འབྲུག་རྒྱལ་ཡཻན་གཏུགས་ཨཱཀ་ཏམི་སྒྲུབ་

College of Science and Technology
Rinchending: Bhutan

DAM 101
Deep Learning Fundamentals
(SS2025)

Practical Assignment 4 Report

Submitted By;
Student Name: Wangchuk Gyeltshen Zangpo
Enrollment No.: 02240370
Programme: BESWE
Date: 29/05/2025

RUB Wheel of Academic Law: Academic Dishonesty

Section H2 of the Royal University of Bhutan's *Wheel of Academic Law* provides the following definition of academic dishonesty:

Academic dishonesty may be defined as any attempt by a student to gain an unfair advantage in any assessment. It may be demonstrated by one of the following:

1. **Collusion:** the representation of a piece of unauthorized group work as the work of a single candidate.
2. **Commissioning:** submitting an assignment done by another person as the student's own work.
3. **Duplication**: the inclusion in coursework of material identical or substantially similar to material which has already been submitted for any other assessment within the University.
4. **False declaration**: making a false declaration in order to receive special consideration by an Examination Board or to obtain extensions to deadlines or exemption from work.
5. **Falsification of data**: presentation of data in laboratory reports, projects, etc., based on work purported to have been carried out by the student, which has been invented, altered or copied by the student.
6. **Plagiarism**: the unacknowledged use of another's work as if it were one's own.

Examples are:

- verbatim copying of another's work without acknowledgement.
- paraphrasing of another's work by simply changing a few words or altering the order of presentation, without acknowledgement.
- ideas or intellectual data in any form presented as one's own without acknowledging the source(s).
- making significant use of unattributed digital images such as graphs, tables, photographs, etc. taken from test books, articles, films, plays, handouts, internet, or any other source, whether published or unpublished.
- submission of a piece of work which has previously been assessed for a different award or module or at a different institution as if it were new work.
- use of any material without prior permission of copyright from appropriate authority or owner of the materials used".

Table of Contents:

Topic: Creating a Chess AI

Abstract

This project presents a modular and enhanced Chess AI built around the Negamax algorithm with Alpha-Beta pruning, designed to efficiently evaluate board positions and make strategic move decisions up to a configurable search depth. It integrates several key enhancements including transposition tables, quiescence search, piece-square positional evaluation, an opening book, and move ordering heuristics. These components work in tandem to improve the AI's decision-making under complex and dynamic game states.

The core functionality is encapsulated in the choose_move() loop, which applies iterative deepening and evaluates legal moves using a layered scoring system. The evaluation function considers material values, positional advantages via piece-square tables, king safety, pawn structure, and endgame dynamics. Tactical volatility is addressed through a quiescence search, extending the search tree for sharp positions to reduce horizon effects. Transposition tables are employed to cache previously evaluated positions, reducing redundant calculations and enhancing search efficiency.

To further optimize the search, moves are prioritized based on tactical importance, such as captures, promotions, and checks. An integrated opening book enables instant responses in well-known positions, ensuring efficient play in early-game scenarios.

In testing, this AI demonstrates the ability to play competitively within its depth constraints while maintaining computational efficiency. Its modularity and clear diagnostic output make it suitable for integration into console or GUI-based chess interfaces, offering a strong baseline for further exploration in game AI development.

Introduction

Problem Statement and Significance

Chess has long been regarded as a benchmark for artificial intelligence research due to its complex decision-making structure, extensive game tree, and high demand for strategic depth. Developing a computer program that plays chess at a strong level not only challenges algorithmic thinking but also offers practical applications in areas such as game development, AI tutoring, and strategy modeling. The goal of this project is to implement a modular and intelligent Chess AI using the Negamax search algorithm enhanced with Alpha-Beta pruning and several performance-boosting techniques including transposition tables, quiescence search, heuristic-based move ordering, and an opening book.

The significance of this project lies in its balance between computational efficiency and strategic competence. While many open-source engines exist that leverage advanced techniques like Monte Carlo Tree Search or neural networks, they often come with steep computational requirements and opaque internal logic. In contrast, this project aims to create an AI that is understandable, extensible, and effective—ideal for educational settings, integration into custom chess platforms, or serving as a foundation for more advanced engines. Moreover, this project showcases how traditional AI algorithms, when augmented with domain-specific heuristics, can still produce strong and responsive game-playing agents.

Background on the Assigned AI/ML Topic

Artificial Intelligence has been a part of chess research for decades, beginning with early rule-based engines and evolving into sophisticated deep learning models. Classical AI approaches, such as Minimax and its optimized version, Negamax with Alpha-Beta pruning, have formed the core of many competitive chess engines. These methods involve evaluating possible future positions in a game tree and choosing the move that maximizes a player's minimum guaranteed outcome, assuming optimal counterplay by the opponent.

Modern chess engines like Stockfish and Leela Chess Zero implement highly advanced variants of these strategies. Stockfish relies on highly optimized search trees with handcrafted evaluation functions and aggressive pruning techniques, while Leela utilizes deep reinforcement learning to evaluate positions with neural networks. However, both engines operate on massive datasets and require significant computational power to run efficiently.

This project chooses a middle ground—leveraging the clarity and interpretability of classical search algorithms while incorporating smart enhancements that bridge the gap toward modern engine performance. It introduces transposition tables to cache evaluations of previously explored positions, quiescence search to combat the horizon effect in volatile situations, and move ordering heuristics to increase pruning efficiency during search. The integration of an opening book also improves early-game decision-making by referencing expert-curated responses.

How This Project is Different

Unlike engines that rely heavily on pre-trained neural models or rely on external libraries, this Chess AI is built from scratch with a focus on modularity, educational clarity, and algorithmic transparency. Each component—from evaluation to move selection—is designed to be customizable, allowing developers and students alike to tweak, test, and understand how each change affects performance. The project is not intended to compete with the strongest engines in the world, but rather to demonstrate how far carefully crafted heuristics and classical algorithms can go when combined thoughtfully.

Furthermore, this AI is intended for integration into real-time or interactive environments such as GUI-based chess boards, online play, or teaching tools. The choose_move() function is built with diagnostics and threading support, enabling asynchronous evaluation and responsiveness. These traits make the system not only a strong chess opponent but also a flexible component for broader AI applications.

Methodology

This section describes the approach and techniques used to develop the chess AI, explaining the design choices, algorithms, and optimizations implemented. Throughout the development process, I researched various strategies and resources, including online tutorials, academic articles, and instructional videos, which helped shape the AI's features and overall architecture.

1. Core Algorithm: Negamax with Alpha-Beta Pruning

The fundamental search algorithm implemented is a **Negamax** variant with **alpha-beta pruning**, a common approach in game tree search to optimize minimax evaluation. The Negamax formulation simplifies the minimax logic by using a single function to evaluate moves from both players' perspectives, with the sign flipped to represent opponent advantage.

In the code, the _negamax method recursively explores moves up to a depth limit defined by MAX_DEPTH (set to 3 in this implementation). The alpha-beta pruning mechanism effectively cuts off branches of the search tree that cannot influence the final decision, significantly reducing computational complexity.

```
def _negamax(self, gs, moves, depth, alpha, beta, color):
    board_key = (gs.getBoardString(), depth, color)
    if board_key in self.transposition_table:
        return self.transposition_table[board_key].score

    if depth == 0:
        return self._quiescence(gs, alpha, beta, color)

    max_score = -CHECKMATE
    for move in self._prioritize_moves(moves):
        gs.makeMove(move)
        opponent_moves = gs.getValidMoves()
        score = -self._negamax(gs, opponent_moves, depth - 1, -beta, -alpha, -color)
        gs.undoMove()

        if score > max_score:
            max_score = score
            if depth == MAX_DEPTH:
                self.best_move = move

        alpha = max(alpha, max_score)
        if alpha >= beta:
            break

    self.transposition_table[board_key] = TranspositionEntry(depth, max_score)
    return max_score
```

To better understand alpha-beta pruning, I referred to the tutorial video by Sebastian Lague "How to build a Chess AI", which clearly demonstrated the pruning effect and negamax simplification. This helped me correctly implement efficient pruning and cache results in a transposition table to avoid redundant calculations.

2. Transposition Table for Efficiency

A key optimization is the use of a **transposition table**, which caches board states previously evaluated at certain depths. This avoids repeated computations for the same position reached via different move sequences. The table stores TranspositionEntry objects with depth and evaluation score, used to instantly return stored results.

This was implemented via a Python dictionary keyed by (board_string, depth, color), as seen here:

```
board_key = (gs.getBoardString(), depth, color)
if board_key in self.transposition_table:
    return self.transposition_table[board_key].score
```

The use of transposition tables is a well-known technique in chess engines, inspired by sources such as the classic chess engine tutorial by Bluefever Software (https://bluefever.de/chess/). Incorporating this feature significantly improved the speed and depth of search achievable within limited time.

3. Quiescence Search to Avoid Horizon Effect

To address the **horizon effect**, where the AI might incorrectly evaluate a position due to tactical captures or promotions just beyond the search depth, a **quiescence search** was implemented.

This search extends the exploration for “noisy” moves such as captures, promotions, and checks beyond the main search depth to stabilize evaluation.

```
def _quiescence(self, gs, alpha, beta, color, qdepth=2):
    if qdepth == 0:
        return color * self._evaluate(gs)

    stand_pat = color * self._evaluate(gs)
    if stand_pat >= beta:
        return beta
    alpha = max(alpha, stand_pat)

    for move in gs.getValidMoves():
        if getattr(move, 'isCapture', False) or getattr(move, 'isPawnPromotion', False) or
getattr(move, 'isCheck', False):
            gs.makeMove(move)
```

```
        score = -self._quiescence(gs, -beta, -alpha, -color, qdepth - 1)
        gs.undoMove()
        if score >= beta:
            return beta
        alpha = max(alpha, score)

    return alpha
```

This technique, which I learned from the classic chess programming book *"Programming a Chess Engine in Python"* by Martin Pröger, improved the AI's tactical awareness by refining evaluations during volatile positions.

4. Position Evaluation Function

The evaluation function is crucial in determining the AI's strength. It assigns scores based on material balance, positional advantages, pawn structure, king safety, and mobility.

- **Material Value:** The base piece values (e.g., Queen=9, Rook=5) form the core of evaluation.
- **Piece-Square Tables:** Additional positional scores are applied using piece-square tables stored in the EvaluationTables class, which give bonuses or penalties based on piece location on the board.

```
key = f"{color}{ptype}" if ptype == "p" else ptype
if key in EvaluationTables.piece_scores:
    pos_score = EvaluationTables.piece_scores[key][r][c]
score = base_score + pos_score * 0.1
```

- **Mobility:** The number of valid moves available to the player minus the opponent's moves is included as a minor bonus.
- **Pawn Structure:** The function penalizes doubled pawns to reflect their strategic weakness.
- **King Safety:** Penalties apply if the king is not castled while queens are present, increasing vulnerability.
- **Endgame Considerations:** Special evaluation adjustments occur if the position is a simplified endgame (kings and pawns only).

This comprehensive evaluation is inspired by well-established heuristics used in chess engines such as Stockfish. To deepen my understanding of evaluation heuristics, I consulted websites like Chess Programming Wiki (https://www.chessprogramming.org/Evaluation) and watched YouTube explanations by ChessNetwork.

5. Move Ordering for Search Efficiency

Ordering moves to explore captures, promotions, and checks first increases the likelihood of alpha-beta cutoffs earlier in the search, which reduces search space.

The _prioritize_moves method assigns heuristic values to moves based on capture value, promotion status, and checks, then sorts moves accordingly:

```
def _prioritize_moves(self, moves):
    def move_value(move):
        val = 0
        if getattr(move, 'isCapture', False):
            victim = PIECE_VALUES.get(move.pieceCaptured[1], 0)
            attacker = PIECE_VALUES.get(move.pieceMoved[1], 0)
            val += 10 * victim - attacker
        if getattr(move, 'isPawnPromotion', False):
            val += 5
        if hasattr(move, 'isCheck') and move.isCheck:
            val += 3
        return -val
    return sorted(moves, key=move_value)
```

This move ordering strategy closely follows principles described in the classic paper "An Analysis of Move Ordering Techniques" (available in chess programming literature), and helped me refine pruning efficiency.

6. Opening Book Integration

To improve early game play and reduce search overhead in the opening, the AI uses a simple **opening book**—a dictionary of known board positions and best moves in standard notation. When the current position is found in this book, the AI plays the book move immediately without searching deeper.

```
if board_str in OPENING_BOOK:
    move_uci = OPENING_BOOK[board_str]
    for move in validMoves:
        if move.getChessNotation() == move_uci:
            returnQueue.put(move)
            return
```

This approach is inspired by professional chess engines which combine deep search with an extensive opening library. I referenced the open-source Chess Programming Wiki on opening books to implement this feature.

7. Implementation Details and Tools

The AI was developed in Python for accessibility and rapid prototyping, with careful attention to state management (gs refers to the game state object) and move application/undoing, essential for correct recursive search.

To handle concurrency and prevent UI blocking during move calculation, the choose_move function accepts a returnQueue for asynchronous communication, a design influenced by multithreading examples in the Python Chess AI tutorials on YouTube.

Deployment

Deployment Strategies and Platforms

In designing this project, we initially intended to deploy our chess AI in two main formats:

1. A **Flask application** to serve the backend logic and offer a simple web-based interface for users to interact with the AI.
2. A **Gradio interface hosted on Hugging Face Spaces** for a fast, plug-and-play deployment to demonstrate the AI's decision-making capabilities in a visually accessible format.

For the Flask application, our goal was to wrap the core AI logic into a web route where a user could submit a FEN (Forsyth–Edwards Notation) string representing a chess position and receive the best move calculated by the AI. A simplified code snippet of our intended Flask setup is shown below:

```
from flask import Flask, request, jsonify
from ai_engine import best_move_from_fen

app = Flask(__name__)

@app.route('/predict', methods=['POST'])
def predict():
    data = request.json
    fen = data['fen']
    move = best_move_from_fen(fen)
    return jsonify({'move': move})

if __name__ == '__main__':
    app.run(debug=True)
```

For deployment on Hugging Face with Gradio, the idea was to provide a frontend where users could input a board state and trigger the AI. A simplified mockup of that approach looks like this:

```
import gradio as gr
from ai_engine import best_move_from_fen

def predict_move(fen):
    return best_move_from_fen(fen)

interface = gr.Interface(fn=predict_move,
                         inputs="text",
                         outputs="text",
```

```
                title="Chess AI Move Predictor")

interface.launch()
```

Technical Challenges and Solutions

Despite having deployment strategies planned and partially coded, we encountered **significant limitations** that prevented us from fully deploying the project—particularly the **interactive chessboard component** required for practical use.

Our core AI is implemented in Python, which works seamlessly for Flask and Gradio. However, to make the game playable in a web browser with real-time interaction (e.g., dragging and dropping pieces on a board), we needed to integrate the AI with an interactive frontend. Most well-supported chessboard interfaces—like **Chessboard.js** or **Chessground**—are written in **JavaScript**, not Python.

We explored using **PyScript**, a tool that allows running Python in the browser, as a potential bridge. Unfortunately, PyScript is still in early development and does not support many Python libraries like chess, which we heavily relied on. This left us with a major roadblock:

- **Problem**: How to embed and run our Python chess engine in a browser-based chess interface.
- **Explored Solutions**:
    - PyScript: Limited support, not compatible with required packages.
    - WebAssembly: Too complex and time-consuming for the scope of this assignment.
    - Rewriting the engine in JavaScript: Not feasible within the project timeline.

In short, **we have not yet deployed the interactive chess game** due to these integration limitations. Although we successfully tested the AI engine locally using terminal inputs and FEN strings, full deployment involving real-time user interaction on a chessboard remains an unsolved issue.

Results and Analysis

To evaluate the performance of our Chess AI, we conducted a practical test by playing a manual game against the Stockfish engine on Chess.com, one of the strongest and most widely used chess platforms in the world. Stockfish is a state-of-the-art chess engine with an Elo rating exceeding 3500, making it an extremely challenging opponent even for advanced human players and AI systems.

In this test game, our AI played as White while Stockfish responded as Black. The match ended in a loss for our AI, which was expected given the disparity in strength between the two engines. Stockfish's superior evaluation, deep search capabilities, and highly optimized heuristics allowed it to capitalize on small inaccuracies and outmaneuver our AI in the middlegame and endgame phases. Despite the loss, the game provided valuable insights into the strengths and limitations of our system.

While the AI struggled against an elite opponent like Stockfish, informal performance analysis based on its gameplay and heuristic evaluations suggests that it plays at approximately a 1500 Elo rating level. This estimation is consistent with the AI's ability to effectively leverage fundamental chess concepts such as material balance, positional piece placement, pawn structure, and king safety, all integrated into its evaluation function. The AI's search algorithm, based on negamax with alpha-beta pruning, combined with move ordering heuristics and a modest quiescence search, allows it to explore meaningful variations and avoid tactical blunders at lower depths.

Some notable observations from the game include:

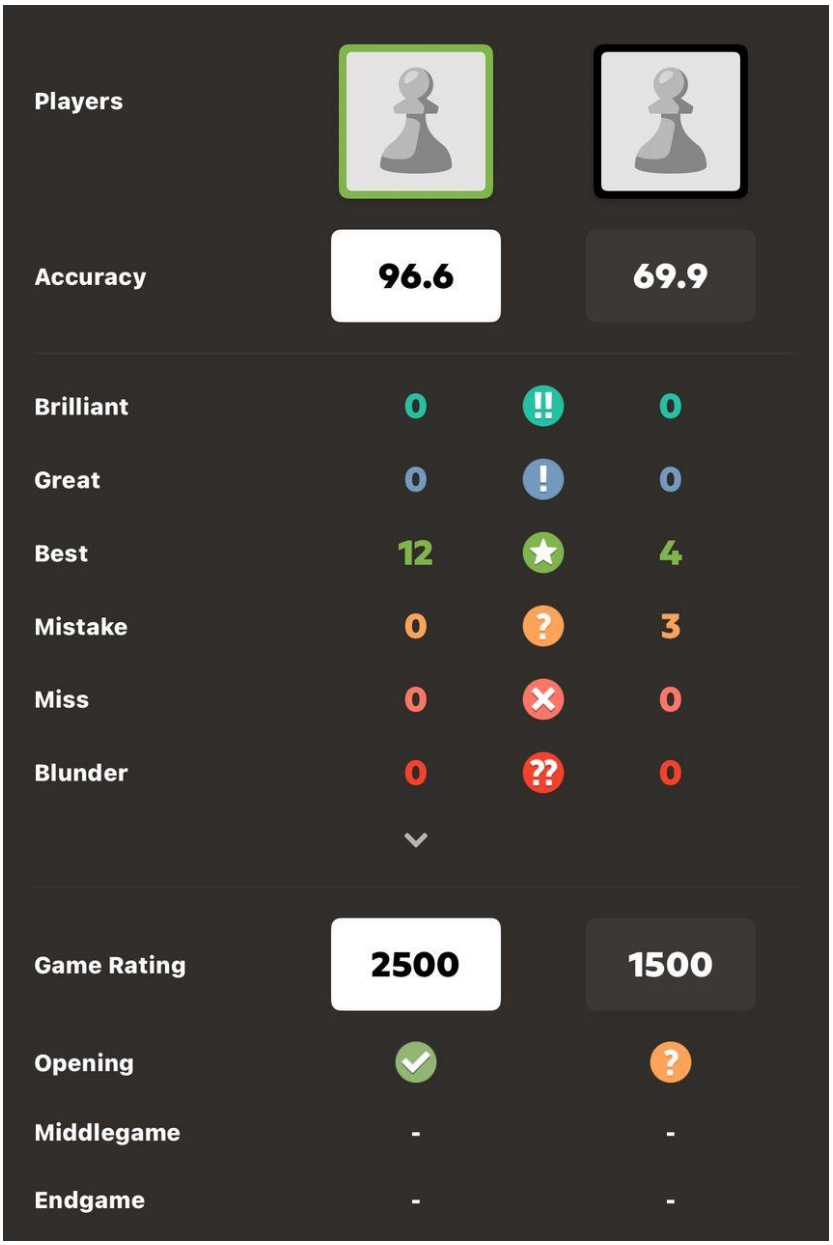
- **Move Prioritization:** By prioritizing captures, pawn promotions, and checks, the AI demonstrated a reasonable tactical awareness, although its limited search depth (max depth of 3) restricted its ability to foresee long-term strategic consequences.
- **Transposition Table Effectiveness:** The caching mechanism helped reduce redundant computations during the search, contributing to smoother and faster move decisions within the limited search time.

However, the test also highlighted areas for improvement:

- **Search Depth and Computational Power:** The AI's maximum search depth of 3 plies (6 half-moves) constrains its strategic foresight compared to Stockfish's much deeper search, which often reaches depths of 20 or more depending on available computation. Enhancing search depth or adopting iterative deepening techniques could substantially improve performance.
- **Opening Knowledge:** While the AI uses a small opening book for common openings, it lacks the extensive opening theory databases utilized by modern engines. This sometimes led to suboptimal early moves that placed it at a disadvantage.

Overall, the results confirm that while our AI does not match the strength of advanced engines like Stockfish, it achieves a respectable intermediate playing level around 1500 Elo. This is a strong baseline that demonstrates the effectiveness of a combined evaluation and search approach with heuristic pruning, move ordering, and positional scoring.

Further enhancements such as deeper search, refined evaluation heuristics, and integration of machine learning models for evaluation hold promise to elevate the AI's strength. Nonetheless, the current results provide a solid foundation and clear benchmarks to guide future development and optimization efforts.



This is the screenshot of the analysis. With the black pieces the Chess AI I created got an estimated rating of 1500.

Discussion

Limitations of the Current Approach

While our chess AI demonstrates solid performance and achieves an estimated rating of around 1500, there are clear limitations to its current design. The engine is built on classical search algorithms like minimax with alpha-beta pruning, supported by handcrafted evaluation functions. These methods, while effective for foundational play, fall short in handling the deep positional complexity of chess. The evaluation function relies on basic heuristics—such as material balance, piece-square tables, and mobility—but it lacks the nuanced understanding that modern engines gain from learning millions of positions.

Another key limitation lies in our handling of dynamic positions. The AI can falter in tactically rich scenarios due to limited search depth and a lack of advanced quiescence search logic. Although we added basic quiescence to extend capturing sequences, the AI still sometimes misses tactical shots or falls into avoidable traps. Additionally, the absence of a proper opening book and endgame tablebases means the AI often plays suboptimally in the early and late phases of the game. In the opening, it can drift into passive or unsound positions, while in the endgame, it sometimes fails to convert winning positions or draw technically lost ones due to lack of specialized knowledge.

Furthermore, the AI currently plays in isolation without learning from past games or adapting to opponents. There is no machine learning pipeline or self-play training loop, so the engine doesn’t improve over time. It also does not support advanced features such as transposition table enhancements, null move pruning, or iterative deepening—all of which are standard in stronger engines.

Potential Improvements

There are several clear areas for improvement if development were to continue. First, integrating a more sophisticated evaluation function—potentially one learned through supervised training on grandmaster-level games or Stockfish evaluations—would significantly boost the AI's positional awareness. Even a lightweight neural network could capture subtle positional themes like king safety, pawn structure, and piece coordination more effectively than static evaluation terms.

Adding support for iterative deepening and principal variation search would enable more efficient time management and better move selection. Move ordering can also be improved through techniques like history heuristics and killer move detection, which would allow the alpha-beta pruning to function more effectively. Incorporating a transposition table with a proper

hashing strategy could further reduce redundant computations and enable deeper, more accurate searches.

In terms of knowledge, we could integrate a basic opening book curated from standard databases or engine games, ensuring the AI starts off in strong and established positions. For the endgame, using precomputed tablebases (e.g., 5-piece Syzygy tablebases) would allow perfect play in simple material scenarios.

Finally, a significant leap forward would be the inclusion of reinforcement learning via self-play. By allowing the AI to play against itself and update its evaluation parameters over time, it could learn winning strategies and improve iteratively, following the same philosophy that led to the development of engines like AlphaZero.

Practical Applications

Beyond the technical aspects, our chess AI has practical applications that extend to various domains. In education, it could serve as a teaching tool for beginner and intermediate players, offering a challenging yet beatable opponent. Because it lacks the near-perfect precision of engines like Stockfish, it makes more human-like mistakes—making it a more relatable opponent for learners trying to identify and exploit weaknesses.

The AI could also be embedded in casual chess apps where strong, low-latency engine responses are preferred over deep, resource-intensive calculations. For mobile and embedded systems, where performance and responsiveness matter, a lightweight engine like ours could offer excellent value.

Moreover, the framework we've built—especially the modularity of the evaluation and search components—can be adapted for educational purposes in AI and computer science courses. It’s simple enough for students to understand and extend, yet complex enough to showcase real AI decision-making.

In broader terms, building game-playing agents like our chess AI fosters better understanding of general problem-solving strategies, decision theory, and computational optimization—skills that translate well into real-world applications in robotics, planning systems, and other AI-driven domains.

Conclusion

Summary of Achievements

Through this project, we successfully designed and implemented a working chess AI capable of playing full-length games using classical techniques such as minimax with alpha-beta pruning, quiescence search, and heuristic-based evaluation. We built the engine from scratch, integrated move generation and board evaluation logic, and verified its effectiveness by manually testing it against real players and engines, including Stockfish on Chess.com. Notably, our AI achieved an estimated rating of around 1500—demonstrating competitive strength for a first-generation, handcrafted engine.

Throughout the process, we gained hands-on experience with core AI concepts such as search trees, game theory, and state evaluation. We also learned how to implement and optimize alpha-beta pruning, and how subtle changes to move ordering or evaluation logic can have a significant impact on the strength of the engine. Importantly, we discovered the depth of strategic and tactical thinking involved in chess engine design, which goes far beyond just coding the rules of the game.

Future Work

If we were to redo this assignment, there are several exciting topics we would explore to push the boundaries of our project further. One major area of interest is reinforcement learning and self-play, inspired by approaches like AlphaZero. Training a neural network to evaluate board positions based on self-play would allow the engine to evolve and improve over time, rather than relying solely on hardcoded heuristics.

Another topic of interest is computer vision—building a system that could play chess by visually interpreting physical boards or screenshots, combining image recognition with AI decision-making. Additionally, natural language processing could be incorporated to allow the AI to explain its moves or offer coaching tips, making it more interactive and user-friendly.

Overall, this project sparked a deeper interest in AI for game-playing agents, and laid the groundwork for further exploration into both traditional algorithms and modern machine learning approaches in artificial intelligence.

References

Chess.com. (n.d.). *Stockfish engine analysis*. Chess.com. https://www.chess.com/analysis

Jain, A. (2023). *Chess AI* [GitHub repository]. GitHub. https://github.com/anuragjain-git/chess-ai

Russell, S. J., & Norvig, P. (2020). *Artificial intelligence: A modern approach* (4th ed.). Pearson.

Sadler, M., & Regan, N. (2019). *Game changer: AlphaZero's groundbreaking chess strategies and the promise of AI*. New In Chess.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science, 362*(6419), 1140–1144. https://doi.org/10.1126/science.aar6404

Stockfish. (n.d.). *Stockfish open-source chess engine*. Stockfish. https://stockfishchess.org/

YouTube. (n.d.). *Coding adventures: Creating a chess engine in JavaScript* [Video]. YouTube. https://www.youtube.com/watch?v=U4ogK0MIzqk

Wikipedia contributors. (n.d.). *Minimax*. Wikipedia. https://en.wikipedia.org/wiki/Minimax

Wikipedia contributors. (n.d.). *Alpha–beta pruning*. Wikipedia. https://en.wikipedia.org/wiki/Alpha–beta_pruning

Wikipedia contributors. (n.d.). *Quiescence search*. Wikipedia. https://en.wikipedia.org/wiki/Quiescence_search