How Chess Engines Employ Search Algorithms

Colby Ryan TJ Schlueter (Advisor)

March 19, 2024

Abstract

This research aims to explore the nuances of the distinct search methods and evaluation equations of the chess engines Stockfish and Leela Chess Zero, by how they differ in their evaluation of random and intricate chess positions with asymmetrical material imbalances. By comparing how these engines employ distinct search methods and evaluation equations, we gain a deeper understanding of how AI perceives and solves problems, or struggles with counter intuitive ones. This research will ultimately shed light on the evolving landscape of intelligent decision-making algorithms beyond the confines of chess.

Contents

1 Introduction

Even though chess has been played for thousands of years, it has only been in the last few decades that the theory behind the game has advanced tremendously. This is because of recent computer programs leap-frogging humans' ability to play the game. These computer programs are called chess engines.

In 1997, the chess computer known as Deep Blue defeated the then-world chess champion, Garry Kasparov, in a series of matches that marked a pivotal moment in the history of chess and artificial intelligence [1]. It was a watershed event that not only established the dominance of computers in abstract strategy games but also ignited a rapid and ongoing evolution in the world of chess engines. These engines have since become invaluable tools for chess players of all levels, offering them the opportunity to analyze their games, learn from their mistakes, and continuously refine their skills.

Since chess engines have cemented themselves as consistently stronger than human players, they still compete with each other and advance their knowledge of chess and abilities [10]. On top of chess engines dominating the chess world and furthering our knowledge of the game, their continued advancement serves as benchmarks and insights into how Artificial Intelligence approach complex problems.

In any given chess board position, there exists the potential for a solution- an elusive "best" move that a player can make. In the realm of game theory, chess is categorized as a 'game of perfect information'. Perfect information refers to the fact that at any given moment in the game, each player has the same information that will be available at the end of the game. This is, each player knows or can see other player's moves. A game of perfect information provides a well-defined environment for testing and developing intelligent systems because there is never luck or guessing involved [11]. Thus, the entirety of the game can be reduced to computations, which is in part why computers will likely always be better than humans in the future.

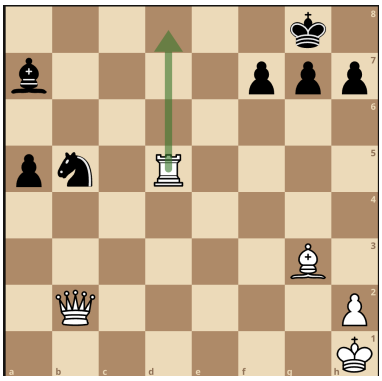


Figure 1: White to move. This position demonstrates a decisive checkmate

This position shows identifying the optimal move is relatively straightforward at times. There are many moves here which the white player can make and still maintain a significant advantage, and likely a forced checkmate pattern, but the 'optimal' move refers to the one in which white decidedly wins the game immediately.

The problem lies in uncovering the optimal move in any given position, and this has been a driving force in artificial intelligence computing and chess engines [3]. This problem of finding the 'optimal' move becomes increasingly difficult in chess 'puzzles' which typically involve a counter intuitive move. These types of moves - ones which look extremely poor upon initial examination - are difficult to spot for humans and computers alike.

So why is chess such a strong benchmark for AI? It is a game that embraces perfect information, where no details are concealed from the players, and every move, be it a brilliant sacrifice or a subtle positional shift, can dramatically alter the course of the game. This characteristic, coupled with the vast number of possible board configurations, makes chess an ideal arena for studying artificial intelligence and problem-solving. In chess, the number of potential positions (from legal moves) is estimated to be between 10^{40} and 10^{50} [19]. This estimation also does not include pawn-promotions, which significantly increases the estimated upper-bound of potential positions. This immense complexity arises from the game's relatively simple rules.

This research seeks to delve deeper into the rich history and ongoing evolution of chess engines, and seeks to answer the question: To what extent do Stockfish and Leela Chess Zero differ in their evaluation of random, complex chess positions with asymmetrical material imbalances? Through a comprehensive analysis of how these engines employ their distinct search methods and evaluation equations, we aim to gain a more profound understanding of how artificial intelligence perceives and resolves complex problems, as well as how it grapples with counter intuitive scenarios. By unraveling the strategies and methodologies utilized by these engines, we hope to shed light on the ever-changing landscape of intelligent decision-making algorithms, with implications that extend far beyond the realm of chess.

It is essential to provide a brief overview of the two chess engines that will be the focus of our analysis. Stockfish and lc0 represent different paradigms in the world of computer chess, each showcasing distinctive strengths and strategies. They are two of the strongest chess-playing engines in the world, and by exploring their contrasting approaches to chess problem-solving, we will gain a deeper understanding of the rich landscape of chess engines and the evolving landscape of intelligent decision-making algorithms.

Stockfish

Stockfish [21] has consistently been one of the most dominant chess engines, and is currently the best engine in the world. Stockfish is a venerable and formidable chess engine, traces its lineage back to the early days of computer chess. It employs traditional search methods and evaluation equations, relying on its ability to explore vast numbers of positions and analyze chess games to a considerable depth. Stockfish has established itself as one of the strongest chess engines globally, often rivaling and surpassing the capabilities of grand masters. Its utilization of brute-force search, combined with an evaluation function, enables it to navigate complex chess scenarios with exceptional precision. It is currently the strongest chess engine in the world.

Stockfish relies on a traditional approach to chess problem-solving. Its methodology is rooted in being able to quickly search because it can evaluate positions so quickly. The evaluation function employed by Stockfish encompasses a wide array of chess principles, including material imbalances, pawn structures, piece mobility, and strategic motifs. This equation, crafted through years of human chess expertise, allows Stockfish to assess positions and make informed decisions based on a comprehensive understanding of the game. Since it's a single equation, it's able to provide a consistent value almost immediately. When faced with a chess position, Stockfish systematically explores a vast number of potential moves, creating a decision tree that extends to considerable depths. This exhaustive search enables Stockfish to analyze positions with remarkable precision.

In the context of our analysis on chess puzzles, Stockfish's methodical search algorithm and well-established evaluation function will be scrutinized. The engine's performance on puzzles with counter-intuitive moves will shed light on the efficiency of traditional chess engine approaches in tackling complex chess challenges [21].

Leela Chess Zero

On the other hand, Leela Chess Zero [12], often referred to as Leela or lc0, represents the new wave of artificial intelligence in chess. Moving forward in this paper, "lc0" will refer to Leela Chess Zero. lc0 is an open-source project based on Google's AlphaZero, which came into the chess scene in 2017 and dominated Stockfish in a 100-game matchup. lc0 uses neural networks and machine learning techniques to develop a more intuitive and human-like understanding of the game. Rather than relying solely on traditional search algorithms, lc0 emphasizes selective searching and deep evaluation, taking inspiration from the way humans approach chess. Its ability to learn and adapt from its own experiences, coupled with its selective searching, has propelled it to new heights in the world of computer chess, often outperforming traditional engines in specific domains. Because of it's emphasis on deep evaluations of a position, it is unable to search nearly as many positions as Stockfish.

Leela Chess Zero represents a departure from traditional chess engine equations, embracing neural networks and machine learning techniques for decision-making. lc0 learns from self-play, continuously refining its understanding of chess dynamics. Unlike rule-based evaluation functions, lc0's evaluation is

driven by deep neural networks, allowing it to capture complex patterns and relationships within chess positions. This comes at the expensive of it's search function.

In our analysis of chess puzzles, lc0's emphasis on neural networks and self-learning will be a focal point. We aim to explore how lc0 navigates puzzles with counter intuitive moves, leveraging its adaptive evaluation to provide a distinctive perspective on chess problem-solving. The comparison between Stockfish and lc0 in puzzle scenarios will contribute valuable insights into the diverse approaches employed by chess engines in addressing complex challenges. [12]

The study of Stockfish and lc0 on chess puzzles will provide us with a unique opportunity to dissect the distinctive approaches they employ when faced with different chess scenarios. By comparing their strategies and methodologies, we can explore how each engine navigates and resolves complex problems and, crucially, how they contend with counterintuitive moves within the puzzle framework. This examination will offer a comprehensive understanding of the strengths and limitations of these engines, shedding light on the ongoing evolution of intelligent decision-making algorithms.

1.1 History

Chess engines and theory have been studied extensively. In 1950, Claude Shannon produced two strategies to implement a chess engine: The first being a brute-force search of all possible positions, and the second being an implementation of weighting favorable moves to simulate the way humans play chess [18]. Modern chess engines still use these techniques.

In 1953, Alan Turing designed a chess-playing program which at the time was impossible to create due to lack of processing power and memory [13]. His innovative program was still a pioneer for early exploration of artificial intelligence, chess programming, and game theory. In his chapter, he discussed the idea of using digital computers to simulate human thought processes and implementing chess theory into a computer. He introduces the idea of board evaluation in a digital computer.

In 1957, Alex Bernstein was the primary author for the Bernstein Chess Program, affiliated with IBM. This was the first ever complete chess program [2]. Although it was able to be beaten by novice players, this was a massive step in the development of chess engines.

Chess engines continued to improve in the 1960's through 1980's while new techniques were being tried and implemented [4]. In 1975, Donald Knuth produced a detailed analysis of alpha-beta pruning, which is a noteworthy search algorithm still used today [7]. Alpha-beta pruning significantly decreases the number of nodes which require evaluation by halting the evaluation of a branch when one possibility proves to be worse than previous ones evaluated. Other search algorithm enhancements such as iterative deepening and move ordering were also discovered and researched during these decades [22].

It was not until 1997, however, for the chess engine known as Deep Blue to defeat then world chess champion Garry Kasparov in a 6-game match, making history [1]. This did not end the research and development of chess engines.

Deep Blue beats chess champ

IBM's computer stages winning attack in first of six-game match

By The Associated Press

PHILADELPHIA — A chess computer turned retreat into a winning attack Saturday to defeat world champion Garry Kasparov in the first of a six-game match.

IBM's Deep Blue can master a move no human can accomplish: shifting through more than 200 million possible chess maneuvers in a second. The duel is the first to pit human against machine for a regulation, six-game chess match.

Kasparov ceded defeat on the 37th move when Deep Blue pinned his king between a knight and a rook.

Playing black, Kasparov was putting heat on Deep Blue's king in the 28th move when the computer managed to maneuver its way out of a defensive posture by capturing a key Kasparov pawn.

By the 29th move, Grandmasters Yasser Seirawan and Maurice Ashley were saying that Kasparov lost.

The glowering, 32-year-old Ukrainian champion left the game site in Philadelphia's Convention center without a word.

The three-hour defeat came much quicker than Deep Blue's design team had even dared to hope, IBM computer engineer Chung Jen Tang said after the match.

Kasparov's defeat came as a surprise to both chess grandmasters and IBM programmers, who thought Kasparov's aggressive attack would give him the day.

"I know my opponent is invisible, but I strongly believe that it's not invincible," Kasparov said before the match.

In 1989, he proved critics wrong by handily defeating Deep Thought, the IBM prototype for Deep Blue.

TODAY'S SCRIPTURE

"And God said, Let there be light: and there was light."

— Genesis 1:3

Figure 2: As one of the most landmark moments in Chess and AI history, it made front page headlines [1]

1.2 Modern Related Work

Many studies explore the tuning of a chess evaluation function using new algorithms and values. In one such study published in 2012, they performed an exploration search through an evolutionary algorithm, and then a tuning algorithm to adjust value weights. This tuning method was able to successfully increase the elo-rating of their chess engine by a substantial 200 points [22].

In 2017, DeepMind (bought by Google) developed a self-playing chess algorithm called AlphaZero, which relied entirely on self-play. After playing itself for only 24 hours, AlphaZero was able to beat the most powerful engine in the world, Stockfish [9]. Interestingly, AlphaZero, plays chess eerily similarly to humans although only learning by itself. Since it was given no theoretical input, AlphaZero developed it's own piece-values and concept values, which are very similar to the values produced by chess theory. This was the first chess engine to introduce neural networks and self-learning concepts, which have since become the norm.

Chess AI's using different machine learning methods can be studied to explore their respective advantages. A 2022 study uses Stockfish and lc0 and examines how they perform at an endgame puzzle with extremely counter-intuitive moves needed to complete it [8]. They found that Stockfish greatly outperformed lc0. This is because Stockfish efficiently searched 1.9 billion positions, whereas lc0 emphasizes evaluation and employs very selective searching.

2 Background

2.1 Search Algorithms

Within the realm of chess engines, search algorithms play a fundamental role in uncovering the optimal moves in a given position [5]. These algorithms determine how engines traverse the vast tree of possible moves and positions, culminating in a decision that identifies the best course of action. Search algorithms enable engines to evaluate positions, calculate potential outcomes, and ultimately select the move that offers the greatest advantage.

One of the most notable search algorithms in the domain of chess engines is alpha-beta pruning, a technique introduced by Donald Knuth in 1975 [7]. This algorithm significantly reduces the number of nodes that require evaluation by ceasing the evaluation of a branch when one possibility proves to be worse than previously evaluated ones. By systematically pruning away less promising branches of the decision tree, alpha-beta pruning efficiently narrows down the search space, ultimately leading to faster and more effective decision-making [20].

In addition to alpha-beta pruning, chess engines have developed various search algorithm enhancements over the years [5]. Techniques such as iterative deepening and move ordering have been implemented to further refine the search process. Iterative deepening involves conducting multiple search iterations with increasing depth, allowing engines to explore promising branches more deeply while conserving computational resources. Move ordering aims to prioritize the evaluation of moves with a higher likelihood of leading to a successful outcome, thus accelerating the search process [9].

The complex interplay between these search algorithms and evaluation functions lies at the heart of chess engines' strategic decision-making. As we delve deeper into our analysis of Stockfish and lc0 on chess puzzles, we will consider how these engines leverage search algorithms to approach complex problems, offering insights into the evolving landscape of intelligent decision-making algorithms within the realm of chess and beyond.

2.2 Board Evaluation

One of the key pillars of chess engines' decision-making processes is board evaluation. The evaluation function in a chess engine is responsible for assigning a numerical value to a given board position, reflecting the engine's assessment of its desirability [17]. This function considers a myriad of factors, ranging from the material balance on the board to the positioning of the pieces, king safety, pawn structure, and various positional motifs.

The evaluation function acts as the engine's "judgment" of the current position, providing a quantitative measure of who stands better or worse. A positive evaluation typically favors one side (e.g., White), while a negative evaluation indicates an advantage for the opposing side (e.g., Black). This numerical evaluation guides the engine in determining its course of action in the game.

In the context of chess engines, board evaluation is a multifaceted endeavor that combines a deep understanding of chess strategy and tactics with computational efficiency. Engine developers craft intricate evaluation functions that capture the essence of chess principles, taking into account both material imbalances and positional subtleties.

Typically, chess engines utilize a multitude of evaluation terms to assess a position. These terms assign values to different aspects of the game. For example, capturing an opponent's queen might be assigned a substantial positive value, while losing one's pawn structure could yield a negative evaluation. Other factors such as piece mobility, king safety, and pawn structure intricacies are all considered within the evaluation process [17]. Understanding the dynamics of board evaluation is crucial for comprehending how chess engines like Stockfish and lc0 approach chess puzzles. These engines rely on their evaluation functions to provide the quantitative foundation for assessing puzzle positions.

Stockfish and lc0 employ distinct approaches to board evaluation. Stockfish relies on a traditional evaluation function that encompasses a vast array of chess knowledge, including material imbalances, pawn structures, piece mobility, and positional motifs. This approach enables Stockfish to assess positions with exceptional precision, emphasizing a comprehensive understanding of the game's dynamics. This equation is hand-written using human-based chess theory. In contrast, lc0 leverages neural networks and ma-

Figure 3: Stockfish determines the black pieces have an advantage worth 7.25 pawns

chine learning techniques to develop its evaluation function. By learning from self-play, lc0 constructs a dynamic and evolving assessment of board positions, striving to mimic human-like intuition. These divergent strategies exemplify the varying philosophies in chess engine development, with Stockfish prioritizing established chess knowledge and lc0 embracing a more adaptive, learning-based approach. The comparison of these methodologies in our analysis sheds light on the evolving landscape of chess engines and their distinctive pathways to board evaluation.

As we delve into our analysis of chess engines on chess puzzles, we will closely examine how these evaluation functions contribute to the engines' decision-making processes.

2.3 Forsyth-Edwards Notation (FEN)

Forsyth-Edwards Notation is a chess notation used to represent a chess position as a single string. The notation system was co-created by Scottish chess players David Forsyth and Steven Edwards, evolving over time to become a universal tool for all chess players, engines, and in literature [6]. Initially designed for transmission efficiency during telegraph games, FEN has become the common notation used to represent chess positions at all times.

The utility of FEN is in its ability to convey the complexity of any given chess position into a human-readable, and more importantly, machine-readable single string. The notation consists of a single string of characters, which captures the entire state of a chess board. This includes every piece placement, castling availability, en passant targets, and move counts. This representation facilitates the sharing and recording of games, making it an invaluable asset for chess literature, databases, and software.

Consider figure 4, the initial position of a standard chess game. In FEN, this is represented as "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1". Lets break down this string. Each forward slash represents a new row, starting with the top row at the leftmost, and the bottom row at the rightmost. Then, r, n, b, q, k, and p represent rooks, knights, bishops, queen(s), kings, and pawns respectively. The capital letters indicate the white pieces. The end of the string indicates whos turn it is, castling and en passant possiblities, and the move count. FEN is an indispensable tool for chess enthusiasts, and extremely helpful in chess software.

Figure 4: The starting position of a chess game

3 Methods and Preparation

3.1 Puzzles

The creation and discovery of chess puzzles that challenge even the most powerful chess engines represent a captivating pursuit within the realm of computer chess. These puzzles often involve counterintuitive moves or intricate scenarios that push the boundaries of traditional chess evaluation and search algorithms [8]. A chess puzzle is a chess position which is crafted, or discovered, such that there exists a specific set of moves which are determined to be the best. This could be through a forced checkmate, gaining a material advantage via winning a piece, or even gaining a strategic advantage.

Two chess puzzles which we are looking at are in positions which lead to a forced checkmate for the white pieces. Both puzzles involve different types of counter-intuitive moves. In chess, a counter-intuitive move is a move which does not seem beneficial on initial examination for humans and computers alike.

In the puzzle we will look at, several types of counter-intuitive moves come up. One rule of chess is that when a pawn reaches the opposite end of the board, it must become a different piece other than another king. As the strongest piece, it is almost always significant to promote the pawn to a queen. There are certain positions, though, where it is advantageous or even necessary for a player to promote to a piece other than a queen. This type of counter-intuitive move is called an under-promotion.

Another type of counter-intuitive move is called a "sacrifice". Sacrifices involve deliberately giving up a valuable piece, like a queen or rook, to gain a strategic advantage or launch a powerful attack. It's the chess version of making a calculated trade-off for a bigger goal. We see examples of this in both puzzles.

Lastly, a "waiting move" is often extremely difficult for humans to see because they defy the instinct to always be on the offensive. Instead, they involve making a subtle, seemingly passive move to see how the opponent reacts before launching a decisive attack. Sometimes, such as in these puzzles, a waiting move is necessary because the only move the opponent can make will weaken their position. These add layers of complexity to the game, requiring players—and even chess engines—to look beyond immediate gains and anticipate the nuanced interplay of pieces on the board.

In the first puzzle, shown in Figure 4, there is a forced checkmate in 9 moves for the white player. The solution to this puzzle is:

1. Nc5 b2 (White knight to c5, black responds with pawn to b2)
2. Nb3+ Kb1 (White knight to b3 with check, black king to b1)
3. Nd5 a1 = N (White knight to d5, black promotes the a pawn to a knight)

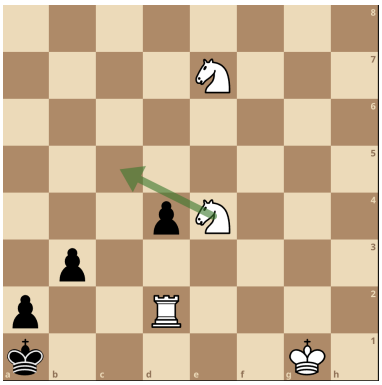


Figure 5: White has forced checkmate in 9 moves, starting with Nc5

4. Nxd4 Kc1 (White knight captures on d4, black king to c1)
5. Rc2+ Kxc2 (White rook to c2 with check, king captures on c2)
6. Nb3+ Kd1 (White knight to b3 with check, black king to d1)
7. Kf1 b1=N (White king to f1, black promotes the b pawn to a knight)
8. Kf2 Nd2 (White king to f2, black knight to d2)
9. Nc6# (White knight to c6, checkmate)

A visual representation of the solution to puzzle 1 can be found in the Appendix A. Although only 9 moves, this puzzle requires the white player to anticipate an under promotion, sacrifice a rook, and play a waiting move to force a checkmate onto the black player.

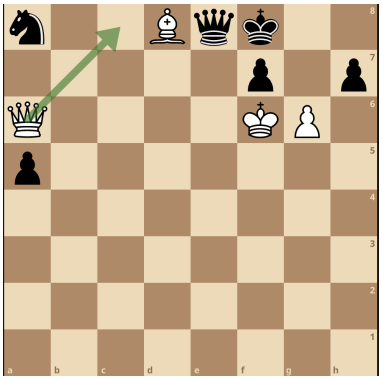


Figure 6: White has forced checkmate in 13 moves, starting with Qc8

In the second puzzle, shown in Figure 5, there is forced checkmate in 13 moves for the white player. The solution to this puzzle is:

1. Qc3 kg8 (White queen to g3, black king to g8)
2. Bc7 Qxc3 (White bishop to c7, black queen captures white queen on c3)
3. Gxf7+ kh8 (White g pawn takes on f7 with check, black king to h8)
4. Be5 Qc5 (White bishop to e5, black queen to c5)
5. Bb2 Nc7 (White bishop to b2, black knight to c7)

6. Ba1 a4 (White bishop to a1, black pawn to a4)
7. Bb2 a3 (White bishop to b2, black pawn to a3)
8. Ba1 a2 (White bishop to a1, black pawn to a2)
9. Bb2 a1=Q (White bishop to b2, black a pawn promotes to queen)
10. Bxa1 Nd5+ (White bishop captures on a1, black knight to d5 with check)
11. Ke6+ Nc3 (White king moves to e6 with a check, black knight to c3)
12. Bxc3+ Qxc3 (White bishop captures on c3 with check, black queen captures on c3)
13. f8=Q# (White f pawn promotes to a queen with checkmate)

A visual representation of the solution to puzzle 2 can be found in the Appendix B. This puzzle requires a queen sacrifice, followed by several waiting moves from the white player, before delivering checkmate. This means that in order for an engine to find this complete line, it would need to analyze the position after the queen sacrifice for 12 more moves. This is abnormal because typically engines will not consider a position so deeply when the move looks so poor.

3.2 Python Implementation

In addition to the deep analysis of both Stockfish and lc0 on these puzzles, we can further our understanding of our both engines analyze positions by giving them random chess positions. This will help us understand the extent to which both engines differ in their evaluation of boards.

In this project, several Python libraries were utilized to facilitate various tasks. Each library played a crucial role in the generation of random positions, as well as running both engines. Here is a brief overview of the libraries employed:

- **chess:** The chess library provided essential functionality for working with chess-related data structures and operations. It facilitated the representation of chess positions and moves, and helped check whether positions were valid when they were created [15].
- **random:** The random library was employed to introduce variability in the generation of random chess positions. It ensured the diversity and randomness of the positions which were generated, therefore contributing to a more comprehensive evaluation [16].
- **asyncio:** Asyncio is a library for asynchronous programming in Python. It was used to run the analysis asynchronously for both Stockfish and lc0, which was necessary for the two to run equally [14].
- **pandas:** Pandas is one of the most popular data manipulation and analysis library. It was used to organize and manage the data collected during the analysis, and outputting the results to csv files [**pandas**].
- **matplotlib:** Matplotlib is a comprehensive plotting library. It played a key role in visualizing the results and generating informative plots to aid the interpretation of data.

These libraries collectively played a vital role in the successful execution and analysis of the chess engine experiments. They were used to assist the python scripts written. Here is a quick description of the directory:

- **run_engines.py:** This file uses the chess library. It has two functions, run_stockfish(position, clock) and run_lc0($position$, $clock$). Both functions take a *valid* FEN position, and a double $clock$ value representing the number of seconds permitting either engine to run. Both return a string value of the best move which the respective engine suggests in the given position.

- **random_FEN_positions.py:** This script uses the libraries chess, and random. It has an entry function called start(), which then places a legal number of pieces per side randomly on a board, and returns a string that is the FEN value of that board. Note, just because there is a legal number of pieces on the board, does not mean the board is in a legal chess position. For example, a pawn could be placed on the first or last rank, which is not possible following the rules of chess.
- **compare_engines.py:** This is the entry python file, and uses libraries pandas, chess, asyncio and imports the two files mentioned. This file uses variables (int) num_of_trials and (double) $time$ to run num_of_trials number of random positions, and analyze each position for $time$ amount of time. It ensures a position is valid before having either engine analyze it, and records the percentage of positions which are valid. Then, using this information, generates a pandas dataframe, recording each FEN position, and the move recommended by each engine, and exports it to a csv file saved using the variable names.
- **read_csv.py:** This last file uses pandas and matplotlib to read several csv files and generate a bar graph of the results, and the percentage of different moves recommended by each engine.

For the purposes of this paper, the tests ran were:

1. 50 trials, 60.0 seconds
2. 100 trials, 15.0 seconds
3. 100 trials, 30.0 seconds
4. 100 trials, 45.0 seconds
5. 500 trials, 7.5 seconds
6. 1000 trials, 5.0 seconds

4 Results

4.1 Puzzle Results

Table 1: Stockfish Performances

Puzzle	Depth	Score	Time	Nodes	Move
Puzzle 1	101/38	M+19	5:00	504M	1. Nc5
Puzzle 2	61/29	+0.00	5:00	145M	1. Qa6+

Table 2: Leela Chess Zero Performances

Puzzle	Depth	Score	Time	Nodes	Move
Puzzle 1	17/36	+0.12	5:00	486K	1. Nc5
Puzzle 2	15/30	+0.07	5:00	711K	1. Qa6+

These tables show Stockfish and lc0's performance on each puzzle. The depth column represents the depth of the search, or how many moves deep each engine searched at least partially. The score column indicates the value which either engine assigns the position. Stockfishs' M+19 score means Stockfish found a guaranteed checkmate in 19 moves for the white pieces. The rest of the scores 0.00 mean the engines valued all the other positions as almost exactly equal. The time allotted for each of these engines to analyze these positions was 5:00 minutes. The nodes column shows the number of nodes which each engine analyze in each position. Lastly, the move column is the chess notation for which move the engine recommends be played in the given position.

4.2 Random positions results

```
Trial: 100 of 100
                                                  FEN Stockfish Move LC0 Move
0   8/1qBq3p/2q1rP2/4Np1p/P2K3Q/P7/2qP1B1Q/5k2 w -...           c7d6     d4e3
1              8/1k2pr2/8/1KQ5/8/8/1b6/8 w - - 0 1\n           c5d5     c5d5
2      7k/1Nn2n2/5K2/3q4/1R1n2B1/2pr4/8/8 w - - 0 1\n           b4b5     b4b5
3                   8/2N3K1/8/8/8/k7/8/8 w - - 0 1\n           g7g6     c7a8
4   8/1qbPr1Q1/1Bp2Pnb/K2p4/2p1nb1n/5k2/1P3pp1/8 w...           g7g6    d7d8q
..                                                ...            ...      ...
95              8/6bk/2P3N1/8/8/K2r4/8/8 w - - 0 1\n           a3b4     a3b4
96  8/Pb4pp/2Q2Q1Q/2q2br1/pK2PB1q/5kqn/r3q2r/8 w -...           b4c5     c6c5
97  8/1pB1Br1Q/k2b2p1/r1B2qb1/n1P1q1nP/1K1QB3/3bP1...           g2e4     g2e4
98  5K2/1nk2q1p/5R1q/6p1/2pb4/1r2b3/1q4bq/8 w - - ...           f8f7     f8f7
99  3K4/qP2Nr1b/pR1r2N1/3Nn3/kp2r1RB/b1q5/1bN1P1QB...           b6d6     b6d6

[100 rows x 3 columns]
Positions generated: 260, percent of valid positions: 38.46153846153847%
```

Figure 7: Pandas Dataframe Generated

This is the printed output from running one instance of the run_engines.py script, printing the pandas dataframe, and the percentage of chess board positions were legal board setups. The dataframe is structured such that the first column is the *legal* FEN position randomly generated, then Stockfish's recommended move, then lc0's recommended move. All this data was saved to a automatically stored to a csv file, tracking the number of trials and the time allowed as well as the results.

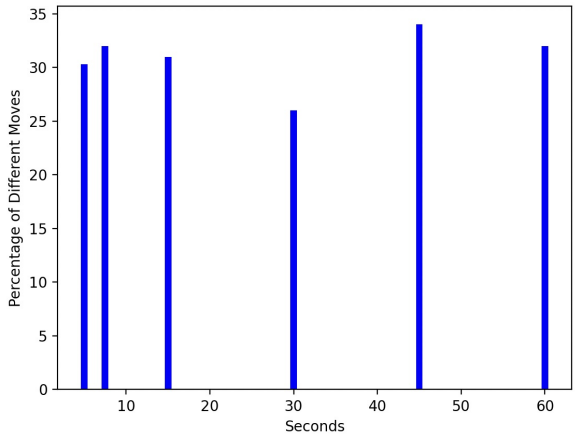


Figure 8: Percent of Different Moves

4.3 Discussion

Both Stockfish and lc0 grappled with the intricate challenges presented by these puzzles within a constrained five-minute timeframe. The struggle by these engines shows the inherent complexity of these puzzles, proving to be a formidable test even for sophisticated chess engines, and also can provide some insight into blind spots in these engines, since these puzzles can be solved in "only" 9, and 13 moves respectively.

Notably, it's remarkable that Stockfish, analyzed nearly 1000 times more nodes than lc0 during the allotted 5-minute span. This stark contrast aligns with Stockfish's strength, rooted in its unparalleled ability to navigate and search through an extensive array of positions. Its prowess lies in the sheer quantity of positions it can explore within a limited timeframe.

In puzzle 1, while neither engine could unearth the full solution within the initial five minutes, they demonstrated a capacity to grasp the essence of the puzzles. Interestingly, both Stockfish and lc0 successfully identified the opening move, but neither could find the rest of the solution after that. Once provided with the first two moves, both engines were able to find the rest of the solution to the puzzle.

These findings go beyond the mere evaluation of whether the engines successfully unraveled the complete puzzle within the imposed time constraint. In Puzzle 1, Stockfish managed to discover a forced checkmate in 19 moves, after searching through a staggering 504 million nodes. lc0 failed to do so, and still evaluated the board as even.

The implications of Stockfish achieving a superior evaluation in Puzzle 1 compared to lc0 are multifaceted. Despite both engines failing to identify the 9-move checkmate pattern within the time limit, Stockfish's ability to find a checkmate in 19 moves speaks to the diverse nature of chess analysis. This finding suggests that Stockfish, with its traditional approach rooted in brute-force search and well-established evaluation functions, possesses a unique strategic insight. This outcome underscores the intricate balance between computational efficiency and strategic understanding in chess engines. While lc0, with its emphasis on neural networks and adaptive learning, may excel in certain domains, Puzzle 1 reveals that Stockfish's methodical and exhaustive exploration of positions can lead to a more accurate evaluation.

In Puzzle 2, a different challenge emerged. Both Stockfish and lc0 initially struggled to identify the optimal opening move, perceiving the position as equal for both black and white. The initial assessment suggested a potential draw after the move Qa6+. It was not until after being given the first two moves that both engines were able to find the complete line. These nuanced findings underscore the intricate nature of chess analysis, where engines navigate a vast and complex search space with the efficiency dictated by time constraints. Our results contribute significantly to the ongoing exploration of chess engines' strengths and limitations when confronted with the complexities of intricate and counterintuitive puzzles. The interplay of strategic insight, adaptability, and computational efficiency showcased by Stockfish and lc0 in these scenarios opens the door to further understanding the dynamics of chess engine decision-making.

In addition to analyzing the puzzles in depth, we tested the consistency of both lc0 and Stockfish's results on these puzzles, and found that after being ran 100 times with 5 minute time limits, not only did

neither engine find the full solutions to either puzzle, but neither engine offered different moves than the ones in their original finding. The variation of their board score's was also minimal.

In the investigation of randomly generated chess positions, a notable revelation emerged as only 38% of the positions constructed were deemed valid. This output can be seen as printed to the console in Figure 7. This highlights the intricacies involved in crafting legally playable chess scenarios, underlining the complexity of generating positions that adhere to the game's rules. This is including restrictions on the number of pieces each color is allowed on the board- that is, no more than one king for each, eight pawns, nine queens, or ten of the other minor pieces (considering those positions are technically achievable).

Surprisingly, the time constraints imposed on Stockfish and lc0 did not substantially influence the frequency of varied moves between the two engines on valid chess board positions, as seen by the matplotlib bar graph generated and seen in Figure 8. Regardless of the time limit, Stockfish and lc0 consistently offered different moves in a substantial portion of cases, ranging from 30% to 35%. This result is remarkably higher than expected. Many instances of disparate moves occurred in positions characterized by highly imbalanced material, intricate complexities, or endgame scenarios where a checkmate was guaranteed. The engines' diverse approaches to these challenging situations suggest a nuanced decision-making process influenced by the specific characteristics of each position.

These findings raise intriguing questions about the adaptability and decision-making processes of both Stockfish and lc0 in the face of diverse and intricate chess scenarios. This also emphasizes the complexity of any chess position, consider that two of the strongest chess engines in the world still provide different moves in such a high percentage of positions. The higher-than-expected frequency of varied moves prompts further exploration into the factors influencing the engines' evaluations.

5 Conclusion

The examination of Stockfish and Leela Chess Zero's performance on complex chess puzzles helps us draw conclusions to the complicated nature of their decision-making processes. The contrast between Stockfish's exhaustive search and lc0's adaptive learning approach reveals the diverse strategies employed by these engines. Puzzle-specific nuances, such as the underpromotion in Puzzle 1 and the waiting moves in Puzzle 2, showcase the engines' ability to navigate counterintuitive scenarios. The significant difference in the number of nodes explored by Stockfish and lc0 emphasizes their distinct methodologies and raises questions about the trade-offs between computational efficiency and strategic insight.

Notably, Puzzle 1's checkmate in 19 moves, successfully identified by Stockfish, highlights the engine's unique strategic insight even within the constraints of a time-limited puzzle. The divergent approaches of Stockfish and lc0 in Puzzle 2, where both struggled with the initial move, underscore the challenges posed by nuanced positions. These findings contribute to our understanding of chess engines' decision-making, emphasizing the balance between computational efficiency and strategic understanding, and highlighting some blind spots which both types of Artificial Intelligence contain in complex puzzles.

Moving forward, further research could expand the scope of puzzle analysis, exploring new challenges or generating a broader set of positions. The potential to delve into the dynamic branches that engines focus on during puzzle-solving offers a promising avenue for understanding their adaptive strategies.

In addition, there are always more puzzles which can be analyzed, which already exist or are yet to be created. Expanding the scope of puzzle examination can provide a more comprehensive understanding of how engines perform across various puzzle types. Inspecting in depth how these engines perform in new puzzles can further support claims being made across the puzzles which have been studied so far. This continued research would be conducted in a similar fashion to the research done in this paper. A limitation in this aspect of future research could be a failure to find existing puzzles which the engines compute differently. Another limitation may be failure to find significant results.

Chess puzzles continue to serve as a valuable arena for unraveling the complexities of artificial intelligence and decision-making algorithms, with exciting implications extending beyond the realm of chess.

References

[1] In: *The Salina Journal Salina, Kan* (1996).

[2] URL: https://www.chessprogramming.org/Alex_Bernstein.

[3] Mathieu Acher and François Esnault. *Large-scale Analysis of Chess Games with Chess Engines: A Preliminary Report*. 2016. arXiv: 1607.04186 [cs.AI].

[4] Pieter Bijl and AP Tiet. "Exploring modern chess engine architectures". In: *Victoria University, Melbourne* (2021).

[5] Ami Hauptman and Moshe Sipper. "Evolution of an efficient search algorithm for the mate-in-N problem in chess". In: *European Conference on Genetic Programming*. Springer. 2007, pp. 78–89.

[6] Azlan Iqbal. "An Algorithm for Automatically Updating a Forsyth-Edwards Notation String Without an Array Board Representation". In: *2020 8th International Conference on Information Technology and Multimedia (ICIMU)*. 2020, pp. 271–276. DOI: 10.1109/ICIMU49871.2020.9243487.

[7] Donald E. Knuth and Ronald W. Moore. "An analysis of alpha-beta pruning". In: *Artificial Intelligence* 6.4 (1975), pp. 293–326. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(75)90019-3. URL: https://www.sciencedirect.com/science/article/pii/0004370275900193.

[8] Shiva Maharaj, Nick Polson, and Alex Turk. "Chess AI: Competing Paradigms for Machine Intelligence". In: *Entropy* 24.4 (2022). ISSN: 1099-4300. DOI: 10.3390/e24040550. URL: https://www.mdpi.com/1099-4300/24/4/550.

[9] Thomas McGrath et al. "Acquisition of chess knowledge in AlphaZero". In: *Proceedings of the National Academy of Sciences* 119.47 (2022), e2206625119. DOI: 10.1073/pnas.2206625119. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.2206625119. URL: https://www.pnas.org/doi/abs/10.1073/pnas.2206625119.

[10] Ethan Modi and Kymberly Acuna. "The Effects of Computer and AI Engines on Competitive Chess". In: *Journal of Student Research* 12.3 (Aug. 2023). DOI: 10.47611/jsrhs.v12i3.4993. URL: https://www.jsr.org/hs/index.php/path/article/view/4993.

[11] Jan Mycielski. "Games with perfect information". In: *Handbook of game theory with economic applications* 1 (1992), pp. 41–70.

[12] Pascutto, Gian-Carlo and Linscott, Gary. *Leela Chess Zero*. Version 0.21.0. Mar. 8, 2019. URL: http://lczero.org/.

[13] F.J. Pitman. *A. Turing. Digital Computers Applied to Games*. Pitman, 1953. Chap. 25, pp. 286–310.

[14] *Python asyncio library*. URL: https://docs.python.org/3/library/asyncio.html (visited on 03/19/2024).

[15] *Python chess package*. URL: https://pypi.org/project/chess/ (visited on 03/19/2024).

[16] *Python random package*. URL: https://docs.python.org/3/library/random.html (visited on 03/19/2024).

[17] Mehdi Samadi, Zohreh Azimifar, and Mansour Zolghadri Jahromi. "Learning: An Effective Approach in Endgame Chess Board Evaluation". In: *Sixth International Conference on Machine Learning and Applications (ICMLA 2007)*. 2007, pp. 464–469. DOI: 10.1109/ICMLA.2007.48.

[18] Claude E. Shannon. "XXII. Programming a computer for playing chess". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314 (1950), pp. 256–275. DOI: 10.1080/14786445008521796. eprint: https://doi.org/10.1080/14786445008521796. URL: https://doi.org/10.1080/14786445008521796.

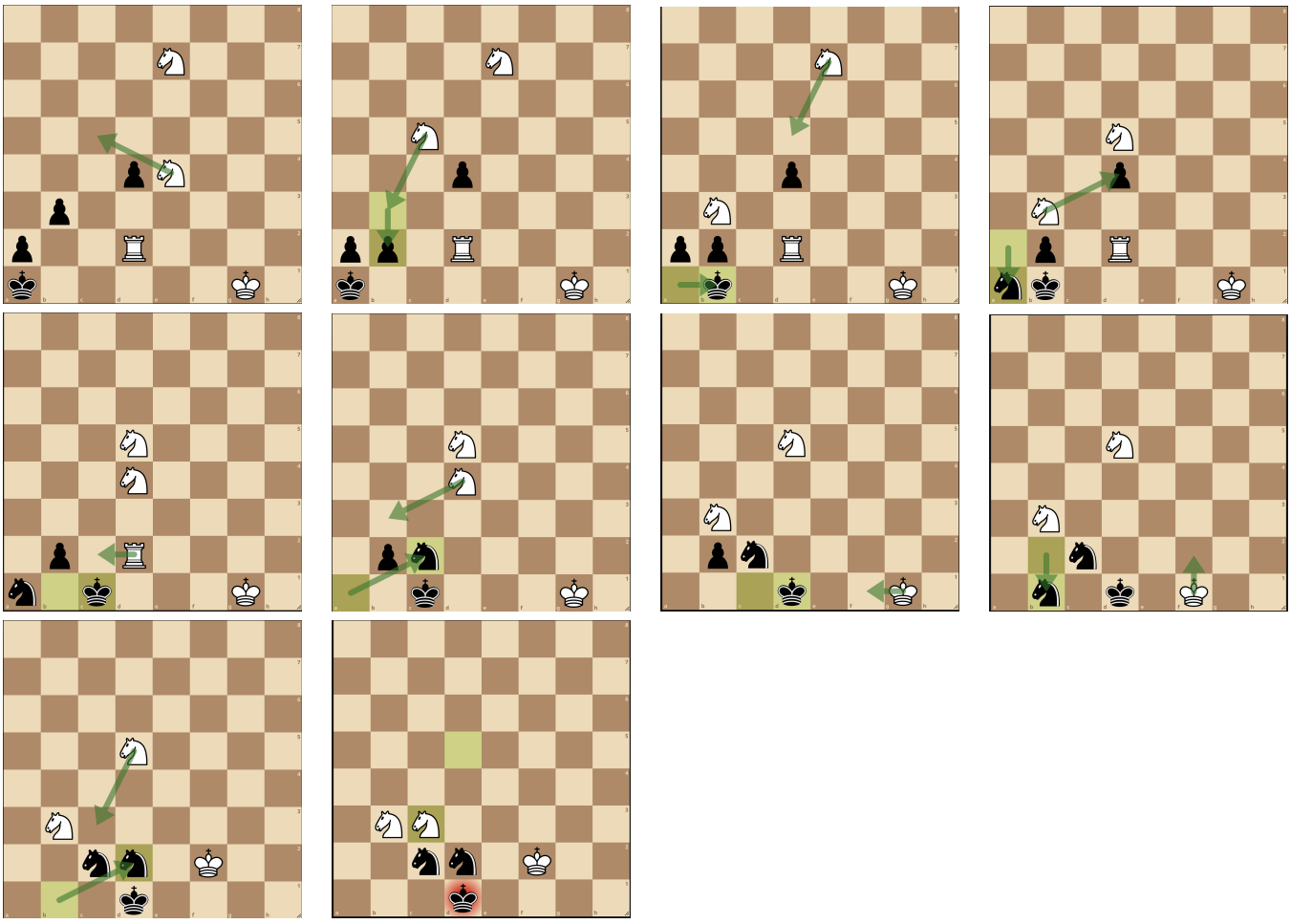
[19] Stefan Steinerberger. "On the Number of Positions in Chess without Promotion". In: *Int. J. Game Theory* 44.3 (Aug. 2015), pp. 761–767. ISSN: 0020-7276. DOI: 10.1007/s00182-014-0453-7. URL: https://doi.org/10.1007/s00182-014-0453-7.

[20] George C. Stockman. "A minimax algorithm better than alpha-beta?" In: *Artificial Intelligence* 12.2 (1979), pp. 179–196.

[21] The Stockfish developers (see AUTHORS file). *Stockfish*. URL: https://github.com/official-stockfish/Stockfish.

[22] Eduardo Vázquez-Fernández, Carlos A. Coello Coello, and Feliú D. Sagols Troncoso. "An evolutionary algorithm coupled with the Hooke-Jeeves algorithm for tuning a chess evaluation function". In: *2012 IEEE Congress on Evolutionary Computation*. 2012, pp. 1–8. DOI: 10.1109/CEC.2012.6252977.

A Puzzle 1 Solution



B Puzzle 2 Solution