Progress Report

Project Mittens

Kate Brown, Dan Hayes, James Parkington

We're in the middle of a massive surge in popularity surrounding the game of chess. What once was considered a niche hobby relegated to clubs of intellectuals has now becomes a prominent thread in the social fabric. This sudden rise in interest, no doubt magnified by the number of us at home, can be attributed to several factors, including:

- the increase in options for accessible chess-playing platforms
- the emergence of popular chess streamers garnering millions of views alongside the top gamers in the country
- the portrayal of chess in mainstream television, particularly *The Queen's Gambit*

With this increase in attention comes a significant increase in players of all backgrounds. The added attention to the game has been amplified by the rapid development of advanced chess bots, evolving the game into a playing ground for machine intelligence, and thereby inviting an entirely new form of competition for eager new users.

This fever pitch for bot play broke the sound barrier with the emergence and sudden decommissioning of **Mittens** in January of 2022. Mittens was, at the time, the most competitive bot available for players to challenge on **chess.com.** Like every player on **chess.com** (*and the other platforms in the space*), each bot is given a rating that informs you of its general strength as a player. These ratings are generally based on ELO, a scoring algorithm devised by Arpad Elo in the 1960s. Players who are adept at finding winning combinations tend to have higher ELO values approaching and sometimes surpassing a value of **2,000**.

However, **chess.com** upon releasing Mittens bestowed it with an ELO of **1**! About **100** times worse than the poorest chess players on its platform. Naturally, this sparked curiosity and attention, followed by thousands of Twitch streams and YouTube videos of grandmasters playing Mittens and attempting to guess its ELO, even pitting Mittens against other well-known bots like Stockfish and AlphaZero to get a stronger sense of its abilities.

This raised some major questions for the **3** of us chess fans of different backgrounds:

- How could someone develop a sense of a competitor's ELO without a value for reference?
- How was Mittens designed to be "better" than other prominent bots?
- Can bots actually make human chess-players more competitive as a whole?
- What's the natural growth of a chess player improving via learning, rather than relying on bots? Can we tell when players cheat by the growth of their ELO?
- And most importantly, did Arpad Elo have any idea what was in store for the future when he devised this algorithm?

Hypotheses

Our goal is to substantiate these **4** premises with available math and research to conclude that **bots are responsible for a measurable growth in the average chess player's ability to play chess to such a degree that it is tempting for top players to cheat**.

1. ELO as a mathematical construct is **positioned to scale** with the introduction of a new "population" of players as bots, which can compute depths of move sequences too large for humans to comprehend.
2. Experienced players have an **intrinsic understanding of ELO** and its variations from platform to platform, and therefore can make educated guesses about a competitor's strength based on their opening choices, demonstration of known winning variations, and overall ability to generate creative patterns of move sequences.
3. The algorithms chess bots use to play at previously unreached levels of precision and accuracy have become **increasingly efficient** at processing move sequences over time, resulting in their accessibility for a wider audience.
4. Humans have shifted their perception of "good" chess and their ability to recreate it in large part to the increased accessibility of said bots, to such a degree that **cheating has become more rampant** at all levels of chess.

Move Sequences

The major reason chess has remained a compelling game to study is because of its unassumingly high complexity. The number of possible chess games is a well-known and frequently cited statistic in the field of chess and computer science. It is often referred to as the **Shannon number** named after the mathematician Claude Shannon who first calculated a figure of 10^{120} in the 1950s[1]. He went on to claim that the number of possible move sequences on a chess board far exceeds the number of atoms in the observable universe.

Estimating the Number of Possible Games

For a rough estimate of the total number of possible chess games, we can use the branching factor formula, $P(d) = b^d$, to calculate the total number of possible games, where b is the branching factor, and d is the average length of a game in ply (or half-turns). You can understand d as the **depth** of known moves at any given point, while b can be thought of as the average number of legal moves available at each point in the game. When advanced players calculate their next moves, they typically review the most competitive move sequences available out to a depth of **8** to **10** ply to determine if a sequence may be winning.

In chess, the branching factor is around **35** and the average length of a game is around **80** ply[2]. We will later verify that figure for the branching factor by using known depths as base cases for a recurrence relation. For now, we can estimate the total number of possible games as follows:

$$\begin{aligned} p(d) &= 35^{80} \\ &= 3.353 \cdot 10^{123} \end{aligned}$$

In other words, this value is **3,353** followed by **120 zeroes**! This estimation assumes that all legal moves are equally likely at each point in the game, which is not entirely accurate since some moves are more strategically

---

[1] Claude Shannon (1950). "Programming a Computer for Playing Chess". p.4. *Philosophical Magazine*. **41** (314). Archived from the original on 2020-05-23.

[2] Victor Allis (1994). *Searching for Solutions in Games and Artificial Intelligence*. p. 171. Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands. ISBN 978-90-900748-8-7.

advantageous than others. However, it still provides a useful approximation of the number of possible chess games.

It's worth nothing that the theoretical endpoint to d has also been previously explored and iterated upon. Fabel, Bonsdorff, and Riihimaa propose that the maximum length of a chess game is **5,898** plies under the established **fifty-move rule** and the **threefold repetition rule**, both of which are designed to prevent endless games[3].

This theoretical maximum takes into account all possible move sequences while adhering to the rules of the game, including stalemate, checkmate, and draw by insufficient material. However, it's important to note that this maximum length is purely theoretical and doesn't account for the strategic choices that players make during a real game.

Comparing Chess with Other Games

Here's how chess compares in raw complexity to some other games that use pieces with limited mobility:

- **Tic-tac-toe**: The branching factor varies depending on the position, but in the early game it is around **3**. Assuming an average game length of **9** moves, $p(d) \approx 3^9 = 19{,}683$.
- **Connect Four**: The branching factor is **7** in the early game. Assuming an average game length of **30** moves, $p(d) \approx 7^{30} = 5.5 \cdot 10^{24}$.
- **Checkers**: Checkers is another popular board game with limited piece mobility. The branching factor is around **5 - 10** in the early game, but decreases as pieces are removed and legal moves disappear. Assuming an average game length of **50** moves, $p(d) \approx 8^{50} = 8.9 \cdot 10^{37}$.
- **Scrabble**: Assuming a standard Scrabble game played with **100** tiles and an average game length of **25** moves, we can make an estimation based on the average number if playable tiles per turn, which is **7** for most of the game.

  Without considering the need to make legal words, $p(d) \approx \binom{100}{7} \cdot 7!^{25} = 5.82 \cdot 10^{75}$

[3] Fabel, Bonsdorff, Riihimaa, (1974). Ajedrez y Matemáticas. p. 13 - 17. Editorial Martínez Roca.

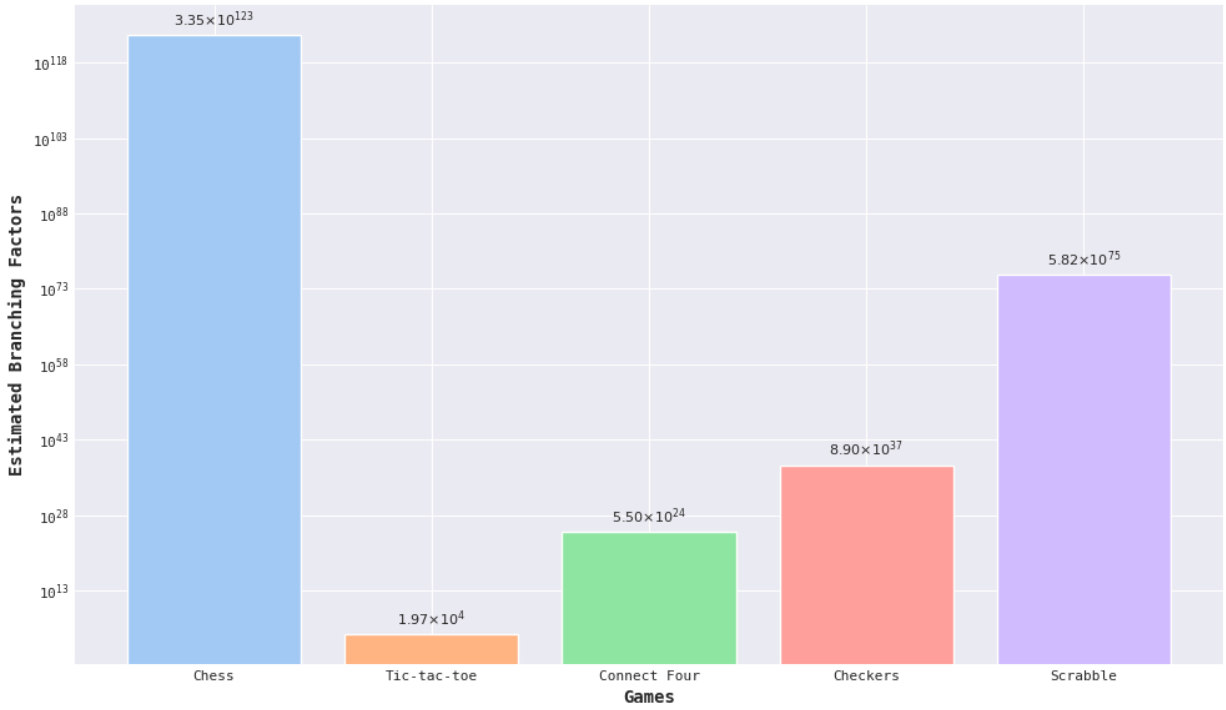

Figure 1: Estimated Branching Factors

Regarding the pursuit of *a winning condition* for each of these games, it must be noted that the branching factor doesn't necessarily correlate to algorithmic complexity.

For example, some games might have a relatively small branching factor but require more sophisticated heuristics to evaluate the strength of different moves, while others might have a larger branching factor but can be more easily pruned or optimized.

However, in general, games with higher game tree complexity and branching factors, like chess, are more computationally difficult than games with lower complexity, like checkers and tic-tac-toe, since evaluating the strength of different moves and find the optimal strategy requires consideration of more move sequences. This difficulty is precisely why chess has become a standard for evaluating the performance of artificial intelligence algorithms, particularly in the area of game-playing agents.

Analyzing Perft Results to Calculate Branching Factors

Earlier, we noted that both Shannon and Allis cite **35** as the average branching factor for a chess game of **80** ply. However, branching factor generally increases as the game progresses, since a moved piece inherently allows the other pieces behind it to also move more freely.

As such, we can back into a branching factor of $b(d)$ using the values of $p(d)$ to compute the ratio of the number of possible games at successive game lengths. Because the branching tree for chess increases one ply at a time, the relationship can be understood as:

$$b(d) = \frac{p(d)}{p(d-1)}$$

From here, we can use verified **Perft** (*performance test, move path enumeration*) results[4] to expand our previous analysis on the growth of the sequence derived from $b(d)$. Using the first **15** ply from analysis by Edwards and Banerjee, we receive the following branching factors:

$$
\begin{aligned}
b(1) &= p(1)/p(0) && = 20 && / && p(0) && = 20 \\
b(2) &= p(2)/p(1) && = 400 && / && p(1) && = 20 \\
b(3) &= p(3)/p(2) && = 8{,}902 && / && p(2) && = 22.255 \\
b(4) &= p(4)/p(3) && = 197{,}281 && / && p(3) && = 22.161 \\
b(5) &= p(5)/p(4) && = 4{,}865{,}609 && / && p(4) && = 24.663 \\
b(6) &= p(6)/p(5) && = 119{,}060{,}324 && / && p(5) && = 24.470 \\
b(7) &= p(7)/p(6) && = 3{,}195{,}901{,}860 && / && p(6) && = 26.843 \\
b(8) &= p(8)/p(7) && = 84{,}998{,}978{,}956 && / && p(7) && = 26.596 \\
b(9) &= p(9)/p(8) && = 2{,}439{,}530{,}234{,}167 && / && p(8) && = 28.701 \\
b(10) &= p(10)/p(9) && = 69{,}352{,}859{,}712{,}417 && / && p(9) && = 28.429 \\
b(11) &= p(11)/p(10) && = 2{,}097{,}651{,}003{,}696{,}806 && / && p(10) && = 30.246 \\
b(12) &= p(12)/p(11) && = 62{,}854{,}969{,}236{,}701{,}747 && / && p(11) && = 29.964 \\
b(13) &= p(13)/p(12) && = 1{,}981{,}066{,}775{,}000{,}396{,}239 && / && p(12) && = 31.518 \\
b(14) &= p(14)/p(13) && = 61{,}885{,}021{,}521{,}585{,}529{,}237 && / && p(13) && = 31.238 \\
b(15) &= p(15)/p(14) && = 2{,}015{,}099{,}950{,}053{,}364{,}471{,}960 && / && p(14) && = 32.562 \\
&\cdots \\
b(80) &= p(80)/p(80-1) && = 3.353 \cdot 10^{123} && / && p(80-1) && \sim 35
\end{aligned}
$$

In other words, we know that after each player has moved a piece **4** times, there are **84,998,978,956** possible games that could have been played.

[4] Chess Programming Wiki. (last modified [29 August 2022]). "Perft Results." chessprogramming.org/Perft_Results.

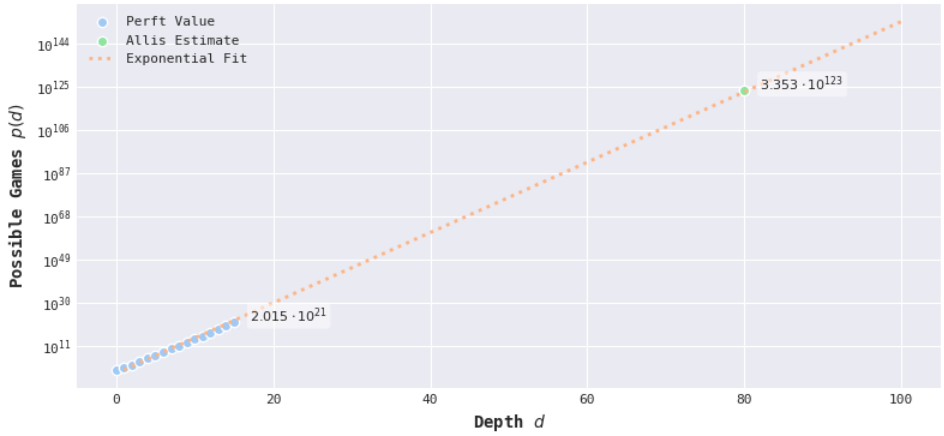

Figure 2 uses the following **exponential regression** to fit values of $P(d)$:

$$p(d) = e^{m \cdot d + b}$$

The fit line has the following slope (m) and y-intercept (b) values:

$$\begin{aligned} m &= 3.5733 \\ b &= -2.8118 \end{aligned}$$

Figure 2: $p(d)$ vs. d

To further quantify the growth of the number of possible games, we can analyze the sequence of branching factors using **big-O notation**, which represents the asymptotic upper bound of a function.

Given $b_{>1} \in \mathbb{R}^+$, and that the number of possible games grows exponentially with the game length, we have:

$$p(d) = \mathcal{O}(b^d)$$

This upper-bounded exponential growth is not unique to chess and can be found in other complex systems, such as the growth of decision trees in artificial intelligence, the development of protein folding patterns, and even the expansion of certain natural phenomena like the growth of crystals.

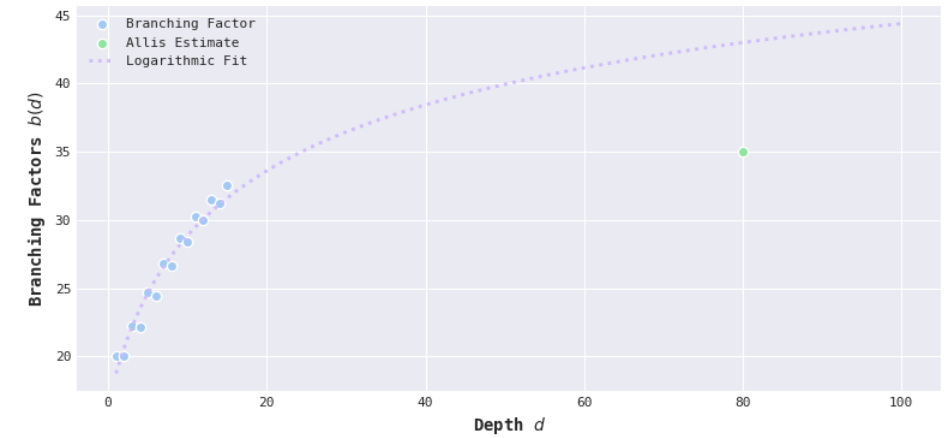

Figure 3 uses the following **logarithmic regression** to fit values of $b(d)$:

$$b(d) = W \log^X(d - Y) + Z$$

W, X, Y, & Z are constants:

- W: Scaling coefficient
- X: Curve-shaping exponent
- Y: Translation constant
- Z: Offset constant

$$\begin{aligned} W &= 21484.6793 \\ X &= 0.0014 \\ Y &= -6.4722 \\ Z &= -21487.1988 \end{aligned}$$

Figure 3: $b(d)$ vs. d

This exponential growth has implications for exceptionally strong players and algorithms alike, as they must efficiently traverse and evaluate vast, quickly growing game trees. Understanding the time complexity of chess not only allows us to appreciate its intricacy but also helps inform the development of more advanced heuristics and artificial intelligence techniques to tackle such a complex system.

Measuring the Growth of Complexity

Note that in spite of the ever-increasing number of move sequences per ply, the rate of increase for $b(d)$ seems to slows down as d increases, especially as it approaches the Allis estimate at $b(80)$. To express this mathematically and to analyze the growth rate of $b(d)$ over increases in d, we can define a separate function, $g(d)$:

$$g(d) = \frac{\log(b(d))}{\log(d)}$$

A higher value for this logarithmic ratio indicates a more rapid growth rate, while a lower value suggests slower growth.

By using the values from the $b(d)$ series, we can calculate $g(d)$ for the same ply depths to better understand and compare the growth behavior of the tree structure over time. Note that $g(1)$ is undefined, because $\log(1) = 0$ for any base, and we cannot divide by zero:

$$\begin{aligned}
g(1) &= \log(b(1))/\log(1) && = 2.996/0 && = \varnothing \\
g(2) &= \log(b(2))/\log(2) && = 2.996/0.693 && \approx 4.322 \\
g(3) &= \log(b(3))/\log(3) && \approx 3.103/1.099 && \approx 2.824 \\
g(4) &= \log(b(4))/\log(4) && \approx 3.098/1.386 && \approx 2.235 \\
g(5) &= \log(b(5))/\log(5) && \approx 3.205/1.609 && \approx 1.992 \\
g(6) &= \log(b(6))/\log(6) && \approx 3.197/1.792 && \approx 1.785 \\
g(7) &= \log(b(7))/\log(7) && \approx 3.290/1.946 && \approx 1.691 \\
g(8) &= \log(b(8))/\log(8) && \approx 3.281/2.079 && \approx 1.578 \\
g(9) &= \log(b(9))/\log(9) && \approx 3.357/2.197 && \approx 1.528 \\
g(10) &= \log(b(10))/\log(10) && \approx 3.347/2.302 && \approx 1.454 \\
g(11) &= \log(b(11))/\log(11) && \approx 3.409/2.397 && \approx 1.422 \\
g(12) &= \log(b(12))/\log(12) && \approx 3.400/2.484 && \approx 1.368 \\
g(13) &= \log(b(13))/\log(13) && \approx 3.451/2.564 && \approx 1.345 \\
g(14) &= \log(b(14))/\log(14) && \approx 3.442/2.639 && \approx 1.304 \\
g(15) &= \log(b(15))/\log(15) && \approx 3.483/2.709 && \approx 1.286 \\
&\cdots \\
g(80) &= \log(b(80))/\log(80) && \approx 3.555/4.382 && \approx 0.811
\end{aligned}$$

Given this calculation, $g(d)$ appears to be asymptotically converging to zero over time. This suggests that the growth of $b(d)$ is **sub-exponential**, or at a slower rate than exponential growth. However, the rate of decrease in $g(d)$ is quite small, so it's possible that $b(d)$ is still growing close to exponentially.

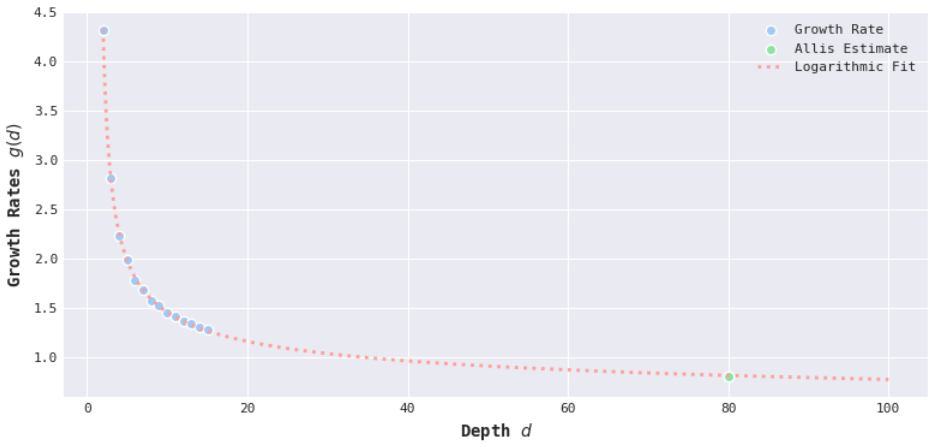

Figure 4 uses the following **logarithmic regression** to fit values of $g(d)$:

$$g(d) = \frac{W(d^X)}{d^Y + Z}$$

W, X, Y, & Z are constants:

- W: Numerator scaling coefficient
- X: Numerator exponent
- Y: Denominator exponent
- Z: Denominator offset constant

$$\begin{aligned} W &= 1.8516 \\ X &= 0.5382 \\ Y &= 0.7339 \\ Z &= -1.0415 \end{aligned}$$

Figure 4: $g(d)$ vs. d

Recall that the theoretical endpoint of chess, $d = 5898$, represents an extreme depth of analysis, which is far beyond the practical reach of even the most advanced chess engines or human players. That said, it is interesting that even at such an immense depth, $g(d) > 0$ using these constants, with an approximate value of **0.339**.

Chess can be considered a finite and deterministic game with perfect information. In principle, it can be solved through exhaustive search, similar to simpler games like tic-tac-toe or Connect Four. However, the unsolved nature of chess stems from its enormous game tree, and the fact that finding an optimal strategy requires searching through this tree more efficiently than today's readily available algorithms and computational resources allow.

The presence of a non-zero growth rate even at the game's theoretical endpoint suggests there is still a degree of complexity and branching that remains to be explored. Understandably, that complexity continues to be a fertile ground for research and innovation in the fields of artificial intelligence, machine learning, and game theory.

Contextualizing Acumen for Move Sequences

To summarize, the data above suggests that the number of possible games available to each player slowly decreases as the game progresses. This makes intuitive sense, since the primary objective is to place the opponent's king in checkmate, which requires restricting the mobility of the king and limiting the available moves of the other pieces. As a game progresses, the number of available moves decreases, and the importance of each move increases, making it more difficult for players to avoid bad move sequences.

The average chess opening lasts between **10** to **12** moves, although this can vary widely depending on the style of play and the specific opening being used. Top chess players must therefore possess an extraordinary level of skill and rote memorization to continue improving their game. Furthermore, they must be able to recognize and evaluate complex and advanced move sequences as the game progresses, and make more precise and effective moves than weaker players. This requires an incredible acumen developed through years of practice and pattern recognition, and the ability to anticipate and respond to their opponent's moves.

Mathematically speaking, we can think of a chess player's improvement as a function of the number of move sequences they are able to recognize, evaluate, and execute in sequence. As they become more skilled, they are able to recognize and evaluate more move sequences, leading to better moves and a higher chance of winning.

This idea of skill level as a function of performance is at the heart of the ELO rating system, which assigns a numerical rating to each player based on their performance in past games, and uses that rating to predict the outcome of future games. As players improve and find more winning move sequences against higher-rated opponents, their rating increases, leading to tougher competition and a greater opportunity to improve further. Ultimately, the ELO system provides a means of quantify a player's knowledge of the strongest move sequences over time.

The History of ELO (Incomplete)

The ELO rating formula is a player rating formula that can be traced back to the 1960s. Arpad Elo, a Hungarian American physics professor and avid chess player, developed a rating system to replace then current system, the Harkness system.

In the 1978 publication of *The Rating of Chessplayers Past & Present* Elo defines the rating system, the formulas, and the mathematics used to create the performance of players and to properly rate them. The system is built on relative inference based on other player's scores.

Breaking Down the Inputs (Incomplete)

The class subdivision into 200 points and choice of 2000 points as reference was preserved when Elo created his calculations.

Since we have an established point system, we can consider the variability in player performance. With the number of players contributing to the system there is a normal distribution curve that can explain the range of expected scores. The statistical probability theory, however, needs to be combined with the competition, Elo suggested. So enters the normal probability function.

The rating system is defined as a logistic function that maps the probability of winning a game between two players to their relative ratings. The formula for his function involves logarithms and summation.

The Elo Rating System can be broken down to the following two equations

These equations calculate the expected score for player A and player B.

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}} \qquad E_B = \frac{1}{1 + 10^{(R_A - R_B)/400}}$$

- E is the expected result of the game
- R_A and R_B are the ratings of player 1 and player 2, respectively.

The formula for his function involves logarithms and summation. We can see that in the ***Current Rating Formula – Continuous Measurement*** which is recognized as

$$\Delta R = K \cdot (S - E)$$

- ΔR is the change in ratings
- K is the constant, set at $K = 16$ for masters and $K = 32$ for weaker players
- S is the actual result of the game (1 for a win, 0 for a loss, 0.5 for a draw)

In this formula, the base-10 logarithm represents the odds of one player winning, given the difference in their ratings. The expected result E is then used to calculate the change in ratings ΔR using that same logistic function.

In understanding these formulas, we'd like to take these equations from Arpad Elo's book on use them to trace the likelihood of different-level players winning matches.

How the Calculation Has Improved (Incomplete)

In its first publication Elo remarked that ratings were subject to the current population. Change in population causes the equations to fluctuate. Hence inevitable change over time.

The change in K-factor for different level of players. The USCF "uses a three-level K-factor for players below 2100, between 2100–2400 and above 2400 of 32, 24 and 16, respectively. FIDE uses similar factors (40, 20 and 10) albeit with somewhat different rules."

Past & Present ELO Values (Incomplete)

Here, we would like to supplement some data on current ratings of top players as well as the agreed upon range of different player levels, and compare them to historical figures.

What was a good ELO when it was first introduced?

- Elo: 5.4 | The Crosstables of 120 Years*
- Elo: 5.5 | The 500 Best Five-Year Averages*

Implementation of the Equation

Start with how ELO works generally, as well as for experienced players using data from chess.com / lichess.org. Then implement the equation and take a look at summations of the equation to see what growth would look like long term.

Getting a look at how long it would take for a strong chess player to reach their accurate strength, and perhaps a bot that never loses and see what would eventually happen there.

How ELO Reflects Chess Intuition (Needs Review)

"We evaluated the relative strength of AlphaGo Zero (Figs 3a, 6) by measuring the Elo rating of each player. We estimate the probability that player a will defeat player b by a logistic function P(a defeats b) = 1 , and 1 + exp(c_elo(e(b) – e(a)) estimate the ratings e(·) by Bayesian logistic regression, computed by the BayesElo program using the standard constant c_elo = 1/400. " - **Mastering the Game of Go**, *p. 361*

The ELO rating system serves as a fairly accurate indicator of a player's chess knowledge, understanding, and intuition, particularly in relation to their mastery of valuable move sequences as a game progresses. Once a player has engaged in a sufficient number of games, their ELO rating can be considered a reliable reflection of their skill level. According to Arpad Elo, a player's rating becomes stable after around 30 games, with a confidence level of approximately 95% after completing 75 games against opponents with established ratings.

For players with exceptionally high ratings, such as those at a master level or above, a greater number of games may be necessary to accurately reflect their skill level. Elo himself suggested that a difference of 200 rating points implies that the higher-rated player would be expected to score about 75% in a match against the lower-rated player.

In his book on the ELO rating system, Arpad Elo posits that from a scientific standpoint, a rating system is essentially a method for pairwise comparison of individual players or teams[5]. Pairwise comparison forms the basis of all scientific measurements, be it physical, biological, or behavioral. The general theory of measurement applies to rating systems and rating scales, allowing for comparisons between two items, one of which serves as the standard.

The primary purpose of the ELO rating system is to predict the outcome of a match based on the past performances of both players, which includes their ability to recognize and execute valuable move sequences as a game progresses. Each player's past performance contributes to their current ELO rating, and once enough games have been played, the rating system can be used to demonstrate a player's overall chess knowledge, understanding, and intuition regarding move sequences.

Projecting a Test Case's Ratings

To illustrate this point, we can examine the case of Player A. Player A creates a new chess account and plays 30 games on it, then compares the new account's rating to the rating of an account they have been playing on for years. The results from the 30 initial 5-minute games on the new account are presented in the figure below. These results, when analyzed in conjunction with the existing account's rating, offer insight into Player A's ability to understand and execute complex move sequences, thereby reflecting their chess intuition as measured by the ELO rating system.

---

[5] Elo, A. (1978). The Rating of Chessplayers, Past and Present. p. 3-4.

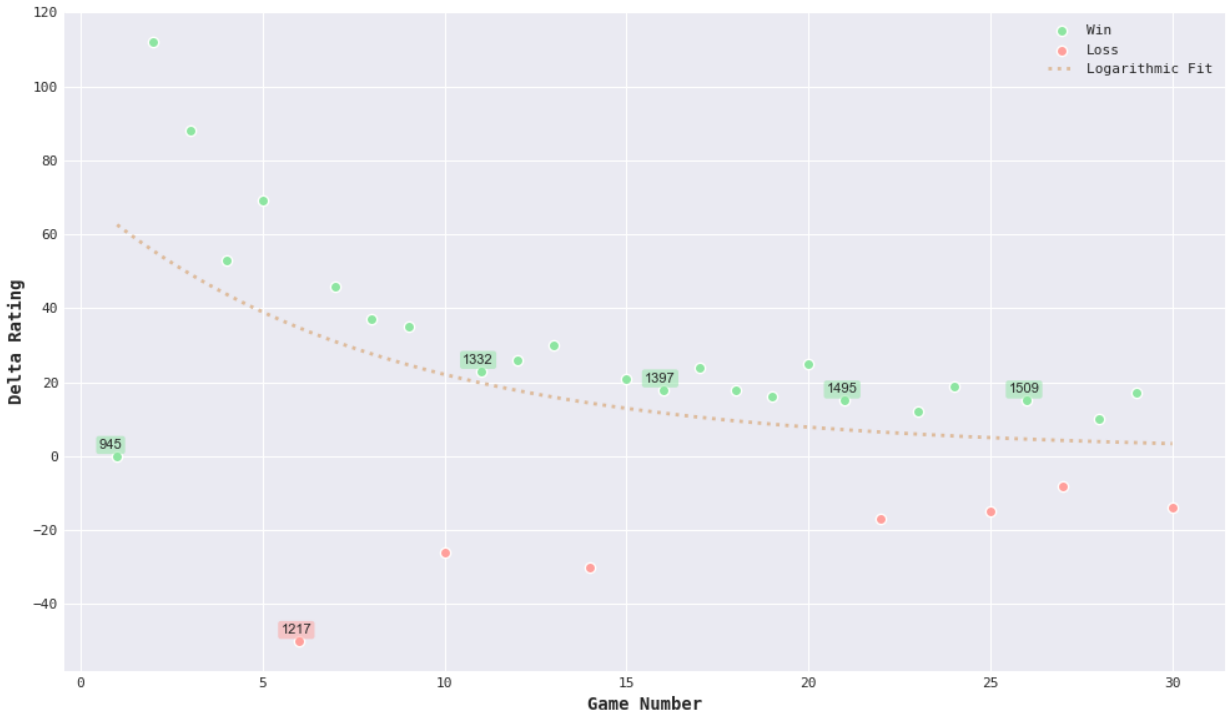

Figure 5: Player A's First 30 Games

Player A's current rapid rating on *chess.com* is **1812**. Although it is unclear whether Player A performed at an **1800** level during this assessment, they managed to reach a rating of **1514** after thirty games. With 23 wins and 7 losses, this data provides an opportunity to examine the ELO equation as presented in Arpad ELO's book.

Using the logarithmic regression derived from Player A's first 30 games, we can estimate their potential rating after 50, 100, 200, and 500 games. It is important to note that these projections are based on the current trend and may not accurately represent future performance. The custom logarithmic curve above generally follows the equation:

$$f(x) = W \cdot \log(x - Y)^X + Z$$

$$\begin{aligned} W &= 2.525 \cdot 10^{96} \\ X &= -130.570 \\ Y &= -200.177 \\ Z &= 1.05g \end{aligned}$$

As Player A's rating increases, it can be expected that their ability to process and select from various move sequences will improve. However, quantifying the exact number of move sequences they can handle is challenging. When compared to an established chess engine like LCZero, a human player's calculation capability is significantly lower. While a chess engine can evaluate millions of positions per second, human players rely on pattern recognition, intuition, and strategic understanding to make decisions.

Nonetheless, let's estimate Player A's rating after the specified number of games:

These projected ratings are purely based on the logarithmic regression and do not account for factors such as improvement in strategic understanding or changes in learning rate. As Player A's rating increases, their ability to calculate and select from move sequences will likely improve, but it will still be far from the capabilities of advanced chess engines.

Objectivity in Good Calculation (Incomplete)

In his book, Arpad Elo poses the intriguing question of whether great performers from one era would surpass those from another. For example, how would chess masters from the 19th century fare against modern grandmasters? Such comparisons assume a common ground, but factors like access to accumulated knowledge and the prevalence of advanced training tools like chess engines and bots must be considered.

Over the past century, the skill level of top chess players has increased significantly, as evidenced by the rise in average ELO ratings. Elo's book provides a sampling of the top players at 25-year intervals, indicating a trend of improvement.

- From 1860 to 1960, the number of top-level players increased from 15 to 50, and their average ELO rating grew by 90 points.
- From 1960 to the present day, the number of players with a 2600 or higher FIDE ELO has risen to approximately 2000* **DAN CHECK THIS ONE**, with an average rating of 2657. This represents an 80-point increase over the 1960 average and a 400% increase in the number of top-level players.

One way to measure the prevalence of grandmasters today is by examining the percentage of grandmasters among all FIDE rated players. According to FIDE data from September 2021, there were approximately 171,000 rated players worldwide, with around 1,700 grandmasters. This means that grandmasters represented nearly 1% of all rated players at the time. As more players join the ranks of FIDE-rated competitors, the number of grandmasters is likely to increase as well, reflecting the continuous improvement of chess skills at the highest level.

Such rapid growth in both ELO ratings and the number of top players can be attributed to factors like better access to resources, more widespread chess education, and the emergence of powerful chess engines and bots that help players refine their skills. These tools have likely contributed to the improvement of chess understanding at the top level.

At what ELO or knowledge of move sequences does it become apparent that someone is "comfortable" not following standard chess conventions?

The Bot Bump (Incomplete)

How has the average ELO rating improved over time as bots have become more endemic and available on major platforms? Can we find data points by decade? Are there obvious spikes?

Is there anything we can say about how humans at the top level THINK more like bots as they progress? At what ratings are players more likely to break chess convention when they play? **Since bots objectively review positions, they don't care about convention at all, so is there an exemplar game where a top-level player breaks all the established rules and is still "performing well" according to the engine?**

Plot a graph of the rise in grandmasters from the beginning of Elo's stats up to the 2020s to see if the total in 2020ish breaks the logarithm.

Bots and Their Algorithms

The rapid development and emergence of advanced chess bots, in tandem with the game's soaring popularity, has profoundly transformed how players interact with the game and substantially influenced the average chess player's capacity to enhance their skills. The complexity of chess lies in the vast number of possible move sequences, with an immense game tree that challenges even the most advanced artificial intelligence algorithms.

In this section, we will delve deep into the open-source chess bot **Leela Chess Zero** (Lc0), which has played a significant role in advancing chess AI due to its transparent methodologies and open-source nature. As Lc0 is an open-source project with a readily accessible GitHub repository and a wealth of academic research behind it, we can explore its construction and methodologies in detail, providing valuable insights into modern chess AI.

As we explore Lc0 and other chess bots, we will address some of the questions raised earlier in this paper, including how Mittens was designed to be "better" than other prominent bots and whether bots can actually make human chess players more competitive as a whole. Moreover, we will discuss the implications of these bots' increasing accessibility to a wider audience, as well as the temptation for top players to cheat by using them.

Introducing Leela

Lc0 was heavily inspired by its closed-source DeepMind predecessor, AlphaZero, which utilizes a combination of deep neural networks and reinforcement learning, and a novel tree search algorithm to achieve groundbreaking performance. The creators of Lc0 adopted a similar combination of methodologies, but with a distributed, community-driven effort to contribute to its training and development.

Lc0's commit history began in January 2018, shortly after AlphaZero's groundbreaking achievements were made public. Since then, multiple versions have been released, with each iteration demonstrating significant improvements over its predecessor. Unlike traditional chess engines, which often focus on material advantage and precise calculations, Lc0 exhibits an intuitive and creative approach to the game, drawing comparisons to human Grandmasters. This has led to a shift in how chess players and enthusiasts perceive the role of AI in chess, with an increased emphasis on understanding, learning from, and emulating Lc0's unique playing style.

Note that the inclusion of "Zero" in the name signifies that no human knowledge has been added to Lc0's understanding of chess[6], apart from the rules for piece movement and victory conditions.

Leela employs a unique combination of neural networking and Monte Carlo Tree Search (MCTS) with a PUCT algorithm (*explained below*) to evaluate positions and explore move sequences. This combination is precisely what differentiated it upon release from other prominent bots, which rely on finely-crafted, human-built systems for generating value and policy.

---

[6] LCZero. (accessed 27 March 2023). "Why Zero?" LCZero Dev Wiki. lczero.org/dev/wiki/why-zero/

Decision-Making Via Neural Network

Lc0 is trained through a combination of supervised learning and reinforcement learning, which serve as the foundation for the bot's decision-making process. Its neural network takes a game state s_t as input and outputs two values[7]:

- $\mathbf{p}$: a **policy vector** representing the probability distribution over possible moves
- v: a scalar **value estimation** of the position between $[-1, 1]$

The network consists of several layers of convolutional and fully connected layers, with residual connections and batch normalization. The final layers of the network are split into two separate "heads," one for the policy and one for the value. The policy head outputs the move probabilities, while the value head estimates the value of the position.

We can understand the computation for a given s_t as:

$$f_\theta(s_t) \mapsto (\mathbf{p}, v)$$

Here, θ represents the network's parameters, which include the weights and biases of each layer. Once trained, the neural network serves as an evaluation function within the MCTS, guiding the search towards promising moves and estimating position values. By integrating the neural network into the MCTS framework, Lc0 can efficiently explore the game tree and make well-informed decisions in complex positions, ultimately leading to a strong and adaptive playing style.

Aligning Predictions with MCTS Policies

Once the neural network is integrated into the MCTS framework, it is crucial to ensure the policy and value heads provide meaningful guidance for the search process. To achieve this, the network is trained to minimize a loss function, which encourages accurate value estimation and a policy that aligns with the target probabilities obtained from the MCTS.

The loss function used for training the neural network is as follows:

$$L(\theta) = (z - v)^2 - \pi^T \log \mathbf{p} + c\|\theta\|^2$$

- $(z - v)^2$: the mean squared error between the predicted value head and the actual game outcome (z) for a given position
- $\pi^T \log \mathbf{p}$: the cross-entropy loss between the predicted probability distribution of legal moves and the search probabilities obtained from the MCTS (π^T), which are derived from the visit counts of the tree search
- $\|\theta\|^2$: an L2 regularization component that helps to prevent overfitting by penalizing large weights in the neural network
- c: a constant that controls the strength of the L2 regularization

This usage of a loss function in training the neural network is essential for Lc0's performance, as it facilitates the alignment of the network's predictions with the MCTS-derived search policies. The combination of mean

---

[7] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Hassabis, D. (2017). Mastering the game of Go without human knowledge. Nature, 550(7676), 354-359. p. 355.

squared error, cross-entropy loss, and L2 regularization has been widely adopted in the field of deep reinforcement learning. It has proven effective in various applications like robotic manipulation and locomotion, Deep Q-Networks, and even some frameworks for autonomous vehicles.

Balancing Exploration and Exploitation

Lc0 employs a variation of MCTS known as the **Predictor + Upper Confidence Bound Tree Search (PUCT)** algorithm[8], striking a balance between exploration and exploitation. While PUCT shares similarities with traditional MCTS, it diverges in its integration of a neural network to inform the search. This neural network supplants the need for rollouts, which are typically employed in standard MCTS models to sample and evaluate games until a terminal state is reached.

Here's a mathematical breakdown of the PUCT algorithm:

$$\mathbf{p} = [p(a_1, s_t),\ p(a_2, s_t),\ \ldots,\ p(a_n, s_t)]$$

$$\forall a \in \mathbf{p} : a_t = \underset{a}{\operatorname{argmax}} \left(Q(a, s_t) + c \cdot p(a, s_t) \cdot \frac{\sqrt{\sum_b \cdot N(b, s_t)}}{1 + N(a, s_t)} \right)$$

$Q(a, s_t)$ is the **exploitation term**, an estimated mean value of taking some action a (*a chess move*) at s_t (*a given board orientation*).

By contrast, $c \cdot p(a, s_t) \cdot \frac{\sqrt{\sum_b \cdot N(b, s_t)}}{1 + N(a, s_t)}$ is the **exploration term**:

- c: a constant that controls the exploration-exploitation trade-off
- $p(a, s_t)$: the prior probability of taking a at s_t, as part of the vector produced by the neural network
- $N(a, s_t)$: the number of visits that a makes to s_t during the tree search
- $\sum_b \cdot N(b, s_t)$: the total number of visits to s_t summed over all actions b

The arguments of the maxima (argmax) are used to find the action that maximizes the balance between the exploitation of high-scoring moves and the exploration of less visited moves. Striking that balance ensures that the search considers a diverse range of move sequences. Because the neural network provides a powerful prior for the search, allowing it to focus on promising moves, argmax ensures that the search does not become too narrow, helping it avoid getting stuck in optimization spaces where a solution is "better" than all its neighboring solutions, but not necessarily the best possible solution.

When considering PUCT's **time complexity**, the algorithms performance is defined by the game's branching factor(s) and the maximum depth of the tree, which we've previously evaluated to be $\mathcal{O}(b^d)$. However, it is important to note that the actual number of iterations and the depth of the tree can be limited by available computational resources or desired search depth, as mentioned earlier.

---

[8] LCZero. (accessed 27 March 2023). "Technical Explanation of Leela Chess Zero." LCZero Dev Wiki. lczero.org/dev/wiki/technical-explanation-of-leela-chess-zero/

On the other hand, while the complexity of evaluating a neural network depends on its architecture, it is generally linear with respect to the number of neurons and connections present in the model.

Once the PUCT algorithm has been executed and the argmax operation is complete, the resulting a_t would contain the move that has the highest combined score from the exploitation and exploration terms, which would be the next move played by the AI in-game.

Fine-Tuning the Algorithm's Search Strategy

During MCTS, Lc0 uses temperature scaling to control the level of exploration. Temperature scaling adjusts the move probabilities by raising them to the power of an inverse temperature parameter, τ, and then normalizing the probabilities[9]. Note that higher values of τ increase the level of exploration, while lower values make the search more focused on the highest-probability moves:

$$\alpha = \frac{1}{\tau}$$

$$p_a^* = \frac{N(a, s_t)^\alpha}{\sum_b N(b, s_t)^\alpha}$$

- $N(a, s_t)$: previously defined in the PUCT algorithm, and used in temperature scaling to adjust the probabilities based on the exploration of the search
- $\sum_b N(b, s_t)^\alpha$: normalizes the adjusted probabilities by summing the raised probabilities to 1

The result of this formula, p_a^*, represents the adjusted probability of action a after applying temperature scaling. These probabilities are used during the selection phase of the search when deciding which child nodes to traverse. A temperature of $\tau = 1$ is used during the first 30 moves of each game in self-play to encourage exploration, while for the remainder of the game, the temperature is set to approach zero effectively choosing the move with the highest visit count.

These adjusted move probabilities will influence which nodes are chosen for traversal and thus have an impact on the overall search. By modulating the temperature parameter, the balance between exploration and exploitation can be fine-tuned, which allows Lc0 to adapt its playing style and decision-making based on the specific position and game context.

Bellman Equation (Incomplete)

A key aspect of solving for the optimal $Q(a, s_t)$ values involves the Bellman equation, which can be used to express the relationships between the current game state and all future game states. The Bellman equation is based on the **principle of optimality**, which states that for an inductive strategy to be optimal, it must make the best possible decisions at every step, considering the new situations arising from previous actions.

By contrast, the more commonly-used **greedy inductive policy** is a strategy that chooses the action with the highest immediate reward at each time step, without considering the long-term consequences of these actions.

---

[9] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Hassabis, D. (2017). Mastering the game of Go without human knowledge. Nature, 550(7676), 354-359. p. 358.

While a greedy inductive policy can sometimes result in good decisions, it often fails to consider the overall impact of its choices, leading to suboptimal performance.

Backward induction is a method used to solve the Bellman equation by identifying the most optimal action at the last node in the decision tree (*in this case, checkmate or a draw*) and working backwards until the best action for every possible situation is determined.

We can understand how the PUCT derives value in terms of the Bellman constraint for Q-values[10]:

$$Q(a, s_t) = u(a, s_t) + \sum p(s_t^* : a, s_t) \cdot \max_a Q(a, s_t^*)$$

- $p(s_t^* : a, s_t)$: the probability of moving to a new state given the current state and action
- s_t^*: the board state after a is taken and the opponent has responded
- $u(a, s_t)$: a move's instantaneous utility, which can be replaced with the value head output, $v(s_t)$

Recall that $v(s_t) \in [-1,1]$ is the output of the value network, which estimates the probability of winning from the current position.

In the Lc0 context, we can take the maximum value of $Q(a, s_t)$ over the current action to represent the total value function $V(s_t)$ after incorporating the value network output into the policy network's probability predictions:

$$V(s) = \max_a \left(v(s) + \sum p(s_t^* : a, s_t) \cdot \max_a Q(s^*, a) \right)$$

Complete this v(s_t) conversation and confirm that the logic still applies. Explain each term in V(S).

Centi-Pawn Conversion (Incomplete)

After applying the temperature-scaled policy and value results from the neural network back to the a_t under consideration, Lc0 takes the $Q(a, s_t)$ in the range of $[-1, 1]$ and converts it into a traditional centipawn (P_c) evaluation.

While these formulas are not readily available, given that the result of the Bellman equation is a win probability, the centi-pawn values generated by Lc0 may resemble the approach for converting win advantage to pawn advantage by Fischer and Kennan in 2007[11], which they devised for approximating ELO values:

$$P_c = 4 \log_{10} \frac{V(s)}{1 - V(s)}$$

---

[10] Maharaj, S. et al. (2022). Chess AI: Competing Paradigms for Machine Intelligence. p. 3-4.

[11] Chess Programming Wiki. (accessed March 27, 2023). "Pawn Advantage, Win Percentage, and Elo." https://www.chessprogramming.org/index.php?title=Pawn_Advantage,_Win_Percentage,_and_Elo

Leela's ability to efficiently explore move sequences and evaluate positions stems from the synergy between the neural network and the MCTS. By leveraging the neural network's capacity for learning and generalization, it can quickly identify promising moves and focus its search efforts on the most relevant parts of the game tree. That said, the ongoing exploration of alternative search algorithms within the intersection of AI and Game Theory highlights the potential for future advancements, with PUCT remaining the dominant choice for the time being.

Is this the correct formula to use for centipawn conversion?

Mittens (Incomplete)

To be started

Other Citations

1. ***(2005) "A Psychometric Analysis of Chess Expertise" by Han L. J. Van Der Maas, Eric-Jan Wagenmakers***

    - This study introduces the Amsterdam Chess Test (ACT) and explains the math behind the 5 tests within, a lot of which uses sets and series to understand large ranges of possible move sequences of moves
    - Their results show that the ACT has high predictive validity, suggesting that strong players have an inherent human understanding of chess complexity and when it's created by other humans
    - ELO is their foundational measurement, which they use as a dependency for much of their reporting

2. ***(2005) "Chess: A Cover Up" by Eric K. Henderson et al.***

    - The goal of this team's mathematical approach was to supply a more "modern" formula set for how possible moves could be "pruned" down to a sequence of logical next steps for a player to make
    - This study does an excellent job of breaking down some of the possible move sequences and piece movements on the board into digestible mathematical formulas
    - Much of this seems like it would be foundational to the programming of an early-stage chess bot, in that its pruning mechanism suggests our current application of "depth"

Links to Revisit

- Reviews calculations for b with nice visuals: growth
- Discusses theoretical endpoint for b: https://wismuth.com/chess/longest-game.html
- OEIS links for possible chess positions: https://oeis.org/wiki/Index_to_OEIS:_Section_Ch#chess
- Perft introduction: https://www.chessprogramming.org/Perft
- Reddit post on Neimann's centipawn loss: https://www.reddit.com/r/chess/comments/xv4rc0/a_more_thorough_look_at_niemanns_centipawn_loss/

- Interesting discussion on combinatorial game theory: https://chess.stackexchange.com/questions/27259/has-anyone-attempted-to-characterize-chess-mathematically

Abstract

Script for Figure 1

```python
import matplotlib.pyplot as plt
import numpy             as np
import seaborn           as sns
plt.style.use('seaborn-darkgrid')

# Prepare the data
games             = ["Chess", "Tic-tac-toe", "Connect Four", "Checkers", "Scrabble"]
branching_factors = [3.353e123, 19683, 5.5e24, 8.9e37, 5.82e75]
colors            = sns.color_palette('pastel')

# Create the bar chart
bars = plt.bar(games, branching_factors, color = colors[:5], edgecolor = 'white')

plt.xlabel("Games", fontweight = 'bold')
plt.ylabel("Estimated Branching Factors", fontweight = 'bold')

# Customize the plot appearance
ax = plt.gca()
ax.spines[:].set_visible(False)
ax.tick_params(axis = 'both', which = 'major', labelsize = 8)
ax.set_yscale("log")

# Add annotations to the top of each bar
for bar, bf in zip(bars, branching_factors):
    height   = bar.get_height()
    exponent = int(np.floor(np.log10(bf)))
    base     = bf / (10 ** exponent)
    ax.annotate(f'${base:.2f} \times 10^{{{exponent}}}$',
                xy         = (bar.get_x() + bar.get_width() / 2, height),
                xytext     = (0, 3),
                textcoords = "offset points",
                ha         = 'center',
                va         = 'bottom',
                fontsize   = 8,
                fontweight = 'bold',
                fontfamily = 'DejaVu Sans Mono')

# Update the font family for all text elements
font_updates = [ax.title, ax.xaxis.label, ax.yaxis.label] + \
               ax.get_xticklabels() + ax.get_yticklabels()
for i in font_updates:
    i.set_fontfamily('DejaVu Sans Mono')

plt.show()
```

Script for Figure 2

```python
from sklearn.linear_model import LinearRegression as lr
import matplotlib.pyplot  as plt
import numpy              as np
import seaborn            as sns
plt.style.use('seaborn-darkgrid')

# Prepare the data
colors      = sns.color_palette('pastel')
depths      = np.array(list(range(0, 16)) + [80])
positions   = np.array([1, 20, 400, 8902, 197281, 4865609, 119060324, 3195901860, 84998978956, 2439530234167, 69352859712417, 2097651003696806,
62854969236701747, 1981066775000396239, 61885021521585529237, 2015099950053364471960, 3.353e123], dtype = float)
annotations = [((15, 2.015e21), r'$2.015 \cdot 10^{21}$'), ((80, 3.353e123), r'$3.353 \cdot 10^{123}$')]

# Fit the linear regression model and generate predictions for the fit line
model       = lr().fit(depths[:, None], np.log(positions))
depth_range = np.linspace(1, 100, 1000)
predictions = np.exp(model.predict(depth_range[:, None]))

# Create the scatter plot
plt.scatter(depths,
            positions,
            color      = colors[0],
            marker     = 'o',
            edgecolors = 'white',
            label      = "Perft Value")

# Isolate the point for the "Allis Estimate"
plt.scatter(np.array([80]),
            np.array([3.353e123]),
            color      = colors[2],
            marker     = 'o',
            edgecolors = 'white',
            label      = "Allis Estimate")

# Create the fit line
plt.plot(depth_range,
         predictions,
         color     = colors[1],
         linestyle = ':',
         linewidth = 2,
         label     = "Exponential Fit")

# Add annotations as defined above to the plot
for pos, text in annotations:
    plt.annotate(text, pos, (pos[0] + 2, pos[1] * 1.5), fontsize = 8,
                 bbox = dict(boxstyle  = "round,pad = 0.2",
                             edgecolor = "none",
                             facecolor = "white",
                             alpha     = 0.5))

plt.xlabel("Depth $d$",             fontweight='bold')
plt.ylabel("Possible Games $p(d)$", fontweight='bold')

# Customize the plot appearance
ax = plt.gca()
ax.spines[:].set_visible(False)
ax.set(yscale = 'log')
ax.tick_params(axis = 'both', which = 'major', labelsize = 8)

# Update the font family for all text elements
font_updates = [ax.title, ax.xaxis.label, ax.yaxis.label] + ax.get_xticklabels() + ax.get_yticklabels() + plt.legend(loc = 'best', fontsize =
8).get_texts()
for i in font_updates:
    i.set_fontfamily('DejaVu Sans Mono')

plt.show()
```

Script for Figure 3

```python
from scipy.optimize import curve_fit
import matplotlib.pyplot  as plt
import numpy              as np
import seaborn            as sns
plt.style.use('seaborn-darkgrid')

# Prepare the data
colors    = sns.color_palette('pastel')
depths    = np.array(list(range(1, 16)) + [80])
branching = np.array([20, 20, 22.255, 22.161, 24.663, 24.470, 26.843, 26.596, 28.701, 28.429, 30.246, 29.964, 31.518, 31.238, 32.562, 42.5],
dtype = float)

# Define a custom logarithmic function
def custom_logarithmic(x, a, b, c, d):
    return a * np.log(x - c) ** b + d

# Fit the custom curve
depth_range = np.linspace(1, 100, 1000)
predictions = custom_logarithmic(depth_range, *curve_fit(custom_logarithmic,
                                                         depths,
                                                         branching,
                                                         maxfev = 100000,
                                                         p0 = (20, 1, 0, 0))[0])
# Create the scatter plot
plt.scatter(depths[:-1],    # Ignore the data pad at the end of the series
            branching[:-1], # Ignore the data pad at the end of the series
            color      = colors[0],
            marker     = 'o',
            edgecolors = 'white',
            label      = "Branching Factor")

# Isolate the point for the "Allis Estimate"
plt.scatter(np.array([80]),
            np.array([35]),
            color      = colors[2],
            marker     = 'o',
            edgecolors = 'white',
            label      = "Allis Estimate")

# Create the fit line
plt.plot(depth_range,
         predictions,
         color     = colors[4],
         linestyle = ':',
         linewidth = 2,
         label     = "Logarithmic Fit")

plt.xlabel("Depth $d$",               fontweight = 'bold')
plt.ylabel("Branching Factor $b(d)$", fontweight = 'bold')

# Customize the plot appearance
ax = plt.gca()
ax.spines[:].set_visible(False)
ax.tick_params(axis = 'both', which = 'major', labelsize = 8)

# Update the font family for all text elements
font_updates = [ax.title, ax.xaxis.label, ax.yaxis.label] + ax.get_xticklabels() + ax.get_yticklabels() + plt.legend(loc = 'best', fontsize =
8).get_texts()
for i in font_updates:
    i.set_fontfamily('DejaVu Sans Mono')

plt.show()
```

Script for Figure 4

```
from scipy.optimize import curve_fit
import matplotlib.pyplot  as plt
import numpy              as np
import seaborn            as sns
plt.style.use('seaborn-darkgrid')

# Prepare the data
colors      = sns.color_palette('pastel')
depths      = np.array(list(range(2, 16)) + [80])
g_values    = np.array([4.322, 2.824, 2.235, 1.992, 1.785, 1.691, 1.578, 1.528, 1.454, 1.422, 1.368, 1.345, 1.304, 1.286, 0.811])

# Define a custom logarithmic function
def custom_log(x, a, b, c, d):
    return a * (x ** b) / (x ** c + d)

# Fit the custom curve
depth_range = np.linspace(2, 100, 1000)
predictions = custom_log(depth_range, *curve_fit(custom_log,
                                                 depths,
                                                 g_values,
                                                 maxfev = 100000,
                                                 p0 = (1, -1, 1, 0))[0])
# Create the scatter plot
plt.scatter(depths,
            g_values,
            color      = colors[0],
            marker     = 'o',
            edgecolors = 'white',
            label      = "Growth Rate")

# Isolate the point for the "Allis Estimate"
plt.scatter(np.array([80]),
            np.array([0.811]),
            color      = colors[2],
            marker     = 'o',
            edgecolors = 'white',
            label      = "Allis Estimate")

# Create the fit line
plt.plot(depth_range,
         predictions,
         color     = colors[3],
         linestyle = ':',
         linewidth = 2,
         label     = "Logarithmic Fit")

plt.xlabel("Depth $d$",            fontweight='bold')
plt.ylabel("Growth Rates $g(d)$",  fontweight='bold')

# Customize the plot appearance
ax = plt.gca()
ax.spines[:].set_visible(False)
ax.tick_params(axis = 'both', which = 'major', labelsize = 8)

# Update the font family for all text elements
font_updates = [ax.title, ax.xaxis.label, ax.yaxis.label] + ax.get_xticklabels() + ax.get_yticklabels() + plt.legend(loc = 'best', fontsize =
8).get_texts()
for i in font_updates:
    i.set_fontfamily('DejaVu Sans Mono')

plt.show()
```

Script for Figure 5

```python
from scipy.optimize import curve_fit
import matplotlib.pyplot  as plt
import numpy              as np
import seaborn            as sns
plt.style.use('seaborn-darkgrid')

# Prepare the data
colors        = sns.color_palette('pastel')
game_numbers  = list(range(1, 31))
delta_ratings = [0, 112, 88, 53, 69, -50, 46, 37, 35, -26, 23, 26, 30, -30, 21, 18, 24, 18, 16, 25, 15, -17, 12, 19, -15, 15, -8, 10, 17, -14]
win_loss      = [2, 2, 2, 2, 2, 3, 2, 2, 2, 3, 2, 2, 2, 3, 2, 2, 2, 2, 2, 2, 2, 3, 2, 2, 3, 2, 3, 2, 2, 3]
ratings       = [945, 1057, 1145, 1198, 1267, 1217, 1263, 1300, 1335, 1309, 1332, 1358, 1388, 1358, 1379, 1397, 1421, 1439, 1455, 1480, 1495,
1478, 1490, 1509, 1494, 1509, 1501, 1511, 1528, 1514]

def custom_log(x, a, b, c, d):
    return a * np.log(x - c) ** b + d

# Fit the custom curve
game_range  = np.linspace(1, 30, 1000)
predictions = custom_log(game_range, *curve_fit(custom_log,
                                                game_numbers,
                                                delta_ratings,
                                                maxfev = 100000,
                                                p0     = (20, 1, 0, 0))[0])
# Create the fit line
plt.plot(game_range,
         predictions,
         color     = colors[5],
         linestyle = ':',
         linewidth = 2)

for idx, (gn, dr, wl, rt) in enumerate(zip(game_numbers, delta_ratings, win_loss, ratings)):
    plt.scatter(gn,
                dr,
                color      = colors[wl],
                marker     = 'o',
                edgecolors = 'white')

    if idx % 5 == 0:
        plt.annotate(rt,
                     (gn, dr),
                     (gn - 0.5, dr + 2),
                     fontsize = 8,
                     bbox = dict(boxstyle  = "round,pad=0.2",
                                 edgecolor = "none",
                                 facecolor = colors[wl],
                                 alpha     = 0.5))

plt.xlabel("Game Number",  fontweight = 'bold')
plt.ylabel("Delta Rating", fontweight = 'bold')

# Create the legend
win_marker  = plt.scatter([], [], color = colors[2], marker    = 'o', edgecolors = 'white', label = 'Win')
loss_marker = plt.scatter([], [], color = colors[3], marker    = 'o', edgecolors = 'white', label = 'Loss')
fit_line    = plt.plot([],    [], color = colors[5], linestyle = ':', linewidth  = 2,       label = 'Logarithmic Fit')[0]

plt.legend(handles = [win_marker,
                      loss_marker,
                      fit_line],
                      loc      = 'best',
                      fontsize = 8)

ax = plt.gca()
ax.spines[:].set_visible(False)
ax.tick_params(axis = 'both', which = 'major', labelsize = 8)

font_updates = [ax.title, ax.xaxis.label, ax.yaxis.label] + \
               ax.get_xticklabels() + ax.get_yticklabels() + \
               plt.legend(loc = 'best', fontsize = 8).get_texts()
for i in font_updates:
    i.set_fontfamily('DejaVu Sans Mono')

plt.show()
```