Project and Professionalism

Student ID: 2431342
Name: Swoyam Pokharel
Course: BSc (Hons) Computer Science
Supervisor: Prakriti Regmi
Reader: Siman Giri

Table Of Contents

Introduction

Chess has long served as a testing ground for computer science, especially in the study of search algorithms, data structures and decision making under constraints.

“Oops!Mate” aims to be a chess engine written from scratch in Rust. The engine aims to have a baseline Elo Rating of at least 1800. The engine itself will follow a more "classical" approach to chess engines, and the main goal of this project is to explore and benchmark different techniques used in chess programming to enhance engine performance.

Problem Statement

This project aims to fill the gap in a couple of things. Firstly, although chess engines like Stockfish, Alpha Zero and many others do exist, they are all finished products, often open sourced and backed by thousands of open-source contributors or proprietary engines in private development. As such, it is often hard for newcomers to answer, "*how did they make this engine*", or how different components in chess engines such as Ordering, Evaluation, Search etc. tie together. Furthermore, there isn't a resource directly benchmarking these different techniques for the most significant increase in each component. Finally, despite the global progress in chess engine programming, as of (September 03, 2025) there is yet to be a chess engine developed in the country Nepal, which opens room for contribution.

This project, therefore, aims to answer the following question:

“How different techniques involved in chess programming affect a chess engine's playing strength and efficiency”

The entire development will be documented here, and the code will be found here providing newcomers with a comprehensive resource to understand and implement a fully functional chess engine.

Aims and Objectives

The Aim

To build a chess engine in Rust from scratch and benchmark incremental performance gains by implying different techniques used in chess programming.

The Objectives

- Implement baseline engine
    - BitBoards
    - NNUE
    - Magics
    - Negamax
    - FEN
    - UCI
- Incrementally integrate different techniques:
    - Evaluation
        - NNUE
        - HCE (Hand Crafted Evaluation)
    - Pruning:
        - Alpha Beta Pruning
        - Null Move Pruning
        - Depending on time constraints I'd like to play around with Futility Pruning too
    - Move Ordering
        - Captures First (eg. MVV-LVA),
        - Promotions First
    - Other Extensions:
        - Quiescence Search
        - Iterative Deepening
    - Benchmark each engine generation against it's own predecessor
    - Document progress and learning in a public blog.

Research

This project aims to be a more "classical" chess engine, focusing on building a solid baseline and then incrementally integrating and benchmarking established techniques. Each chosen technique is selected for it's proven effectiveness in chess programming. The project also emphasizes benchmarking each iteration against its predecessor to objectively measure improvements, and documenting progress to contribute to both personal learning and the broader chess programming community.

Board Representation

Representing the board is the foundational step in setting up the path for the rest of the chess engine. There are two different main types of board representation, Piece Centric and Square Centric.

Square Centric

This approach focuses on the board itself, typically using a 2D (eg. 8x8) or a 1D (eg. 64) array where each square stores what piece occupies it, if any. It is particularly useful because it is the most intuitive, easy to visualize and simple to implement as it's basically just board [8][8]. But it struggles with large scale computations like generating all moves for multiple pieces.

Piece Centric

In this approach, each piece type is tracked separately, often using techniques like BitBoards, and they shine making certain operations like generating moves or checking attacks on a square faster.

This project aims to use the BitBoards representation. This is because BitBoards are performant because they allow parallel bitwise operations such as AND, OR, NOT to quickly set or query the game states [1]. This, paired with the fact that most architectures these days are 64 bits, and conveniently enough there are exactly 64 squares in a chess board. This alignment makes it so that each bitboard fits in a single machine word. Furthermore, the use of bitboards enables the use of a technique called Magics, which allows move generation for sliding pieces like queens, bishops and rooks to be constant time O(1)

Evaluation

For a engine to come up with good moves, it needs to first know what makes a position good or bad. Each engine defines a score function: eval(position) -> number, but there are different ways to do so.

Hand Crafted Evaluation

Initially, the evaluation function was handcrafted (HCE) which considered factors such as material, piece-square tables, king safety, mobility etc. These metrics were often weighted so that it would look like this at the end:

$$eval = 9 * queens + 5 * rooks + 3 * knights + \ldots + mobilityBonus + kingSafetyPenalty$$

Efficiently Updatable Neural Networks (NNUE / ЭUИИ)

However, a newer approach to this evaluation functions are efficiency updatable neural networks; NNUEs or sometimes styled as ЭUИИ. Originally from Shogi, this technique has made its way to the world of chess programming, with one of the first introductions of it being in Stockfish. NNUEs learn to evaluate positions from games instead of handcrafted weights, where the inputs are usually - BitBoards or piece-square occupancy maps and the output is a score for the position. They are fast enough to update incrementally, during move search, hence the name "*Efficiently Updatabale* neural networks".

Self-Play Based Learning

Finally, engines like Alpha Zero use pure reinforcement learning, and their evaluation of position comes from self-play. This project aims to explore both classical HCE and modern NNUE approaches for the evaluation function, while deliberately not pursuing the self-play alternative due to its substantial computational requirements and the extensive data and infrastructure necessary for effective implementation.

Search

To search for the best move, chess engines explore potential sequences of moves in the background and call the evaluation function, to get a metric of the position. The core challenge is

that the number of possible move sequences grows exponentially with depth. For a zero-sum game like chess, Minimax is the natural approach, it recursively evaluates the game tree by alternating between maximizing the score from the evaluation function for itself and minimizing the score for the opponent. Russell, S., & Norvig, P. (2021). A commonly used variation is Negamax, which simplifies the implementation of Minimax by taking advantage of the zero-sum property of chess.

However, while naive Minimax or Negamax evaluates all nodes in the game tree, engines improve upon this using pruning techniques such as: - Alpha beta pruning, which eliminates branches that cannot affect the final decision. [2] - Null Move Pruning, which assumes doing nothing (passing a move) is worse than playing the *best possible legal move*, allowing engines to prune branches that are unlikely to improve the position [3]

This project aims to explore both of these techniques to improve search efficiency while maintaining optimal decision making.

Other Extensions

To further enhance search quality and efficiency, modern chess engines implement additional techniques beyond standard pruning and Negamax/Minimax.

- Quiescence Search : This technique extends the search at nodes for "noisy" positions, such as captures or checks to avoid the horizon effect.
- Iterative Deepening : This technique repeatedly searches the game tree to increasing depths, using results from shallower searches to guide move ordering in deeper searches. This improves pruning efficiency and allows "anytime behavior", which basically means that the engine can return the last analyzed best move even if interrupted, say in a time-based setting.

This project aims to integrate both of these techniques to improve move selection under practical constraints.

Communication & Notations

To standardize this engine, the project aims to integrate the standard protocol UCI and as such, it's dependency FEN.

Artifacts

Crates For:

- Base Engine
  - NNUE
  - Bitboards
  - UCI
  - FEN
  - Minimax
  - Magic Bitboards
  - Lookup Tables
  - Legality
- Evaluation
  - HCE
- Search
  - Alpha-Beta Pruning
  - Null Move Pruning
- Ordering
  - MVV-LVA,
  - Promotions First

Along documentation for each stage.

Testing

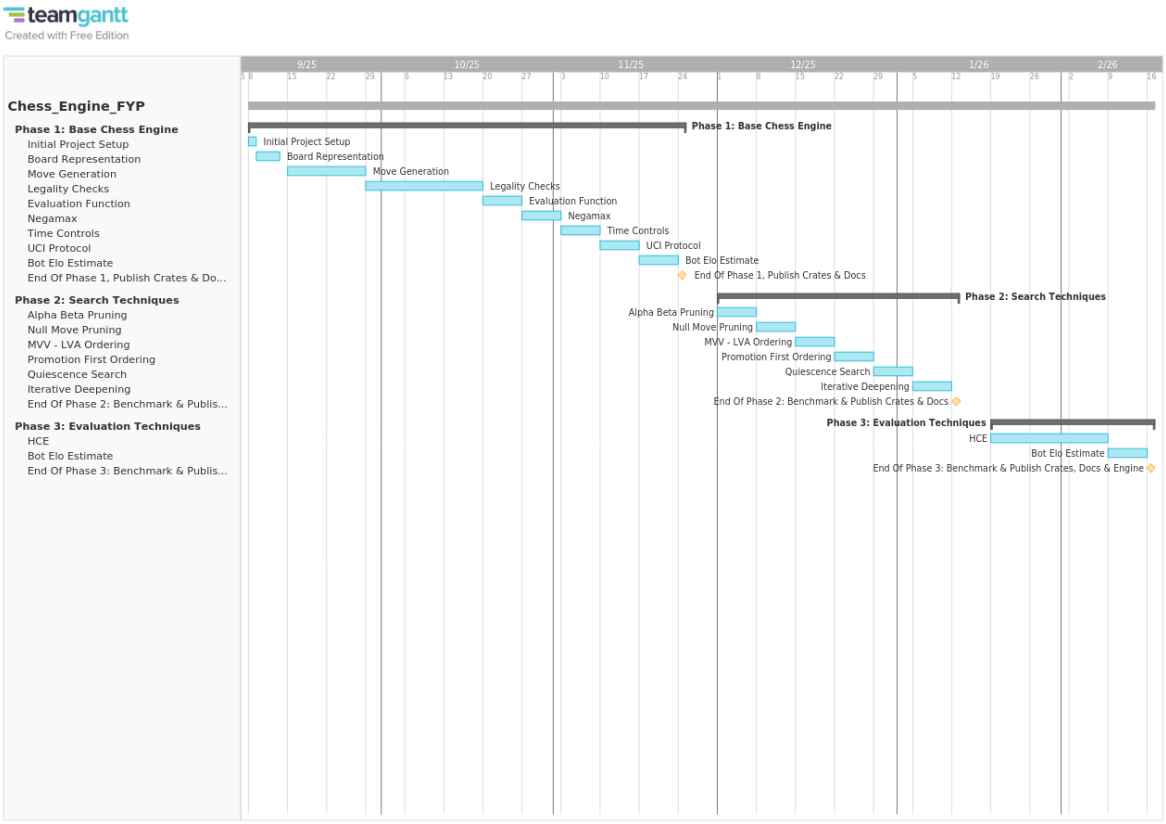
Testing will proceed generation by generation. While alternatives such as using a bot account on the Lichess platform were considered, this approach was ultimately set aside due to the logistical and financial challenge of maintaining a personal VPS for extended testing sessions, each potentially lasting several hours.

For the initial baseline engine, a rough estimate of its Elo will be obtained by having it play against human opponents of varying skill levels. Additionally, techniques like Perft will be used to verify the correctness of move generation.

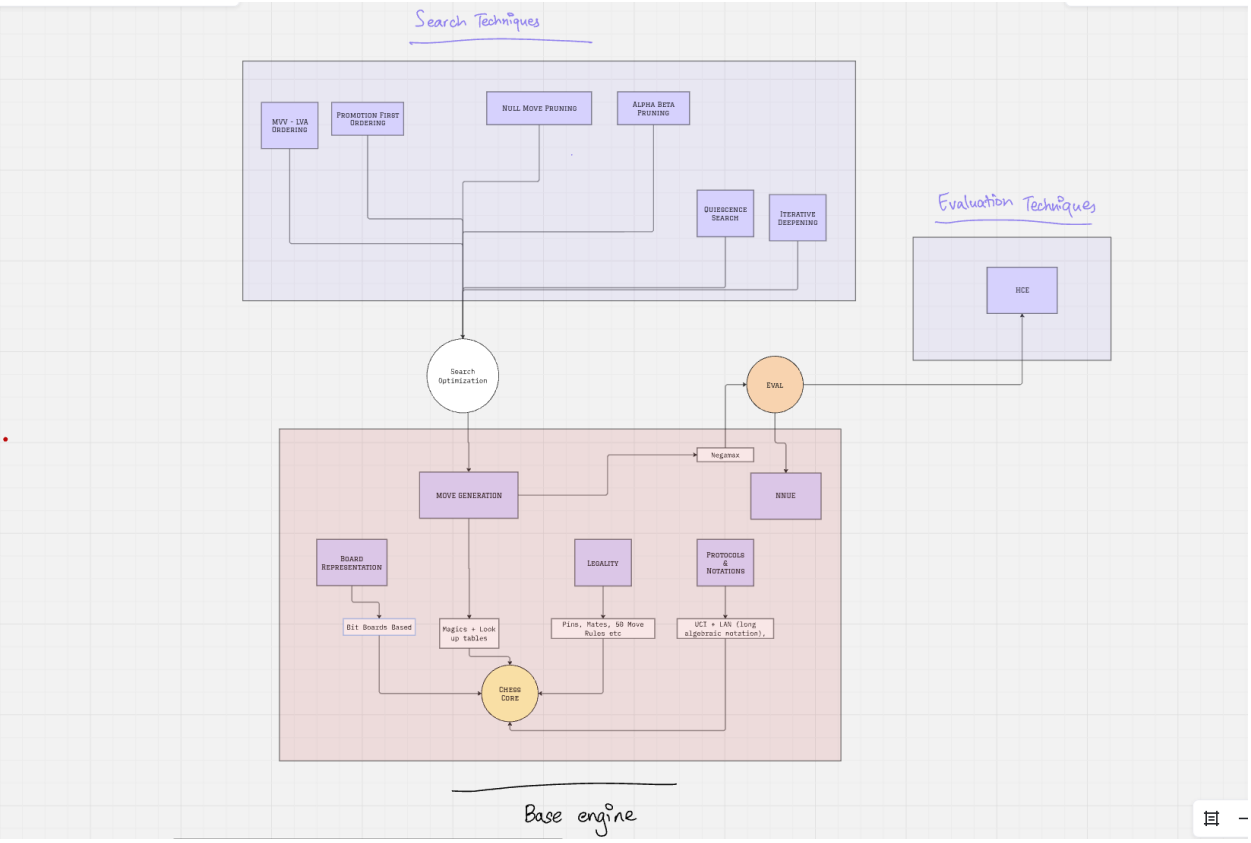
Project Management Plan

The project will use Kanban as the framework for task tracking but following a Waterfall approach for overall planning. This approach is aimed because the requirements for the chess engine are well-defined and unlikely to change, allowing for a structured, phase-based development process while still visually managing tasks in Trello.

The timeline is for ~ 7 months, 10 days.


teamgantt
Created with Free Edition
9/25
10/25
11/25
12/25
1/26
2/26
Chess_Engine_FYP
Phase 1: Base Chess Engine
Initial Project Setup
Board Representation
Move Generation
Legality Checks
Evaluation Function
Negamax
Time Controls
UCI Protocol
Bot Elo Estimate
End Of Phase 1, Publish Crates & Do...
Phase 2: Search Techniques
Alpha Beta Pruning
Null Move Pruning
MVV - LVA Ordering
Promotion First Ordering
Quiescence Search
Iterative Deepening
End Of Phase 2: Benchmark & Publis...
Phase 3: Evaluation Techniques
HCE
Bot Elo Estimate
End Of Phase 3: Benchmark & Publis...
Phase 1: Base Chess Engine
Initial Project Setup
Board Representation
Move Generation
Legality Checks
Evaluation Function
Negamax
Time Controls
UCI Protocol
Bot Elo Estimate
End Of Phase 1, Publish Crates & Docs
Phase 2: Search Techniques
Alpha Beta Pruning
Null Move Pruning
MVV - LVA Ordering
Promotion First Ordering
Quiescence Search
Iterative Deepening
End Of Phase 2: Benchmark & Publish Crates & Docs
Phase 3: Evaluation Techniques
HCE
Bot Elo Estimate
End Of Phase 3: Benchmark & Publish Crates, Docs & Engine

Design Diagram & Tech stack



As mentioned, this project will be implemented in Rust. Rust was chosen because of its balance of performance, safety, and learning opportunity. It delivers performance on par with low-level languages like C and C++, making it well-suited for the computation-heavy tasks of move generation and evaluation.

However, unlike C and C++, Rust provides strong memory safety guarantees through its ownership and borrowing model, eliminating common classes of errors without a garbage collector. In addition, with Rust's growing adoption and recognition in industry, this project serves as an ideal opportunity to both explore a modern systems language and gain practical experience in applying it to a problem.

References

- https://en.wikipedia.org/wiki/Elo_rating_system
- https://stockfishchess.org/
- https://www.chess.com/terms/alphazero-chess-engine
- https://www.chessprogramming.org/Move_Ordering
- https://www.chessprogramming.org/Evaluation
- https://www.chessprogramming.org/Search
- https://www.chessprogramming.org/Bitboards
- https://www.chessprogramming.org/NNUE
- https://www.chessprogramming.org/Magic_Bitboards
- https://www.chessprogramming.org/Negamax
- https://www.chessprogramming.org/Forsyth-Edwards_Notation
- https://www.chessprogramming.org/UCI
- https://www.chessprogramming.org/Pruning
- https://www.chessprogramming.org/Alpha-Beta
- https://www.chessprogramming.org/Null_Move_Pruning
- https://www.chessprogramming.org/Futility_Pruning
- https://www.chessprogramming.org/MVV-LVA
- https://www.chessprogramming.org/Quiescence_Search
- https://www.chessprogramming.org/Iterative_Deepening
- https://lichess.org/@/likeawizard/blog/review-of-different-board-representations-in-computer-chess/S9eQCAWa#one-dimensional-64array
- https://en.wikipedia.org/wiki/Bitboard
- https://en.wikipedia.org/wiki/Word_(computer_architecture)
- https://en.wikipedia.org/wiki/Shogi
- https://www.chessprogramming.org/Stockfish_NNUE
- https://www.netlib.org/utk/lsi/pcwLSI/text/node351.html
- https://www.chessprogramming.org/Null_Move_Pruning#core-idea
- https://en.wikipedia.org/wiki/Horizon_effect
- https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search
- https://www.chessprogramming.org/Perft