# Design of a Multi-Level Chess AI System and performance evaluation of 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. The system implements four distinct difficulty levels by progressively increasing search depth and evaluation complexity. Our approach 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 of/from XXXXX. 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 difficulty levels through systematic control of search depth, pruning aggressiveness, 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 distinct difficulty levels by varying 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 modes for training 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 the lower difficulty levels, we use depth-limited minimax search with alpha–beta pruning and iterative deepening, which provides conceptual clarity and fine-grained control over search depth and pruning behavior. For the highest difficulty level, we employ Monte Carlo Tree Search guided by a CNN-based evaluation, allowing more selective exploration in complex positions under a fixed computation budget. The techniques reviewed above directly inform our choice of search algorithms and our design of depth limits, rollout settings, and time-control strategies across the four 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,

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

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.

Our work follows this line by incorporating a CNN-based evaluator trained on self-play games as the learned component of our hybrid evaluation framework. In the highest difficulty level of our system, this CNN provides position scores or policy hints that guide Monte Carlo Tree Search under a fixed computation budget, while lower levels rely more heavily on theory-driven heuristic evaluation. This hybrid design allows us to explore the trade-off between heuristic and learned evaluation in a resource-constrained, multi-level chess AI setting.

## D. Practical Engine Architectures and Time Management

While the previous subsection focuses on individual search techniques, practical chess engines must integrate these components into a coherent architecture that can operate under real-time constraints. A typical engine is organized into several modules, including position representation and move generation, a search driver with time-management logic, an evaluation module, and auxiliary components such as transposition tables and opening books. The search driver coordinates iterative deepening, alpha–beta search, and move ordering, repeatedly invoking the search routine to increasing depths while monitoring time and node budgets.

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. Transposition tables cache previously evaluated positions keyed by Zobrist hashes, enabling the search to reuse results when the same position is reached via different move sequences. In addition, 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, and employ iterative deepening with basic time limits to control search in the higher difficulty levels. Although we do not implement all advanced features such as full-fledged opening books or endgame tablebases, the architectural ideas from practical engines—modularity, caching, and time-aware iterative deepening—directly inform our design for operating under limited hardware and for 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], [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]. Commercial applications such as the Play Magnus app similarly offer a range of playing styles and strengths aimed at providing a more engaging 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 but related perspective on difficulty scaling. Instead of targeting specific Elo ratings, we define four discrete difficulty levels by jointly controlling search depth, pruning aggressiveness, evaluation complexity, and randomness in move selection, as well as the use of CNN-guided Monte Carlo Tree Search at the highest level. 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.

1) **Organize by Themes** / 按主题分块，而不是按论文顺序：
   建议分成 2–3 个小节，例如：
   - Game-tree search algorithms (Minimax, Negamax, Alpha–Beta, MCTS, etc.)
   - Heuristic evaluation in board-game AI
   - Difficulty scaling and opponent modeling in game AI

   每个小节先用英文一两句说明主题，再总结相关工作。中文用于提示：解释本小节的逻辑和筛选标准。
2) **Three-Sentence Rule** / 每篇文献“三句话原则”：对于重要文献，尽量用 3 句话概括：
   - 做了什么（问题与方法）
   - 怎么做（关键技术点）
   - 好坏点和与你工作的关系（优点/局限/启发）

   这三句可以完全用英文，括号内配一句简短中文解释写作意图。
3) **Critical Analysis** / 批判性分析而不是“流水账”：合理比较不同方法的优缺点，指出例如：
   - 传统手写 heuristic 的可解释性 vs 深度学习评估函数的性能优势；
   - 只调搜索深度的难度分级会导致棋力跨度不均匀；
   - 一些工作缺乏与人类体验相关的评估指标等。
4) **Identify Gaps** / 引出研究空缺：用一段总结：
   - 哪些问题已经被充分研究（你就少做）；
   - 哪些方面相对薄弱（你会重点补足），例如：在教学项目场景下对多等级棋力和资源受限 AI 的系统分析较少。

   最后自然过渡到下一节 Methodology，说明你将如何在这些文献基础上设计自己的系统。

## III. Methodology

This section explains the *theoretical design and algorithms* used in the work, without going into low-level implementation details.

Our multi-level chess AI system is built on a classical game-tree search framework, where each difficulty level is realized through systematic variations in search depth, pruning strategies, evaluation complexity, and move selection rationale. The methodology consists of four main components: (1) formalization of chess as a game-tree search problem, (2) implementation of minimax-based search algorithms with alpha–beta pruning and iterative deepening, (3) design of heuristic evaluation functions combining material, positional, and strategic features, and (4) a principled difficulty scaling mechanism that controls playing strength through parameterized search and evaluation configurations.

The search algorithms form the core decision-making engine, progressing from basic minimax at Level 1 to sophisticated iterative deepening with quiescence search at higher levels. We employ alpha–beta pruning to reduce the effective branching factor, move ordering heuristics to maximize pruning efficiency, and iterative deepening to provide natural time management and robustness under computational constraints. The quiescence search extension addresses the horizon effect by continuing to explore tactical sequences (primarily captures) beyond the nominal search depth, ensuring that evaluation occurs at relatively stable positions.

The evaluation framework and difficulty design work together to create distinct playing strengths. Our evaluation functions combine theory-driven positional heuristics (material balance, piece-square tables, mobility, pawn structure, and king safety) with systematic feature weighting. Difficulty scaling is achieved through four orthogonal mechanisms: (1) search depth variation (from depth 2 at Level 1 to iterative deepening up to depth 5 at Ultimate), (2) pruning strategy (no pruning at Level 1, full alpha–beta at higher levels), (3) evaluation complexity (material-only at Level 1, progressively richer features at higher levels), and (4) move selection randomness (random tie-breaking among top moves at Level 1, deterministic best-move selection at higher levels). This multi-dimensional approach ensures clear separation between difficulty levels while maintaining computational efficiency.

本节是「方法论/算法设计」，偏抽象（conceptual + mathematical），而不讨论具体代码文件。

可以分为如下子节（仅示例）：

### A. Game Formalization

用英文形式化游戏模型，例如 state, action, transition, terminal condition 等。可以写成：

- Define the board representation, players, legal moves, and winning conditions.
- Optionally introduce a game tree notation, where nodes represent states and edges represent moves.

中文注释：说明这是在帮读者建立一个统一的符号与问题设定（问题建模）。

### B. Search Algorithm

在此描述 Minimax/Negamax/Alpha–Beta 之类：

- 用文字 + 必要的伪代码（pseudo-code）展示递归结构和剪枝逻辑。
- 解释算法复杂度和对分支因子、深度的敏感性（大 O 分析可选）。
- 中文可以解释写作层面：突出「为什么选择这个算法」以及「与课程内容的对应关系」。

### C. Heuristic Evaluation Function

介绍你的评估函数设计：

- 用数学表达式给出评分公式，例如 $Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots$。
- 定义各个特征 $f_i(s)$ 的含义（比如连珠数、威胁模式、材料优势等）。
- 讨论权重 $w_i$ 的设定方式（经验设定、简单调参、数据统计等）。
- 中文写作提示：强调「为什么这些特征合理」「与文献中的特征设计有何联系」。

### D. Difficulty Design

说明多等级难度如何实现（difficulty scaling）：

- For each level, specify search depth, evaluation complexity, randomness, and any additional constraints.
- Explain the rationale behind the design, e.g., “Level 1 trades playing strength for faster response time to suit beginners.”
- 中文提示：这里要让设计看起来有理论依据，而不是“瞎调参数”，可以引用文献中的典型做法作为支撑。

## IV. Implementation

This section describes how the methodology is turned into an actual software system.

本节偏工程实现（“How we built it”），可以包含结构图、模块划分等。

建议内容：

- **Architecture Overview** / 系统架构总览：用一张图或一小段，说明前端、后端、AI 引擎、数据模块之间的关系。中文提示：清楚展示数据流（棋局状态如何传递、AI 如何被调用）。
- **Modules** / 核心模块：用小段分别介绍各个模块，例如 `Board`, `Search`, `Evaluation`, `UI` 等。说明每个模块的职责，而不是列代码。
- **Technologies** / 使用的技术栈：列出编程语言、框架和主要第三方库，解释选择理由（简洁、课程要求、易于部署等）。
- **Game Modes** / 对战模式实现：说明 H2M（human vs. machine）、M2M（machine vs. machine）是如何在架构中实现的，如何记录对局数据用于实验。
- **Frontend/UI Design** / 前端设计：简述棋盘渲染、走子交互、计时/悔棋/提示按钮、走子记录（PGN/简单列表）、以及前端如何调用后端 API/引擎（同步/异步、长轮询或本地调用）。若采用桌面 UI（如 PyQt/Tkinter）或浏览器 UI（如 React/Vue），说明选择理由与部署方式。

写作提示：前端部分保持高层描述，强调交互流与状态同步，而非低层代码实现；可用一张示意图表示前端与后端的事件流（用户输入 $\rightarrow$ 请求引擎 $\rightarrow$ 返回最佳着法/评估 $\rightarrow$ 更新棋盘与日志）。

写作时注意：尽量用结构图和清晰的小节标题帮助读者快速理解整体系统，而不是陷入具体代码细节。

## V. Experimental Results

This section presents the experiments and quantitative/qualitative results.

本节展示实验设计与结果，是支撑你结论的「证据」。

建议分两块：

### A. Experimental Setup

介绍实验设置：

- **Environment** / 实验环境：例如 CPU 型号、内存、操作系统、编程语言版本等。
- **Baselines** / 基线对手：例如 random agent、greedy one-step agent 或文献中的实现。中文提示：说明选择这些 baselines 的合理性。
- **Protocols** / 对战设定：例如每组对战 50 盘，双方轮流执先，时间限制等。

### B. Results and Observations

展示并分析结果：

- 使用表格和图（胜率、平均步数、搜索节点数、思考时间等）呈现数据。
- 每张图/表后用 1–2 段英文解释关键观察点，例如强度随深度上升、不同 level 的分层是否明显。
- 中文提示：避免只“报数字”，要回答“这些结果说明了什么？是否和预期一致？如不一致，原因可能是什么？”。

## VI. Performance Analysis

This section goes beyond raw numbers to interpret the performance and trade-offs.

本节是「解释与剖析」，比上一节更偏“为什么”。

可以包括：

- **Depth vs. Strength** / 搜索深度与棋力的关系：讨论深度增加带来的胜率提升和时间开销，指出收益递减点。
- **Heuristic Impact** / 评估函数影响：对比不同评估配置下的表现，说明哪些特征最关键。
- **Difficulty Separation** / 难度分级有效性：说明 Level 1,2,3 之间是否形成清晰的棋力阶梯，并结合数据解释。
- **Limitations** / 局限性：诚实讨论系统的不足，如对特定开局脆弱、对长远战术不敏感等。中文提示：这部分为后面的 Future Work 做铺垫。

写作时：多用“因果语句”和“对比语句”，例如“Because we increase X, the agent tends to Y⋯”，而不仅是“X is bigger than Y.”。

## VII. Future Work

This section outlines plausible extensions and improvements.

本节写「如果有更多时间/资源，我们会做什么」，要合理、不虚。

推荐写：

- 把现有系统扩展到更复杂的棋类或更大棋盘；
- 引入强化学习或深度学习来自动学习评估函数；
- 优化工程性能（C++ 重写、并行搜索、GPU 加速等）；
- 加入更“人性化”的错误模型，提升人机对战体验。

中文提示：未来工作不要过于空泛，要与当前系统的局限性一一对应，看起来像是自然的下一步，而不是另一个完全无关的新项目。

## VIII. Conclusion

This section briefly summarizes what has been done and what has been learned.

本节是“收尾总结”，通常 1–3 段。

建议结构：

- **Restate the Problem** / 重申问题：用 1 句重述任务和场景。
- **Summarize the Approach and Results** / 总结方法与结果：用 2–3 句概括核心方法（Minimax + heuristic + difficulty scaling）以及最重要的实验结论。
- **Highlight the Contribution** / 强调贡献和意义：说明本项目在教学、实践或后续研究上的价值，并可用一句话连接到 Future Work。

中文提示：不要在 Conclusion 引入新结果或新细节，只对已有内容做“高层复盘”。

## Acknowledgment

(可选) 致谢本课程老师、助教、队友等。中文提示：简短即可，注意语气正式。

## References

[1] C. E. Shannon, “Programming a computer for playing chess,” *Philosophical Magazine*, vol. 41, no. 314, pp. 256–275, 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, pp. 210–229, 1959.

[3] J. von Neumann, “Zur Theorie der Gesellschaftsspiele,” *Mathematische Annalen*, vol. 100, pp. 295–320, 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, pp. 293–326, 1975.

[6] R. E. Korf, “Depth-first iterative-deepening: An optimal admissible tree search,” *Artificial Intelligence*, vol. 27, no. 1, pp. 97–109, 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, pp. 669–677, 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, pp. 82–118.

[10] G. C. Stockman, “A minimax algorithm better than alpha-beta?,” *Artificial Intelligence*, vol. 12, no. 2, pp. 179–196, 1979.

[11] H. J. Berliner, “The B* tree search algorithm: A best-first proof procedure,” *Artificial Intelligence*, vol. 12, no. 1, pp. 23–40, 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, pp. 72–83.

[13] L. Kocsis and C. Szepesvári, “Bandit based Monte-Carlo planning,” in *Machine Learning: ECML 2006*, Berlin, Germany: Springer, 2006, pp. 282–293.

[14] D. Silver *et al.*, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, 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, pp. 19–36.

[16] M. Campbell, A. J. Hoenigman, and F. Hsu, “Deep blue,” *Artificial Intelligence*, vol. 134, no. 1-2, pp. 57–83, 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, pp. 151–155, 2007.

[19] G. Tesauro, “Temporal difference learning and TD-Gammon,” *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 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, pp. 1140–1144, 2018.

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

[23] S. Gligoric, “Universal chess interface protocol,” 2000. [Online]. Available: http://wbec-ridderkerk.nl/html/UCIProtocol.html

[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] M. Lai, “Strength and skill: The scientific study of chess engine playing strength,” in *Advances in Computer Games*, 2015.

[28] J. Forbes, “Node limits in chess engines,” *ICGA Journal*, vol. 38, no. 1, pp. 33–45, 2015.

[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