16], [17]. This paradigm is less common, because it demands substantially more computation than typical alpha–beta engines. In those systems, a neural network proposes promising moves (policy) and evaluates positions (value); MCTS then simulates many possible continuations and grows a search tree biased toward stronger moves. Rather than exhaustive width-first expansion like alpha–beta, MCTS performs selective, statistically guided exploration.

2.5 Existing Solutions

Among contemporary chess engines, Stockfish has been the most dominant over the past decade [18]. According to CCRL ratings [19], it has achieved an Elo exceeding 3600, placing it well beyond human competitive levels. For comparison, the current best player Magnus Carlsen has a classical rating of approximately 2840 [20]. While such comparisons are not directly equivalent due to differing rating pools, they provide a useful illustration of the performance gap between top engines and human players.

Stockfish relies on highly optimized alpha-beta search, capable of evaluating hundreds of millions of positions per second, combined with an NNUE for fast and accurate position evaluation. This hybrid approach preserves the strengths of classical search while incorporating neural network guidance.

In contrast, Leela Chess Zero (LC0) follows a fundamentally different paradigm. Inspired by reinforcement learning systems such as AlphaZero [21], LC0 replaces alpha-beta search with Monte-Carlo Tree Search (MCTS) and relies on a deep neural network trained through self-play. This approach typically leverages GPU acceleration to efficiently evaluate positions.

Chapter 3: System Design

3.1 Design Philosophy

Irida is organized, with performance and modularity in mind. Board representation and move generation and other correctness-critical mechanics (rules, move generation, make/unmake, repetition and draw state) are handled by **castro** [5], while the engine core consentrates on search, ordering, evaluation and tablebases. This seperation of concerns ensures, the thinking modules can never corrupt the board state, while also allowing them to evolve without rewritting low-level move logic.

The codebased is also shaped for *iterative optimization*: features are grouped into coherent compilation units (search, quiescence, transposition table, move ordering, evaluation, UCI front-end) with a smaill composition root (thin-main pattern). The main binary wires the engine together through function pointers (`SearchFn`, `EvalFn`, `OrderFn`), so alternate search modes (e.g., benchmarking or simplified drivers) or evaluation backends (classical PeSTO, NNUE, material-only debugging) can be selected without changing the negamax loop's contract. Optional subsystems (NNUE via an external probe library, Syzygy via Fathom [22]) integrate at the edges rather than spreading conditionals through the tree.

3.2 Architectural Overview: Module Analysis

At the center of the architecture is an `Engine` aggregate (`include/core.h`): metadata (name, author), the active chess `Board` (provided by the **castro** library [5]), and three function pointers, search, evaluation, and move ordering, that define the engine's behavior at runtime. The UCI entry point initializes this structure and dispatches protocol traffic; when a search is requested, the driver invokes the configured `search` implementation with the current board and the injected `eval` and `order` callbacks.

Move generation and board representation live outside Irida's core sources, in **castro**: legal and pseudo-legal move generation, captures vs. quiet flags, null moves, hashing, and FEN import/export. The search layer treats castro as the single source of truth for position legality and state updates; Irida consumes a stable C API (`deps/include/castro.h`) rather than embedding its own generator.

Search (`src/search/`) implements the iterative-deepening driver, recursive negamax with alpha-beta and optional PVS, quiescence, null-move and LMR policies, aspiration windows at the root, and integration hooks for Syzygy WDL probes [2], [23]. The **transposition table** (`src/search/tt.c`, `include/tt.h`) is a self-contained subsystem keyed by the board hash [24]–[26], with generation-based invalidation between searches.

Move ordering (`src/moveordering.c`) is deliberately isolated: the search routine never hard-codes ordering heuristics; it calls the `OrderFn` on generated move lists and updates killer/history state on cutoffs. This keeps ordering experiments localized.

Evaluation (`src/eval/`) provides classical scoring (PeSTO-style tapered eval [27] plus supplementary terms and optional material-only paths) and an optional NNUE adapter (`src/eval/nnue.c`) that delegates inference to a linked probe [13], [14]. Evaluation is invoked only through the `EvalFn` indirection, matching the same boundary used by quiescence stand-pat.

Interface and services : the **UCI** module (`src/uci/`) parses commands, maps options (hash size, tablebase paths, eval file, search flags) into a `SearchConfig`, synchronizes with a search thread where applicable, and reports `info` lines [4]. **Syzygy** support (`src/syzygy.c`) wraps the Fathom probe layer for root and in-tree WDL [2]. Smaller **applications** under `apps/` reuse the same engine wiring for benchmarks, validation, or ad-hoc tools; **tests** under `test/` exercise symmetry, search, and evaluation in isolation.

In sum, the architecture is a *thin orchestration layer* (UCI + `Engine`) around three replaceable pillars, **search**, **ordering**, and **evaluation**, all operating on a shared **board** abstraction owned by castro, with **tablebase** and **NNUE** components attached as optional services rather than entangled with the core tree search. Figure 3.1 summarizes the major dependencies at a glance.

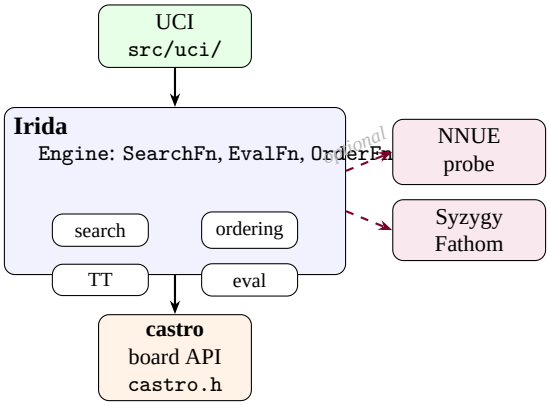


Figure 3.1: High-level module layout: UCI drives the Irida core (search, ordering, TT, evaluation) on a **castro** board; NNUE and Syzygy attach at the edges rather than inside move generation.

3.3 The Universal Chess Interface (UCI) Protocol

The Universal Chess Interface was first developed in 2000 by Rudolf Huber and Stefan Meyer-Kahlen. It specifies a text-based command channel between a chess engine and a separate graphical user interface or tournament manager: the GUI owns the board display and clock, while the engine receives positions and search constraints and returns principal variations, scores, and best moves [4]. Irida follows this contract so it can plug into standard arenas (Cutechess, Banksia, Lichess bot bridges, and similar tools) without bespoke wiring [28]–[30].

3.3.1 Architectural model

UCI adopts a *client–server* style: the GUI drives the session; the engine remains passive until commanded. The engine boots quickly and waits for commands; heavy initialization is deferred until after the initial

handshake, so the GUI may query engine identity and options and then issue `quit` without forcing a full setup. The engine must accept input from standard input even while searching, and must remain in *forced mode*: it never begins calculation (including pondering) without an explicit `go`.

3.3.2 Lexical and syntactic conventions

Commands are newline-terminated text lines; line endings may vary by platform (`0x0a`, `0x0d0a`, etc.), which matters when mixing OS boundaries. Arbitrary whitespace may appear between tokens; unknown commands or tokens are ignored where possible, and misplaced commands (e.g. `stop` when not searching) are also ignored. Before any search, the current position is always communicated with a `position` command.

Table 3.1: Selected UCI commands (GUI → engine)

Command	Role
`uci`	Enable UCI mode; triggers `id`, `option`, `uciok`
`debug on|off`	Verbose debugging output
`isready`	Synchronization; answer `readyok`
`setoption name …[value …]`	Change engine parameters
`register …`	Registration flow for commercial engines
`ucinewgame`	Next search is a new game/context
`position …`	Set board and move list
`go …`	Start search with optional constraints
`stop`	Stop search; still emit `bestmove`
`ponderhit`	Opponent played expected ponder move
`quit`	Terminate engine

3.3.3 Move representation

Moves use *long algebraic notation*: from-square and to-square (four characters), with a promotion suffix where needed. Castling is encoded as the king's move (e.g. `e1g1` for short castling in normal chess). The engine signals a null move to the GUI as `0000`.

3.3.4 Handshake and configuration

After startup, the GUI sends `uci` once. The engine replies with `id name …` and `id author …`, advertises configurable parameters via `option` lines, and finishes with `uciok`. If `uciok` does not arrive within a GUI-defined timeout, the GUI may terminate the engine.

The `isready` command synchronizes the engine with the GUI: the engine must answer `readyok` after processing pending input. It is required before the first search so initialization can complete, and may be sent during search, in that case the engine answers immediately without stopping the search.

`setoption name` *id* `[value` *x*`]` changes engine-internal settings; option names and values are case-insensitive; the substrings `name` and `value` should be avoided inside identifiers to keep parsing unambiguous. `setoption` is only sent while the engine is waiting (not mid-search).

3.3.5 Game lifecycle

`ucinewgame` marks that the next `position`/`go` sequence starts a new analytical or competitive context; engines must not rely on receiving it, because older GUIs might omit it. GUIs should send `isready` after `ucinewgame` if the engine's response may be slow.

`position [fen` *fenstring* `| startpos] moves` $move_1 \dots move_i$ sets the internal board from FEN or the standard start position, then applies the move list. If the position belongs to a different game than the previous one, a `ucinewgame` should have been sent in between.

3.3.6 Search control

`go` begins analysis on the current position. Optional parameters in the same line restrict or bound the search, including: `searchmoves`, `ponder`, clock data (`wtime`, `btime`, `winc`, `binc`, `movestogo`), depth/node/time limits (`depth`, `nodes`, `mate`, `movetime`), and `infinite`. In `ponder` mode the last move in the `position` line is the ponder move; the engine must not exit search on its own even if mate is found; `ponderhit` switches from pondering to normal search when the expected move occurs. `stop` requests termination as soon as possible; every `go` must eventually be answered with `bestmove`.

3.3.7 Engine-to-GUI feedback

During search, the engine may emit `info` lines with structured tokens (`depth`, `seldepth`, `time`, `nodes`, `pv`, `multipv`, `score`, `currmove`, `hashfull`, `nps`, tablebase hits, CPU load, free-form `string`, etc.). PV-related fields should be kept consistent; the specification suggests throttling verbose diagnostics (`currmove`, `currmovenumber`, `currline`, `refutation`) until after one second of search to limit traffic.

When the search ends, the engine sends `bestmove` *m* `[ponder` *p*`]`, optionally with a ponder suggestion; a final `info` line should precede `bestmove` so the GUI has complete statistics. Optional commercial features appear as `copyprotection` and `registration` exchanges; the GUI may need to send `register` after certain engine messages.

3.3.8 Configurable options (`option`)

Each `option` describes a user-tunable parameter: `name`, `type` (`check`, `spin`, `combo`, `button`, `string`), and optionally `default`, `min`, `max`, and repeated `var` for combo boxes. Several names have standardized semantics (`Hash`, `NalimovPath`, `Ponder`, `OwnBook`, `MultiPV`, and many `UCI_*` options); GUIs may hide unknown `UCI_*` options.

3.3.9 Chess960 extension

The specification states that UCI can be extended to Chess960 [31] via `option name UCI_Chess960 type check default false`. In Chess960, castling is encoded as the king "capturing" its own rook (e.g. `e1h1`); FEN castling rights use file letters of the castling rooks because `kq` / `KQ` alone are insufficient when multiple rooks exist on one side.

A simple UCI example can be found in Appendix A.2.2.

3.4 Technical Stack: Languages and Tooling

The project is implemented in **C**, adhering to modern standards while maintaining high portability. The development environment utilizes a cross-platform compilation strategy: `gcc` [32] is employed for Linux environments, while `clang` [33] is used for macOS.

Unlike many modern C projects, the build process relies on **custom Makefiles** rather than CMake to provide more granular control over compilation flags and dependencies. Version control is managed via **Git** [34], with releases and changelogs automated through **changelogger** [35] to ensure consistency.

For quality assurance and documentation, the project integrates:

- **Documentation:** Generated using `tinydocs` [36] for a lightweight, source-integrated manual.
- **Testing:** Data-driven unit testing is handled by `test.h`, a minimalist header-only framework [37].

Finally, custom tooling is written in both bash and python.

Chapter 4: Implementation: The Engine Core

4.1 Move Generation

4.1.1 Board Representation: Bitboards and Magic Bitboards

The board uses a *hybrid* representation: twelve `uint64` bitboards (one per piece kind and colour), together with an 8×8 character grid for `PieceAt`-style queries, and cached aggregates `white`, `black`, and `empty` recomputed whenever occupancy changes. Square-centric loops use `poplsb` / `lsb` to iterate set bits, which maps cleanly to 64-bit word operations on typical architectures. The hybrid design trades a small amount of redundant state (bitboards plus a square grid plus cached colour/occupancy aggregates) for simpler hot paths: `PieceAt` and other square-indexed logic avoid reconstructing piece type from twelve bitboards on every access, while move generation still benefits from bit-parallel attacks and `poplsb` walks. The aggregates keep common occupancy tests $O(1)$ without repeatedly OR-ing all piece bitboards. This is a standard memory-for-clarity pattern in bitboard engines; a pure bitboard design would save bytes but often increase branches in evaluation and make/unmake.

Sliding pieces (bishops, rooks; queens combine both) use *magic bitboards* [38], [39]. At initialisation, for each square a mask of relevant blockers is built; a random 64-bit *magic* multiplier is searched so that all occupancy subsets of that mask map injectively into a small precomputed attack table (rook tables are larger than bishop tables, reflecting different blocker counts). At runtime, attacks are

$$\texttt{table}[s]\big[((\texttt{occ} \,\&\, \texttt{mask}[s]) \cdot \texttt{magic}[s]) \gg \texttt{shift}\big],$$

giving $O(1)$ lookup given occupied squares `occ`. If no satisfactory magic is found for a square, the implementation falls back to classical on-the-fly ray casting (`AttacksFromOccupancy`). Non-sliding pieces use precomputed jump or step masks (knight, king) and rank/file/diagonal masks for pawns. Magic bitboards pay a one-time initialisation cost (magic search and table fill) for attack tables that are read-only at runtime; rook tables are larger than bishop tables because more blocker combinations arise on ranks and files. The asymptotic attack query remains $O(1)$ per square after setup. The ray-casting fallback for a stubborn square adds a second implementation path, so that path should remain cold; it preserves correctness if a magic is not found without blocking engine startup. Other sliding-attack schemes (e.g. masked occupancies with different decompositions) offer different size/speed tradeoffs; magic tables are a common choice when a few hundred kilobytes of precomputed data is acceptable.

4.1.2 Game Logic and Draw Rules

Moves are applied with `MakeMove` and reversed with `UnmakeMove`. Before mutating the board, an `Undo` record captures castling rights and en passant square *before* the move, the halfmove clock before the move, the encoded move, and any captured piece type. Castling, en passant, and promotion are handled as distinct paths: the king-rook rearrangement updates both bitboards and grid; en passant removes the captured pawn on its true square and resets the halfmove clock; promotion replaces the pawn bitboard with the chosen promoted piece and updates hash contributions accordingly. After the piece update, castling rights and the en passant target are recomputed from the move (including rook or king moves

that forfeit rights, and captures on corner squares). The side to move and fullmove number follow FEN-style conventions.

The *fifty-move rule* is implemented as a halfmove clock (`halfmove`): it is reset to zero when a pawn moves or when any capture occurs (the latter detected by comparing total piece counts before and after the move); otherwise it is incremented once per halfmove. A draw is declared when this counter reaches 100 halfmoves (i.e. fifty full moves without a pawn move or capture).

Separately, a repetition structure keyed by the position hash maintains occurrence counts along the current line; unmaking decrements the entry for the child position. The draw-by-repetition predicate follows the usual *threefold repetition* idea: a position that has already appeared twice on the line from the root to the current node (i.e. the third occurrence) is treated as a draw, under the same encoding of full game state (side to move, castling, en passant, and piece placement) that the Zobrist key represents. As with any 64-bit hash, accidental collisions are exceedingly rare but theoretically possible; the implementation assumes identical keys imply identical rule-relevant state. Insufficient material is detected by a dedicated bitmap-based material test. Together, these checks feed the general result / draw logic used after move generation.

4.1.3 Zobrist Hashing

Positions are identified by a 64-bit Zobrist key [24]. The scheme follows the *Polyglot* layout of fixed pseudorandom words `Random64[781]` [40]: one word per piece type and square for the twelve piece classes, four words for the individual castling rights, eight for en passant *files*, and one word toggled when Black is to move. The en passant contribution is *not* keyed solely from FEN: a file's random word is XORed in only when a pawn of the side to move could capture en passant on that file (mirroring Polyglot's rule).

The full hash can be recomputed from scratch by XORing every occupied piece key and the meta-keys. For performance, `MakeMove` and `UnmakeMove` update `board->hash` incrementally: piece movements XOR out the old square and XOR in the new (and captured) pieces; castling and en passant components are removed with the pre-move meta, then re-XORed after rights and the ep target are updated; the side-to-move word is flipped every move. This incremental update is kept consistent with the full recomputation (verified in tests). The same key drives opening-book lookup in Polyglot binary files. Incremental Zobrist updates are an *invariant*: after every `MakeMove`/`UnmakeMove`, the stored hash must equal the xor of the same key material that a from-scratch xor would use. The test suite's full recomputation check catches any drift in castling, en passant, or captured pieces. The TT and Polyglot book therefore see the same position identifier the evaluation and repetition logic see, which is prerequisite for correct transposition-table use and for opening preparation that matches the engine's own rule encoding.

4.1.4 Pseudo-legal Generation, Legality, and the Public API

`GenerateMoves` dispatches to pseudo-legal move generation (all moves that ignore check), legal generation, or legal captures only. Pseudo-legal targets are built per piece from bitboards: pawns (pushes, captures, double push, promotions), knights, bishops, rooks, queens, and kings, with sliding attacks ob-

tained via magic bitboards masked by empty and enemy squares.

Legal moves reuse a precomputed `LegalityContext`: enemy sliders and pieces on the same rank, file, or diagonal as the king yield either a directed check (with a `check_mask` of squares that block or capture the checker) or a pin (per-source `pin_masks` intersected when multiple pinners apply). Knight and pawn checks are merged into the same mask semantics. A fast filter rejects destinations that violate check evasion or pin geometry; pinned knights cannot move. King moves and some pawn cases (en passant, promotions) still use `MakeMove` plus an in-check test on a board copy, because discovered attacks and ep special-cases are not fully captured by the bit mask alone. The legality precomputation turns most moves into cheap mask intersection and filtering, which is $O(1)$ per generated move aside from building the context once per node. The remaining cases (king moves, some pawn moves including en passant and certain promotions) fall back to a copy, `MakeMove`, and in-check detection because discovered check or ep is awkward to fold into a single set of static masks. That hybrid gives a predictable fast path in tactical positions (many check evasions filtered by masks) while still guaranteeing correctness. Worst-case cost rises where many moves need the slow path, but such nodes are a minority relative to nodes that fail fast in the bit filter.

4.2 Search Algorithm

4.2.1 Negamax, Alpha-Beta, and Iterative Deepening

The recursive search core is a *negamax* formulation of minimax [41], [42]. At each node the side to move chooses a legal move that maximizes the backed-up score, where every child search returns a value in the *child's* frame; the parent negates that return so both White and Black plies use the same comparison direction and one routine suffices (Equation (2) and Figure 2.12 in Section 2). Irida keeps scores in centipawns from White's perspective at the root, but the negamax layer only cares about the sign flip across plies.

Alpha-beta pruning bounds the search with parameters α and β in negamax coordinates at the current node [9], [43], [44]. Whenever a child proves a value $\leq \alpha$, remaining siblings cannot raise the node's value (alpha cutoff); whenever a child reaches $\geq \beta$, the parent can stop searching further siblings (beta cutoff). Pruning is correct because those branches cannot change the minimax outcome once $\alpha \geq \beta$.

The root driver implements *iterative deepening* [45]–[47]: the engine repeats a full-width search for increasing nominal depths, reusing transposition-table information from earlier iterations to improve move ordering and time management. For numerical stability with the underlying board representation, the root position is re-initialized from a FEN snapshot at the start of each depth iteration before searching again. In theory, with perfect move ordering, alpha–beta reduces the effective branching factor from b toward $\sqrt{b}$ in many models, so doubling nominal depth is far cheaper than the naive b^d count [43], [44]. In practice ordering is imperfect, so the gain is less than the ideal but still large enough that late improvements (TT, killers, history) have outsized impact. Iterative deepening recycles the TT and ordering hints from depth $d-1$ when starting depth d, so later iterations are not "from scratch" despite the re-search. Re-initialising the board from a FEN snapshot at each new nominal depth is a safety trade: it costs a few synchronous state writes but rules out hard-to-debug drift between iterations in long test runs and

tournament games.

4.2.2 Quiescence Search

When the main search reaches depth zero, evaluating the position statically would often stop in the middle of volatile tactical sequences (the *horizon effect* [48]). Irida therefore extends the search with *quiescence search* [49]: a secondary recursion that applies stand-pat logic using the same static evaluator as the main tree, then considers *capture-only* continuations. Captures are generated explicitly, ordered with the same ordering pipeline as the main search (without a transposition-table move hint), and searched with negated bounds in the usual alpha-beta style. Stand-pat returns immediately if the static score is at least β, a local *fail high* that proves the quiet position already refutes the current window without expanding captures; otherwise α may be raised before captures are tried. A maximum ply cap guards pathological capture chains; at that limit the routine returns a full static evaluation. Quiescence nodes are accounted for separately from main-tree nodes for diagnostics and time polling. The implementation deliberately extends only *captures* in quiescence, not all checks or all noisy quiet moves, which is a node-count trade: adding non-captures (even checks) would better resolve some tactical sequences but can explode the quiescence tree. Stand-pat with a static evaluation is the standard compromise: it allows a cutoff when a quiet position is already provably good enough ($\geq \beta$) before paying for more capture plies. The hard ply cap is a backstop against infinite or near-infinite capture–recapture lines; at the cap, returning a full static eval accepts a small risk of *horizon* error in pathological cases but keeps time bounds manageable.

4.2.3 Transposition Tables

Irida uses a *transposition table* (TT) keyed by the position hash [25], [26]. Each entry stores depth, score bound type (exact, lower, or upper), mate-distance, adjusted scores, the best move from the node when known, and a *generation* tag so entries from previous searches are not reused incorrectly. Probes may return a cutoff score only when the stored depth is at least the current remaining depth, but the stored best move is always exposed for ordering even when the score is too shallow to cut off. Replacement favors empty slots, updates to the same key, deeper results, or stale generations. A TT hit is a dual artifact: the score is only trusted to cut off the search when the stored search depth is at least the depth still *needed* at this node, otherwise a shallow value might mis-rank a quiet line (mate-distance and exact-value handling mitigate gross errors). The *best move* field is useful even from shallow entries, because it seeds search ordering: playing that move first improves alpha–beta's pruning rate even when the score is not deep enough to prune. The generation tag implements *aging*: entries from a previous search pass are not confused with the current pass, so replacement and probe semantics stay aligned with iterative-depth cycles. The replacement policy (prefer empty, same key, deeper, or stale) is a practical compromise between retaining deep results and letting the table reflect new subtrees; alternative schemes (always-replace depth, two-tier buckets) are mainly tuning choices.

4.2.4 Move Ordering

Move ordering combines several layers [50], [51]. Quiet and tactical moves receive a composite integer score and are sorted in descending order. The TT move is ranked highest when present. Captures use

an *MVV-LVA* (most valuable victim / least valuable attacker) table [52] with dedicated treatment for en-passant and promotion bonuses. Among non-captures, quiet queen promotions, then *killer moves* (two slots per ply: recent quiet cutoffs at the same depth), then a *history heuristic* indexed by side, from-square, and to-square. On beta cutoffs caused by quiet moves, Irida stores killers and increases history for the cutoff move while decreasing history for other quiet moves tried earlier at the same node, scaled by the square of the remaining depth and clamped to fixed limits. The order of heuristics mirrors how often they fire: the transposition best move, when present, is a near-optimal first try because it encodes the conclusion of a previous search of the same or parent position. *MVV-LVA* is cheap to compute and good at ranking obvious *tactical* tries among captures. *Killers* and *history* are complementary: killers are depth-local, whereas history summarises long-run success of (side, from, to) over many nodes. On quiet beta cutoffs, boosting the cutoff and damping other tried quiets (scaled by depth squared) concentrates credit on moves that *actually* caused a cutoff at this depth band.

4.2.5 Null Move Pruning

Null-move pruning [53], [54] passes the turn with a "null" ply when it is safe: not at the root, sufficient remaining depth, not in check, the side retains non-pawn material, and the position is not too sparse in pieces. A fast null-window child search uses a fixed reduction beyond the usual decrement. If the preliminary result does not justify a cutoff (including mate-score guards), no pruning occurs; otherwise a verification search at reduced but less aggressive depth can confirm the cutoff before returning a beta-bound score. Null-move pruning is *unsound* in zugzwang-heavy endgames where "passing" would be a legal win for the opponent; the non-pawn-material and piece-sparsity guards reduce the risk of pruning in positions where the side to move might *need* a null move to lose. The reduced-depth null search is a fast *signal*: if even that lazy line refutes the current β window, the real line is likely at least as strong. Mate-score checks avoid turning a spurious null score into a false mate claim. Optional verification (reduced, less aggressive than the null probe) is a second line of defence: it trades a few extra nodes to avoid a rare spurious beta cutoff that would be wrong in exceptional fortresses. Together, preconditions, reduction, and verification are the standard engineering response to the fact that NMP is a heuristic, not a theorem.

4.2.6 Late Move Reductions

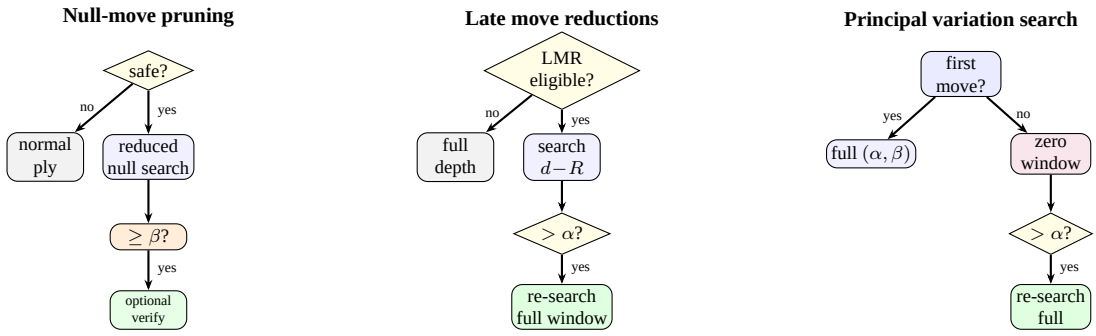
Late move reductions [55] shrink the search depth for late-ordered moves that are unlikely to be best: beyond a minimum depth and move index, on non-PV nodes, when the parent is not in check, the move is neither a capture nor a check-giver, and the move is not the TT move, Irida subtracts a reduction that grows with depth and move index, clamping so the child still receives at least one ply. If the reduced search returns a score above α, a full-depth re-search restores correctness. LMR formalises the observation that *after* good move ordering, late moves are rarely best. The reduction is therefore applied only when the move is neither obviously tactical (capture, check) nor the TT's favourite, and the parent is not in check (where every legal move can defend a narrow eval margin). A reduced search that still exceeds α is a *fail high* in spirit: the quiet move is better than a lazy search suggested, so a full-width or full-window follow-up re-establishes correctness. LMR is intentionally coupled to PVS: many late moves are first touched as zero-window scouts at reduced depth, so the cost of the reduction is paid only when the move orders badly or surprises the scout.

4.2.7 Aspiration Windows

From moderate depths onward, Irida narrows the root window around the previous iteration's score using an aspiration offset proportional to that score's magnitude [56]–[58]. Let v_{d-1} be the fully searched value at the previous nominal depth. The next iteration starts with a window $[\alpha, \beta]$ straddling v_{d-1}. If the true root value lies *below* α, the search stops with an outcome that is only an upper bound relative to the attempted window, a *fail low*. If it lies *above* β, the returned score is only a lower bound, a *fail high*. Either case means the narrow window missed the minimax value; Irida then widens the bounds multiplicatively and re-runs the same depth until the value lands inside the window or the guard triggers a fallback to the full-line window $(-\infty, +\infty)$ for robustness. Aspiration windows are most valuable when the root value changes only slowly between depths; they shrink the search space by causing quick cutoffs on fail-high or fail-low transpositions. When the tactical character of the position shifts (a sacrifice appears, a fortress cracks), the previous-iteration value can be a poor centre for the next window, so the search may fail low or high repeatedly until the window widens enough. Multiplicative widening is a simple policy that recovers from that instability; the fall-back to an infinite window guarantees termination of the *same* depth with a numerically complete minimax value under time control. Tuning the offset as a function of score magnitude addresses the observation that centipawn scale is not uniform across game phases: large material swings tolerate wider bands than dead-equal endings.

4.2.8 Principal Variation Search

Principal Variation Search (PVS) [59], [60] searches the first successfully played legal move (the PV candidate after ordering) with a full (α, β) window. Subsequent moves are probed with a *zero window* $(\alpha, \alpha+1)$ in negamax coordinates; only when a scout proves unexpectedly strong does Irida re-search with the full window. The implementation couples PVS with LMR: a reduced-depth scout that becomes interesting triggers verification at full depth, and zero-window re-search is guarded so that obvious fail-highs do not explode the node count. Figure 4.1 sketches the high-level control flow for these three mechanisms and how they interact at a node. PVS is the outer ordering of move trials: the first move after `TT`+killers+sorting is treated as the putative principal variation; it receives a full window so the true minimax value of that subtree is known. Siblings are probed with a null window; only a result strictly better than α (a surprise) forces a re-search with the parent's real window. LMR typically applies to those late siblings, so the "scout" is often a reduced-depth null-window pass first; a high score there triggers a full-depth confirmation, not an immediate wide window on every move. That stacking is what keeps node count near the practical optimum of alpha–beta while retaining soundness. The same structure applies in the quiescence search for capture ordering, though without a TT move hint the first capture is a weaker stand-in for a true PV.



(a) Pass turn when preconditions hold; a threatening null score may trigger verification before a cutoff.

(b) Quiet, late-ordered moves use a reduced depth; a strong reply forces a full-depth confirmation.

(c) After the PV candidate, scouts use a narrow window; failures require a full-window re-search.

Figure 4.1: Schematic control flow for three interacting search refinements (not every guard or mate score check is shown).

The three panels are not a strict left-to-right pipeline: null-move applies at most once per node before child generation (when its preconditions hold), whereas LMR and PVS apply *per move child*. A single child can be both LMR-reduced and probed with a zero window under PVS; a score that fails high for that stacked probe is what triggers full-depth or full-window re-search. NMP is therefore a node-level early exit, while LMR and PVS shape the cost of exploring the move list.

4.3 Hybrid Evaluation

4.3.1 Hand-Crafted Evaluation (HCE)

The primary hand-crafted evaluator follows the *PeSTO* paradigm [27]: separate midgame and endgame piece values and piece-square tables, accumulated per square for both colors. A discrete game phase in $[0, 24]$ (from material composition) *tapers* the midgame and endgame contributions into a single blended material-PST score. On top of this kernel, Irida adds structured heuristics, including pawn structure, mobility, king safety, piece activity, spatial pressure, tactical threats, specialized endgame handling, and rook activity, and combines them into a total score expressed in centipawns from the side to move's perspective. A separate material-only function exists for debugging and tooling, and the PeSTO tables can be retuned externally (e.g., Texel-style optimization [61]) by reloading tuned midgame/endgame piece weights. The discrete phase in $[0, 24]$ implements a *tapered eval*: material and piece–square tables contribute both a midgame and an endgame component, and the phase weighting (roughly, "more endgame phase as material comes off") blends them so that the same position does not use opening king safety heuristics in a pure pawn endgame. The additional terms (mobility, structure, threats, king safety, specialised endgames) are additive on top of that kernel; their relative scale is a design choice that Texel-style tuning can adjust by optimising against game databases. The important analysis point is *separation of concerns*: PeSTO provides a smooth material surface, while the hand-tuned terms inject strategic and tactical shape that pure PSTs under-express.

4.3.2 NNUE Integration

Irida can replace or augment classical evaluation with a *NNUE* network loaded at runtime from a binary `.nnue` file (UCI `EvalFile`) [11], [12], [14], [18]. Integration is implemented as a thin adapter around an external probe library: the board is exported to FEN, the network returns a score in centipawns for the side to move, and the engine negates appropriately when Black is to move. If no network is loaded, the FEN fails structural validation, or probing fails, evaluation transparently falls back to the PeSTO-based hand-crafted function, yielding a practical *hybrid* pipeline without duplicating search logic.

The deployed NNUE path evaluates the *full position* on each call through the probe rather than maintaining a custom incremental accumulator inside Irida; efficient NNUE inference and any low-level incremental updates are delegated to the linked probe implementation. Training and conversion tooling in the repository can produce compatible networks and optional diagnostics (e.g., evaluation breakdown logging remains centered on the classical evaluator). Exporting a full FEN and invoking the external probe on every evaluation call is simple and keeps Irida's core free of accumulator state, but it implies per-node string formatting and a full forward pass through the network (handled inside the library) instead of incremental feature updates as pieces move, as in high-performance Stockfish-style integrators. The trade is implementation complexity versus raw nodes per second: the present path favours a maintainable adapter and correct side-to-move semantics at the cost of higher per-eval overhead. The binary `.nnue` layout is that of the loaded probe (typically UCI-compatible `EvalFile` networks in the `nnue` training ecosystem); the engine does not re-implement the format, it passes bytes to the library, so any format discussion belongs with the probe or training tooling documentation. When loading fails, transparent fallback to HCE keeps search and time control well-defined for development and for users who omit a network file.

Chapter 5: Experimental Methodology

5.1 Test Environment

To ensure the reproducibility of the results, all experiments were conducted in a controlled environment. The hardware and software specifications are as follows:

- **CPU:** AMD Ryzen 5 4000
- **RAM:** 24GB DDR4
- **OS:** Void Linux
- **Compiler:** GCC version 14.2.1 with -O3 and -march=native flags
- **Engine Settings:** Hash size was fixed at 64MB, and all tests were conducted on a single thread to eliminate SMP (Symmetric Multiprocessing) variance.

5.2 Performance Metrics

Reported **nodes per second (NPS)** and *search depth* (iterative deepening under a wall-clock or depth limit) summarise engine throughput and foresight; Chapter 6.2 interprets them alongside the measurements in this thesis. Fine-grained *empirical* tree growth on the starting position, including the geometric effective branching factor $\mathrm{EBF}_{\mathrm{geom}}$, incremental work between deepening steps, and the single-number proxy $\bar{b} = N^{1/D}$, is defined operationally in Section 6.7; those diagnostics quantify pruning efficiency for the staged binaries there and should not be confused with informal uses of "branching factor" elsewhere.

5.3 Competitive Evaluation

5.3.1 Elo

The **Elo** rating system, developed by Arpad Elo, was introduced to replace the Harkness system used in zero-sum games. Although originally designed to rate human chess players, it is now widely used to evaluate chess engines as well. Elo is a relative rating system: the difference in rating between two players determines the expected outcome of a match. When a player wins, their rating increases, while the opponent's rating decreases. In the case of a draw, rating changes are small, especially when the players have similar ratings [62].

The expected score of a player is given by:

$$E = \frac{1}{1 + 10^{-D/400}}$$

where D is the rating difference between the two players. For example, a player rated 100 Elo points higher than their opponent is expected to win approximately 64% of the time.

5.3.2 SPRT

Statistical significance was evaluated using the **Sequential Probability Ratio Test (SPRT)** [63], [64], a method introduced by Abraham Wald for sequential hypothesis testing. Unlike fixed-sample tests, SPRT evaluates data as it is collected and allows early termination once sufficient evidence has been accumulated.

The test compares two competing hypotheses: the null hypothesis H_0, typically representing no performance difference, and the alternative hypothesis H_1, representing a meaningful improvement. After each observation, a Log-Likelihood Ratio (LLR) is computed, measuring how much more likely the observed results are under H_1 than under H_0.

The LLR is compared against two decision thresholds. If it exceeds the upper bound, H_1 is accepted; if it falls below the lower bound, H_0 is accepted. Otherwise, testing continues. These thresholds are determined by predefined Type I and Type II error probabilities, controlling the risk of incorrect conclusions.

In the context of chess engine evaluation, game outcomes are typically modeled using probabilistic frameworks such as the Elo system, where the expected score depends on the rating difference between players. Because match results are inherently noisy, SPRT provides an efficient and statistically grounded way to determine whether observed performance differences are significant.

More generally, parameter optimisation in chess engines can be viewed as a noisy black-box optimisation problem, where improvements must be validated under uncertainty.

5.4 Head-to-Head Match Protocol

Strength comparisons between staged *Irida* builds use paired games with balanced openings unless a later chapter specifies otherwise. Resource limits follow Section 5.1: 64 MB hash and one thread. Each ablation pairing in Chapter 6 uses the game counts stated in each ablation table (movetime control: $5{,}000$ games per pairing, except $2{,}180$ games for the aspiration-vs.-PVS pairing; fixed-depth control: 100, $1{,}000$, or $2{,}030$ games depending on the stage). Those figures are the *realised* totals from each run: significance is assessed online with SPRT (Section 5.3.2), so a pairing can *stop early* as soon as the log-likelihood ratio crosses an acceptance or rejection boundary, without playing out a larger nominal schedule (e.g. $10{,}000$ games) to the end whenever the evidence is already conclusive. Two scheduling regimes are observed in parallel: a **fixed per-move time** of 0.1 s, and a **fixed nominal iterative-deepening depth** of 4 main-line plies (both sides stopped at the same outer depth). Outcomes are mapped to logistic ΔElo with confidence intervals; sequential testing uses SPRT and reported LLR values (Section 5.3). Other competitive matches, notably the reference games against an external engine in Section 6.8, use the opponent, sample size, and controls stated there instead of this template.

Chapter 6: Results and Analysis

6.1 Move Generation Benchmarking

Move-generation correctness was checked with *perft* counts (legal move enumeration to fixed depth) [65], [66]. The pseudo-legal version of the Perft test is shown in Appendix D.

As demonstrated by the benchmarks in Table 6.1, the move generator delivers competitive performance, exceeding two million nodes per second (MN/s) across all tested positions. Notably, in the notoriously complex ”Kiwipete” position, the engine achieves a throughput of 6.15 MN/s for depth 3.

Table 6.1: Move Generation and Lifecycle Performance Benchmarks

Test Position	Nodes/Cycles	Time (s)	MN/s (MCy/s)
Perft Verification			
Start Pos (d4)	197 281	0.085	2.33
Start Pos (d5)	4 865 609	1.814	2.68
Kiwipete (d3)	97 862	0.016	6.15
Kiwipete (d4)	4 085 603	0.747	5.47
Endgame (d5)	674 624	0.221	3.06
Pos5 (d4)	2 103 487	0.484	4.35
Movegen Throughput			
Starting Pos	4 000 000	1.912	2.09
Kiwipete	9 600 000	1.560	6.15
Middlegame	7 200 000	1.741	4.14
Lifecycle Throughput (MCy/s)			
Starting Pos	10 000 000	7.158	1.40
Kiwipete	24 000 000	16.418	1.46
Middlegame	18 000 000	13.457	1.34

All positions used above are listed in Appendix B.

6.2 Search Dynamics: NPS, Depth, and Win Probability

Section 5.2 summarises how throughput and depth are reported; here they are interpreted for reading the experiments below. Three quantities recur throughout this chapter: **nodes per second** (NPS), **reached depth** (iterative deepening), and **match score** (wins, draws, losses, and derived Elo). They are not independent: together they describe how raw hardware effort translates into playing strength.

Nodes per second. NPS is the rate at which the search visits positions, including quiescence and all pruning paths. It is always lower than the move-generation figures in Table 6.1 because every node pays for evaluation, `MakeMove`/`UnmakeMove`, transposition-table probes, and heuristic overhead (see the lifecycle rows in the same table). Reported “info” NPS therefore reflects the *whole* search stack, not a single kernel. Comparing NPS across binaries is only meaningful when hash size, thread count, and position are fixed, as in the EBF harness (Section 6.7).

Search depth under time and under depth limits. With a **per-move time cap** (Section 6.3), iterative deepening increases the nominal depth until the clock stops the search; a faster or more frugal build reaches a *higher* completed depth on average, which usually correlates with strength. Heuristics that add occasional re-searches (aspiration, PVS) can lower effective depth in the same wall time even when they reduce nodes at a *given* fixed depth, the tradeoff analysed in Section 7.1.1. With a **fixed depth** control, both sides search the same horizon, so differences in NPS do not change the plies; they only change wall time and CPU use.

Win probability and Elo. Ablation tables report logistic ΔElo and confidence intervals, with SPRT and LLR as in Section 5.3. A positive estimate for E_2 means the enhanced build scores a higher *expected* tournament score against E_1 under the stated protocol, not a guarantee in every game: short ties and opening noise remain, which is why intervals and sample sizes matter. Across the two time-control regimes, the same feature can in principle help under one schedule and hurt under the other, which is why the figures in Section 6.5 disaggregate results rather than reporting a single win-rate for the whole experiment.

In summary, NPS measures *computational effort*, search depth (under time) measures *foresight* available before the move must be sent, and Elo measures *outcomes* in self-play. Strength gains require that improvements in the tree (fewer nodes, better ordering, sounder pruning) not come at the cost of so much per-node work or re-search that the engine loses its depth advantage under the time control that matters for the experiment.

6.3 Performance Evaluation: Benchmarking Across Time Controls

The micro-benchmarks in Section 6.1 characterise *local* throughput (legal move generation and make/unmake). Competitive evaluation, by contrast, requires a **time model**: a rule for how long the engine may think per move, or an alternative schedule that fixes search depth instead of wall clock. Those choices determine which aspect of engine quality is measured, roughly, *strength per fixed thinking time* versus *strength at a fixed forward horizon*.

Ablation matches (Section 6.5) use the dual schedules of Section 5.4: a **fixed per-move time** of 0.1 s, and a **fixed nominal search depth** of 4 plies. The first asks which build converts a short real-time budget into better decisions; the second asks which build is stronger when both sides are allowed to search the *same* number of main-line plies, regardless of how long that takes. Neither mode is "more correct" in isolation: the time cap stresses scheduling, iterative deepening, and overhead from heuristics that trigger re-searches; the depth cap removes that clock pressure and isolates search *quality* at a fixed horizon, at the cost of ignoring tournament-realistic time management.

The lifecycle rows of Table 6.1 already hint at the gap between raw movegen speed and end-to-end search cost: throughput drops from several MN/s in the pure move-generation trials to on the order of 1–1.5 M cycles/s when the full board update path is exercised. The ablation and EBF analyses that follow should therefore be read with that hierarchy in mind: optimisations that look decisive at fixed depth can still interact badly with a tight wall clock, as quantified for aspiration and PVS in Section 7.1.1.

6.4 Ablations: Quantifying Feature Impact

Section 5.4 fixes hardware limits, sample size, openings, the dual schedules (movetime and fixed nominal depth), and the statistical framework (Section 5.3). Each row below is a **head-to-head** between two staged builds of *Irida* in the same order as Table 6.16: baseline E_1, candidate E_2 adds exactly one mechanism. Pooled ΔElo, intervals, and SPRT outcomes appear in the *tables*; *figures* split outcomes by schedule so time vs. depth effects are visible without treating sub-panels as separate tests.

6.5 Fixed Movetime Ablation Studies

The first head-to-head uses alpha–beta with iterative deepening and move ordering as E_1; each subsequent row compares two *adjacent* stages, so later E_1 builds already contain all earlier accepted mechanisms. Results are summarised in Table 6.2 and then grouped as (i) extensions and memory, (ii) aggressive pruning and reductions, and (iii) re-search and root windows. This organisation highlights a practical pattern under the 0.1 s budget: early search stabilisation features produce the largest gains, while late-stage windowing features become sensitive to clock overhead.

Table 6.2: Ablation chain: pooled logical Elo advantage of E_2 and SPRT outcome. Negative Δ favours E_1.

Added on E_1 ($\rightarrow E_2$)	**ΔElo**	**SPRT**
Quiescence	$+172.5$	H_1 accepted
Transposition table	$+182.2$	H_1 accepted
Null move pruning	$+26.5$	H_1 accepted
LMR	$+136.1$	H_1 accepted
Aspiration	-7.2	H_0 accepted
PVS	-5.5	H_0 accepted

6.5.1 Extensions and memory: quiescence and the transposition table

Table 6.3: Ablation Results: Base Engine (E_1) vs. Quiescence Search (E_2)

Metric	**Value**
E_1 (baseline)	Base
E_2 (candidate)	Quiescence
Total games (paired)	5000 (2500 pairs)
Outcome distribution*	[192, 242, 1548, 817, 2201]
Mean score per game*	0.7297
Elo difference Δ (E_2 vs. E_1, logistic)	+172.48
95% confidence interval for Δ	[+164.81, +180.14]
Log-likelihood ratio (LLR)	101.4659
SPRT conclusion	H_1 accepted

**Outcomes are five frequency counts for weighted scores* $0, 0.5, 1, 1.5, 2$ *(tournament points).*

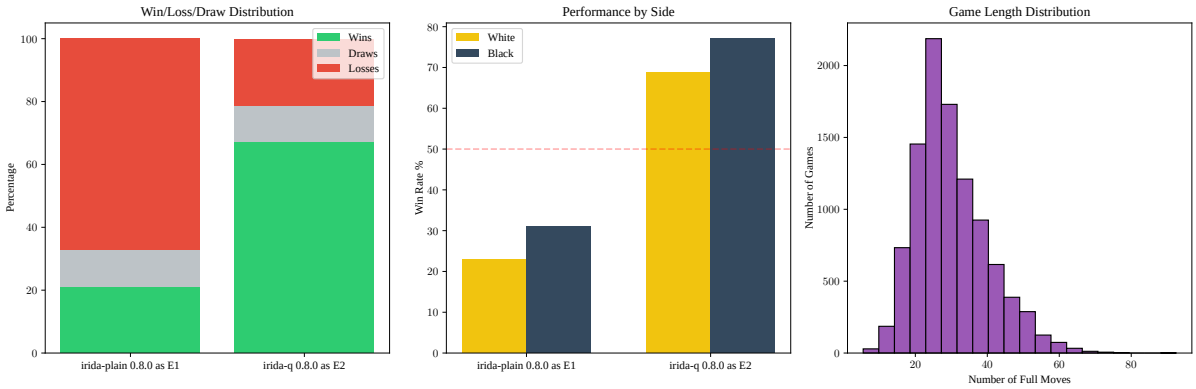


Figure 6.1: Performance impact of adding Quiescence Search to the baseline.

Quiescence search. Table 6.3 shows the largest single jump in the chain ($\Delta\text{Elo} = +172.48$, H_1 accepted): extending tactical lines is decisive relative to the plain horizon-limited search. Figure 6.1 indicates that this advantage is not a narrow artifact of one panel; instead, it reflects a broad reduction in tactical blunders that the fixed horizon baseline cannot repair in time.

Table 6.4: Ablation Results: Quiescence Search (E_1) vs. Transposition Table (E_2)

Metric	**Value**
E_1 (baseline)	Q
E_2 (candidate)	TT
Total games (paired)	5000 (2500 pairs)
Outcome distribution*	[85, 505, 1155, 1025, 2230]
Mean score per game*	0.7405
Elo difference Δ (E_2 vs. E_1, logistic)	+182.16
95% confidence interval for Δ	[+164.79, +199.52]
Log-likelihood ratio (LLR)	22.1268
SPRT conclusion	H_1 accepted

**Outcomes are five frequency counts for weighted scores* $0, 0.5, 1, 1.5, 2$ *(tournament points).*

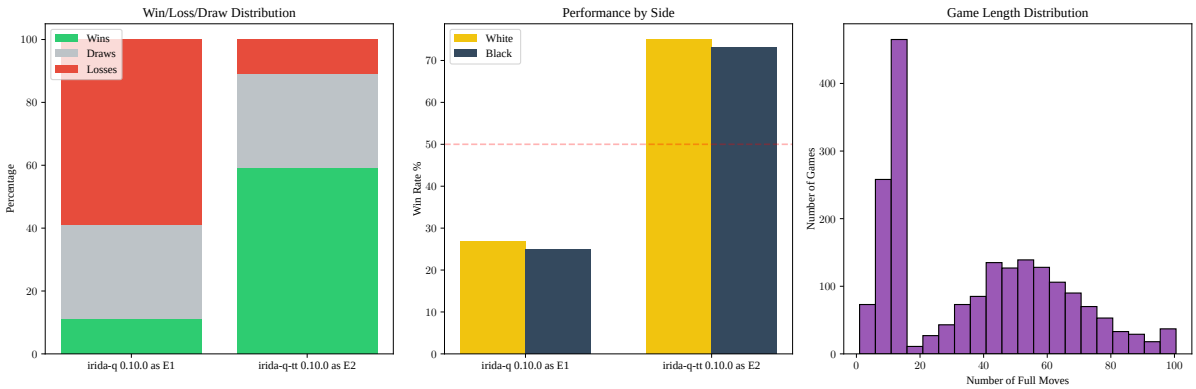


Figure 6.2: Comparative gain of integrating Transposition Tables.

Transposition table. The TT pairing (Table 6.4) produces another large gain ($\Delta\text{Elo} = +182.16$, H_1 accepted), confirming that transposition reuse is both an efficiency and *decision-quality* improvement in this setup. Figure 6.2 is consistent with the interpretation that TT hits convert directly into extra effective depth under tight per-move timing.

6.5.2 Pruning and reductions: null move and LMR

The next two rows add *selective* depth reduction: null move pruning discards likely non-critical branches with a pass-move test, while LMR reduces depth for late quiet moves. Both target tree-size control, but with different risk profiles. In this chain, both pooled rows favour E_2; NMP is a modest gain, while LMR is a major jump, implying that move-order-dependent selective reduction contributes more than the standalone null-move step at this control.

Table 6.5: Ablation Results: Transposition Tables (E_1) vs. Null Move Pruning (E_2)

Metric	**Value**
E_1 (baseline)	TT
E_2 (candidate)	NMP
Total games (paired)	5000 (2500 pairs)
Outcome distribution*	[714, 152, 2860, 208, 1066]
Mean score per game*	0.5380
Elo difference Δ (E_2 vs. E_1, logistic)	+26.46
95% confidence interval for Δ	[+19.63, +33.29]
Log-likelihood ratio (LLR)	13.4445
SPRT conclusion	H_1 accepted

**Outcomes are five frequency counts for weighted scores 0, 0.5, 1, 1.5, 2 (tournament points).*

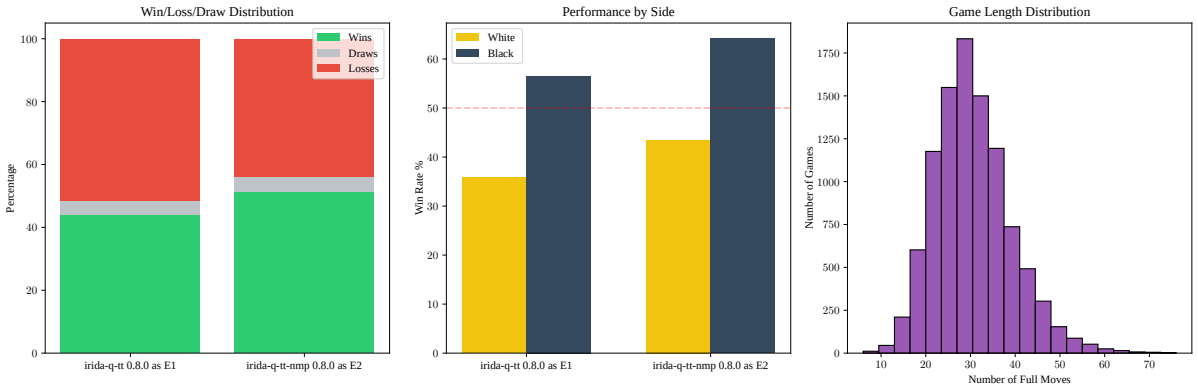


Figure 6.3: Evaluation of NMP performance against a TT baseline.

Null move pruning. NMP (Table 6.5, Figure 6.3) yields a clear but smaller increase ($\Delta\text{Elo} = +26.46$, H_1 accepted). The direction matches the EBF diagnostics in Section 6.7: pruning lowers search work, but its standalone strength effect is limited compared to quiescence, TT, or LMR in this staged pipeline.

Table 6.6: Ablation Results: NMP (E_1) vs. LMR (E_2)

Metric	**Value**
E_1 (baseline)	NMP
E_2 (candidate)	LMR
Total games (paired)	5000 (2500 pairs)
Outcome distribution*	[346, 148, 2064, 315, 2127]
Mean score per game*	0.5310
Elo difference Δ (E_2 vs. E_1, logistic)	+136.12
95% confidence interval for Δ	[+128.78, +143.46]
Log-likelihood ratio (LLR)	69.4826
SPRT conclusion	H_1 accepted

**Outcomes are five frequency counts for weighted scores* $0, 0.5, 1, 1.5, 2$ *(tournament points).*

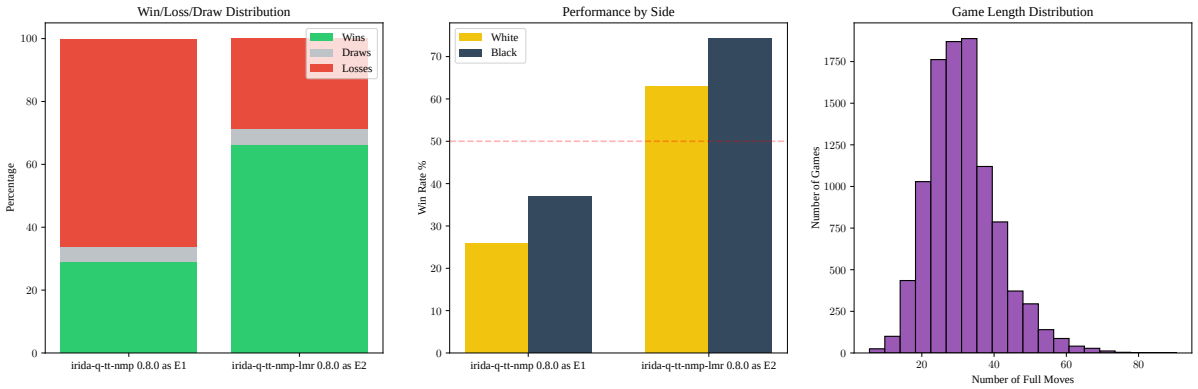


Figure 6.4: Analysis of LMR efficiency compared to NMP.

Late move reductions. LMR (Table 6.6, Figure 6.4) is one of the largest gains in the full sequence ($\Delta\text{Elo} = +136.12$, H_1 accepted), showing that deeper integration of ordering-driven reductions is highly productive in short-time play. The final two rows then test whether extra window-management logic adds value *on top* of this already selective search core.

6.5.3 Re-search and root windows: aspiration and PVS

Root aspiration narrows the score band before full-width re-search; PVS uses null-window scouts and full re-searches on failure. In theory, both reduce nodes at a fixed depth. In practice, both also introduce conditional re-search paths, so under a strict wall clock they can consume iteration budget even if they are asymptotically efficient. Consistent with that trade-off, pooled Elo in Tables 6.7–6.8 is no longer favourable to E_2.

Table 6.7: Ablation Results: LMR (E_1) vs. Aspiration (E_2)

Metric	**Value**
E_1 (baseline)	LMR
E_2 (candidate)	AW
Total games (paired)	5000 (2500 pairs)
Outcome distribution*	[983, 306, 2547, 262, 902]
Mean score per game*	0.4897
Elo difference Δ (E_2 vs. E_1, logistic)	-7.16
95% confidence interval for Δ	[-13.97, -0.35]
Log-likelihood ratio (LLR)	-4.9383
SPRT conclusion	H_0 accepted

**Outcomes are five frequency counts for weighted scores* $0, 0.5, 1, 1.5, 2$ *(tournament points).*

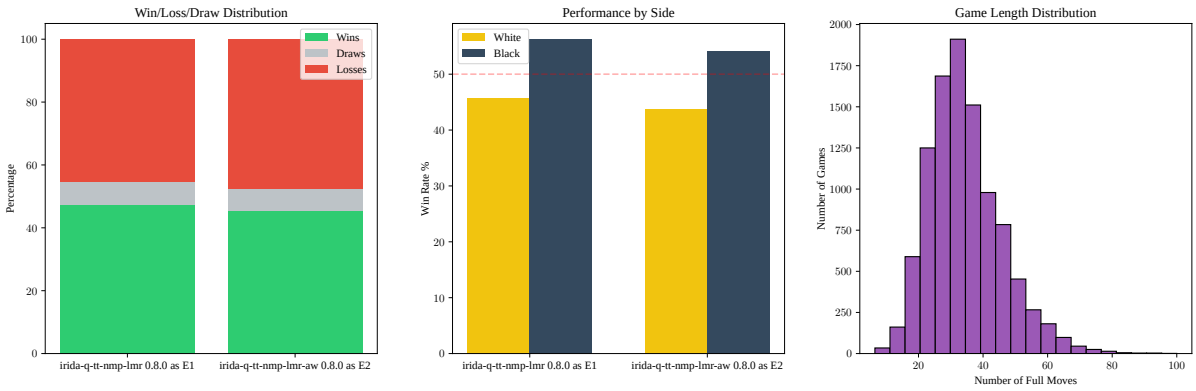


Figure 6.5: Performance delta: Aspiration Windows vs. LMR.

Aspiration windows. The aspiration row (Table 6.7) is the clearest negative result in the movetime chain ($\Delta\text{Elo} = -7.16$, CI entirely below zero, H_0 accepted): at 0.1 s, the extra fail-high/fail-low recovery cost outweighs nominal windowing savings. Figure 6.5 aligns with this clock-driven interpretation and motivates the fixed-depth cross-check in Section 6.6.

Table 6.8: Ablation Results: Aspiration (E_1) vs. PVS (E_2)

Metric	**Value**
E_1 (baseline)	AW
E_2 (candidate)	PVS
Total games (paired)	2180 (1090 pairs)
Outcome distribution*	[442, 138, 1064, 119, 417]
Mean score per game*	0.4921
Elo difference Δ (E_2 vs. E_1, logistic)	-5.50
95% confidence interval for Δ	[-15.81, +4.82]
Log-likelihood ratio (LLR)	-1.7062
SPRT conclusion	H_0 accepted

**Outcomes are five frequency counts for weighted scores* $0, 0.5, 1, 1.5, 2$ *(tournament points).*

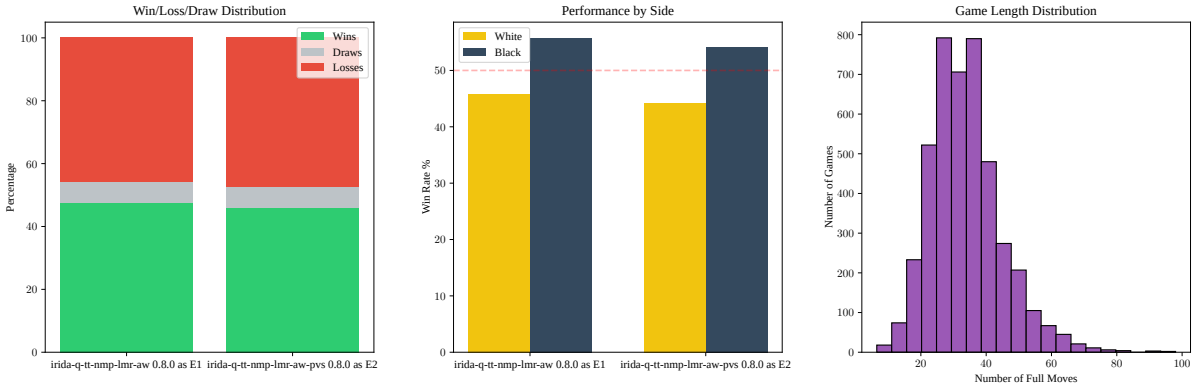


Figure 6.6: Comparison of Principal Variation Search (PVS) against Aspiration Windows.

Principal Variation search. For PVS, Table 6.8 reports a small negative estimate ($\Delta\text{Elo} = -5.50$) with a CI crossing zero and H_0 accepted; this is best read as *non-confirmed benefit* under this short-time protocol rather than a definitive degradation. Figure 6.6 suggests the same timing sensitivity seen for aspiration, and Section 7.1.1 connects that behaviour to re-search overhead and ordering quality.

6.6 Fixed Depth Ablation Studies

This subsection repeats the same staged chain as Section 6.5, but under a **fixed nominal depth** constraint only (4 main-line plies). Because both engines must finish the same principal horizon, these results reduce the role of wall-clock scheduling and isolate quality/efficiency at equal nominal depth. Pooled summaries appear in Table 6.9; per-pairing detail is in Tables 6.10–6.15 and Figures 6.7–6.12.

Table 6.9: Ablation chain (fixed nominal depth): pooled logical Elo advantage of E_2 and SPRT outcome. Negative Δ favours E_1.

Added on E_1 ($\rightarrow E_2$)	**ΔElo**	**SPRT**
Quiescence	$+798.25$	H_1 accepted
Transposition table	-3.82	H_0 accepted
Null move pruning	-4.52	H_0 accepted
LMR	-19.48	H_0 accepted
Aspiration	$+2.43$	H_1 accepted
PVS	$+3.01$	H_1 accepted

6.6.1 Extensions and memory: quiescence and the transposition table

Table 6.10: Ablation Results (fixed nominal depth 4): Base Engine (E_1) vs. Quiescence Search (E_2)

Metric	**Value**
E_1 (baseline)	Base
E_2 (candidate)	Quiescence
Total games (paired)	100 (50 pairs)
Outcome distribution*	[0, 0, 0, 4, 96]
Mean score per game*	0.9900
Elo difference Δ (E_2 vs. E_1, logistic)	+798.25
95% confidence interval for Δ	[+556.28, +1040.23]
Log-likelihood ratio (LLR)	145.8211
SPRT conclusion	H_1 accepted

**Outcomes are five frequency counts for weighted scores $0, 0.5, 1, 1.5, 2$ (tournament points).*

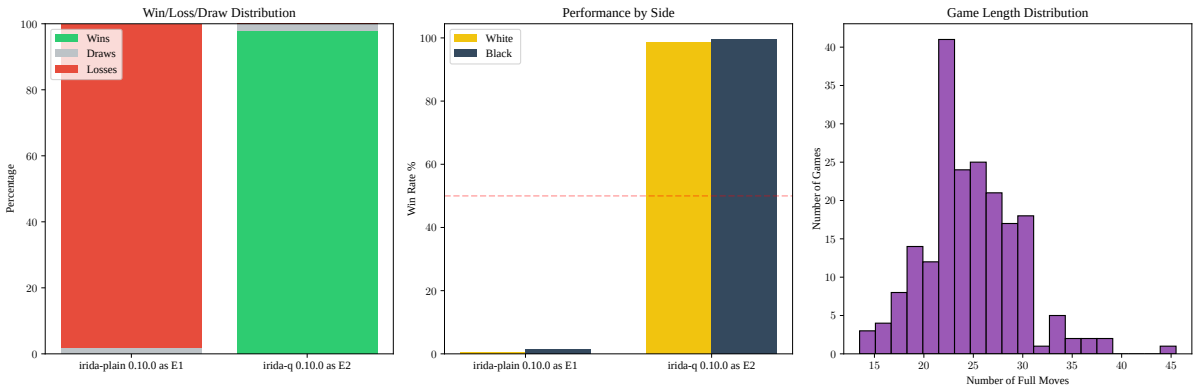


Figure 6.7: Fixed-depth control: performance impact of adding Quiescence Search to the baseline.

Quiescence search. Quiescence remains overwhelmingly favourable at fixed depth (Table 6.10, H_1 accepted). This confirms that its benefit is not merely "more plies under time", but higher tactical stability even when both engines are constrained to the same nominal horizon.

Table 6.11: Ablation Results (fixed nominal depth 4): Quiescence Search (E_1) vs. Transposition Table (E_2)

Metric	**Value**
E_1 (baseline)	Q
E_2 (candidate)	TT
Total games (paired)	1000 (500 pairs)
Outcome distribution*	[3, 81, 854, 59, 3]
Mean score per game*	0.4945
Elo difference Δ (E_2 vs. E_1, logistic)	-3.82
95% confidence interval for Δ	[-19.05, +11.41]
Log-likelihood ratio (LLR)	-6.4050
SPRT conclusion	H_0 accepted

**Outcomes are five frequency counts for weighted scores 0, 0.5, 1, 1.5, 2 (tournament points).*

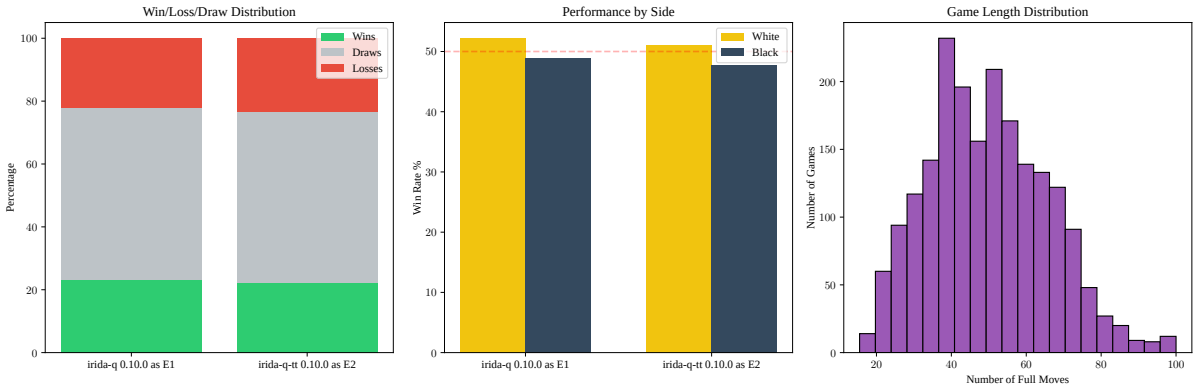


Figure 6.8: Fixed-depth control: comparative gain of integrating transposition tables.

Transposition table. By contrast, the TT row is statistically neutral in this control (Table 6.11, CI spanning zero, H_0 accepted). Relative to the strong positive movetime result, this suggests that a meaningful share of TT's practical value in this project comes from time-budget conversion (more complete iterations), not only from fixed-horizon move quality.

6.6.2 Pruning and reductions: null move and LMR

Table 6.12: Ablation Results (fixed nominal depth 4): Transposition Tables (E_1) vs. Null Move Pruning (E_2)

Metric	Value
E_1 (baseline)	TT
E_2 (candidate)	NMP
Total games (paired)	1000 (500 pairs)
Outcome distribution*	[14, 140, 721, 108, 17]
Mean score per game*	0.4935
Elo difference Δ (E_2 vs. E_1, logistic)	-4.52
95% confidence interval for Δ	[-19.75, +10.71]
Log-likelihood ratio (LLR)	-3.1305
SPRT conclusion	H_0 accepted

**Outcomes are five frequency counts for weighted scores 0, 0.5, 1, 1.5, 2 (tournament points).*

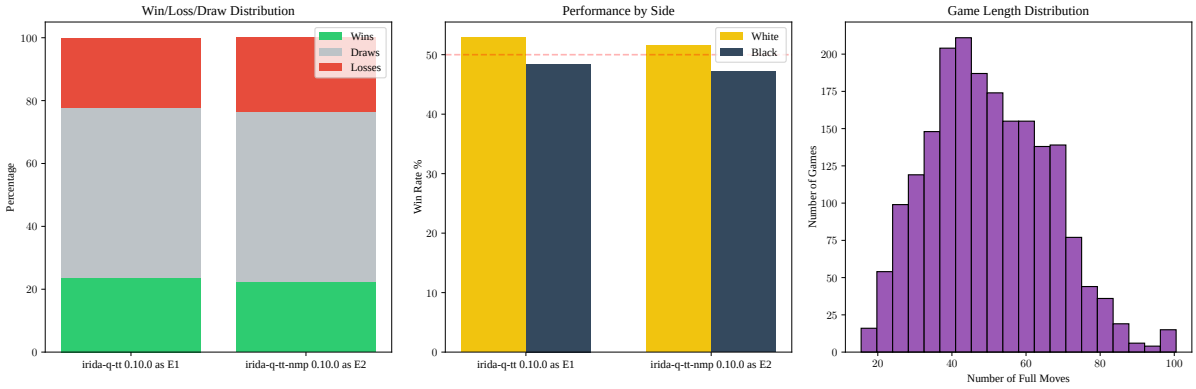


Figure 6.9: Fixed-depth control: NMP performance against a TT baseline.

Null move pruning. NMP is likewise inconclusive at fixed depth (Table 6.12, H_0 accepted). The positive movetime gain therefore appears schedule-sensitive: NMP helps when saved nodes can be reinvested into extra completed iterations, but yields little measurable advantage when both sides are locked to the same nominal depth.

Table 6.13: Ablation Results (fixed nominal depth 4): NMP (E_1) vs. LMR (E_2)

Metric	**Value**
E_1 (baseline)	NMP
E_2 (candidate)	LMR
Total games (paired)	1000 (500 pairs)
Outcome distribution*	[66, 275, 416, 191, 52]
Mean score per game*	0.4720
Elo difference Δ (E_2 vs. E_1, logistic)	-19.48
95% confidence interval for Δ	[-34.73, -4.23]
Log-likelihood ratio (LLR)	-3.9305
SPRT conclusion	H_0 accepted

**Outcomes are five frequency counts for weighted scores* $0, 0.5, 1, 1.5, 2$ *(tournament points).*

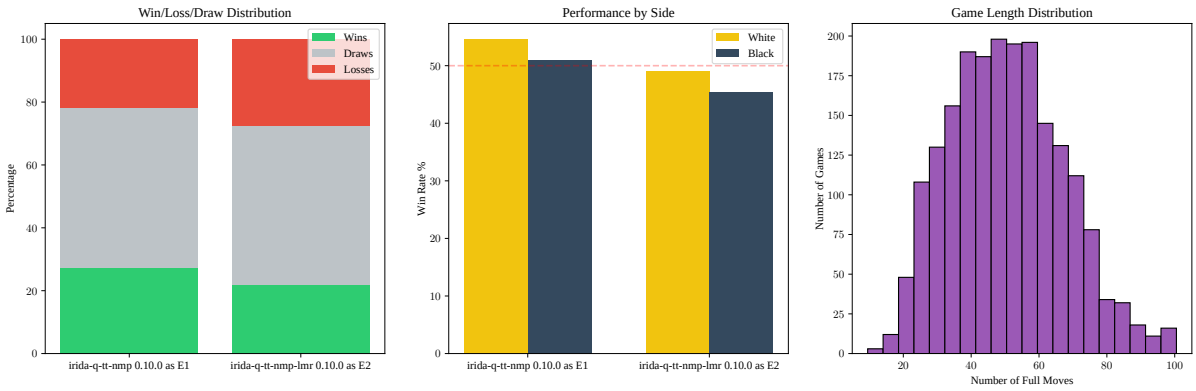


Figure 6.10: Fixed-depth control: LMR efficiency compared to NMP.

Late move reductions. LMR is the notable exception in this middle block: the estimate is negative for E_2 and the interval excludes zero (Table 6.13, H_0 accepted). Together with the strong positive movetime row, this indicates a trade-off: aggressive reductions are excellent for short-time throughput, but can sacrifice fixed-horizon quality when no extra depth can be "bought" from the saved work.

6.6.3 Re-search and root windows: aspiration and PVS

Table 6.14: Ablation Results (fixed nominal depth 4): LMR (E_1) vs. Aspiration (E_2)

Metric	**Value**
E_1 (baseline)	LMR
E_2 (candidate)	AW
Total games (paired)	1000 (500 pairs)
Outcome distribution*	[33, 203, 529, 187, 48]
Mean score per game*	0.5035
Elo difference Δ (E_2 vs. E_1, logistic)	+2.43
95% confidence interval for Δ	[-12.80, +17.66]
Log-likelihood ratio (LLR)	-0.0157
SPRT conclusion	H_1 accepted

**Outcomes are five frequency counts for weighted scores 0, 0.5, 1, 1.5, 2 (tournament points).*

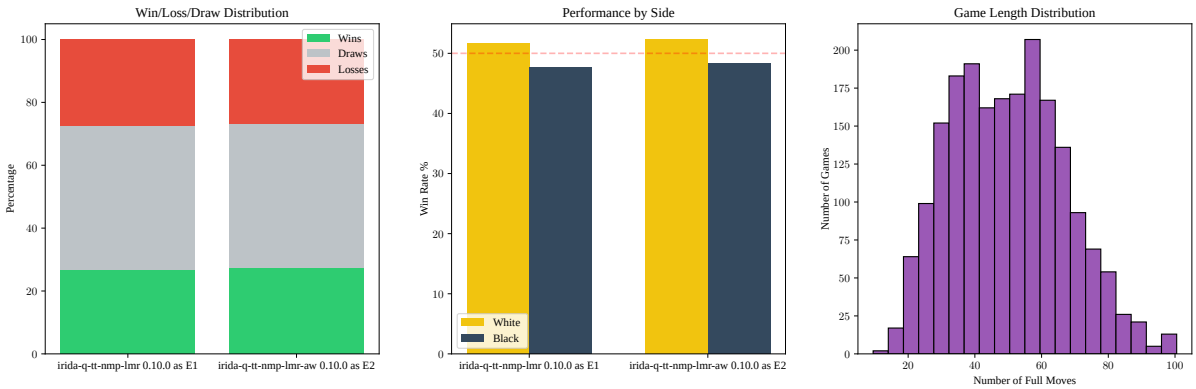


Figure 6.11: Fixed-depth control: aspiration windows vs. LMR.

Aspiration windows. Aspiration is close to neutral at fixed depth (Table 6.14): the point estimate is slightly positive, but uncertainty is wide. Compared with the clear movetime regression, this supports the interpretation that aspiration's main failure mode here is *clock overhead* from re-searches rather than a systematic fixed-horizon search defect.

Table 6.15: Ablation Results (fixed nominal depth 4): Aspiration (E_1) vs. PVS (E_2)

Metric	**Value**
E_1 (baseline)	AW
E_2 (candidate)	PVS
Total games (paired)	2030 (1015 pairs)
Outcome distribution*	[87, 358, 1098, 412, 75]
Mean score per game*	0.5043
Elo difference Δ (E_2 vs. E_1, logistic)	+3.01
95% confidence interval for Δ	[-7.80, +13.75]
Log-likelihood ratio (LLR)	+2.10
SPRT conclusion	H_1 accepted

**Outcomes are five frequency counts for weighted scores* $0, 0.5, 1, 1.5, 2$ *(tournament points).*

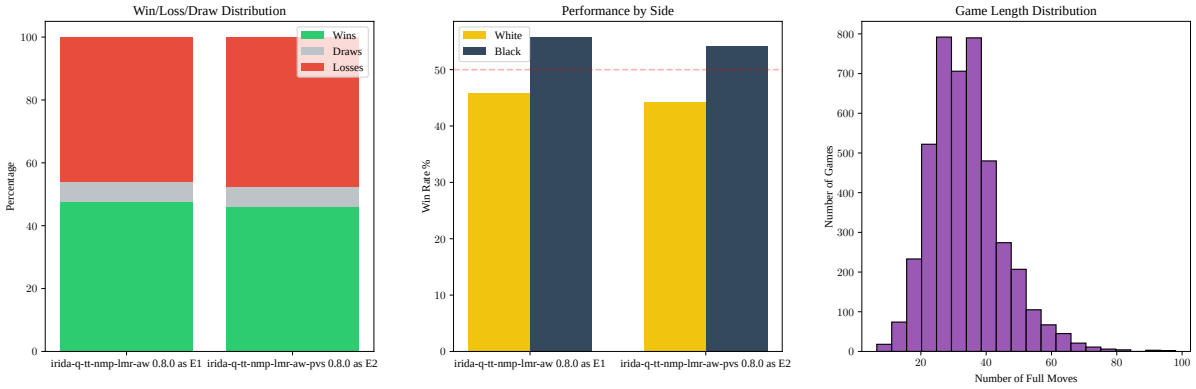


Figure 6.12: Fixed-depth control: principal variation search (PVS) vs. aspiration windows.

Principal Variation search. PVS follows the same pattern as aspiration: near-neutral to slightly positive at fixed depth (Table 6.15), but not decisively separated from zero. In combination with the movetime row, the practical conclusion is that late-stage windowing benefits are contingent on robust move ordering and enough time per move to absorb occasional re-search bursts.

6.7 Empirical Analysis of Search Growth and Branching Metrics

This subsection evaluates the impact of incremental search optimizations on the *Irida* engine's search efficiency. By measuring the effective branching factor (EBF) and node growth across staged binary builds, we quantify the reduction in the state-space search achieved by specific heuristic implementations.

6.7.1 Methodology and Metrics

The evaluation was conducted using a standardized automated harness on the engine's starting position (`startpos`). To isolate the effects of search heuristics, the transposition table (TT) size was fixed at 16 MB, and the search depth was capped at $D = 8$. Nodes are sampled from the UCI `info` stream, representing cumulative counts inclusive of quiescence search nodes.

To characterize tree growth, we define the following metrics:

- **Incremental Work (Δ_d):** The number of nodes explored at depth d that were not explored during the previous iterative deepening step, defined as $\Delta_d = N_d - N_{d-1}$.
- **Growth Ratio (R_d):** The ratio of incremental work between successive depths, $R_d = \Delta_d / \Delta_{d-1}$.
- **Effective Branching Factor (EBF_{geom}):** The geometric mean of R_d for $d \in \{2, \ldots, D\}$. This provides a measure of the multiplicative growth rate of the search tree.
- **Average Branching Factor ($\bar{b}$):** A simplified metric calculated as $N^{1/D}$, providing a global approximation of the tree's expansion.

It is critical to distinguish these empirical diagnostics from theoretical branching factors. Heuristics such as Late Move Reductions (LMR) and Alpha-Beta pruning dynamically reshape the search tree, meaning these values reflect the efficiency of the implementation rather than the absolute complexity of the game state.

6.7.2 Experimental Setup: Build Configurations

A feature matrix was constructed to isolate the contributions of specific search algorithms. Each binary build, summarized in Table 6.16, represents a cumulative addition of features following the standard *Irida* optimization path.

Table 6.16: Feature composition of evaluated engine binaries.

Binary Configuration	Quiescence	TT	NMP	LMR	Asp.	PVS	Syzygy
`irida` (Default)	✓	✓	✓	✓	✓	—	✓
`irida-plain`	—	—	—	—	—	—	—
`irida-q`	✓	—	—	—	—	—	—
`irida-q-tt`	✓	✓	—	—	—	—	—
`irida-q-tt-nmp`	✓	✓	✓	—	—	—	—
`irida-q-tt-nmp-lmr`	✓	✓	✓	✓	—	—	—
`irida-q-tt-nmp-lmr-aw`	✓	✓	✓	✓	✓	—	—
`irida-q-tt-nmp-lmr-aw-pvs`	✓	✓	✓	✓	✓	✓	—

6.7.3 Results

The results of the depth-8 search benchmarks are presented in Table 6.17. Analysis of these metrics reveals several key trends regarding search efficiency.

Table 6.17: Node growth and EBF metrics at depth 8 (initial position).

Binary	Cumulative Nodes (N)	$\bar{b}$ ($N^{1/D}$)	EBF_{geom}
`irida`	362,823	4.954	3.166
`irida-plain`	1,297,709	5.810	4.634
`irida-q`	5,731,889	6.995	5.273
`irida-q-tt`	1,639,226	5.982	4.330
`irida-q-tt-nmp`	2,261,276	6.227	4.607
`irida-q-tt-nmp-lmr`	389,612	4.998	3.513
`irida-q-tt-nmp-lmr-aw`	362,823	4.954	3.166
`irida-q-tt-nmp-lmr-aw-pvs`	335,834	4.906	3.138

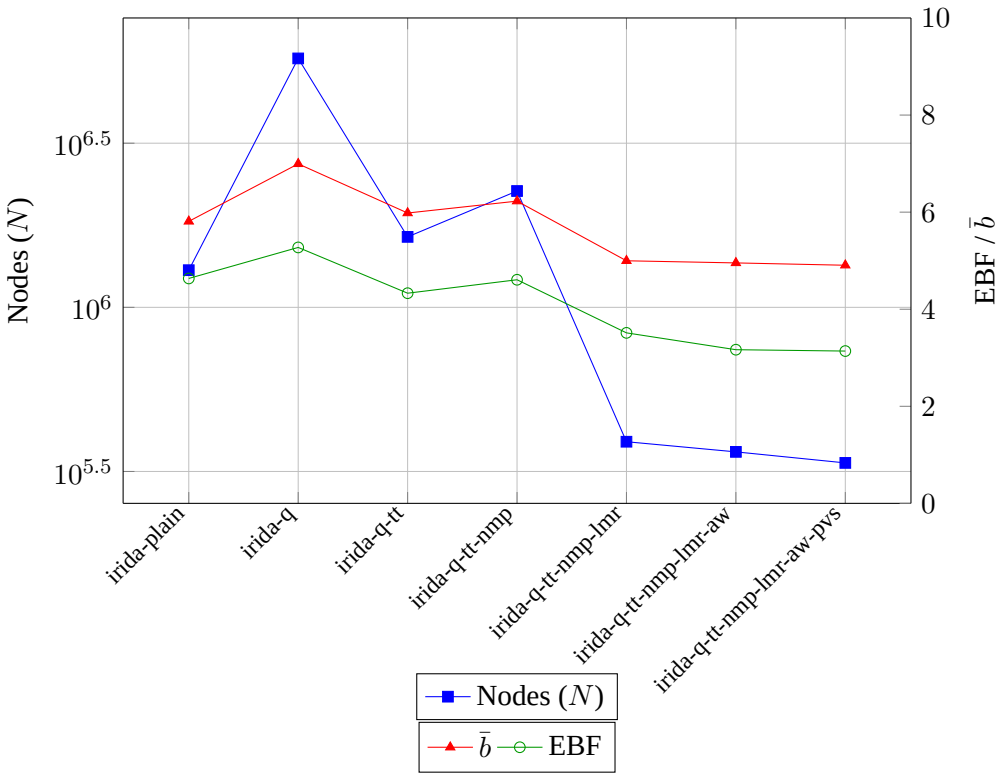


Figure 6.13: Comparison of search metrics across engine iterations.

6.8 Elo Estimation

To estimate Irida's Elo rating, a match of 2,000 paired games was conducted against **Stockfish 18**, which was configured to a fixed strength of **2000 Elo** using the **UCI_LimitStrength** and **UCI_Elo** UCI options.

Statistical analysis of the match results using SPRT (Section 5.3.2) yielded an Elo difference (ΔElo) of **+110.94**. Under the assumption of independent and identically distributed (i.i.d.) games, the 95% confidence interval (CI) for this ΔElo is **[+79.48, +142.40]**. Based on these findings, Irida's performance results in an estimated rating of **2110 Elo**.

Chapter 7: Discussion

7.1 Interpreting the Results: Search, Strength, and Measurements

7.1.1 Ablation Observations

The LMR $\rightarrow$ aspiration and aspiration $\rightarrow$ PVS pairings are the only rows in this chain where E_2 is estimated below E_1 in the pooled tables (Tables 6.7–6.8). The multi-panel figures for these matchups (Figures 6.5–6.6) make the mechanism clear: the **0.1 s per-move** control is where the win-rate and side-performance panels show a *relative* drop for the augmented build, whereas the **fixed depth-**4 control is much closer because both engines are forced to the *same* nominal horizon.

All three effects below are time-budget phenomena; they largely disappear when depth, not the clock, is what is held fixed.

Work versus depth under a wall clock. Aspiration windows and PVS do not change the *rules* of chess; they change *how many* nodes are visited *and* in what order, by inserting small-window probes and, on failure, **re-searches** (root restarts with a widened window for aspiration; full-window re-searches for PVS when a zero-width scout is too good). When the only budget is wall-clock time, any extra pass that does not end in an immediate cutoff *subtracts* from how many iterative-deepening cycles complete before the UCI time handler stops the move. A build that prunes more *in principle* can therefore reach a **shallower completed depth** in 0.1 s than a simpler build that does fewer "meta" retries. In the fixed-depth control, by contrast, both programs may search the same horizon; the comparison is then about efficiency at that depth, not about how many plies fit in the per-move time budget.

Aspiration and noisy priors at short time. Narrow root windows are anchored to the value of the *previous* iteration. At 0.1 s, iterative deepening may stop after only a few outer iterations, so the previous score is a noisier centre for the next window. That raises the rate of *fail low* and *fail high* root outcomes and forces expensive full-window re-runs of the *same* nominal depth, which is exactly the overhead aspiration is meant to avoid when the prior is well-calibrated.

PVS and move ordering at short time. Principal variation search pays off when the first move after ordering is usually best, so most siblings are dismissed after a cheap null-window try. With very short thinking time, the transposition table and history heuristics are less fully populated: more scouts fail into full-window re-searches, and the intended node savings of PVS erode. Table 6.17 shows that, on a single diagnostic position, the AW and default profiles already converge in node count; PVS nudges nodes down only slightly, so the match is won or lost in *scheduling* and ordering noise more than in gross per-position tree size. In the pooled ablation row (Table 6.8) this shows up as a muddled ΔElo: the 95% interval straddles zero, unlike the aspiration case (Table 6.7, entirely below zero), so Table 6.2 should be read as a *clear* loss for added aspiration, but an inconclusive (and schedule-driven) PVS comparison at this sample size.

7.1.2 Heuristic Impact Analysis

The staged EBF results (Table 6.17) and dual-control ablations (Tables 6.2 and 6.9) show that each mechanism contributes in a different way, and raw node reduction alone is not a sufficient predictor of strength:

1. **Quiescence is quality-first, not speed-first.** In the EBF harness, quiescence increases node count relative to `irida-plain` because tactical leaves are expanded further, yet it is the strongest gain in both ablation regimes. The interpretation is that better terminal stability dominates extra search work.

2. **TT mostly converts efficiency into depth under time caps.** TT sharply lowers node growth after quiescence in the fixed-position harness, and it is strongly positive in the movetime chain. Under fixed depth, however, TT is near-neutral, indicating that much of its competitive value appears when saved work can be reinvested into more completed iterations before timeout.

3. **NMP and LMR are schedule-dependent.** NMP is a moderate gain in movetime but inconclusive at fixed depth. LMR is strongly positive in movetime yet negative at fixed depth. Together, these rows indicate that selective pruning/reduction is excellent for short-time efficiency, but can trade off fixed-horizon search fidelity.

4. **AW and PVS are marginal in this budget regime.** Node counts in Table 6.17 show only a small late-stage difference between AW and PVS builds. Correspondingly, both are neutral-to-negative in movetime and near-neutral in fixed depth, which is consistent with sensitivity to ordering quality and re-search overhead.

The convergence of `irida-q-tt-nmp-lmr-aw` and the default `irida` profile suggests that Syzygy tablebase integration does not affect node counts at the early-game start position, as no tablebase probes are triggered.

7.2 Design Trade-offs: Throughput, Depth, and Heuristic Overhead

The results of Chapter 6 reinforce a key engineering distinction: **throughput-oriented optimisations** and **decision-quality optimisations** are not interchangeable, and each is mediated by the active time control.

Move-generation and lifecycle benchmarks (Table 6.1) establish the local performance envelope: raw move generation is fast, but end-to-end search throughput is materially lower once evaluation, state updates, and table lookups are included. Consequently, search strength cannot be inferred from MN/s alone. What matters in play is how efficiently that throughput is converted into completed iterative-deepening layers before the clock stops the search (Section 6.2).

The ablation chain then quantifies where that conversion works. Quiescence, TT, and LMR provide the largest practical gains under 0.1 s movetime, indicating that tactical stability, transposition reuse, and aggressive late-move control collectively improve strength-per-second. However, fixed-depth results expose their limits: TT and NMP become statistically neutral, and LMR can turn negative. This is

evidence that some gains are primarily *time-management gains* (more effective depth for the same wall clock), not unconditional improvements in equal-horizon move quality.

The late-stage AW/PVS rows sharpen this trade-off. Their theoretical purpose is to reduce work via tighter windows, but in short time controls the cost of fail-high/fail-low recovery and scout re-searches competes directly with iterative deepening. In other words, reducing nodes *conditional on stable bounds* is not sufficient when bounds are noisy and the remaining time is small. This motivates practical tuning priorities for student engines: robust ordering, conservative window management at short TC, and parameter sets conditioned on the expected playing schedule rather than a single "global best" profile.

Finally, the Elo estimate against strength-limited Stockfish situates these trade-offs in an external reference frame. The net system is competitive and benefits from the chosen search stack, but the internal ablations show that "add all standard heuristics" is not automatically optimal unless each component is validated under the same time model as intended deployment.

7.3 Lessons for Building a Student Engine

Three methodological lessons are especially transferable to future student projects.

1) Evaluate in stages, not only final-vs-final. The cumulative chain design makes causal interpretation possible. Comparing `irida-plain` directly with the final build would show a large total gain, but would hide that some late additions are neutral or adverse under specific controls. Stage-wise ablation therefore functions as both a research tool and a debugging method for search regressions.

2) Use at least two complementary test regimes. Movetime and fixed-depth controls answer different questions. The first reflects tournament-like constraints (strength per second), while the second probes equal-horizon quality. Running both prevents overfitting to one protocol and helps separate algorithmic merit from time-allocation artifacts.

3) Combine statistical outcomes with search diagnostics. Elo/SPRT outputs tell whether a change is likely beneficial in match play; EBF and node-growth diagnostics help explain *why*. Interpreting them together improves tuning decisions, for example identifying LMR as time-efficient but fixed-depth-fragile, or identifying AW/PVS as sensitive to re-search overhead despite small node-count gains.

Chapter 8: Conclusions and Future Work

8.1 What Was Built and What Was Shown

This thesis presented the design, implementation, and empirical evaluation of a UCI-compatible chess engine in C, with emphasis on transparent search engineering rather than opaque end-to-end learning. The resulting system, *Irida*, combines legal move generation, handcrafted evaluation with NNUE support, and a staged alpha–beta search stack including quiescence, TT, NMP, LMR, aspiration windows, and PVS.

The main empirical contribution is not only the final engine strength, but the *measurement process* used to justify each component. Chapter 6 showed that the largest practical gains under short per-move time come from quiescence, transposition reuse, and LMR, while late-stage windowing heuristics (aspiration and PVS) are sensitive to time budget and do not automatically improve results under every control. The paired movetime/fixed-depth protocol, together with EBF diagnostics and SPRT-based decisions, made this trade-off visible and reproducible.

In summary, the thesis demonstrates that competitive strength in a student engine emerges from the *interaction* of implementation efficiency, time-control-aware search design, and statistically grounded evaluation, rather than from adding heuristics in isolation.

8.2 Limits of the Study

The conclusions should be interpreted with the following boundaries:

- **Hardware and runtime scope.** All experiments were conducted on a single machine, single thread, and fixed hash settings (Chapter 5). Results are therefore internally consistent but not a full cross-hardware performance map.
- **Protocol scope.** The ablation study uses two controls (0.1 s movetime and fixed nominal depth), which intentionally capture two complementary viewpoints. Additional controls (increment, sudden death, longer TCs) may shift the relative value of some heuristics.
- **Statistical uncertainty.** Although sample sizes are large and SPRT/CI reporting is used throughout, engine match outcomes remain noisy and opening-dependent. Borderline rows, especially among late-stage heuristics, should be treated as context-specific rather than universal.
- **Evaluation and learning scope.** The project does not include custom NNUE training or large-scale automated parameter tuning. This keeps the implementation focused and interpretable, but likely leaves strength on the table relative to modern auto-tuned pipelines.
- **External Elo anchoring.** The external Elo estimate is tied to the chosen reference setup and should be read as a practical benchmark, not an absolute rating transferable to all pools.

8.3 Future Work

Several extensions follow naturally from the observed results:

- **Time-control-aware tuning.** Introduce parameter profiles for short and long controls (e.g., aspiration width, reduction tables, null move margins), selected dynamically at runtime.
- **Evaluation refinement.** Expand and tune handcrafted terms, then perform controlled comparisons with stronger NNUE integration paths (network choice, update policy, fallback logic).
- **Parallel search support.** Add SMP search (e.g., Lazy SMP) and repeat ablations under multi-thread settings, where TT dynamics and move-order quality can differ substantially.
- **Broader benchmark matrix.** Extend testing to additional time controls, opening suites, and external opponents to improve generalisability and reduce schedule-specific bias.
- **Automated tuning workflow.** Integrate black-box optimization (for search and evaluation parameters) with SPRT gating, so candidate configurations are accepted or rejected automatically.
- **Tooling and reproducibility.** Package scripts, configuration presets, and experiment manifests for repeatable runs and easier educational reuse by other students.
- **Accumulator–based NNUE.** NNUE that updates as the game progresses to avoid full re-evaluation of the position.

Taken together, these steps would move the project from a strong single-machine research prototype to a more broadly tuned and deployable chess engine framework, while preserving the thesis emphasis on explainability and evidence-based development.

BIBLIOGRAPHY

[1] "Endgame Tablebases Online," Accessed: Mar. 17, 2026. [Online]. Available: https://kirill-kryukov.com/chess/tablebases-online/.

[2] "Syzygy Bases - Chessprogramming wiki," Accessed: Mar. 24, 2026. [Online]. Available: https://www.chessprogramming.org/Syzygy_Bases.

[3] V. Mikkonen. "The Evolution of Chess Engines in the Era of Artificial Intelligence," Accessed: Mar. 17, 2026. [Online]. Available: http://www.theseus.fi/handle/10024/903102.

[4] "UCI protocol," Accessed: Mar. 17, 2026. [Online]. Available: https://www.wbec-ridderkerk.nl/html/UCIProtocol.html.

[5] K. Despoinidis, *KDesp73/castro*, Mar. 15, 2026. Accessed: Mar. 17, 2026. [Online]. Available: https://github.com/KDesp73/castro.

[6] K. Despoinidis, *KDesp73/irida*, Apr. 16, 2026. Accessed: Apr. 16, 2026. [Online]. Available: https://github.com/KDesp73/irida.

[7] "E. MISCELLANEOUS / 01. Laws of Chess / FIDE Laws of Chess taking effect from 1 January 2023 / FIDE Handbook," International Chess Federation (FIDE), Accessed: Apr. 16, 2026. [Online]. Available: https://handbook.fide.com/chapter/E012023.

[8] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," Stanford University, Department of Computer Science, Tech. Rep. STAN-CS-75-530, Oct. 1975. Accessed: May 6, 2026. [Online]. Available: https://stanford.edu/~knuth/papers/alphabeta.pdf.

[9] G. M. Baudet, "An analysis of the full alpha-beta pruning algorithm," in *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, ser. STOC '78, New York, NY, USA: Association for Computing Machinery, Feb. 1, 1978, pp. 296–313, isbn: 978-1-4503-7437-8. doi: 10.1145/800133.804359. Accessed: Mar. 17, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/800133.804359.

[10] C. E. Shannon, "XXII. Programming a computer for playing chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, no. 314, pp. 256–275, Mar. 1950, issn: 1941-5982, 1941-5990. doi: 10.1080/14786445008521796. Accessed: Apr. 11, 2026. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/14786445008521796.

[11] P. Triebold, M. Moll, H.-G. Enkler, and S. Pickl, "From Shogi and Chess to Reinforcement Learning: A Study of NNUEs in More General Settings," in *Operations Research Proceedings 2023*, G. Voigt, M. Fliedner, K. Haase, W. Brüggemann, K. Hoberg, and J. Meissner, Eds., Cham: Springer Nature Switzerland, 2025, pp. 567–572, isbn: 978-3-031-58405-3. doi: 10.1007/978-3-031-58405-3_72.

[12] D. Klein, *Asdfjkl/nnue*, Mar. 2, 2026. Accessed: Mar. 17, 2026. [Online]. Available: https://github.com/asdfjkl/nnue.

[13] D. Klein. "Neural Networks for Chess." arXiv: 2209.01506 [cs], Accessed: Mar. 17, 2026. [Online]. Available: http://arxiv.org/abs/2209.01506, pre-published.

[14] D. Tan and N. W. Medina. “Study of the Proper NNUE Dataset.” arXiv: 2412.17948 [cs], Accessed: Mar. 17, 2026. [Online]. Available: http://arxiv.org/abs/2412.17948, prepublished.

[15] S. Maharaj, N. Polson, and A. Turk, “Chess AI: Competing Paradigms for Machine Intelligence,” *Entropy*, vol. 24, no. 4, p. 550, Apr. 14, 2022, issn: 1099-4300. doi: 10.3390/e24040550. arXiv: 2109.11602 [cs]. Accessed: Mar. 17, 2026. [Online]. Available: http://arxiv.org/abs/2109.11602.

[16] M. C. Fu, “A tutorial introduction to Monte Carlo tree search,” in *2020 Winter Simulation Conference (WSC)*, Dec. 2020, pp. 1178–1193. doi: 10.1109/WSC48552.2020.9384090. Accessed: May 6, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/9384090.

[17] “Monte-Carlo Tree Search - Chessprogramming wiki,” Accessed: Apr. 16, 2026. [Online]. Available: https://www.chessprogramming.org/Monte-Carlo_Tree_Search.

[18] “Official-stockfish/Stockfish: A free and strong UCI chess engine,” Accessed: Mar. 24, 2026. [Online]. Available: https://github.com/official-stockfish/Stockfish.

[19] “CCRL - Index,” Accessed: Apr. 16, 2026. [Online]. Available: https://computerchess.org.uk/4040/.

[20] “Carlsen, Magnus FIDE Chess Profile,” International Chess Federation (FIDE), Accessed: Apr. 16, 2026. [Online]. Available: https://ratings.fide.com/profile/1503014.

[21] D. Silver et al., “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play,” *Science*, vol. 362, no. 6419, pp. 1140–1144, Dec. 7, 2018. doi: 10.1126/science.aar6404. Accessed: May 6, 2026. [Online]. Available: https://www.science.org/doi/10.1126/science.aar6404.

[22] J. Dart, *Jdart1/Fathom*, Apr. 6, 2026. Accessed: Apr. 16, 2026. [Online]. Available: https://github.com/jdart1/Fathom.

[23] jordiruizcentelles. “Syzygy Endgame impact on Chess Engine Performance,” IJCCRL, Accessed: Mar. 17, 2026. [Online]. Available: https://ijccrl.com/syzygy-endgame-impact-on-chess-engine-performance/.

[24] A. L. Zobrist, “A New Hashing Method with Application for Game Playing1,” *ICGA Journal*, vol. 13, no. 2, pp. 69–73, Jun. 1, 1990, issn: 24682438, 13896911. doi: 10.3233/ICG-1990-13203. Accessed: Mar. 17, 2026. [Online]. Available: https://journals.sagepub.com/doi/full/10.3233/ICG-1990-13203.

[25] “Transposition Table - Chessprogramming wiki,” Accessed: Mar. 24, 2026. [Online]. Available: https://www.chessprogramming.org/Transposition_Table.

[26] D. M. Breuker, J. W. H. M. Uiterwijk, and H. J. van den Herik, “Replacement schemes for transposition tables,” *ICCA Journal*, vol. 17, no. 4, pp. 183–204, 1994. doi: 10.3233/ICG-1994-17402.

[27] “PeSTO’s Evaluation Function - Chessprogramming wiki,” Accessed: Mar. 24, 2026. [Online]. Available: https://www.chessprogramming.org/PeSTO%27s_Evaluation_Function.

[28] “Cute Chess,” Accessed: Apr. 16, 2026. [Online]. Available: https://cutechess.com/.

[29] “BanksiaGUI – A Free Chess Graphical User Interface,” Accessed: Apr. 16, 2026. [Online]. Available: https://banksiagui.com/.

[30] I. Pantidis, M. Harrison, S. Choksi, and T. Duplessis, *Lichess-bot*, Mar. 19, 2026. Accessed: Mar. 19, 2026. [Online]. Available: https://github.com/lichess-bot-devs/lichess-bot.

[31] *Chess960*, in *Wikipedia*, Apr. 11, 2026. Accessed: Apr. 20, 2026. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Chess960&oldid=1348290161.

[32] "GCC, the GNU Compiler Collection - GNU Project," Accessed: Apr. 15, 2026. [Online]. Available: https://gcc.gnu.org/.

[33] "Clang C Language Family Frontend for LLVM," Accessed: Apr. 15, 2026. [Online]. Available: https://clang.llvm.org/.

[34] "Git," Accessed: Apr. 15, 2026. [Online]. Available: https://git-scm.com/.

[35] K. Despoinidis, *KDesp73/changelogger*, Feb. 11, 2026. Accessed: Apr. 15, 2026. [Online]. Available: https://github.com/KDesp73/changelogger.

[36] K. Despoinidis, *KDesp73/tinydocs*, Mar. 3, 2026. Accessed: Apr. 15, 2026. [Online]. Available: https://github.com/KDesp73/tinydocs.

[37] "IncludeOnly/libs/test.h at main · KDesp73/IncludeOnly," Accessed: Apr. 15, 2026. [Online]. Available: https://github.com/KDesp73/IncludeOnly/blob/main/libs/test.h.

[38] P. Kannan, "Magic Move-Bitboard Generation in Computer Chess," [Online]. Available: https://pradu.us/old/Nov27_2008/Buzz/research/magic/Bitboards.pdf.

[39] D. R. Rasmussen, "Parallel Chess Searching and Bitboards," 2004, Master's thesis, University of Southern Denmark.

[40] "Polyglot book format," Accessed: Mar. 17, 2026. [Online]. Available: http://hgm.nubati.net/book_format.html.

[41] "Negamax - Chessprogramming wiki," Accessed: Mar. 24, 2026. [Online]. Available: https://www.chessprogramming.org/Negamax.

[42] M. Maschler, S. Zamir, and E. Solan, *Game Theory*. Cambridge University Press, Jun. 25, 2020, 1053 pp., isbn: 978-1-108-49345-1. Google Books: o6UlEAAAQBAJ.

[43] D. J. Edwards and T. P. Hart, "The Alpha-Beta Heuristic," Dec. 1, 1961. Accessed: Apr. 11, 2026. [Online]. Available: https://dspace.mit.edu/handle/1721.1/6098.

[44] S. H. Fuller, J. G. Gaschnig, and J. J. Gillogly, "Analysis of the alpha-beta pruning algorithm," Nov. 1, 2010. doi: 10.1184/R1/6603488.v1. Accessed: Mar. 17, 2026. [Online]. Available: https://kilthub.cmu.edu/articles/journal_contribution/Analysis_of_the_alpha-beta_pruning_algorithm/6603488/1.

[45] "Iterative Deepening - Chessprogramming wiki," Accessed: Mar. 24, 2026. [Online]. Available: https://www.chessprogramming.org/Iterative_Deepening.

[46] E. Kodhai, E. Bhuvaneswari, V. A. Devi, and S. Preethi, "Iterative Deepening Chess Engine with Alpha Beta Pruning," in *2024 International Conference on Smart Technologies for Sustainable Development Goals (ICSTSDG)*, Aug. 2024, pp. 1–4. doi: 10.1109/ICSTSDG61998.2024.11026648. Accessed: Mar. 17, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/11026648.

[47] R. R. Nair, T. Babu, S. Kishore, K. Pavithra, and S. G. Sumana, "Improving Alpha-Beta Pruning Efficiency in Chess Engines," in *2024 International Conference on IT Innovation and Knowledge Discovery (ITIKD)*, Apr. 2025, pp. 1–6. doi: 10.1109/ITIKD63574.2025.11004942. Accessed: Mar. 17, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/11004942.

[48] H. J. Berliner, "Search and knowledge," in *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, vol. 2, 1977. Accessed: May 6, 2026. [Online]. Available: https://www.ijcai.org/Proceedings/77-2/Papers/087.pdf.

[49] "Quiescence Search - Chessprogramming wiki," Accessed: Mar. 24, 2026. [Online]. Available: https://www.chessprogramming.org/Quiescence_Search.

[50] "Move Ordering - Chessprogramming wiki," Accessed: Mar. 24, 2026. [Online]. Available: https://www.chessprogramming.org/Move_Ordering.

[51] A. Cook and W. Edmondson, "Using Chunking to Optimise an Alpha-Beta Search," in *2010 International Conference on Technologies and Applications of Artificial Intelligence*, Aug. 2010, pp. 166–171. doi: 10.1109/TAAI.2010.36. Accessed: Mar. 17, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/5695448.

[52] "MVV-LVA - Chessprogramming wiki," Accessed: Mar. 24, 2026. [Online]. Available: https://www.chessprogramming.org/MVV-LVA.

[53] O. David-Tabibi and N. S. Netanyahu. "Verified Null-Move Pruning." arXiv: 0808.1125 [cs], Accessed: Mar. 17, 2026. [Online]. Available: http://arxiv.org/abs/0808.1125, pre-published.

[54] E. A. Heinz, "Extended null-move reductions," in *Advances in Computer Games, Many Games, Many Challenges*, H. J. van den Herik, H. Iida, and E. A. Heinz, Eds., Berlin, Heidelberg: Springer, 2004, ch. 19, pp. 325–353. doi: 10.1007/978-3-540-87608-3_19. Accessed: May 6, 2026. [Online]. Available: https://doi.org/10.1007/978-3-540-87608-3_19.

[55] "Late Move Reductions - Chessprogramming wiki," Accessed: Mar. 24, 2026. [Online]. Available: https://www.chessprogramming.org/Late_Move_Reductions.

[56] "Aspiration Windows - Chessprogramming wiki," Accessed: Mar. 24, 2026. [Online]. Available: https://www.chessprogramming.org/Aspiration_Windows.

[57] H. Kaindl, R. Shams, and H. Horacek, "Minimax Search Algorithms With and Without Aspiration Windows," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 13, no. 12, pp. 1225–1235, Sep. 1, 1991, issn: 0162-8828. doi: 10.1109/34.106996. Accessed: Mar. 24, 2026. [Online]. Available: https://doi.org/10.1109/34.106996.

[58] R. Shams, H. Kaindl, and H. Horacek, "Using Aspiration Windows for Minimax Algorithms,"

[59] "Principal Variation Search - Chessprogramming wiki," Accessed: Mar. 24, 2026. [Online]. Available: https://www.chessprogramming.org/Principal_Variation_Search.

[60] "NegaScout - Chessprogramming wiki," Accessed: Mar. 24, 2026. [Online]. Available: https://www.chessprogramming.org/NegaScout.

[61] "Texel's Tuning Method - Chessprogramming wiki," Accessed: Mar. 24, 2026. [Online]. Available: https://www.chessprogramming.org/Texel%27s_Tuning_Method.

[62] A. E. Elo, *The Rating of Chessplayers, Past and Present*. New York: Arco Publishing, 1978, isbn: 978-0668047210.

[63] A. Wald and J. Wolfowitz, "Optimum character of the sequential probability ratio test," *The Annals of Mathematical Statistics*, vol. 19, no. 3, pp. 326–339, 1948. doi: 10.1214/aoms/1177730197. Accessed: May 6, 2026. [Online]. Available: https://doi.org/10.1214/aoms/1177730197.

[64] "Sequential Probability Ratio Test - Chessprogramming wiki," Accessed: Apr. 9, 2026. [Online]. Available: https://www.chessprogramming.org/Sequential_Probability_Ratio_Test.

[65] "Perft - Chessprogramming wiki," Accessed: Apr. 9, 2026. [Online]. Available: https://www.chessprogramming.org/Perft.

[66] "Perft Results - Chessprogramming wiki," Accessed: Apr. 9, 2026. [Online]. Available: https://www.chessprogramming.org/Perft_Results.

Appendix A: Usage Guide

A.1 Castro Library

Castro is a high-performance move generation library. To build the library from source, clone the repository and use the provided `Makefile`:

```
git clone https://github.com/KDesp73/castro && cd castro
make all -j2
```

A.1.1 C API Example

The following example (`example.c`) demonstrates how to initialize the global lookup tables, load a board from a FEN string, and generate legal moves. Note that we include `castro.h`, which contains the core board structures and function prototypes.

example.c

```c
#include <stdio.h>
#include "castro.h"

int main(void)
{
    // Initialize global lookup tables (Zobrist, Bitmasks, Magic Bitboards)
    castro_InitZobrist();
    castro_InitMasks();
    castro_InitMagic();

    const char* fen = "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1";
    Board board = {0};
    castro_BoardInitFen(&board, fen);

    // Generate all legal moves for the current position
    Moves moves = castro_GenerateMoves(&board, MOVE_LEGAL);

    printf("Legal moves in the starting position:\n");
    for (size_t i = 0; i < moves.count; ++i) {
        char move_str[8];
        castro_MoveToString(moves.list[i], move_str);
        printf("%s\n", move_str);
    }

    castro_BoardFree(&board);
    return 0;
}
```

A.1.2 Compilation and Execution

To compile your application, link against the static library generated in the previous step. Ensure the include path (`-I`) points to the directory containing `castro.h`.

```
gcc example.c -o example -L. -l:libcastro.a -I src/
./example
```

Expected Output:

```
Legal moves in the starting position:
a2a3
a2a4
b2b3
b2b4
c2c3
c2c4
d2d3
d2d4
e2e3
e2e4
f2f3
f2f4
g2g3
g2g4
h2h3
h2h4
b1a3
b1c3
g1f3
g1h3
```

A.2 Irida Engine

Irida is a UCI-compliant chess engine. It can be used via any standard Chess GUI (such as Arena or Cutechess) or directly through the terminal for debugging and testing.

A.2.1 Building from Source

To build the engine, clone the repository and run the `make` command:

```
git clone https://github.com/KDesp73/irida && cd irida
make irida
./irida
```

A.2.2 Manual UCI Usage

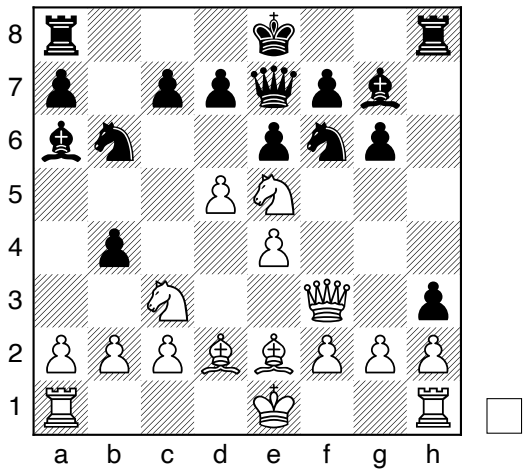
Once the engine is running, you can communicate with it using the UCI protocol. In the example below, > denotes user input and < denotes engine output.

UCI Session

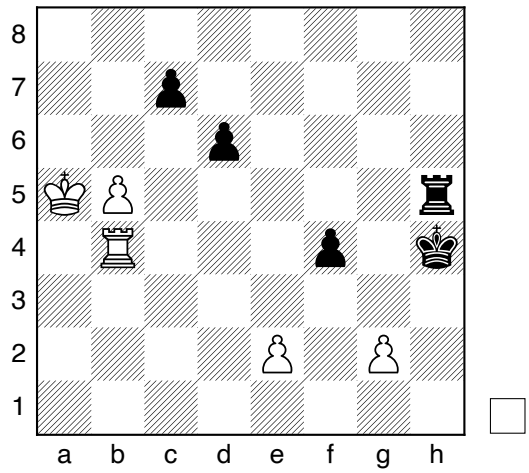
```
< irida v0.9.0 by Konstantinos Despoinidis (KDesp73)
> uci
< id name irida
< id author Konstantinos Despoinidis (KDesp73)
< id version 0.9.0
< ... [options omitted for brevity] ...
< uciok

# Command the engine to search the current position to depth 6
> go depth 6
< info depth 1 seldepth 1 score cp 162 nodes 40 nps 0 pv g1f3
< info depth 2 seldepth 3 score cp 0 nodes 197 nps 0 pv g1f3 g8f6
< info depth 3 seldepth 6 score cp 144 nodes 1732 nps 0 pv g1f3 g8f6 b1c3
< info depth 4 seldepth 7 score cp 0 nodes 4009 nps 0 pv b1c3 g8f6 g1f3 b8c6
< info depth 5 seldepth 9 score cp 140 nodes 14982 nps 2 pv e2e3 e7e6 d1f3 d8f6
↪  f1d3
< info depth 6 seldepth 12 score cp 0 nodes 46862 nps 8 pv b1c3 b8c6 c3b5 c6b4 g1f3
↪  g8f6
< bestmove b1c3
```

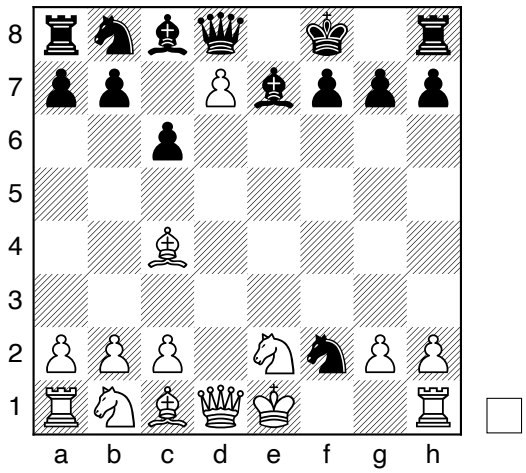
Appendix B: Games and Positions



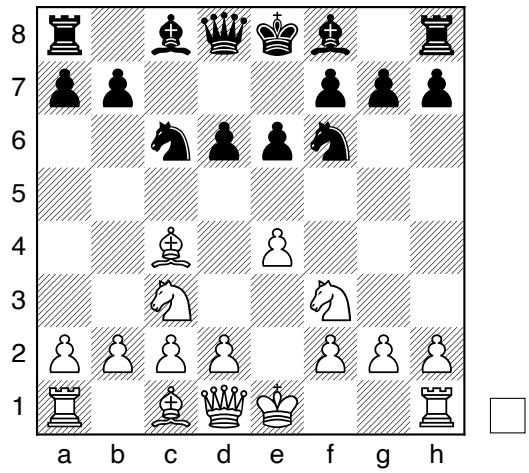
(a) Kiwipete



(b) Endgame



(c) Position 5



(d) Middlegame

Figure B.1: Benchmark test positions used for Perft verification and throughput analysis.

```
[Event "Engine V Engine"]
[Site "Local Arena"]
[Date "2026.04.22"]
[Round "6"]
[White "irida 0.10.0"]
[Black "Stockfish 18 (2000 Elo)"]
[Result "1-0"]

1. e4 c5 2. Nf3 d6 3. Bc4 Nf6 4. d3 e6 5. Bb3 Be7 6. c3 O-O 7. O-O Nc6
8. Be3 e5 9. Ba4 Na5 10. b4 c4 11. bxa5 cxd3 12. Qxd3 Qxa5 13. Bb3 b6
14. Qc4 Ba6 15. Qc7 Qb5 16. Rd1 Rfe8 17. Ng5 Rac8 18. Bxf7+ Kf8
19. Qxa7 Red8 20. Be6 Rc6 21. Bd5 Qe2 22. Ne6+ Kf7 23. Nxd8+ Kg6 24. Bf7# 1-0
```

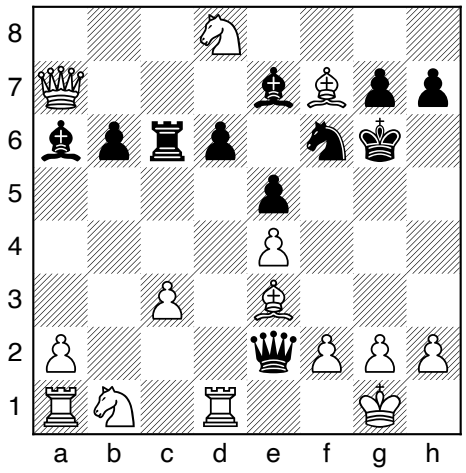


Figure B.2: Irida 0.10.0 delivering mate with white against Stockfish 18. Stockfish was limited to 2000 Elo with 0.2s per move for both engines.

```
[Event "Engine V Engine"]
[Site "Local Arena"]
[Date "2026.04.23"]
[Round "9"]
[White "Stockfish 18 (2000 Elo)"]
[Black "irida 0.10.0"]
[Result "0-1"]

1. Nf3 d5 2. d4 Nf6 3. c4 e6 4. Nc3 c6 5. Bg5 h6 6. Bxf6 Qxf6 7. e3 Nd7 8. Bd3
dxc4 9. Bxc4 g6 10. O-O Bg7 11. Rc1 O-O 12. Re1 Rd8 13. Qe2 Qf5 14. Red1 Qh5 15.
b4 Nb6 16. Ne4 Nxc4 17. Qxc4 a5 18. Rd2 Qg4 19. Qc2 f6 20. h3 Qf5 21. a3 axb4
22. Nh4 Qh5 23. Nf3 Qa5 24. Nc5 Qxa3 25. Rb1 f5 26. Ne5 Qc3 27. Qd3 Bxe5 28. Qe2
b3 29. Rd3 Qc4 30. dxe5 Rxd3 31. Qxd3 Qxc5 32. Qd8+ Qf8 33. Qc7 Qg7 34. Qb6 Qxe5
35. Rxb3 Qd6 36. Rd3 Ra1+ 37. Qb1 Rxb1+ 38. Rd1 Rxd1# 0-1
```

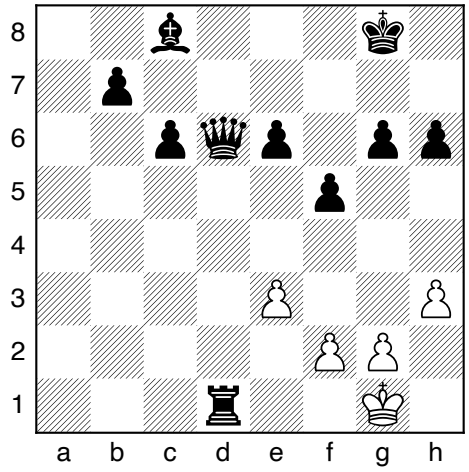


Figure B.3: Irida 0.10.0 delivering mate with black against Stockfish 18. Stockfish was limited to 2000 Elo with 0.2s per move for both engines.

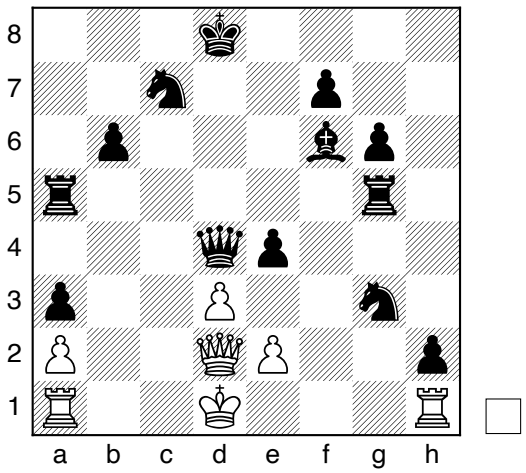


Figure B.4: Die Schachspieler painting position. Irida evaluates this position as -23.61 so the devil is winning the game.

Appendix C: Figures

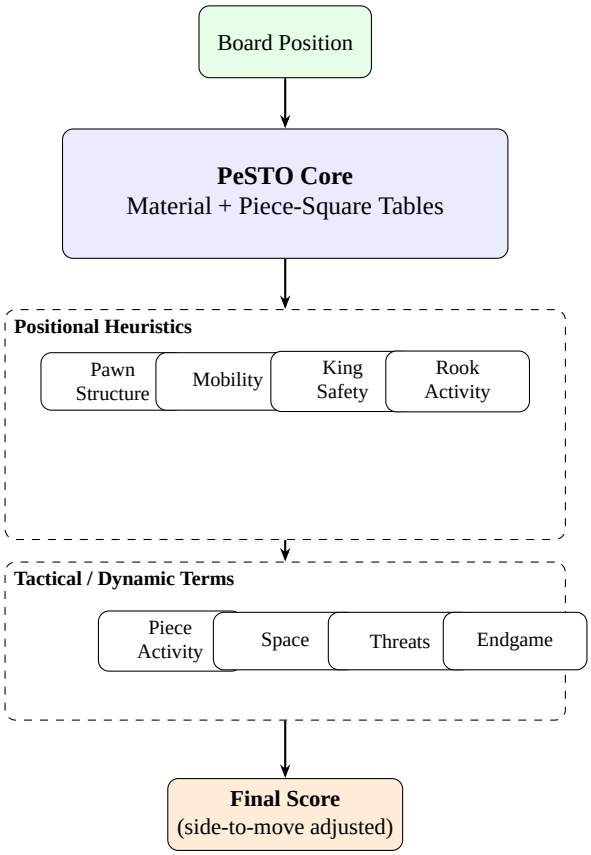


Figure C.1: Layered structure of the evaluation function. A PeSTO-based material and piece-square table core is augmented with positional heuristics and tactical/dynamic terms before producing the final score.

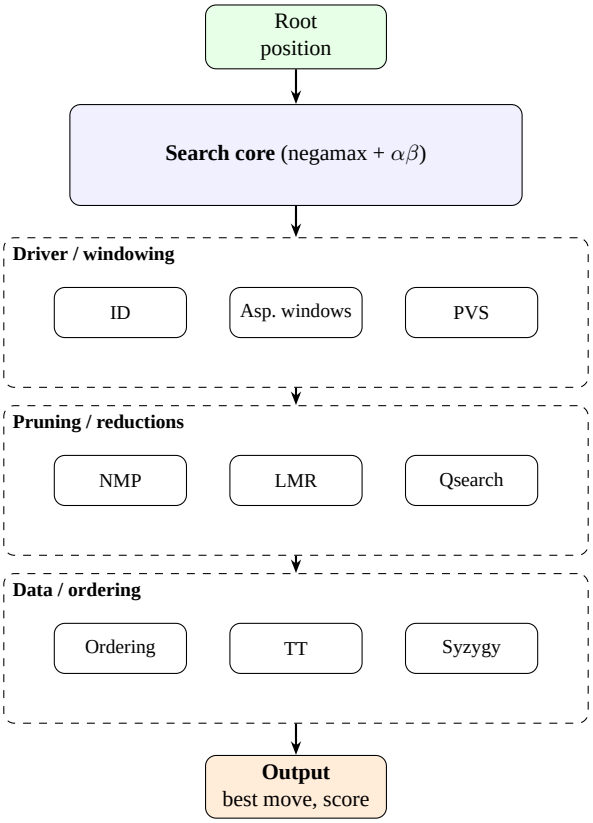


Figure C.2: High-level structure of the search system: a single negamax–alpha-beta core with grouped enhancements (schematic, not a call graph).

Appendix D: Code Snippets

perft.c

```c
#include "castro.h"

typedef unsigned long long u64;

u64 castro_PerftPseudoLegal(Board* board, int depth)
{
    if (depth <= 0) return 1;

    Moves moves = castro_GenerateMoves(board, MOVE_PSEUDO);
    u64 total = 0;

    for (size_t i = 0; i < moves.count; i++) {
        if (!castro_MakeMove(board, moves.list[i]))
            continue;

        // After MakeMove, board->turn is the opponent;
        // the side that just moved is !board->turn
        if (!castro_IsInCheckColor(board, (PieceColor)(!board->turn))) {
            if (depth == 1)
                total++;
            else
                total += castro_PerftPseudoLegal(board, depth - 1);
        }
        castro_UnmakeMove(board);
    }
    return total;
}
```

castro.h

```c
// @type Board
// @desc Core board structure combining bitboards and grid for performance and
↪  simplicity.
typedef struct {
    Bitboard bitboards[PIECE_TYPES]; ///< One bitboard per piece type (white/black)
    char grid[8][8];                 ///< ASCII piece grid for quick access
    Bitboard white;                  ///< Cached: all white pieces
    Bitboard black;                  ///< Cached: all black pieces
    Bitboard empty;                  ///< Cached: empty squares (~(white|black))

    // Game state
    Square enpassant_square;         ///< En passant target square, if any
    bool turn;                       ///< true = white to move, false = black
    uint8_t castling_rights;         ///< Castling rights bitfield
    size_t halfmove;                 ///< Halfmove clock for 50-move rule
    size_t fullmove;                 ///< Fullmove number (starts at 1)
```

```

    History history;                ///< Move history
    uint64_t hash;                  ///< Zobrist hash of current position

    struct {
        bool turn;
        int halfmoveClock;
        int fullmoveNumber;
        Square epSquare;
    } null_move_state;
} Board;
```

core.h

```
// @struct Engine
// @desc Engine descriptor: name, author, function pointers (search, eval, order),
↪  and board.
typedef struct {
    char name[128];
    char author[128];

    SearchFn search;
    EvalFn eval;
    OrderFn order;

    Board board;
} Engine;
```