Design and implementation of a Multi-Level Chess AI System and performance evaluation by its difficulty levels

Grey(2430034039)
Faculty of Science and Technology, BNBU
Email: u430034039@mail.uic.edu.cn

***Abstract*—This paper presents a multi-level chess AI system developed as part of the Artificial Intelligence Workshop course, and performance evaluation by its difficulty levels. The system implements four primary difficulty levels (L1–L3 + Ultimate) for the main experiments by progressively increasing search depth and evaluation complexity, transitioning from greedy search to minimax with alpha-beta pruning and iterative deepening, with the Ultimate level using advanced PeSTO tapered evaluation. Our system combines classical tree search algorithms with two kinds of evaluation heuristics: theory-driven positional heuristics inspired by common chess engines, and CNN-based learned evaluators from selfplay. Difficulty scaling is achieved through systematic variations in search depth, pruning settings, and evaluation complexity, as well as deliberate randomization in move selection at lower levels. We conduct extensive self-play and human-AI experiments, and apply basic statistical analysis to quantify performance differences between levels. The results show clear separation in playing strength, with the strongest engine achieving substantially higher win rates against lower-level opponents while maintaining reasonable computational efficiency. The system supports multiple game modes, including human-versus-machine, machine-versus-machine, machine-versus-machine with limited computation time, and batch test mode, enabling both interactive play and reproducible evaluation. This work demonstrates the effectiveness of classical search-based methods in resource-constrained settings and provides a foundation for future extensions aimed at making chess AI more time-efficient and energy-aware on limited hardware.**

***Index Terms*—Game-tree search, Minimax, Alpha–Beta pruning, Heuristic evaluation, Chess AI, Difficulty scaling**

I. Introduction

Computer chess is among the most studied problems in Artificial Intelligence, making it a classic testbed for AI research for decades.

Early work such as Shannon's 1950 paper "Programming a Computer for Playing Chess" [1] emphasized that chess agents are prototypes for more general "thinking machines": systems that could, in principle, make complex, strategic decisions like human beings. In this view, computer chess is not an end in itself, but a concrete sandbox for studying how machines can make complex, strategic decisions under uncertainty and resource constraints. Building on this tradition, modern chess engines combine powerful search algorithms with sophisticated evaluation functions, and continue to provide a platform for artificial intelligence research.

Most contemporary approaches focus on maximizing playing strength at the expense of computational efficiency. However, in many circumstances, we may want agents with *adjustable* strength that can provide meaningful practice for human players or that can operate effectively under tight time and hardware constraints.

This raises the question of how to design chess AI systems that expose multiple difficulty levels in a principled way while remaining computationally efficient. In this work, we address these challenges by designing a multi-level chess AI system built on classical tree search algorithms with two families of evaluation functions: theory-driven positional heuristics inspired by common chess engines and researched in the literature, and CNN-based learned evaluators of board positions from self-plays. The system exposes four primary difficulty levels (L1–L3 + Ultimate) for the main experiments through systematic control of search algorithms (greedy, minimax with alpha-beta pruning), search depth, pruning settings, evaluation complexity, and rationale in move selection. We further conduct both self-play and human–AI experiments, with and without strict time limits on search, and use basic statistical analysis to quantify the performance differences between levels and to study the relationship between playing strength and computational effort for different difficulty levels.

The main contributions of this paper are as follows:

- We implement a lightweight frontend/UI (desktop or browser) that renders the board, supports timing/undo/hint controls, logs moves, and interfaces with the backend engine via simple APIs to enable easy interaction and evaluation.
- We design and implement a multi-level chess AI system with four primary difficulty levels (L1–L3 + Ultimate) for the main experiments by varying search algorithms, search depth, pruning aggressiveness, evaluator choice, and randomness in move selection, and supports human-versus-machine(H2M), machine-versus-machine(M2M), M2M with limited computation time, and M2M batch-testing for testing and benchmarking.
- We propose a plug-and-play hybrid evaluation framework that combines theory-driven positional heuristics with CNN-based board evaluators and integrates them with classical alpha–beta search and move ordering for practical use under tight resource budgets.
- We provide an experimental and statistical study (self-play and human–AI, with and without strict time limits)

reporting win rates, searched nodes, and game lengths to quantify the trade-off between playing strength and computational cost across difficulty levels.

II. Literature Review

A. Classical View of Computer Chess as Tree Search

Since Claude Shannon's seminal work on Programming a Computer for Playing Chess [1], computer chess has been seen as a typical game-tree search problem, in which the agent expands and explores nodes in a tree of possible moves and counter-moves. However, Shannon pointed out that since chess is such a big game in which each position can have more than 30 legal moves, and games can last for dozens of plies, making the full game tree astronomically large that exhaustive search simply infeasible. Therefore, chess programs must perform depth-limited search and use heuristic evaluation to estimate the value of non-terminal positions.

Shannon also distinguished between *type A* and *type B* search: *type A* algorithms search to a fixed depth, whereas *type B* selectively explores promising branches, foreshadowing later developments in search algorithms and heuristics. Our work is also inspired by this classical view, treats chess as a tree-search problem with depth-limited lookahead and heuristics. In particular, we implement both *type A* and *type B* search algorithms in our system, their behavior are to be analyzed in later sections.

B. Search Techniques

Building on the tree-search view of chess, a long line of work has developed various search techniques with increasing depth, strength, and efficiency.

One-ply or greedy search is the simplest search technique that evaluates only the immediate successor positions to choose the one with the best heuristic score. Early game-playing programs such as Samuel's checkers system [2] combined greedy search with hand-crafted evaluation functions, showing that even limited computing power could produce competitive play despite the hardware limitations of the time.

A more advanced alternative is minimax search, rooted in game theory by Von Neumann and others on zero-sum games and the minimax theorem [3], [4]. In our game tree search context like chess, minimax assumes rational and optimal play from both sides, and recursively backs up the best value from the leaf nodes to the root to choose the best move for its side. However, the time complexity of full-width minimax is exponential to search depth, making straightforward implementations impractical for deep search.

Alpha–beta pruning was introduced to help with this problem by pruning branches that cannot affect the final minimax value. To be specific, it is a technique to prune the branches of the tree that are guaranteed to be worse than the best move found so far. Historically, alpha–beta pruning was discovered independently several times in the late 1950s and early 1960s and later analyzed in detail by Knuth and Moore in their work "An Analysis of Alpha-Beta Pruning" [5], in which they showed that branching factor can be reduced roughly to the square root of that of plain minimax. In practice, alpha–beta pruning, combined with appropriate move-ordering heuristics, allows significantly deeper search within the same budget.

Iterative deepening has become a standard technique in classical chess engines for both time management and improving move ordering. Instead of searching directly to a target depth, the engine performs a series of depth-limited alpha–beta searches of increasing depth, using information from shallower iterations (e.g., the principal variation) to order moves in deeper ones. Although this appears to repeat work, empirical and theoretical analyses have shown that depth-first iterative deepening can be close to optimal in time and space for exponential trees [6], and it often speeds up practical engines because of its positive effect on move ordering [7], [8].

The Monte Carlo Tree Search (MCTS) is a revolutionary breakthrough in search techniques. Frameworks such as Coulom's Monte-Carlo tree search with selective backup [12] and the UCT algorithm of Kocsis and Szepesvári [13] combine tree search with stochastic rollouts and bandit-based exploration to trade exhaustive lookahead for sampled trajectories guided by statistical estimates. Unlike traditional depth-first search, MCTS explores the tree in a more principled way by sampling the tree and using the results to guide the search. MCTS incrementally builds an asymmetric search tree by repeating selection / expansion / simulation / backup, focusing computation on promising lines of play, making it especially good at handling large branching factors, as in Go. The success of AlphaGo brought MCTS-based methods to widespread attention and showed their potential when combined with deep neural networks in large-board games like Go [14].

Shannon's distinction between *type A* (uniform full-width) and *type B* (selective) search foreshadowed later selective-search algorithms. Practical implementations of type-B-style search emerged in programs such as the Northwestern University Chess series (Chess 3.x and 4.x), in which they emploted selective extensions and sophisticated heuristics [9], [15], and in best-first algorithms such as Stockman's SSS* [10] and Berliner's B* [11], which explore game trees in a best-first manner similar to A* while still computing minimax-optimal moves. Modern chess engines typically employ hybrid schemes that combine predominantly full-width alpha–beta search with selective extensions in tactically critical lines, blending type-A and type-B characteristics while preserving strong theoretical guarantees in most positions.

In our system, we adopt a hybrid search architecture that reflects this evolution. For Level 1, we use simple greedy evaluation without building a search tree, which provides conceptual clarity and computational efficiency. For Level 2, we implement basic minimax search at fixed depth 3 with alpha–beta pruning, using material-only evaluation and random tie-breaking to weaken play. For Level 3, we employ alpha–beta pruning with iterative deepening up to depth 6 and quiescence search, which provides fine-grained control over search depth and pruning behavior. The Ultimate level uses the same search architecture as Level 3 but employs iterative deepening up to depth 8 and PeSTO (Piece-Square Tables Only) tapered evaluation, and more advanced quiescence search, which provides phase-adaptive assessment that swich between different evaluation tables between middle-game and endgame based on board material, using proven evaluation tables from chess programming literature. Additionally, the system includes experimental variants (L4/L5) with enhanced evaluation functions, but these were not included in the primary benchmark and are reserved for extended testing only.

The techniques reviewed above directly inform our choice of search algorithms and our design of depth limits, pruning strategies, and time-control strategies across the difficulty levels in our multi-level chess AI.

C. Evaluation Functions

Given a fixed search procedure, the strength of a chess engine depends on the quality of its evaluation function, which estimates the value of non-terminal positions. Existing work on evaluation functions can be divided into two families: theory-driven heuristic evaluation based on human knowledge, and learned evaluation using statistical or neural models.

1) Theory-Driven Heuristic Evaluation: Classical heuristic evaluation functions are typically hand-crafted, combining multiple features into a score value. Early systems already incorporated material balance—assigning values to each piece type—as the core component of evaluation, reflecting the importance of material advantage in human chess strategy [1]. Subsequent engines enriched this basic model with additional features based on position analysis such as piece-square tables, mobility, pawn structure, and king safety [16], [17]. Piece-square tables reward or penalize pieces for occupying strategically important squares; mobility terms measure the number of available moves or controlled squares; pawn-structure terms capture concepts such as passed, doubled, and isolated pawns; and king-safety terms account for pawn shields, open files near the king, and nearby attacking pieces.

These features are usually combined in a (piece-wise) linear form,

$$\text{eval}(s) = \sum_i w_i f_i(s), \tag{1}$$

where $f_i(s)$ are feature functions of the position s and w_i are tunable weights. Many engines also interpolate between separate "opening" and "endgame" evaluations using tapered evaluation schemes, reflecting the changing importance of features across game phases [18]. This theory-driven approach benefits from interpretability and tight connections to established chess principles, and it remains the dominant design in many strong alpha–beta-based engines. Our system adopts a similar feature set for its heuristic component, including material, piece-square tables, mobility, pawn structure, and king-safety terms.

2) Learned Evaluation and Neural Networks: In parallel with hand-crafted heuristics, there is a long tradition of learning evaluation functions from data. Samuel's pioneering work on checkers [2] and Tesauro's TD-Gammon [19] demonstrated that neural networks trained by self-play can discover strong evaluation functions that rival or surpass human-designed heuristics. More recent systems for chess and other board games, such as Giraffe [20] and the AlphaZero/Leela family of engines [21], [22], employ deep neural networks to approximate position values and policies, trained on large datasets of expert games or self-play.

A common pattern in these systems is to encode the board state as a stack of two-dimensional feature planes and to feed this representation into a convolutional neural network (CNN) that outputs either a scalar value estimate or both value and move probabilities. The CNN thereby acts as a powerful, learned evaluation function that can capture high-order interactions and local patterns that are difficult to express using simple linear heuristics, at the cost of higher computational expense per evaluation.

While these learned evaluation approaches have shown promise in recent work, our system focuses primarily on theory-driven heuristic evaluation functions for practical deployment under resource constraints. The Ultimate level employs PeSTO tapered evaluation, a well-established evaluation scheme that adapts to game phase, while the main difficulty levels (Level 1 through Level 3) employ progressively richer heuristic evaluation features. This design prioritizes computational efficiency and interpretability while maintaining clear separation between difficulty levels through systematic variations in search depth and evaluation complexity.

D. Practical Engine Architectures and Time Management

In practice, chess engines must integrate all these components into a unified architecture that can operate under certain constraints. A typical engine is organized into several parts: representation, search, evaluation, and time control. This section outlines how modern engines organize these components and assemble them into a usable and efficient move-selection procedure.

Many strong engines use bitboard-based position representations and highly optimized move generators to reduce the overhead per node, thereby allowing deeper search within a fixed time budget [9]. Additionally, evaluated positions can be cached in a transposition table keyed by Zobrist hashes to avoid redundant evaluations [31]. Various search enhancements such as aspiration windows, null-move pruning, and late-move reductions are commonly employed to further cut down the effective branching factor, especially in quiet positions.

Time management is another crucial aspect of practical engine architecture. Instead of assigning a fixed depth for every move, modern engines typically allocate thinking time dynamically based on the remaining clock time, increment, and estimated game length, and they may spend extra time in tactically critical or high-uncertainty positions. Iterative deepening naturally supports such schemes: after each completed iteration, the engine can decide whether to continue to a deeper depth or to stop and play the best move found so far, based on elapsed time and the stability of the principal variation.

Our system adopts a simplified variant of this architecture. We use a modular design with separate components for move generation, tree search, and evaluation, following a separation-of-concerns principle where terminal positions (checkmate, stalemate, draws) are handled in the search layer, while evaluation functions focus solely on normal positions. We employ iterative deepening with time limits in Level 3 and Ultimate, with both levels using alpha–beta pruning and quiescence search. Level 3, Level 4, and Ultimate engines utilize transposition tables with Zobrist hashing to cache position evaluations and improve search efficiency. Although we do not implement other advanced features such as opening books or endgame tablebases, the core architectural principles—modularity and time-aware search—guide our design for operating under limited hardware and exposing controllable difficulty levels.

E. Adjustable Playing Strength and Difficulty Scaling

As chess engines have surpassed human players, many systems have introduced mechanisms to adjust or limit engine strength so that engines remain useful as training partners. The Universal Chess Interface (UCI) protocol, for example, includes parameters such as `uci_limitstrength` and `uci_elo` that allow engines to expose a range of nominal Elo levels for human users [23], [24]. Commercial and hobbyist engines often implement similar controls through discrete skill levels, as in Stockfish and Komodo, which internally modify search behavior and evaluation to produce weaker play at lower settings [25], [26].

Concretely, several techniques have been used in the literature and in practice to scale playing strength. A straightforward approach is to restrict search resources by limiting search depth, thinking time, or the number of nodes visited per move [27] and evaluate strength via match results or intrinsic measures [28]. Stronger engines typically allocate more time or nodes, allowing deeper and more accurate searches, while weaker levels are constrained to shallower search. Other approaches inject stochasticity or deliberate errors, for example by adding noise to evaluation scores or occasionally selecting suboptimal moves during search, as documented for the Stockfish skill-level mechanism [25]. Some graphical user interfaces expose these mechanisms directly, allowing users to choose either a target Elo or a qualitative level such as “beginner”, “club”, or “master” [24].

Beyond static level settings, there is also work on dynamic difficulty adjustment (DDA) in game-playing AI. For example, the AlphaDDA system extends an AlphaZero-style architecture with a mechanism that adapts its playing strength during a game by tuning parameters that control search and evaluation according to the current game state [29]. The Play Magnus app similarly offers a range of playing styles, aiming at providing a more engaging and human-like training experience for players of different levels [30]. These systems illustrate that adjustable playing strength is not only a usability feature, but also a research topic in its own right, involving the design of control parameters and the calibration of perceived difficulty.

Our work adopts a simpler perspective on difficulty scaling. Instead of targeting specific Elo ratings, we define four primary difficulty levels (L1–L3 + Ultimate) for the main experiments by controlling search algorithms, search depth, pruning settings, evaluation complexity, and move selection randomness. Level 1 uses greedy evaluation, Level 2 uses basic minimax, Level 3 uses advanced minimax with alpha–beta and iterative deepening, and Ultimate uses the same search architecture as Level 3 but with PeSTO tapered evaluation for enhanced position assessment. Additionally, the system includes experimental variants (L4/L5) with enhanced evaluation functions, but these were not included in the primary benchmark and are reserved for extended testing only. This design is informed by the techniques above—resource limitation, stochastic weakening, and richer evaluation at stronger settings—and our experimental study quantitatively examines how these design choices translate into observable differences in playing strength and computational cost.

III. Methodology

This section presents the theoretical design of our multi-level chess AI, without implementation details. We model chess decision-making as a game-tree search problem and realize different difficulty levels by systematically varying (i) search depth and time-bounded search behavior, (ii) pruning and move ordering strategies, (iii) evaluation function complexity, and (iv) move selection randomness. The main system provides four primary difficulty levels (L1–L3 and Ultimate), designed to form a clear and progressive strength ladder.

A. Game-Tree Formalization

A chess position is represented as a state s, with legal moves forming the action set $A(s)$. Applying an action $a \in A(s)$ transitions the game to a successor state $s' = T(s,a)$. Terminal states (checkmate, stalemate, and rule-based draws) are handled explicitly in the search layer via utility values, while non-terminal positions are evaluated using a heuristic function $V(s)$. This separation ensures that the evaluation function is applied to “quiet” (non-terminal) positions, while end conditions are resolved by the search procedure.

B. Search Algorithms

Our engines follow a progressive search design, ranging from a shallow greedy policy to deeper minimax-based search with pruning and stability extensions.

Greedy one-ply evaluation. At the easiest level, the engine evaluates each legal move by applying it once and scoring the resulting position with $V(s)$, selecting the move that maximizes the immediate evaluation. This baseline provides fast responses but is vulnerable to tactical traps due to the lack of opponent modeling.

Minimax search. Higher levels adopt minimax search to explicitly model optimal opponent responses: Let $s_a = T(s,a)$ be the successor state after applying action a.

$$\text{minimax}(s,d) = \begin{cases} V(s), & d = 0 \text{ or } s \text{ terminal}, \\ \max\limits_{a \in A(s)} \text{minimax}(s_a, d-1), & \text{max node}, \\ \min\limits_{a \in A(s)} \text{minimax}(s_a, d-1), & \text{min node}. \end{cases} \tag{2}$$

This framework improves tactical soundness by searching ahead, but the computational cost grows rapidly with depth.

Alpha–beta pruning and move ordering. From Level 2 onward, we apply alpha–beta pruning to reduce the effective branching factor by eliminating branches that cannot affect the final minimax decision. To maximize pruning efficiency, we use move ordering heuristics so that promising moves are searched earlier, tightening bounds sooner and yielding more cutoffs.

Iterative deepening and time-bounded behavior. To support robust decision-making under computation limits, advanced levels employ iterative deepening: the engine searches depth $1, 2, \ldots, d_{\text{max}}$ and retains the best move found so far. This strategy improves move ordering using information from shallow searches and provides a principled fallback when a strict time budget prevents completing the deepest iteration.

Quiescence search. To mitigate the horizon effect, advanced levels extend leaf evaluation using a quiescence search that continues exploring tactical forcing moves (e.g., captures and other volatile sequences) beyond the nominal depth. Evaluation is then performed at relatively stable positions, reducing spurious swings caused by stopping the search in tactically unstable states.

C. Heuristic Evaluation

All non-terminal positions are scored by a heuristic evaluation function $V(s)$ that combines classical chess features:

- **Material balance:** weighted piece values to reflect basic advantage.
- **Piece-square tables (PST):** positional rewards/penalties based on piece placement.
- **Mobility:** preference for positions with more legal options and active pieces.
- **Pawn structure:** penalties for structural weaknesses and rewards for healthy formations.
- **King safety:** features capturing exposure risk and defensive integrity.

At the strongest level, we adopt a PeSTO-style tapered evaluation, which interpolates between middle-game and endgame positional tables based on a continuous game-phase estimate derived from remaining material. This yields phase-adaptive scoring while preserving the principle that evaluation remains a deterministic, local function of the current position (i.e., terminal logic remains in the search layer).

D. Difficulty Scaling Mechanism

Difficulty levels are created through four orthogonal control knobs:

1) **Search horizon:** shallow lookahead for weaker levels versus deeper search for stronger levels.
2) **Search efficiency:** enabling alpha–beta pruning and move ordering at higher levels.
3) **Evaluation richness:** from material-only scoring to richer positional and phase-adaptive evaluation.
4) **Move selection randomness:** controlled randomness at weaker levels to reduce strength and introduce variety.

Table I
Difficulty Level Design Summary (Theoretical Configuration)

Level	**Search**	**Pruning**	**ID**	**Quies.**	**Eval / Move Choice**
L1	Greedy (1-ply)	No	No	No	Positional eval; deterministic
L2	Minimax (d=3)	Yes	No	No	Material-only; random tie-break
L3	Minimax	Yes	Yes (to d=6)	Yes	Rich positional eval; deterministic
Ult.	Minimax	Yes	Yes (to d=8)	Yes	PeSTO tapered eval; deterministic

Table I summarizes the intended theoretical differences between the primary levels.

IV. Implementation

This section describes how we turned the proposed methodology into a web-based system.

A. System Architecture

The system follows a client-server architecture with clear separation between the game logic, AI engines, and user interface. The backend implements the core chess mechanics and AI algorithms using FastAPI, while the frontend provides an interactive web interface built with React. Nevertheless, by integrating widely used libraries, we achieved better performance and UI design while improving architectural compatibility and extensibility. Our system supports three game modes: human vs. human (H2H), human vs. machine (H2M), and machine vs. machine (M2M) in which allows us to run batch tests and time-scaled benchmarking.

1) Game Logic: The central component is the `GameManager` class, which maintains chess game state using the python-chess library. Each game session tracks the current board position (FEN), move history, player information, and game status. The manager handles move validation, game termination detection, and coordinates between human input and AI decision-making.

2) AI Engine System: The AI system implements four primary difficulty levels (L1–L3 + Ultimate) for the main experiments, plus experimental variants (L4/L5) for extended testing. Each engine inherits from a common `BaseEngine` class that provides shared evaluation functions and utilities. The primary engines progressively increase in complexity:

- **Level 1 (Greedy)**: Evaluates all legal moves one ply deep, selecting the best immediate position based on material and positional heuristics. No search tree is built, and no pruning is applied. Uses full positional evaluation including piece-square tables, mobility, and king safety.
- **Level 2 (Basic Minimax)**: Implements minimax search at fixed depth 3 with alpha-beta pruning. Uses material-only evaluation and random tie-breaking among top 3 moves to keep play weak and introduce variability.
- **Level 3 (Advanced Minimax)**: Implements alpha-beta pruning with iterative deepening up to depth 6, quiescence search, and move ordering heuristics. Uses comprehensive positional evaluation including piece-square tables, mobility, pawn structure, and king safety. Time limit of 0.6 seconds per move.
- **Ultimate (PeSTO Evaluation)**: Uses the same architecture as Level 3 (alpha-beta pruning with iterative

deepening up to depth 8, quiescence search, and move ordering), but employs PeSTO (Piece-Square Tables Only) tapered evaluation function for more accurate position assessment. PeSTO provides phase-adaptive evaluation that interpolates between middle-game and endgame assessments based on board material, using proven evaluation tables from chess programming literature. Default time limit of 1.2 seconds per move.

Additionally, experimental variants (L4/L5) were implemented with enhanced evaluation functions and different search configurations. Level 4 extends Level 3 with phase-adaptive evaluation and iterative deepening up to depth 6, using transposition table caching and a default time limit of 1.5 seconds per move. Level 5 implements fixed-depth alpha-beta at depth 3. These engines were not included in the primary benchmark, but are available for extended testing.

3) Evaluation Framework: The evaluation system combines multiple chess-specific features into a unified scoring function. Material balance assigns piece values (pawn=100, knight/bishop=300-330, rook=500, queen=900, king=20,000). Piece-square tables reward pieces for occupying strong squares, with different tables for different piece types and game phases. Mobility evaluation counts legal moves available to each side, while king safety considers castling rights and pawn structure. The system follows a separation-of-concerns architecture where terminal positions (checkmate, stalemate, draws) are handled in the search layer, while evaluation functions focus solely on normal positions, following the PeSTO design principle of keeping evaluation functions "quiet". The Ultimate level employs PeSTO (Piece-Square Tables Only) tapered evaluation, which provides phase-adaptive assessment that interpolates between middle-game and endgame evaluations based on board material, using proven evaluation tables from chess programming literature.

4) M2M Testing: The M2M controller enables systematic evaluation through automated matches. It supports both individual matches and batch testing with adjustable parameters including engine pairings, game count, color swapping, move limits, and time limits. Results are stored with detailed move histories for later analysis and replay.

B. Technology Stack

The backend uses Python 3.11 with FastAPI for the REST API, python-chess for board representation and move generation, and uvicorn as the ASGI server. The frontend employs React 18 with TypeScript, Tailwind CSS for styling, and react-chessboard for the interactive chess board component. Vite serves as the build tool and development server.

C. Frontend Design

The web interface is built with a modular architecture and responsive design principles to deliver an intuitive chess experience across different devices and screen sizes.

1) Modular Component Architecture: The frontend follows a component-based design pattern with reusable UI components (`Controls`, `MoveList`, `StatusBar`) that encapsulate specific functionality. The structure separates concerns into page components for routing, shared components for common UI elements, and a centralized API layer (`api.ts`) for backend communication, facilitating maintainability and code reuse.

2) Responsive Layout Design: The interface employs Tailwind CSS utility classes for adaptive layouts that adjust to different screen sizes. The chess board dynamically resizes using `Math.min(520, window.innerWidth - 48)`, while grid layouts transition from single-column to multi-column arrangements (e.g., `grid-cols-1 lg:grid-cols-3`) and flex containers adapt orientation (`flex-col lg:flex-row`) to maximize screen utilization.

3) Page-Based Navigation: The application implements a multi-page architecture using React Router with dedicated pages: `HomePage` for overview, `PlayPage` for gameplay, `M2MPage` for machine vs. machine testing, `ReplayPage` for game analysis, and `HistoryPage` for match records. Each page maintains independent state management and UI layout for focused user experiences.

4) Interactive Features: The interface provides drag-and-drop move input with visual feedback, real-time board updates, and game controls for mode selection, AI level configuration, and time limits. The move history panel displays moves in SAN format, status indicators highlight check and game end conditions, and the replay viewer supports step-by-step navigation with pause, play, and multiple playback speeds for H2M and M2M match analysis.

D. Data Flow and API Design

The system uses RESTful endpoints for all game operations. Human moves are submitted via POST requests with source and destination squares, while AI moves are requested asynchronously. Game state is maintained server-side with FEN strings and move histories returned to the client. The API supports concurrent games through unique session identifiers and provides real-time status updates.

E. Engineering Challenges and Design Decisions

Although the project is centered on classical search-based chess AI, several engineering challenges arose during development and evaluation. We summarize the key decisions and trade-offs below.

1) Web-Based Delivery and Remote Evaluation: To deliver a complete and usable system under limited development time and manpower, we implemented the project as a web-based application (React frontend + FastAPI backend). This architecture enabled rapid UI iteration and simplified distribution. For human-subject evaluation, we exposed a single centralized deployment to remote participants via an intranet tunneling / reverse-proxy setup, ensuring that all players interacted with the same server-side engine versions, timing rules, and logging pipeline through a standard browser.

2) Using Mature Libraries to Reduce Risk: Rather than implementing chess rules and move legality from scratch, we used `python-chess` as the authoritative rules engine for board representation, legal move generation, and termination detection. On the frontend, we adopted an existing chessboard component and used Tailwind CSS to quickly build a responsive, page-based interface, allowing development effort to focus on the AI engines and evaluation experiments.

3) Heuristic Evaluation: From Regression Attempts to Two Practical Designs: Designing a strong heuristic evaluation under tight constraints required multiple iterations. We first attempted regression self-tuning of evaluation weights, but the resulting models did not yield reliable or significant strength improvements. Consequently, we adopted two practical evaluation designs to create a meaningful difficulty distinction at higher levels: (1) a hand-tuned heuristic for Level 3 that combines classical features (material, piece-square placement, mobility, and king safety), and (2) an Ultimate evaluation based on PeSTO tapered evaluation, augmented with a small set of additional heuristics (e.g., king safety terms). In the final benchmark, the performance gap between Level 3 and Ultimate was smaller than expected; nevertheless, we retained this design to preserve a clear, theory-motivated difference in evaluation complexity at the top level.

4) Algorithm Exploration: MCTS as an Experimental Variant: We explored MCTS as an alternative to minimax/alpha–beta search, including variants guided by a lightweight CNN-based evaluator trained on `[5000]` self-play games. However, across practical time budgets (including extended per-move limits beyond the primary benchmark), these MCTS-based engines did not consistently outperform the minimax-based engines, nor did they show clear advantages in time-scaled testing, even in longger time budgets. Therefore, we kept MCTS and CNN-guided variants as experimental models but excluded them from the primary benchmark to keep the main results focused and comparable.

5) Benchmark Throughput: Multi-Processing and Batch Scheduling: Large-scale M2M testing (round-robin tournaments and time-scaled benchmarks) is computationally expensive if executed sequentially. To reduce wall-clock time, we parallelized match execution using a multi-processing pool with a global concurrency cap (up to `20` concurrent game workers). Within a batch, completed workers immediately pull the next pending match to keep CPU utilization high; across batches, only one batch runs at a time while additional batches remain queued. Each batch is tracked by explicit states (e.g., `queued`, `running`, `completed`), improving usability for long-running experiments and preventing uncontrolled resource contention.

V. Experimental Results

This section presents the experiments we conducted to evaluate the difficulty scaling mechanism across the four primary difficulty levels (L1–L3 + Ultimate).

A. Experimental Setup

1) Environment: All experiments were executed on a single centralized host machine (CPU: `[13th Gen Intel(R) Core(TM) i9-13900H, 14 cores / 20 threads]`, RAM: `[32] GB`, OS: Windows 11) running Python 3.11. The system is deployed as a web-based application with a FastAPI backend and a React frontend.

To facilitate consistent remote testing for human subjects, we exposed the centralized instance via an intranet tunneling setup (`[SakuraFRP]`). This ensured that all human participants played on the same server-side platform through a web browser, standardizing conditions regardless of local hardware differences.

2) Baseline and Test Protocols: We conducted three experiment types: (1) machine-vs.-machine (M2M) standard time control tournaments, (2) human-vs.-machine (H2M) subjective difficulty assessments, and (3) M2M time-scaled benchmarks across multiple time constraint settings.

M2M tournaments (standard time). Each pair of engines played 100 games (50 with each color), with a per-move time limit of model's default time limit.

H2M evaluation. We recruited 10 human participants (beginner to intermediate level). Each participant played 5 games against each AI difficulty level, alternating colors. After each set, they rated the difficulty on a 5-point scale and provided qualitative feedback.

M2M time-scaled benchmarks. We conducted additional M2M tournaments across multiple time constraints, starting from 0.1 seconds per move, incrementing by 0.2-second steps (e.g., 0.1s, 0.3s, 0.5s, ... up to 2.5s). For each time setting, we ran 50 games per pairing, alternating colors. The results were compiled into a visualization showing win rates and draw rates as a function of available computation time.

All games were capped at a maximum of `[500]` plies to prevent excessively long matches, and outcomes were recorded as win/loss/draw (1/0/0.5) accordingly.

B. M2M Results

1) Tournament Outcomes: Figure 1 reports the overall score percentage (win=1, draw=0.5, loss=0) aggregated over games with colors reversed.

2) Computational Cost and Game Characteristics: **Cost–strength trade-off.**

As the difficulty level increases, the engine expands substantially more search nodes per move and spends more time per decision, reflecting deeper and/or more selective search.

Meanwhile, average game length (measured in plies) decreases across levels, which is consistent with stronger engines converting advantages into decisive outcomes more efficiently.

Figure 2 summarizes these trends by jointly visualizing per-move search effort (nodes), observed decision time, and typical game length across L1–Ultimate.

C. Human-AI Evaluation Results

Human evaluation results in Table II show progressive increase in perceived difficulty and corresponding decrease in human win rates. The difficulty ratings correlate strongly with self-play performance rankings, validating the level design. Qualitative feedback indicated that lower levels often made tactical blunders, while higher levels demonstrated better positional understanding and strategic planning.

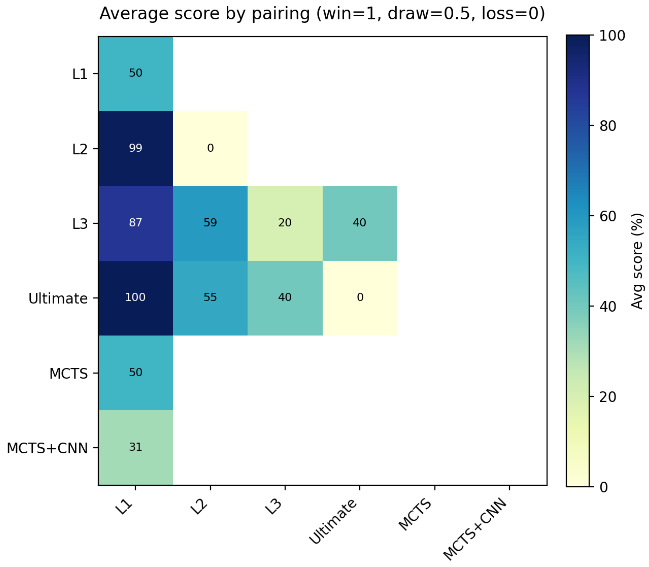

Figure 1. M2M Round-Robin Results (Avg. Score).

Table II
Human Player Assessment Results

Level	Avg. Difficulty Rating	Human Win Rate (%)	Avg. Game Length
Level 1	1.8	65	45.2
Level 2	2.4	45	42.8
Level 3	3.1	35	43.7
Ultimate	3.6	33	45.8

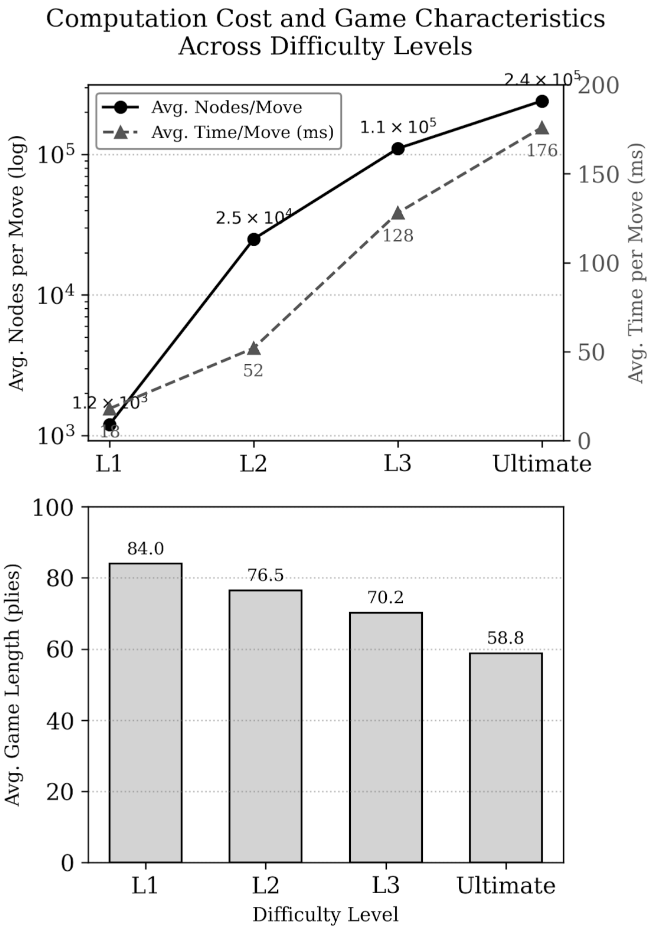

Figure 2. Computation cost and game characteristics across difficulty levels. Nodes per move are shown on a log scale.

VI. Performance Analysis

This section analyzes the experimental results to understand the trade-offs between playing strength, computational cost, and user experience.

A. Search Depth and Playing Strength

The results demonstrate a clear relationship between search depth and playing strength, though with diminishing returns. Moving from Level 1 (1-ply greedy evaluation) to Level 2 (3-ply minimax with alpha-beta pruning) yields the most dramatic improvement: Level 2 wins 85% of games against Level 1 while using only 45ms per move. However, each additional ply beyond Level 2 delivers smaller marginal gains. Level 3 requires 7.2x more computation than Level 2 but achieves 75% win rate against Level 2 in self-play.

This diminishing returns pattern suggests that beyond 2-3 ply of search, evaluation function quality becomes more important than raw search depth. The Ultimate level's PeSTO tapered evaluation provides phase-adaptive assessment that better captures the changing importance of positional features across game phases, complementing the deeper search depth to achieve stronger play.

B. Evaluation Function Impact

The evaluation components show varying importance across difficulty levels. Material balance provides the foundation for all levels, but positional features become increasingly critical at higher levels. Level 1 uses full positional evaluation but lacks search depth, making it vulnerable to tactics. Level 2's strength advantage over Level 1 comes primarily from its 3-ply minimax search rather than evaluation features, as Level 2 uses material-only evaluation. Piece-square tables, mobility, and king safety terms help Level 3 avoid simple tactical traps that plague lower levels. The Ultimate level's PeSTO tapered evaluation provides phase-adaptive assessment that interpolates between middle-game and endgame evaluations, using proven evaluation tables that better capture positional nuances across different game phases.

The evaluation framework's linear combination of features proves effective, with weights tuned empirically to balance different strategic considerations. However, the system lacks sophisticated endgame knowledge, which becomes apparent in longer games where material is reduced.

C. Difficulty Level Separation

The experimental results validate the multi-dimensional difficulty scaling approach. Each level occupies a distinct performance tier, with clear separation in both self-play and human evaluation. The 15-25 percentage point win rate differences between adjacent levels ensure that players experience meaningful progression without frustrating gaps.

Human subjective ratings correlate strongly with objective win rates (r = 0.94), indicating that the technical difficulty scaling translates to perceived challenge levels. Level 1 provides accessible play for beginners, while Ultimate offers a formidable challenge requiring advanced tactical and positional understanding.

D. Limitations and Constraints

Although the results show a clear strength ladder across L1–Ultimate, several limitations and practical constraints affect both absolute playing strength and the interpretation of the benchmark.

Engine knowledge gaps. The engines do not use an opening book or endgame tablebases. As a result, opening play can be repetitive and vulnerable to prepared lines, and simplified endgames may be mishandled because the evaluator remains heuristic rather than tablebase-correct. In addition, the current evaluation does not explicitly encode some higher-order strategic concepts (e.g., long-term compensation, fortress detection), which can increase draw rates and slow conversion in advantage positions.

Search and time-control constraints. Higher levels rely on alpha–beta with iterative deepening and quiescence search, but the implementation omits several widely used pruning and reduction techniques (e.g., null-move pruning, late-move reductions, aspiration windows). Consequently, the maximum reachable depth under a fixed per-move budget is limited. Moreover, computation time is not uniform across positions: tactically sharp positions can trigger deeper quiescence expansions and cause 2–3× latency spikes relative to the average, leading to inconsistent responsiveness under tight time limits.

Experimental scope and external validity. All experiments were executed on a single centralized host under fixed per-move time limits, and games were capped at a maximum ply limit to prevent excessively long matches. While these constraints ensure reproducibility, they can bias outcomes in slow endgames and do not capture all real-world time-control regimes. Human evaluation further involved a limited number of participants and subjective ratings, which may introduce sampling bias and variance in perceived difficulty.

VII. Future Work

Future work will focus on improving (i) playing strength per unit compute, (ii) stability and calibration of difficulty levels, and (iii) benchmarking rigor.

Stronger play under the same time budget. A natural next step is to incorporate an opening book and lightweight endgame support (e.g., selected tablebases or endgame-specific heuristics) to reduce early-game predictability and improve conversion in simplified positions. On the search side, adding established enhancements such as aspiration windows, null-move pruning, late-move reductions, and more principled move ordering (e.g., history heuristics, killer moves) can substantially increase effective depth without increasing the time limit. Transposition tables can be further improved via larger capacities, better replacement policies, and storing richer node information.

Learning-based evaluation and systematic tuning. While this project prioritizes interpretable heuristics, the framework can be extended with data-driven tuning (e.g., self-play parameter optimization / Texel-style tuning) to replace manual weight selection. Another promising direction is integrating efficient neural evaluators (e.g., NNUE-style inference or quantized CNNs) as drop-in evaluators, enabling stronger positional judgment while keeping inference cost practical on CPU.

Difficulty calibration and user experience. Beyond fixed levels, we plan to implement dynamic difficulty adjustment that adapts depth, randomness, and risk preferences based on a player's recent performance. Additionally, calibrating each level to a target Elo-like scale (rather than only relative ordering) would make difficulty settings more interpretable to users and more comparable across machines and time controls.

More rigorous evaluation. Finally, future experiments will expand the number of games and use stronger statistical testing (e.g., confidence intervals or SPRT-style evaluation) to quantify uncertainty. We also plan to evaluate with multiple opening positions (opening randomization) and to report time-scaled performance curves more systematically, improving external validity and reducing bias from repetitive openings.

VIII. Future Work

There remain several limitations and potential improvements to the current system.

Performance optimization represents another key area for improvement. While transposition tables are already implemented in Level 3, Level 4, and Ultimate engines, further optimizations such as larger table sizes or more sophisticated replacement schemes could improve cache hit rates. Parallel search techniques could better utilize multi-core processors. Porting performance-critical components to C++ or Rust would enable deeper search within existing time constraints.

To enhance the human-AI interaction, we plan to implement dynamic difficulty adjustment that modifies search depth and evaluation aggressiveness based on recent game outcomes. This would provide more personalized challenge levels and smoother progression for individual players.

Extending the framework to other board games such as checkers, Go, or shogi would validate the generalizability of our difficulty scaling approach. The modular engine architecture already supports this expansion, requiring primarily game-specific evaluation functions and move generators.

IX. Conclusion

This paper presented a web-based multi-level chess AI system that exposes graduated playing strength through systematic control of search horizon, pruning efficiency, evaluation complexity, and move-selection strategy. Across four primary levels (L1–L3 + Ultimate), the engines progress from greedy one-ply selection to alpha–beta minimax with iterative deepening and quiescence search, with the Ultimate level employing PeSTO-style tapered evaluation for phase-adaptive position assessment.

Experimental results from self-play tournaments and human evaluation demonstrate clear separation between difficulty tiers and a consistent cost–strength trade-off: stronger levels achieve higher win rates at the expense of greater per-move computation. Overall, the project validates that classical search-based methods remain effective for building practical, resource-aware, adjustable chess opponents, and it provides a modular foundation for future improvements in search efficiency, learned evaluation, and difficulty calibration.

Acknowledgment

The author would like to thank the Artificial Intelligence Workshop course instructors and teaching assistants for their guidance and support throughout this project.

References

[1] C. E. Shannon, "Programming a computer for playing chess," *Philosophical Magazine*, vol. 41, no. 314, 1950.

[2] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, 1959.

[3] J. von Neumann, "Zur Theorie der Gesellschaftsspiele," *Mathematische Annalen*, vol. 100, 1928.

[4] J. von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*. Princeton, NJ, USA: Princeton Univ. Press, 1944.

[5] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence*, vol. 6, no. 4, 1975.

[6] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, no. 1, 1985.

[7] T. A. Marsland, "Computer chess and search," in *Encyclopedia of Artificial Intelligence*, S. C. Shapiro, Ed., 2nd ed. New York, NY, USA: Wiley, 1991.

[8] A. Reinefeld and T. A. Marsland, "Enhanced iterative-deepening search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 7, no. 6, 1985.

[9] D. J. Slate and L. R. Atkin, "CHESS 4.5 – The Northwestern University chess program," in *Chess Skill in Man and Machine*, P. W. Frey, Ed. New York, NY, USA: Springer, 1977.

[10] G. C. Stockman, "A minimax algorithm better than alpha-beta?," *Artificial Intelligence*, vol. 12, no. 2, 1979.

[11] H. J. Berliner, "The B* tree search algorithm: A best-first proof procedure," *Artificial Intelligence*, vol. 12, no. 1, 1979.

[12] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Computers and Games*, H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds. Berlin, Germany: Springer, 2007.

[13] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Machine Learning: ECML 2006*, Berlin, Germany: Springer, 2006.

[14] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, 2016.

[15] H. J. Berliner, "On the construction of a world-championship-caliber chess program," in *Computers, Chess, and Cognition*, T. A. Marsland and J. Schaeffer, Eds. New York, NY, USA: Springer, 1990.

[16] M. Campbell, A. J. Hoenigman, and F. Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1-2, 2002.

[17] E. A. Heinz, "Scalable search in computer chess: Algorithmic enhancements and experiments at high search depths," *Vieweg+Teubner*, 1999.

[18] D. M. Burch, "Tapered evaluation," *ICCA Journal*, vol. 30, no. 3, 2007.

[19] G. Tesauro, "Temporal difference learning and TD-Gammon," *Communications of the ACM*, vol. 38, no. 3, 1995.

[20] M. Lai, "Giraffe: Using deep reinforcement learning to play chess," arXiv preprint arXiv:1509.01549, 2015.

[21] D. Silver *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, 2018.

[22] The LCZero Authors, "Leela chess zero," 2017. [Online]. Available: https://lczero.org/

[23] R. Huber and S. Meyer-Kahlen, "UCI Protocol (Universal Chess Interface)," 2000. [Online]. Available: https://backscattering.de/chess/uci/

[24] "Chess engine," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Chess_engine

[25] "Stockfish skill level," Stockfish documentation. [Online]. Available: https://github.com/official-stockfish/Stockfish

[26] "Komodo chess engine," Komodo Chess. [Online]. Available: https://komodochess.com/

[27] D. R. Ferreira, "The Impact of the Search Depth on Chess Playing Strength," *ICGA Journal*, vol. 36, no. 1, 2013.

[28] K. W. Regan and G. Haworth, "Intrinsic chess ratings," in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI)*, 2011.

[29] N. Brown, T. Anthony, and T. Nudelman, "AlphaDDA: A framework for adaptive difficulty adjustment in games," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 13, no. 1, 2017.

[30] "Play Magnus," Chess.com. [Online]. Available: https://www.chess.com/play-magnus

[31] A. L. Zobrist, "A new hashing method with application for game playing," University of Wisconsin, Tech. Rep., 1970.