Chess Engine “Gambit” Development

Project Plan

Callum Moss
Royal Holloway, University of London
Supervised by Eduard Eiben

ABSTRACT

Motivation:

Chess engines have long dominated human players, ever since 1997 when IBM’s Deep Blue famously defeated the world chess champion Garry Kasparov [1], officially marking the end of human superiority in chess. Since then, engines have become increasingly powerful, vastly surpassing human capabilities. However, despite their overwhelming strength, engines do not defeat humans as quickly as one might expect. If an engine is so strong, why doesn’t it deliver a checkmate in just a few moves? This is because of how an engine decides on moves. Most chess engines rely on variations of the **minimax algorithm** [2], which searches for optimal **lines** by assuming that the opponent will respond with their best possible move. The engine would never choose to play a move which could be exploited, even if it could also lead to a quick checkmate based on inaccuracies by its opponent.

While the minimax algorithm is logical in a two-player, zero-sum game with perfect information, it overlooks the fact that human players are fallible [3] and have a very different thought process than engines. For humans, the problem of which move is best is too complex to be solved even in analysis, therefore they will have certain predictive hunches about risks and possible advantages. This means that it is predictable as to how a human may pick a move [4]. A means of taking advantage of this is producing an **opponent model**, such that we can begin to predict which moves a player may play, and which they could miss [5].

We propose that chess engines should account for human fallibility, and their superior capabilities. Engines should play "**risky moves**" that assume their human opponents will fail to counter effectively. This strategy would enable engines to achieve stronger positions and win more quickly and dominantly. These types of moves could be closely compared to gambits, a clever action in a game or other situation that is intended to achieve an advantage and usually involves taking a risk [6]. Hence naming this engine “Gambit”. These gambits if not properly diffused result in a much stronger position than if the engine had played the "safest" move. This project aims to explore whether engines can adopt this strategy to defeat human players faster and more dominantly than by following traditional **alpha-beta** lines. Furthermore, we aim to help discover some factors at play to determine the likelihood of an opponent missing the counter moves and factors that should affect the engine’s decision making. It would also be interesting to see if this engine emulates human behaviour at all, and whether there is a noticeable behavioural difference between this engine and a human.

It is important to note that this approach will only work if our engine is stronger than its human opponents. This should be the case. This is assumption is based off the mean average **Elo** rating of the 41 independently created chess engines, from a chess engine development discord server, being higher than the average Elo of the top 10 chess players. The average Elo of independently created chess engines was ~3223 **CCRL rating** in Blitz [7]. Using the formula: FIDE rating = 0.70*CCRL rating + 840 ELO points [8], we can determine that the average **FIDE rating** is ~3103. In comparison, the average FIDE rating of the top 10 blitz chess players in the world is ~2813 [9].

Some closely related concepts are opponent model search [10] and the **contempt** factor [11]. Explained briefly, in opponent model search the engine creates a profile of its opponent, and uses this to predict what moves their opponent may respond with in various positions. This can be helpful to us as this can help us predict whether or not the opponent will spot the counter to our risky move. The contempt factor measures the level of respect an engine has for its opponent. If it has low respect, it will temporarily accept a worse position to avoid a draw with the assumption that if the game carries on, it will be able to regain its advantage and win. These aspects will be important when implementing **speculative play**.

A useful metric to calculate how quickly our engine can beat humans would be using an **EAS-Score** [12]. This can help us determine the average number of moves a game finishes in when we change any part of the decision process for making moves. Furthermore, it can tell us the number of short games it wins, number of sacrifices made, and number of **bad draws**, which gives us a greater idea of how the engine thinks throughout the game. Not only can we use EAS to gauge the level of success each of our changes has in reaching our goal of quick, dominant games vs humans, we can also use **SPRT**s [13]. This allows us to see whether our engine is outperforming previous iterations of itself. This will be helpful for big changes like random mover to alpha beta, but it will be less helpful near the end of the project as our engine is designed to play against a human rather than engines.

In 1983, Reibman et al suggests three conditions that must be met for a successful speculative play. The three principles are:

1. the fallible opponent makes mistakes with a probability close to 50%

2. the size of mistakes made by the opponent is (much) larger than the size of the mistakes made by the program (risk)

3. the program can predict with reasonable accuracy whether the opponent will make a mistake” [14, p141].

There are two approaches I have considered to determine whether a move should be played.

Approach 1:

It is relatively simple to identify where conditions one and two apply to a position. We can use the following formula I created to calculate the necessary likelihood of a better position than alpha beta required for a line to be worth playing:

$$\boldsymbol{NSL = \frac{AB - F}{S - AB}}$$

$$\boldsymbol{NecessarySuccessLikelihood = \frac{AlphaBetaLineEvaluation - FailedLineEvaluation}{SuccessLineEvaluation - AlphaBetaLineEvaluation}}$$

We then multiply the necessary success likelihood by its evaluation, then store it in a 2d array. We pick the line with the highest likelihood to evaluation ratio to play.

Approach 2:

We will calculate two sets of moves which are either successful or a failure. We will store these in two 2d arrays, where we will store the successful or failed line (whether it is positive or negative) and its corresponding ratio, calculated by the following:

$$\boldsymbol{R = L * (E - BLE)}$$

$$\boldsymbol{LineLikelihoodRatio = LikelihoodOfOccurance * (BestLineEvaluation - ThisLineEvaluation)}$$

We then add the two ratios of entire sets. If above 0 then we should take a risk (it is more likely that a risky move will turn out well then not). We then play line with highest ratio. If below 0, it is not worth the risk. We then play alpha beta line.

Using either approach could easily affirm conditions 1 and 2 by adding the condition to ignore lines with less than a 50% likelihood. The main challenge comes when addressing the third condition; to make a program to predict with reasonable accuracy whether the opponent will make a mistake. Therefore, to make Gambit effective, we must design a sophisticated opponent modelling system that makes use of various factors, such as opponent skill, without using a neural net which would be out of the scope of this project. A neural net would not be as helpful here as we would not be able to effectively assess what factors are helpful when developing an opponent model. For example, Leela Chess Zero uses a similar concept, contempt, to evaluate whether a move is worth playing based on whether Lc0 would be happy with a draw, given the strength of their opponent. Lc0 uses a sophisticated method of determining the size of the contempt factor, which makes use of WDL (win/draw/loss likelihoods). As Lc0 was trained on a large database of real games, it is more than likely that Lc0 incorporates some version of opponent modelling when determining WDL [15]. However, the factors and patterns that Leela takes into account when deciding the WDL is unknown as it is a large, complex neural network. Therefore, to determine what conditions should be met when deciding on a risky move, we must make use of **HCE**.

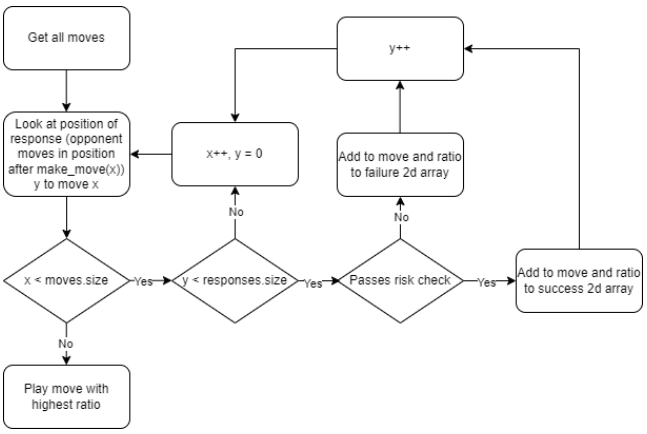


Figure 1: Search of up to depth 2

Some things to consider during the calculation of success likelihood by the opponent model are:

- Bad players are hard to predict but lower risk. Good players are easier to predict but higher risk
- Opponent skill will be determined throughout the game but could have an initial value set by a puzzle quiz or Elo input. Otherwise, start with the average skill of a chess player and adjust from there.
- Complexity of position (if more move nodes, they are less likely to make the desired mistake although they are more likely to make a mistake in general)
- Complexity of line (patterns of tactic spotting like knight forks, pieces of higher value sacrifices etc)
- Number of missed best moves or number of successful risky moves would indicate a higher likelihood of error.
- Stats about player like gender, age, strength compared to engine, self-described style. Can use this to look at stats for chess psychology.

Some things to consider during the evaluation of a position are:

- Standard principles like centipawn advantage, piece value sum, activity of pieces (how many attacks can be generated via counting the nodes of attacks), piece-square tables etc.
- Engine vs Human strengths and weaknesses (for example, engines favour complexity, can do this by spotting of complex tactics like forks, and mid-end games whereas

humans tend to excel at early game, or the solving of chess for 8 or less pieces on the board (only 5 would be viable as they exponentially get massive in size)).

Aims and Goals:

To summarize, our goal is to develop a chess engine that integrates standard chess programming principles and practices. It should also adhere to the Google C++ programming standard and follow a **TDD** workflow. We aim to design a new move decision algorithm to both enhance the engine versus human experience, but also to see if by taking risks, engines can beat humans quicker and more dominantly than they already do. The new move decision algorithm should account for various factors, such as its current position (whether it is winning or losing) and whether the opponent is strong or weak. Not only will this make a more interesting experience for its opponent, but it will also attempt to capitalize on human fallibility, therefore winning quicker than other engines would by taking risks. It is important to note that this engine is not intended to compete with other chess engines (although it will do so to gain a rough evaluation of engine Elo) but is intended to make for a much more interesting experience for humans. In the process, we will understand standard AI techniques and algorithms and how they can be applied or adapted to play chess. We will also experiment with and contribute unexplored ideas for chess engine development.

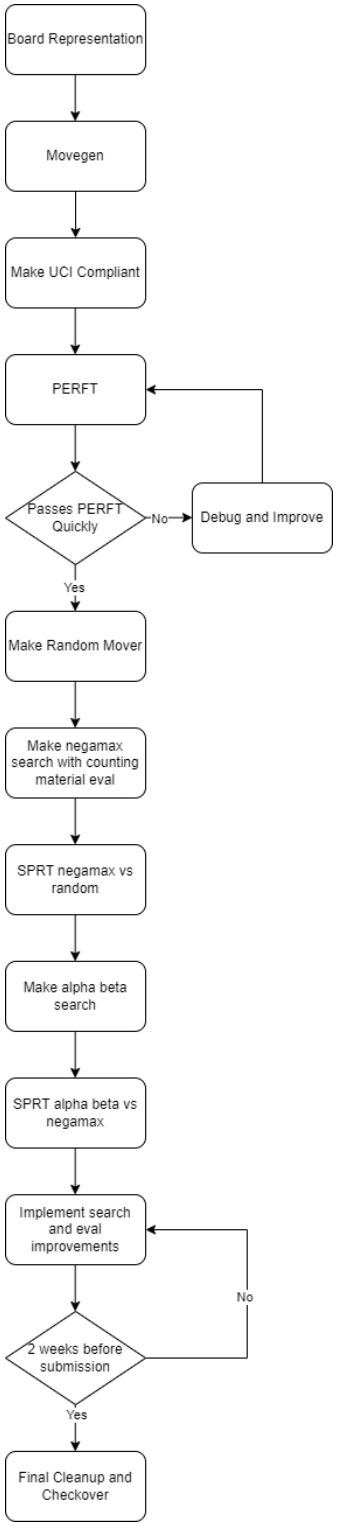
We aim to answer the following questions: Can we use HCE to beat humans quicker than the average chess engine? As gambits are very human in nature, can a chess engine emulate human behaviour, such that it is able to pass a **Turing test** against varying skill levels of chess players? In what scenarios should an engine use a risky move?

Project Value:

Since the first publication of the idea of speculative play and opponent modelling in 1983, there has been little research in these ideas. Reibman et al calls for more research on this: “In sum, it seems clear that further research has good prospects of demonstrating the validity of the claim that speculative play can obtain better overall”. I believe that by applying modern knowledge of chess programming and human psychology we can find a better approach. Some core concepts of human chess psychology have been discovered by Adriaan de Groot, who suggests for example that humans tend to scan the board and find the best move best on intuition, then spend time proving to themselves that it is the best [16]. This is a rather different approach than engines, as via the alpha beta algorithm, the engine brute forces all possibilities and cut off unpromising lines early, continuing to look at a higher depth on promising lines for a given amount of time. Although the results were not promising, this niche of chess engine programming is under-represented and researched. I believe with some expansion and development, there is a lot of potential in modelling an opponent.

1 Timeline and Milestones

1.1 Development Flowchart



1.2 Term 1: Core Development

Working on report throughout and making notes as I go.

Week 0:

- Pushed work done over the summer (Board representation and move gen for knights, kings and some pawn moves, as well as progress made on sliders)

Week 1 (Begins 30th September)

- Begin work on **magics** (**bitboard** optimizations, attack masks, etc.).

Week 2

- Complete implementation of magics.

Week 3

- Task: Implement pawn promotions, en-passant, and castling logic.

Week 4

- Task: Perform legality checks (ensure legal move generation, valid board states, etc.).

Week 5

- Task: Implement **UCI** [17] (Universal Chess Interface) compliance for engine communication.

Week 6

- Tools: Use **CuteChess** [18] or similar tools to:
  - Test **perft** (performance test) for move generation accuracy.
  - Run **SPRTs** (Sequential Probability Ratio Tests).
- Goal: Debug move generation to ensure accuracy.

Week 7

- Task: Implement a basic move generator and a random mover to test move generation.

Week 8

- Goal: Start implementing **negamax** algorithm and basic evaluation function.

Week 9

- Milestone: Complete negamax and basic evaluation function.

Weeks 10–11

- Deliverables:
  - Prepare for presentation.
  - Submit interim report.

1.3 Term 2: Advanced Features and Testing

Working on report throughout and making notes as I go.

Week 0 (Over the break)

- Task: Run SPRT to compare negamax versus random mover.

Week 1 (Begins 13th January)

- Task: Implement alpha-beta pruning to improve negamax efficiency.

Week 2

- Task: Run SPRT to compare alpha-beta versus negamax.

Weeks 3–6

- Development Focus: Implement and refine gambit-related functionality (specialized move sequences and strategies).

Week 7

- Task: Continue work on improving gambit handling.

Week 8

- Milestone: Finalize gambit functionality.

Week 9

- Task: Conduct user testing by:
  - Testing with real players.
  - Collect findings and feedback.
  - Host engine on **Lichess** for public testing and feedback.

Weeks 10–11

- Final Deliverables:
  - Final cleanup of code and functionality.
  - Prepare for final submission and documentation.

1.3 Project Checklist

1. Board Representation
   a. 6, 2 bitboards
   b. Parse **FEN**s – consider UCI throughout design process
   c. Print board
2. Move Generation
   a. Generate **pseudo legal** moves, then add legality checks:
   b. Pseudo legal move gen using magic bitboards
   c. Knights: Precompute lookup table for possible moves a knight could make for all 64 squares.
   d. Kings: Precompute lookup table for possible moves a king could make for all 64 squares.
   e. Pawns: Shift them all set wise and use **masks**
   f. Bishops: Magics
   g. Rooks: Magics
   h. Queens: Magics (Rook glued to a bishop)
   i. Pawn promotions
   j. En Passant
   k. Castling
   l. Add legality checks
   m. “Copy-Make move” (function for searching)

    n. Implement make move as a random mover (which would be considered a v1)
3. Make UCI Compliant
4. Perft – For various positions, generates the expected number of move nodes in a reasonable amount of time. (UCI software generally has a perft command)
5. Search Function
    a. Negamax
    b. Alpha Beta
    c. Consider legality of the move just after you make it. Check for legality, if its legal, then proceed with the search, else go and take it back. Keep a flag to have at least one legal move played, if not then either its a check mate (if the king is in a check) or stalemate.
6. Eval Function
    a. Count material first, can make use of **piece square tables** and core chess principles, like castle before its too late, develop etc
    b. Give it an aggressive opening book after random mover has been developed. Download the book and type a few characters into cute chess then its done.
    c. Incorporate idea of Gambit (look at contempt, opponent model search etc)
7. Testing
    a. Test with Lichess
    b. Test with chess society people
    c. Conduct “Turing Test”

1.4 Extensions

Evaluate at the end of the game how the opponent played, and whether it was better or worse than expected, and perhaps to point out the best moves where they missed them.

Develop a sophisticated Elo assignment protocol as not all players will know their Elo.

Be able to toggle on and off the Gambit features, such that it will be strong for **EvH** and **EvE**. This way we can also compare the two approaches.

Make a custom UI for interacting with Gambit.

2 Risks and Mitigations:

2.1 Engine Depth

Risk:

Engine will likely be very limited by depth, due to the nature of probabilities. Just at depth 4, two moves of 50% likelihood drops to 25% likelihood to occur. Depth is very important for not foreseeing a blunder several moves down the line

Mitigation:

However, humans can’t search in much depth either, so the effect of this is limited. We could generate an optimistic and pessimistic guess predictions for each move in the sets to help mitigate false predictions. We could explore up to an infinite depth until likelihood of a line is below (for example) 20%. This way, the engine should not play lines that are very unlikely to occur. We must also consider the set of moves rather than the best and the desired mistake move. Many moves could prove to be mistakes, raising the likelihood. We care about the likelihood that the engine will get a better position than if it had followed the alpha beta.

2.2 Alpha Beta Superiority

Risk:

Determining that using regular alpha beta is simply better than using gambits, due to the inherent risk of gambits.

Mitigation:

Although likely, this is of little importance as it still contributes more ideas to the area of speculative play and opponent modelling.

2.3 Ambition

Risk:

This is a highly ambitious project with a lot of moving parts, complex theory and some new ideas.

Mitigation:

I have lots of support from a plethora of resources, such as forums and discord servers where I can ask questions. There is lots of information out there, and much of what I am doing here is not new, so I do not have to reinvent the wheel.

2.4 Opponent Model Deception

Risk:

Several means of building the opponent model could be intentionally tricked or be inconceivable to provide a reliable conclusion (for example saying you’re worse than you are).

Mitigation:

Whilst this is the case for any initial data required, the opponent model will be updated throughout the game, therefore any impact of deception should be minimised once past the opening, where the engine should simply be following an opening book so risky moves shouldn’t be played until the opponent model is accurate. If an opponent purposely plays bad to lure the engine to make a mistake, I believe the consequences of these bad moves to cause the risky move would be greater than the consequence of capitalising on the risky move.

GLOSSARY

Minimax: A search algorithm where player A attempts to maximize their position and player B attempts to minimize player A’s position.

Line: A sequence of chess moves which lead to a given position.

Opponent Model: The engine creates a profile of its opponent, and uses this to predict what moves their opponent may respond with in various positions.

Risky Moves: Moves that make use of speculative play and an opponent model to choose moves more likely to succeed than the alpha beta line.

Alpha-Beta: An enhancement of minimax that prunes branches where there is a possible response to move B that results in a worse position than any response for move A.

Elo: A rating given to a chess player that represents their skill level.

CCRL Rating: The standard for an Elo assigned to engines to represent their skill level.

FIDE Rating: The standard for an Elo assigned to humans to represent their skill level.

Contempt: The contempt factor measures the level of respect an engine has for its opponent. With high respect it will accept moves which lead closer to a draw. Whereas with low respect, it will temporarily accept a worse position to avoid a draw, with the assumption it will win later in the game.

Speculative Play: Choosing to play non minimax moves, based on the assumption that your opponent will miss the correct counter move, with the goal to gain a better position than the alpha beta sequence.

EAS-Score: A score given to engines to gauge its aggression and success.

Bad Draws: Draws that should not have occurred, whether it happened too early in the game or off the back of an earlier advantage.

SPRTs: Compares the last version of your engine to your current version by playing many games against each other, to determine whether your recent modifications are impactful.

HCE: Hand-crafted evaluation method of determining which player is winning in a given position and by how much, typically in value of centipawns.

TDD: Test-driven development

Turing Test: A test conducted to determine whether a person cannot accurately determine whether they are interacting with a human or a machine.

Magics: a technique where you hash the pieces that can block a slider's attacks and use that hash as an index to look up a precomputed bitboard of attacks.

Bitboard: a 64 bit representation of a chess board state.

UCI: Universal chess interface, used for engine and user interface communication

CuteChess: An interface to conduct engine matches and also has a GUI.

Perft: Tests move generation, time taken to generate moves and tests makemove.

Negamax: Simplified version of minimax that produces the same output.
Lichess: a popular chess website where we can host the engine to play real people.

FEN: Forsyth-Edwards Notation used as the standard for describing chess positions.

Pseudo legal: All legal moves that do not consider whether you are in check, or it results in you being in check

Mask: Board which shows the status of each square in a specific region. A mask could be for one file, or it could be for attacks of a piece. A knight attack mask would have 1s in the L positions and 0 elsewhere.

Piece-square Tables: Used to determine the rough value of a given piece type on a given square. For example, a pawn on e4 is stronger than a pawn on a2.

EvH: Engine versus human

EvE: Engine versus engine

REFERENCES

IEEE Format (using https://www.mybib.com/tools/ieee-citation-generator)

[1] Chess.com Team, “Kasparov vs. Deep Blue | The Match That Changed History,” *Chess.com*, Oct. 12, 2018. https://www.chess.com/article/view/deep-blue-kasparov-chess (accessed Oct. 05, 2024).

[2] “Search - Chessprogramming wiki,” *Chessprogramming.org*, 2010. https://www.chessprogramming.org/Search (accessed Oct. 05, 2024).

[3] “The Game Theory of Chess | Tidings Media,” *www.tidingsmedia.org*. https://www.tidingsmedia.org/blog/the-game-theory-of-chess (accessed Oct. 06, 2024).

[4] D. de and Fernand Gobet, “Perception and memory in chess: Heuristics of the professional eye,” *ResearchGate*, 1996, Chapter 9, pp.18. https://www.researchgate.net/publication/247130036_Perception_and_memory_in_chess_Heuristics_of_the_professional_eye (accessed Oct. 06, 2024).

[5] A. L. Reibman, “Non-Minimax Search Strategies for Use Against Fallible Opponents - AAAI,” *AAAI*, Oct. 16, 2023. https://aaai.org/papers/00338-AAAI83-084-non-minimax-search-strategies-for-use-against-fallible-opponents/ (accessed Oct. 06, 2024).

[6] Cambridge Dictionary, “gambit,” *@CambridgeWords*, Oct. 02, 2024. https://dictionary.cambridge.org/dictionary/english/gambit#google_vignette. (accessed Oct. 07, 2024)

[7] EngineProgramming, “GitHub - EngineProgramming/engine-list,” *GitHub*, 2023. https://github.com/EngineProgramming/engine-list (accessed Oct. 07, 2024).

[8] “a direct comparison of FIDE and CCRL rating systems - Page 4 - TalkChess.com,” *Talkchess.com*, Feb. 24, 2016. https://talkchess.com/forum3/viewtopic.php?p=661573#p661573 (accessed Oct. 07, 2024).

[9] “FIDE Ratings,” *ratings.fide.com*. https://ratings.fide.com/top_lists.phtml. (accessed Oct. 08, 2024).

[10] “Opponent Model Search - Chessprogramming wiki,” *Chessprogramming.org*, 2015. https://www.chessprogramming.org/Opponent_Model_Search (accessed Oct. 08, 2024).

[11] “Contempt Factor - Chessprogramming wiki,” *Chessprogramming.org*, 2015. https://www.chessprogramming.org/Contempt_Factor

[12] *Sp-cc.de*, 2024. https://www.sp-cc.de/files/eas_scoring_explanation.txt (accessed Oct. 08, 2024).

[13] “Sequential Probability Ratio Test - Chessprogramming wiki,” *Chessprogramming.org*, 2024. https://www.chessprogramming.org/Sequential_Probability_Ratio_Test (accessed Oct. 09, 2024).

[14] A. L. Reibman, “Non-Minimax Search Strategies for Use Against Fallible Opponents - AAAI,” *AAAI*, Oct. 16, 2023. https://aaai.org/papers/00338-AAAI83-084-non-minimax-search-strategies-for-use-against-fallible-opponents/ (accessed Oct. 10, 2024).

[15] “Win-Draw-Loss evaluation - Leela Chess Zero,” *Lczero.org*, 2020. https://lczero.org/blog/2020/04/wdl-head/ (accessed Oct. 11, 2024).

[16] Thought and Choice in Chess. DE GRUYTER, 1978. doi: https://doi.org/10.1515/9783110800647 (accessed Oct. 11, 2024).

[17] “UCI Protocol,” *backscattering.de*. https://backscattering.de/chess/uci/ (accessed Oct. 09, 2024).

[18] “Cute Chess,” *cutechess.com*. https://cutechess.com/ (accessed Oct. 09, 2024).

Other Helpful Resources:

Aggressive and gambit opening books:

- https://www.chess.com/forum/view/general/need-very-aggressive-abk-book-can-t-find-anywhere

Forums

- Talk Chess A forum for chess talk, including engine programming: https://talkchess.com/ https://discord.gg/mJuKQbrEmN
- Discord Server for Chess Engine Development: https://discord.gg/uTDs7U5Vh7

Engine Evaluation:

- Comparing Strength with Other Engines: https://www.chessprogramming.org/Cutechess-cli
- Engine used to determine strength: https://www.chessprogramming.org/Stash
- Engine Rating List: https://computerchess.org.uk/ccrl/4040
- Open Source Engine High Standard Engine: https://github.com/AndyGrant/Ethereal
- Open Bench for Evaluation: https://github.com/AndyGrant/OpenBench

Perft

- Move generation accuracy is verified using known numbers of legal moves from a given position.
- Information: https://www.chessprogramming.org/Perft
- Online Perft Interface: https://analog-hors.github.io/webperft/
- More Info on Test Results: http://www.rocechess.ch/perft.html

General Resources

- Pre-existing Open Source Engines: https://github.com/EngineProgramming/engine-list
- A Survey of Monte Carlo Tree Search Methods https://www.researchgate.net/publication/235985858_A_Survey_of_Monte_Carlo_Tree_Search_Methods
- Opening Books: https://github.com/official-stockfish/books