The Effect of Endgame Tablebases on Modern Chess Engines

By Christopher Peterson

CPE 462: Senior Project II

Dr. Helen Yu

June 11, 2018

Abstract

Modern chess engines have the ability to augment their evaluation by using massive tables containing billions of positions and their memorized solutions. This report examines the importance of these tables to better understand the circumstances under which they should be used.

The analysis conducted in this paper empirically examines differences in size and speed of memorized positions and their impacts on engine strength. Using this technique, situations where memorized tables improve play (and situations where they do not) are discovered.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Memorized Positions in Chess

Memorizing positions is a common technique used in modern chess engine development. By saving a board state and a value or optimal "solution" move to disk, a properly-coded engine can ensure that the position is correctly handled in all future cases.

Not all engines employ memorization-based techniques, but the authors of those that do cite improvements to playing strength, speed, and variety.

Format	5 piece	6 piece	7 piece
Nalimov	7.1 GiB	1.2 TiB	-
Lomonsov	-	-	140 TiB
Syzygy	939 MiB	150.2 GiB	-

Table 1: Sizes of commonly used endgame tablebases [1]

On the other hand, memorized tables can be extremely large and introduce access time delays. The recently released 7-piece Lomonsov tablebases, for instance, are a cumbersome 140 Terabytes [2]. It is believed by some that chess engines are sophisticated enough that the performance gained from memorizing vast amounts is no longer enough to justify their size.

Because of the large state-space complexity of chess, it is typically only feasible to memorize positions in the first several moves or after many pieces have been removed from the board. Although mid-game moves can be memorized, it is difficult to come up with optimal solutions and there is little guarantee a position will ever be seen more than once.

1.2 Problem Statement

Despite their relatively common inclusion in modern engines, the pros and cons to the use of large amounts of memorized information are not well understood. How can the use of memorization in chess be optimized as storage becomes faster and cheaper, as engines improve, and as networked cloud storage becomes more widely used? What are the different strategies that a programmer should employ when developing on a supercomputer versus a mobile phone? Are memorized tables even necessary at all for modern engines, which do an excellent job of quickly finding good moves?

This report seeks to examine and potentially resolve these issues by examining some of the following questions:

- What performance gains/losses can be expected when using memorized positions? Where do they come from?
- Will memorized positions give a larger performance boost to stronger or weaker engines?
- What effect does the amount of time an engine is given have on the usefulness of memorized positions?
- Are memorized positions required to compete at the highest level of chess engine performance?
- What effect does tablebase completeness have on performance? Are entire tablebases required for an engine to gain the benefits of tablebases?
- How important is the access speed of the medium that the memorized information is stored on? Will slower mediums negatively impact performance, and if so, by how much?

A precise answer and explanation for these questions regarding the importance (or lack thereof) of memorized positions would benefit engine creators, who must balance the strength and size of their programs, as well as engine users, who must make informed decisions on which engines/tablebases to use and/or buy.

Interestingly, better knowledge about endgame memorization could also benefit human players. Although humans are not able to memorize positions to the same accuracy or extent as computers, knowledge about the most common and/or tricky endgames could still be helpful. Some commonly occuring endgames could be worth memorizing for a human player, while others might be workable from other knowledge and therefore not require memorization.

1.3 Topic Overview

The following topics will be addressed in this design report:

1.3.1 Literature Review

The literature review section of this report covers a variety of topics and publications related to memorized positions in chess. Some of these papers and articles are briefly described below:

- Programming a Computer for Playing Chess - Described the statistical complexity of "solving" 32-piece chess
- Retrograde analysis of Certain Endgames - Outlined (and later used) an algorithm to completely "solve" certain endgames
- Space-Efficient Indexing of Chess Endgame Tables - Contained optimizations which drastically reduced the amount of space required to store memorized information about endgame positions
- Can Endgame Tablebases Weaken Play? - Described the five cases in which memorized endgame information could hurt an engine's performance
- Mechanisms for Comparing Chess Programs - Outlined ways in which chess engines could be tested and compared

1.3.2 Background

The background section of this report covers topics that are important to understanding its content and analysis. Broadly speaking, these topics are:

- Complexity in games and chess, and how it puts constraints on what can be memorized
- Evaluating the playing strength of chess engines
- Comparing chess engines using metrics like Elo rating and likelihood of superiority
- The common protocols used by modern chess engines

1.3.3 Analysis

The analysis section of this report will empirically test a variety of chess engines in order to better understand how they respond to changes in memorized information.

The following three tests will be performed:

- Analyze tablebaes usefulness at different time controls in order to determine the effect of time control on tablebase importance
- Test smaller, incomplete tablebase performance against complete tablebase performance to determine a relationship between tablebase size and tablebase performance
- Examine performance loss when storing tablebases on a slower storage medium to determine the relationship between tablebase access time and engine performance

By carefully conducting these experiments and analyzing their effects on various engines, conclusions can be drawn regarding when and where tables of memorized information is most (or least) useful. This will allow engine developers and users to better optimize the performance of their chess engines.

2 Literature Review

2.1 Comparing Chess Players

2.1.1 Selection of the Best Treatment in a Paired-Comparison Experiment

Published in 1963, *Selection of the Best Treatment in a Paired-Comparison Experiment* by Trawinski and David laid out much of the mathematical and statistical framework that would later be applied to chess players.

This paper outlined the comparison of two "treatments" which are presented in pairwise comparisons to some judge. Based on the judge's rulings, the "treatments" are scored and the best treatment is defined as the one with the highest score. This paper omitted expressions of no preference, which would be necessary in its adaptation to chess.

This paper and several like it became the foundation for the Elo rating system, which was developed around 1960 by Arpad Elo to simply and accurately compare an arbitrary number of chess players. The Elo rating system has been the most widely used method to rate chess players since. This system and its mathematical foundations are described in much further detail in Section 3.4.2.

2.2 Complexity in Chess

2.2.1 Programming a Computer for Playing Chess

In 1949, Claude E. Shannon published his paper *Programming a Computer for Playing Chess* [17]. This paper is commonly regarded as the first in-depth discussion of computer chess and contains much of the groundwork for future advancements in the field.

This paper estimated the state-space complexity of chess (the number of unique boards states that can be achieved in real games) to be approximately 10^{43}, using the following formula:

$$\frac{64!}{32!8!^2 2!^6} \approx 10^{43} \tag{1}$$

This calculation provided an improvement on previous ones, but did not account for captures or promotions and included many redundant or impossible combinations of material.

More information about the state-space and game-tree complexity of chess can be found in Section 3.1.

2.2.2 Searching for Solutions in Games and Artificial Intelligence

Victor Allis's 1994 thesis *Searching for Solutions in Games and Artificial Intelligence* improved on previous complexity estimates and expanded them to include more games [9]. This thesis attempted to better understand non-trivial zero-sum games and their likelihood of being solved by humans in the near or far future. This paper sought to examine current AI topics in these games and potential future issues that would need to be overcome.

In his thesis, Allis estimated the state-space complexity of chess to be bounded below 5×10^{52}, which included captures and promotions and therefore improved on Shannon's calculation. Based on this upper bound, he estimated the actual state-space complexity to be on the order of 10^{50}.

More information about the state-space and game-tree complexity of chess can be found in Section 3.1.

2.3 Endgame Tablebases

2.3.1 Retrograde analysis of Certain Endgames

In 1986, Ken Thompson published *Retrograde Analysis of Certain Endgames*, one of the first papers to offer a strategy for memorizing massive amounts of optimal endgame board states [18]. This paper pioneered the technique of exhaustive retrograde analysis that is still used today. Thompson would later publish further papers covering various 5 and 6 piece endgame tablebases as they were created in 1990, 1991, and 1997.

The retrograde analysis algorithm proposed by Thompson works roughly as follows to generate distance-to-mate metrics:

- Begin with all board states that represent checkmate. Consider these positions as “mate in 0”.
- All positions that can be reached from “mate in 0” positions by the current side to move are marked as ”mate in 1”.
- All positions that only lead to “mate in 1” positions are marked as “mate in 2” positions.
- This continues until all positions are covered and determined as an unavoidable “mate in X”or as a draw.

This algorithm presents a number of advantages, including its simplicity, ease of parallelizing, and guarantee of optimality. This algorithm and algorithms like it are commonly used today to calculate tablebases.

On the contrary, this algorithm and algorithms like it require a vast amount of RAM and CPU time to calculate larger tablebases. Creation of 6-piece Syzygy tablebases requires an estimated 16 GB of RAM and creation of 7-piece tables requires roughly 1 TB.

2.3.2 Space-Efficient Indexing of Chess Endgame Tables

The 2000 paper *Space-Efficient Indexing of Chess Endgame Tables*, written by Nalimov, Haworth, and Heinz, examined and improved on tablebase compression techniques [14]. This paper suggested and quantified methods for improving tablebase indexing, removing more redundant or impossible positions from tables. As a result of significantly reducing tablebase size, these methods were later adopted and improved on in future tablebase design.

Prior to this paper's publication, many tablebases were indexed using the following simple, but naive scheme (or schemes like it):

```
index = position\_white\_King + position\_black\_King * 64
for each piece {
   index = index * 64 + position
}
```

This paper suggested removing the many redundant or impossible positions, including:

- Symmetrical positions
- Positions with adjacent kings
- Positions with pawns on ranks 1 or 8
- Full indexing up to 64 is not needed after the first piece: further pieces only have 63 possible locations.

Because of their usefulness in compacting endgame tablebases, these improvements and improvements like them have been integrated into most modern endgame tablebase formats, including Nalimov's own tablebases (discussed further in Section 3.2.3.2).

2.3.3 Reference Fallible Endgame Play

Published in 2002, G.M C. Haworth's paper, *Reference Fallible Endgame Play* attempted to address a discovered issue of endgame tablebase use: among positions that were theoretically drawn or lost, a "perfect" player according to endgame tablebases would not attempt to win [12]. Although this result is expected against a perfect opponent, it felt unrealistic when playing against a fallible player. This paper presents a solution to the issue, which attempts to make optimal moves (that do not make the position theoretically worse) while also trying win or tie otherwise lost positions.

These tablebase probing algorithms that attempt to punish non-optimal savvy players are now used in many chess engines, often known as "contempt" or "swindling." With these features activated, engines will use their analysis in tablebase positions in order to attempt to force wins or draws.

Although these features make engines *feel* more human-like, they do not improve playing strength against optimal opponents and can therefore be considered unnecessary or even superfluous at the highest levels of play.

2.3.4 Can Endgame Tablebases Weaken Play?

In 2003, Horizon Chess published an article titled *Can use of endgame tablebases weaken play?* [4]. This article addressed experimental results that occasionally showed a slowdown in engine speed when using tablebases. This article addressed and laid out several cases in which endgame tablebases could *weaken* positional strength:

- Incomplete tablebase issue
- Tablebases don't account for castling
- Tablebases don't account for the 50-move rule
- "Knowing too much" problem
- Disk access slows down a search

By outlining these ways in which endgame tablebase use could potentially weaken play, engine users and programmers were able to better understand where performance gains/losses originated from. Some of these potential sources of errors will be addressed in this report.

2.4 Creating and Testing Engines

2.4.1 Programming a Computer for Playing Chess

Claude E. Shannon's 1949 paper, *Programming a Computer for Playing Chess*, is commonly credited as the first in-depth analysis of computer chess [17]. This paper considers chess and how computers can . For its time, this paper was extremely detailed and accurate, and as a result would heavily influence future chess engines and publications.

This paper proposes a separation of chess engines into two tasks, which very closely resemble engine tasks today:

- Evaluation - A computer chess program would be able to evaluate a board state. Based on material and piece position, the computer could estimate the utility for a given board. Shannon proposes some simple evaluation functions, including material, pawn strength, rook mobility, and king safety.
- Searching - Given approximate evaluations of positions, a computer can search between

This paper also predicted that future engines would follow one of two types: Type A, engines that used a brute-force approach, and Type B, those that used a best-first approach. Today, most top engines are iterative deepening "Type A" versions, as a best-first search "Type B" engines are too prone to missing crucial moves.

Although some its estimations and predictions are inaccurate, Shannon's paper carefully analyzed many aspects of computer chess and laid much of the groundwork for research that would come in the future.

2.4.2 Mechanisms for Comparing Chess Programs

In 1973, Tony Marsland and Paul Rushton published *Mechanisms for Comparing Chess Programs* [13]. This article theorized different ways in which engines could be tested and their playing strength determined, before engines could be reliably connected together. Although advances in technology and processing power have largely obsoleted some techniques described in this paper, others remain relevant today.

This paper suggested the following methods of testing and comparing chess engines:

- Measuring resource utilization - By keeping track of an engine's memory and CPU usage, it is possible to get a better idea of the engine's performance in comparison to other engines.
- Playing engine games from a starting position - An engine's response to a variety of starting positions can be tested to get a wider idea of its chess-playing ability.
- Games between two engines - Although not feasible at the time, Marsland and Rushton propose this as a method of quickly comparing two engines.
- Comparison to master-level games - By using the same moves and comparing an engine's response to that of a recorded master-level player, the strength of an engine can be deterministically evaluated using only one computer.

Despite being published well before large-scale automatic chess engine testing or games between multiple automated engines was viable, some of the methods presented in this paper are still in use today (Section 3.3 goes into more detail on modern comparison strategies).

However, the rise of computers as super-human chess players has largely obsoleted comparisons to master-level games, and massive advances in computing power means that limiting games between engines to partial ones is no longer necessary.

2.4.3 Statistical Minefields with Version Testing

Jeff Rollason's 2007 article *Statistical Minefields with Version Testing* discusses some of the primary issues with modern engine development and how to improve on them [15]. The article describes the issues with testing for small improvements and the large sample size required to make sure a change is a positive one. Because modern engine development typically revolves around large amount of these additive small improvements (and small hindrances to be avoided), this topic is crucial for engine development.

The article then suggests the following techniques to mitigate the aforementioned issues:

- Using statistical analysis of results to determine the impact and significance of changes
- Maximizing sample size of test games to increase sample size

- Using continual round-robin testing between different archived engine versions to ensure changes are positive
- Using a consistent testing plan and schedule to prevent testing issues from causing unexpected results

The techniques and potential pitfalls described in this article are now widely understood adopted. Modern engine development typically involves round-robin version testing and extensive parameter tuning and testing. Some of the techniques described in this article will be used in this report's analysis.

3 Background

3.1 Complexity in Chess

3.1.1 State-space Complexity of Chess

State-space complexity is measured by the total number of legal positions that are possible to reach in a particular game. In chess, this value has traditionally been best approximated by the number of ways to legally arrange pieces on the board, which provides an estimate to the number of positions that are actually possible to reach in real games.

The state-space complexity of chess was originally approximated by Claude Shannon in his landmark 1949 paper, *Programming a Computer for Playing Chess.* Shannon used the following equation, which included some illegal positions and omitted legal ones which followed promotions and captures, for his estimate [17]:

$$\frac{64!}{32!8!^2 2!^6} \approx 10^{43} \tag{2}$$

This estimate was further refined by Victor Allis in his 1994 paper *Searching for Solutions in Games and Artificial Intelligence.* In this paper, Allis calculated an upper bound of $5x10^{52}$ possible unique positions for the state-space complexity of chess, with captures and promotions included [9].

3.1.2 Game-tree Complexity of Chess

Game-tree complexity is a measurement of the total number of unique games that can be played. Due to the 50-move rule, every chess game must eventually end and thus this complexity is bound to a finite number.

In the same 1949 paper, Claude Shannon provided a lower bound for the game-tree complexity of chess, which became known as Shannon's Number. His estimate of 10^{120} was based on an average game length of 40 moves per side and an average of 10^3 possible outcomes per pair of turns [17].

The number of possible chess games can be greatly reduced by limiting games to "sensible" ones. By assuming 3-4 reasonable moves per turn, the estimate is lowered to around 10^{45}.

3.1.3 Solving Chess

A "strongly solved game" is one where an optimal move is known for every possible turn. Strongly solving chess would thus require an optimal move to be given for every available position, as described in Section 3.1.1.

If chess was ever strongly solved, the generation of memorized 32-piece tablebases would enable engines to play the game perfectly. This has been described as "playing chess with god", a perfect opponent. Accomplishing this is considered fairly unlikely based on our current understanding of physics, due to two major issues that stem from chess's state-space complexity:

- Calculating solutions to $5x10^{52}$ different positions would take an exceedingly long time. While theoretically conceivable given enough time, this would require time on the order of an estimated 10^{90} years with current technology [17].
- Should they be calculated somehow, storing information about that many positions in a database is not expected to ever be feasible.

3.2 Memorized Positions in Chess Engines

3.2.1 Why is Memorization Used in Chess Engines?

Memorizing positions is a common technique used in modern chess engine development. By saving a board state and a value or optimal "solution" move to disk, a properly-coded engine can ensure that the position is correctly handled in all future cases.

Not all engines employ memorization-based techniques, but the authors of those that do cite the method's many improvements to playing strength, including:

- Faster Analysis of Memorized Positions - Looking up a known solution is often faster than calculating one on the fly, especially for complex board states. Theoretically, this is even more true for low-end computers or engines, where deeply calculating optimal moves is more difficult.
- More Precise - Memorized solutions to board states can be calculated using extremely precise and specialized metrics which often aren't available or suitable to an engine normally, such as exhaustive retrograde analysis. A weaker engine can, for instance, be made to copy a known strong engine's opening or endgame play-style to improve its own play.

- Encourages Diversity - Having multiple memorized solutions to randomly choose from can make an engine unpredictable and more interesting to play against. Without a random selection of moves, an engine can become deterministic or repetitive in similar-looking games.
- Faster Analysis of Near-Memorized Positions - Because calculating the utility of an unknown position in chess requires analysis of potential resultant positions, memorizing information about positions can also improve play in near-memorized ones. If an engine can preemptively discard a possible line as "losing" using a memorized endgame analysis, it's more likely to find a stronger move.

Despite these cited benefits of memorized solutions to common board states, issues can also arise which can negatively impact playing strength, such as:

- Imperfect Moves - Aside from exhaustive retrograde analysis, there's no guarantee that a particular memorized move or strategy is truly optimal. As a result, any non-perfect memorized move can potentially introduce error into an engine's analysis.
- Incomplete Memorization - An engine following a powerful memorized line or technique might get itself into a position it cannot play well after the memorized line ends. Even though a memorized line is optimal for a skilled engine, a weaker one that attempts to follow it can be lead astray.
- Assumes Perfect Play - Typically, memorized lines assume perfect reactions by an opponent. As a result, unexpected responses are less likely to be memorized. Additionally, memorized lines do not account for an opponent's potential mistakes and might lead an engine to preemptively call or play for a loss or draw.
- Drive Access Time - Accessing a large database of memorized moves introduces file overhead. In some cases, this overhead can cause an engine to slow down and play worse moves.

Further complications with the general memorization of positions, as well as specific cases, are discussed in the following sections.

3.2.2 Current Limits of Memorization in Chess

As discussed in Section 3.1, chess games rapidly increase in state-space complexity after the first several moves. Even when limited to 'good' moves, the amount of different board positions quickly becomes intractable and impossible to thoroughly and completely analyze with current technology. Despite memorization of these complicated middlegame positions appearing useful,

The rapidly expanding state-space complexity of chess also places limits on what information can be stored on modern storage drives. Even with state-of-the-art compression techniques, maintaining optimal moves for boards of 7 pieces requires 140,000 gigabytes of drive space [2]. This issue is further compounded by the fact that, as discussed in Section 3.2.1 and 3.4.1.3, drive access time plays a factor in engine performance–limiting maximum performance memorized information to faster, more expensive drives.

As a result of these issues and constraints, memorization in chess is typically only feasible during the first several moves along commonly played lines, and after many pieces are removed and the number of possible unique positions decreases. The following section, 3.2.3, discusses and further analyzes these places where memorization is viable for chess engines using current technology.

3.2.3 Places Where Memorization is Currently Feasible

3.2.3.1 Opening Book

An opening book contains a memorized list of commonly occurring opening positions or strategies which an engine can use and access. Additionally, opening books can be swapped between engines to encourage or practice certain playstyles.

This memorized information found in opening books is typically either handcrafted or created by an analysis of many highly rated games games. These can either come from professional human games, or games played between top engines.

Opening books can improve engine speed when dealing with common positions in the opening, but can also cause slowdowns due to file I/O and are not guaranteed to produce optimal moves. Despite this, their importance will be examined in the analysis of this report, which will be primarily focusing on endgame tablebases.

3.2.3.2 Endgame Tablebase

An endgame tablebase is an exhaustive database of all possible board states below a certain number of pieces. These tables are calculated using retrograde analysis and contain an expected game result and a distance to achieve that result from a given position. Because they are created using exhaustive analysis, endgame tablebases are guaranteed to be optimal, assuming perfect play from both players.

Endgame tablebases typically keep track of positions using two measurements, which allow an engine consulting them to identify which moves improve the position and which degrade it:

- Depth to Mate (DTM) - The number of moves until checkmate is reached.
- Depth to Zeroing (DTZ) - The number of moves until a pawn move or a capture, upon which the 50-move rule will be reset.
- Win/Loss/Draw (WDL) - Whether a position is won, lost, or drawn, assuming optimal play from both players.

Because they cover every possible board state below a certain number of pieces, endgame tablebases become large quickly. Table 2 shows the sizes of some of the ones most commonly used in modern engines.

Format	Metric	5 piece	6 piece	7 piece
Nalimov	DTM	7.1 GiB	1.2 TiB	-
Lomonsov	DTM	-	-	140 TiB
Syzygy	WDL + DTZ	939 MiB	150.2 GiB	-

Table 2: Sizes and metrics of commonly used endgame tablebases [1]

More information regarding the creation and compaction of endgame tablebases can be found in the Literature Review, Section 2.

Endgame tablebases are commonly used in engines and have potential to improve a chess engine's play by saving time and providing optimal moves. However, they are often very large and the file I/O associated with them can slow down an engine's play as well. As a result, their importance will be carefully examined in the analysis of this report.

3.2.3.3 Endgame Bitbases

Endgame bitbases are compacted versions of endgame tablebases that trade a much smaller size for less information about the positions they describe. An endgame bitbase will usually encode the result of a board position in a single bit as *won* or *not won* or in two bits as *won*, *drawn*, *lost*, or *invalid*.

As a result of their smaller size, endgame bitbases can more easily fit into faster memory, increasing search speed while still providing some of the benefits of exhaustive endgame tablebases. Having a fast reference to whether a position is won, lost, or drawn can allow an engine to navigate endgames more carefully and prune off losing moves quickly. However, endgame bitbase's lack of direction or progress in won positions can lead to mistakes or wasted moves.

Because fewer engines support them in favor of the more widely used endgame tablebases, the importance of endgame bitbases will not be thoroughly examined in the analysis of this report.

3.3 Evaluating Chess Engines

Beyond what is possible to analyze through exhaustive retrograde analysis, playing chess well is not yet perfectly understood. The vast majority of moves, positions, and players cannot be evaluated exactly, but are understood and approximated in terms of how likely they are to win, lose, or tie a given game. This can lead to unresolvable disagreements in positions or moves, where two equally rated engines or players will come up with different evaluations for the same issue.

However, this lack of a universal, perfectly accurate method of evaluating the ability of an agent to play chess does not disqualify chess from being analyzed in a logical way. This subsection will go over various quantifiable ways in which chess engines can be more precisely understood and ultimately compared.

3.3.1 What Makes a Chess Engine Good?

Before being able to answer the question of "How strong is a chess engine?" it is necessary to quantify the distinguishing characteristics of engines. The most important of these can then be examined and used to estimate playing strength.

There are many different characteristics which engines are currently understood and evaluated by. Engines will often exemplify one or more of these aspects, and can be tweaked to focus them as necessary:

- Middlegame Strength - Ability to search accurately and deeply to find material or positional advantages in middlegame positions. An engine with a strong middlegame can find advantages and snowball them to wins.
- Endgame Strength - Ability to find wins/ties in positions with few pieces. An engine with a strong endgame can play out technical positions that

- Aggressiveness - Desire to attack and pressure the opponent's king. An engine with an aggressive playstyle will try to apply pressure and gain tempo when possible.
- Overall Strength - Overall ability to beat other engines in full games. This metric provides a realistic overall estimate of an engine's validity and skill at chess.
- Adaptation - Ability of an engine to adapt to its opponent's playstyle and make changes to its own where necessary.
- Optimism - Desire to play into positions the engine is unsure of. An optimistic engine will play lines that are more "risky".

Although characteristics like aggressiveness and adaptability might make an engine more interesting to use, the overall strength of a chess engine is best understood in terms of its ability to win games of chess against a wide range of opponents. Metrics like "Endgame Strength" and "Middlegame Strength" can be used to estimate an engine's playing capacity in a particular phase of the game, but these metrics are inherently biased when being used to evaluate overall playing strength because they ignore the other important phases of the game.

Additionally, it is important that engines are tested with a variety of opponents, to eliminate biases caused by one engine's playstyle naturally doing well against another's.

3.3.2 Determining Strength by Evaluating Board Positions

Evaluating the best move for a particular board position is the main function of modern engines. Theoretically, the best engine will be the one that can find and make the best move at each board position it encounters, from the start to the end of a game.

To this end, board position tests are an extremely common and lightweight method of testing engines. These tests, often referred to as "tactics puzzles" or "debugging suites" are contrived board layouts with an intended best move (or sequence of moves). Although this intended solution is subjective and prone to error, it is usually verified by top humans and engines given hours of thinking time. Coming up with the solution to these puzzles typically requires a search of reasonable depth and/or a difficult decision between two or three good-looking candidate moves.

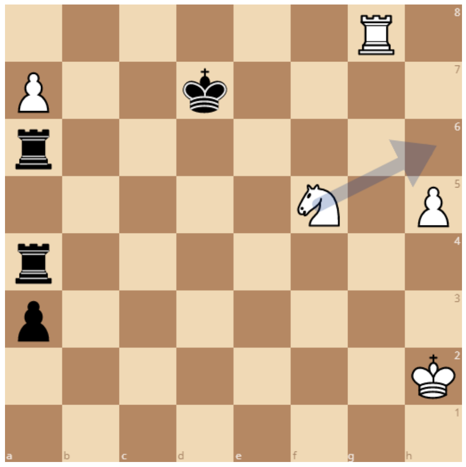


Figure 1: The Djaja position. White to move and draw

Figure 1, the "Djaja position," is an example of a well-known tactics puzzle that can be used to quickly test engines. The solution, Nh6!, is not obvious by any means, and requires careful analysis to discover. By moving the knight out of the way, White gains access to a perpetual check using their rook which prevents Black from promoting his/her pawn and winning the game.

Like every method of testing engines, however, solving contrived tactics puzzles has downsides and pitfalls that must be understood when using them:

- Tactics puzzles typically resemble middlegame positions, and might not give an accurate representation of an engine's ability to play the earlygame or endgame.
- Often, tactics puzzles are fairly short-sighted, and do not require a deep search to solve. This puts them contrary to many real chess positions, where an optimal solution can require a deep search and evaluation.
- The solution to a tactics puzzle is typically the best move by a wide margin. As a result, the puzzles often do not correspond to actual chess positions, where the best move is uncertain and an engine must pick between several seemingly viable moves.

Some of these issues can be mitigated by smart test development and usage, but others will always remain an issue for evaluating via board positions. Overall, tactics puzzles are useful as very lightweight tests for an "at-a-glance" overview of an engine's playing power, but not as an in-depth, all-inclusive analysis of an engine's strength. As a result, they will be used for a quick estimate of strength in the later testing.

3.3.3 Determining Strength by Playing Partial Games

Playing partial games attempts to determine an engine's overall aptitude in a fixed scenario. These tests begin at a predetermined state and have an expected result. Unlike simple tactics puzzles, these positions will not have markedly "correct" and "incorrect" moves, but a certain outcome will be expected, which still makes them useful in a quantitative analysis.

Partial games attempts to remedy some of the issues presented by evaluating static board states, while preserving their lightweight and simple evaluation. These games are more flexible and can be used to estimate an engine's "long-term" playing prowess. Still, however, playing non-full games can introduce bias into the evaluation of an engine, as certain parts of a game might be over-represented or under-represented.

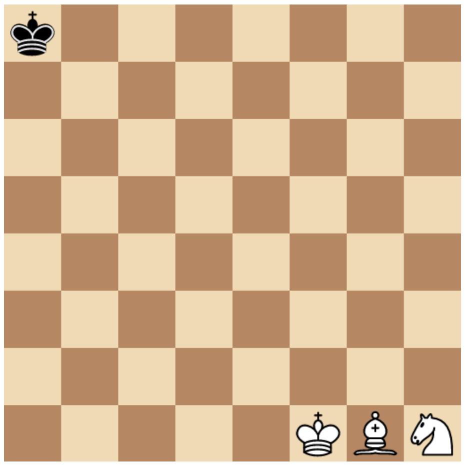


Figure 2: The BNK vs K checkmate. White to win in 29 moves

Figure 2 features one of the more difficult forced checkmate sequences. With proper play, White can force a win in 29 moves in this position or any like it. This checkmate sequence requires foresight and careful square control and manipulation of the Black king. With improper or impatient play from White, however, the game will be drawn due to the 50-move rule. Positions like these can test an engine's ability to find and execute long-term strategies that might not be immediately apparent with a simple low-depth search.

Overall, partial games are useful to gather a slightly more detailed estimation of an engine's playing strength, but are not as useful or unbiased as playing full games. As a result, they will be omitted in place of full games for the testing and analysis of this report.

3.3.4 Determining Strength by Playing Full Games

Playing entire, start-to-end games seeks to determine an engine's overall strength at playing chess. By comparing an engine's results with other engines or humans, it is possible to get an estimate of their overall ability to play chess well and produce wins. This approach does not restrict the evaluation to any specific phase of the game, but rather tests an engine's ability to navigate positions and game phases as they arise during actual play.

As a result, this approach eliminates many of the biases which can be introduced using other testing methods. Because of this, testing an engine's strength by playing full games is typically considered the most accurate and faithful method for determining overall playing strength.

Despite its increased overall fidelity and immunity to being biased by uncommon game states, using full games to determine an engine's strength has a few notable downsides:

- Hundreds, if not thousands of games are required in order to get accurate results. Since engine games have an element of uncertainty and randomness, a large sample size is usually required to be sure of two engine's relative strength.
- Testing full games against another engine uses the most CPU time, RAM, and other resources out of any testing method. Large numbers of longer games, as well as the requirement of running a reference engine, causes this method to be very costly in terms of computing resources. As a result, thorough tests will often require hours to days.
- Often this method requires many other engines to test against for a sufficient sample size. Only testing against a few other engines will potentially introduce a selection bias–an engine may appear strong when it is actually just strong against the opponents that were selected for it. In order to avoid this bias, a variety of opponent engines must be considered and tested against.

These downsides, however, can typically be mitigated with careful testing and planning. Playing full games will always have value for determining the overall ability of an engine to play chess. As a result, this report will make extensive use of this testing technique to ensure accurate, complete, and unbiased results.

3.4 Comparing Chess Engines

3.4.1 Variables to Consider when Evaluating Engines

The ability of a chess engine to play well can be evaluated across many variables. Depending on an engine's purpose, requirements, and resources, certain variables will inevitably be prioritized over others. Because of this, it is necessary to understand and evaluate each of these variables as they relate to an engine's performance as a whole.

The following section provides an overview of some of the more common variables and outlines their use in this report.

3.4.1.1 Move Strength

Move strength is a measurement of the overall "correctness" of an engine's moves. An engine that finds and makes stronger moves will be more likely to gain advantages and ultimately win games. This report will consider move strength primarily in terms of Elo and LOS, discussed in Sections 3.4.2 and 3.4.3 respectively.

Most engines will attempt to optimize move strength at the cost of other variables. Making strong moves that promote winning is the goal of most engines, which will attempt to do so within whatever other constraints are presented.

For the purpose of this report, move strength will be treated as a dependent variable. As other variables are changed, move strength will be used to determine the validity and importance of changes made.

3.4.1.2 Move Difficulty

Move difficulty is a measurement of the complexity of a board position that an engine must analyze. "Easier" positions have a smaller (or more easily prune-able) search space, due to checks, recaptures, mate threats, or other similarly forcing features. On the other hand, "harder" positions might involve many seemingly-valid but important decision points that will impact the game many turns into the future. Move difficulty is not measured empirically, but rather is measured in terms of how much trouble the moves present for other engines and/or human players.

Generally, more difficult moves will come at the cost of move strength and time taken. Difficult moves will take longer to evaluate and are less likely to result in optimal moves, with all else equal.

For the purpose of this report, move difficulty will be treated as a constant. All engines will be evaluated using the same techniques to ensure that they are accurately compared to one another.

3.4.1.3 Hardware Speed

Hardware speed is a measurement of the performance of the various components a chess engine runs on. Performance is measured in terms of GHz, Cores, ms, RPM, response time, and others.

An engine that uses higher performing hardware is generally expected to make moves faster and more accurately than one using slower, weaker hardware.

For this report, most hardware speed will be treated as a constant and will not change between tests. However, storage speed for memorized positions will be tested in order to better understand how overall performance depends on .

3.4.1.4 Time Taken

Time taken is a measurement of the amount of time it takes for an engine to determine the best move. On modern hardware, time taken is usually measured in either seconds or minutes, but can also sometimes be taken in terms of hours or days.

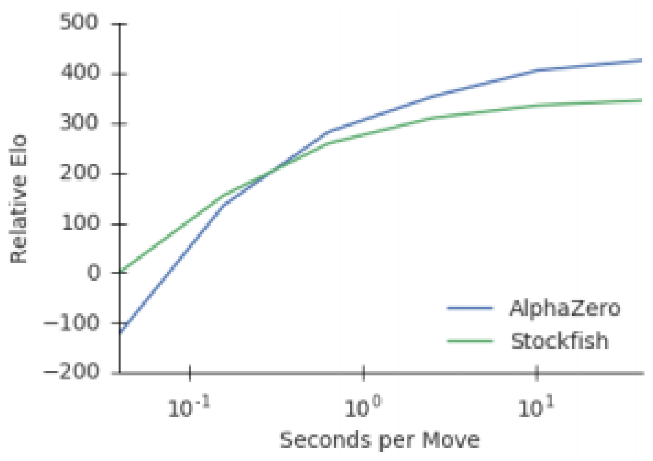


Figure 3: Relative Elo vs. Seconds per Move for AlphaZero and Stockfish [11]

In standard chess games, time is allocated on either a per-turn basis, a per-game basis, or a combination of the two. Typically, engines that are given more time to search will be able to find stronger moves, with all other factors equal. Figure 3 illustrates this trade-off for the engines AlphaZero and Stockfish.

For the purpose of this report, an engine's time will be used as an independent variable. Multiple trials will be performed using different time settings in order to determine the relevance of time to the results and how other variables depend on it.

3.4.1.5 Memory Usage

Memory usage is the space on RAM and on disk used by an engine. For modern chess applications, memory usage is considered in terms of Megabytes, Gigabytes, or Terabytes.

Typically, the less memory a program has to keep track of important information, the slower and less accurate its moves will be.

For the purpose of this report, disk space allocated to memorized information will be used as an independent variable. Tests and trials will be performed to better understand the influence of limited disk space on engine performance.

3.4.2 Elo Rating System

The Elo rating system is a mathematical model which attempts to approximate an agent's skill in a zero-sum game based on its performance in games against other players. This system was originally developed by Arpad Elo in 1939 as a statistically sound improvement to the existing Harkness system, which was prone to misrepresent player strength under certain circumstances. Although the Elo rating system makes a number of simplifying assumptions, its simplicity and general accuracy have made it the most commonly accepted method of comparing chess players today.

Under the Elo rating system, a player's skill is represented as a single number, which is adjusted as they win or lose games. A player's new rating after a match is calculated using the following formula [10]:

$$R_{new} = R_{old} + \frac{K}{2}\Big(W - L + \frac{1}{2}\frac{\Sigma_i D_i}{C}\Big) \tag{3}$$

Where R_{old} is the player's old rating, W is the number of wins, L is the number of losses, D_i is the difference in rating between players, and C and K are tuning constants (typically C = 200, K = 32). Larger K values allow faster rating changes and are therefore used when a player's rating is more uncertain.

The expected score of a player, E_A with rating R_A versus an opponent with a rating R_B can be calculated as follows [10]:

$$R_{new} = \frac{1}{1 + 10^{(R_B - R_A)/400}} \tag{4}$$

For a 100-point Elo advantage, the more highly rated player is expected to score 64%. For a 200-point gap, the expected score increases to 76%.

In order to use this mathematical model to predict player outcomes and condense player skill into a single number, a number of simplifying assumptions must be made:

- Since performance can only be inferred from wins, losses, or draws, it is assumed that the player who wins any given game performed at a higher skill level than their opponent.
- Since a player may perform above or below his/her “true skill” in any single game, it is assumed that a player’s performance is normally distributed around his/her rating. Therefore, stronger players are expected to occasionally have weak performances, and vise-versa.

Because more points are awarded for upset wins and upset losses, the Elo rating system is self-correcting and generally statistically sound. A player who goes on a lucky streak and ends up above his/her “skill level” will eventually fall back down once his/her luck changes.

Despite its popularity and simplicity, the Elo rating system has a few weaknesses. The system can often punish players whose “true skill” is below their current rating. For these players, playing a rated game against any opponent is a losing proposition. Likewise, savvy players can attempt to exploit the Elo rating system by targeting players who they perceive to be overrated, and avoiding those who seem underrated. Additionally, accurately determining a player’s rating typically requires a large sample of games, which can be problematic if a player is inactive or otherwise cannot play a statistically significant number of games.

Because of its widespread acceptance in comparing human chess players, the Elo rating system has become the de facto method used to compare engines. Although it loses some nuance and precision, abstracting an engine’s ability to play well into a single number makes comparisons between them much simpler. The issues presented by the Elo rating system are largely overcome when applied to engines that can play hundreds or thousands of games against each other.

Because of its usefulness when comparing engines, Elo rating estimates will be used in this report’s analysis.

3.4.3 Likelihood of Superiority

Likelihood of superiority, often abbreviated as LOS, is a measurement of statistical significance that attempts to estimate the probability that one player is more skilled than another. LOS does not attempt to indicate *how much more skilled* a player is their opponent, only the statistical likelihood of them being a stronger player based on a particular set of results. LOS can also be understood in terms of the Elo rating system discussed in 3.4.2 as a probability that a player’s Elo rating is at least one point higher than their opponent’s.

From a mathematical perspective, LOS can be approximated using the following formula, developed by Remi Coulom and Kai Laskos in 2009 [6]:

$$LOS = \frac{1}{2}\Big[\, 1 + erf\Big(\frac{wins - losses}{\sqrt{2(wins + losses)}}\Big)\Big] \tag{5}$$

Because White has a first-move advantage in chess, accurate usage of this formula to calculate LOS requires that both players have played an equal number of games as black and white. Failing to do so will bias this LOS calculation towards whichever player played as White more. Additionally, this formula omits drawn games as a simplifying assumption.

LOS is a very specialized measurement for comparing two engines or players that appear roughly equal in skill. Because LOS is a statistical measurement of probability, it is possible to quantitatively compare two engines that would otherwise be within the Elo rating system's margin of error.

In circumstances where two players are not equal in skill, however, LOS is not typically a useful measurement and will only reiterate what is apparent from Elo ratings.

Because of its usefulness when analyzing engines, especially those that are nearby in skill level, LOS estimates will be used in this report's analysis to determine which results are statistically significant.

3.5 Chess Engine Communication Protocols

In order to test engines automatically, a common communication protocol for interfacing between engines or between a testing suite is required. Two of such protocols are currently in use: the Universal Chess Interface and the Chess Engine Communication Protocol.

What follows is a general overview of each protocol and a brief analysis of their pros and cons.

3.5.1 Chess Engine Communication Protocol

The Chess Engine Communication Protocol, also known as XBoard or WinBoard (hence referred to as XBoard for brevity), was first developed in the 1990s by Frank Quinsinsky as a method to connect chess programs absent of a GUI. The XBoard protocol regulates chess engines using plaintext commands send over a program's standard input and output. The protocol was not centrally created, and grew over time as an extension to GNU chess. As a result, the protocol contained "several bugs and deficiencies" but was nonetheless used for automatic chess engine communication [3].

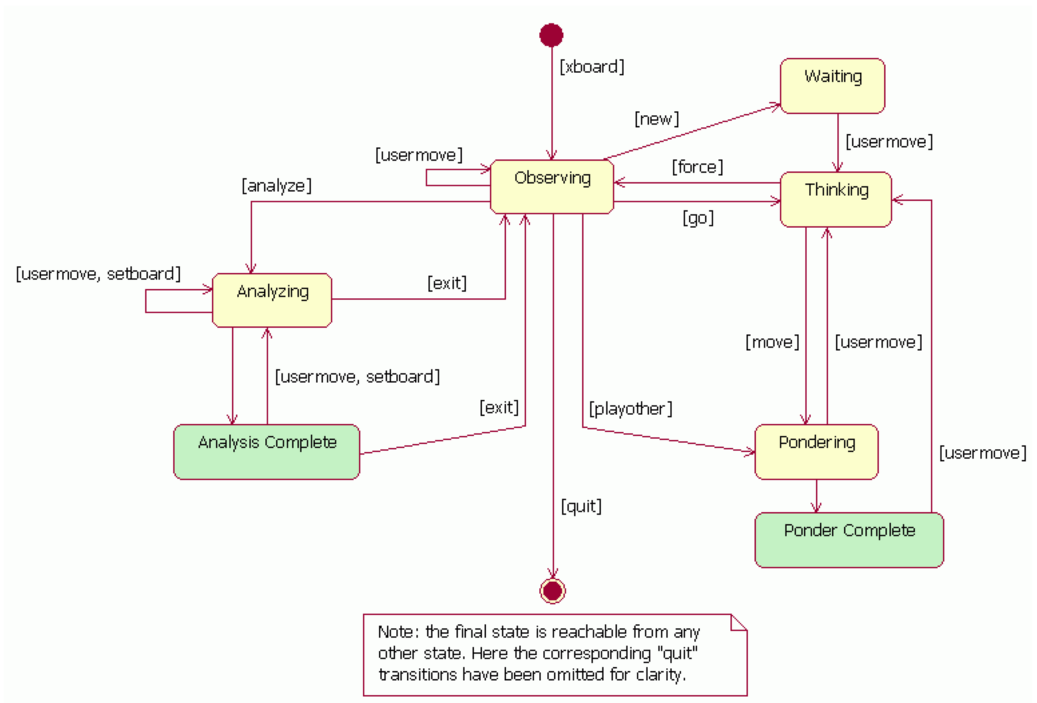


Figure 4: UML State Diagram of the XBoard Protocol by Alessandro Scotti [16]

As shown in Figure 4, XBoard is a stated protocol and requires that an engine match its expected state to establish correct communication.

In 2009, Harm Geert Muller established a Version 2 of the protocol, which modernized the protocol and supported nonstandard chess varients.

3.5.2 Universal Chess Interface

The Universal Chess Interface, or UCI, was released in November 2000 by Rudolf Huber and Stefan Meyer-Kahlen. UCI was designed from the ground-up to be a chess engine communication protocol, and has become the de facto modern replacement for XBoard.

Much like the XBoard protocol, UCI regulates a chess game using commands sent over an engine's standard input and output using plaintext commands. UCI also supports non-standard chess variants and allows the customization of engine parameters, like the modern revision to XBoard [5].

Unlike XBoard, UCI is a stateless protocol. Because of this, UCI is regarded as simpler to implement and less prone to bugs. Some developers have criticized UCI's stateless design, but by and large UCI has become the predominantly supported protocol amongst newer chess engines.

UCI options are set using the following syntax, in plaintext communication. The most commonly used arguments are tablebase locations and hash sizes (setting them is discussed in Sections B.5 and B.6):

```
setoption name <optionname> value <optionvalue>
```

Because of its wider support and more common adoption, the engines and software used in this report will all conform to the UCI protocol to ensure cross-functionality and cross-compatibility.

4 Analysis

4.1 Goals of Analysis

This report seeks to examine the importance of memorizing information in computer chess. By carefully analyzing the positions and situations where memorized positions have a small, large, or negligible impact, engine developers and users can make more informed design decisions.

The main questions this analysis seeks to examine and provide an answer for are as follows:

- What performance gains/losses can be expected when using memorized positions? Where do they come from?
- Will memorized positions give a larger performance boost to stronger or weaker engines?
- What effect does the amount of time an engine is given have on the usefulness of memorized positions?
- Are memorized positions required to compete at the highest level of chess engine performance?
- What effect does tablebase completeness have on performance? Are entire tablebases required for an engine to gain the benefits of tablebases?
- How important is the access speed of the medium that the memorized information is stored on? Will slower mediums negatively impact performance, and if so, by how much?

Answering these questions will help to better outline situations where the gains for using memorized positions is worth the drawbacks associated with them.

The approach, tools, and methods used to answer these questions are examined below.

4.2 Overall Approach

Because of the massive complexity involved in analyzing chess positions (discussed in Section 3.1), the semi-random nature of engines, and the wide variation between different strategies and techniques, it is difficult if not impossible to conclusively solve the above questions using pure theory. As a result, the questions proposed in Section 4.1 will be examined by empirically testing chess engines as they respond to changes in memorized information.

By carefully conducting and examining experiments whereby an engine's knowledge or environment is changed, it is possible to gain quantitative information about the stated goals and the usefulness of certain memorized information techniques. The exact tools and methods by which these experiments will be run are discussed below.

4.3 Tools Used

4.3.1 Strategic Test Suite

The Strategic Test Suite, or STS, is a collection of 1500 chess positions and expected moves used to test and debug engines (See Section 3.3.2 for more information on how this is done). The STS was originally constructed by Dann Corbit and Swaminathan Natarajan and covers a variety of middlegame topics.

The STS is one of the largest testing suites available and its solutions have been verified by top engines and players with hours of thinking time [7].

In 2015, Ferdinand Mosca published an analysis tool named STS_Ratings_v3, which can be used to automatically test and and evaluate an engine using the Strategic Test Suite.

Because of its large sample size, ease of use, relatively fast runtime, and accuracy, this analysis will use STS_Ratings_V3 to provide a quick estimate of an engine's midgame playing strength.

4.3.2 Cutechess-CLI

Cutechess-CLI is an open-source command-line interfaced tool that was first published by Arto Jonsson in 2009. It can be used to organize games, matches, or tournaments between two or more engines and record the results.

Cutechess-CLI supports the two most commonly used chess engine protocols, the Chess Engine Communication Protocol and the Universal Chess Interface, which are discussed further in Section 3.5. As a result, it is compatible with most modern engines when properly configured. Cutechess-CLI also supports a variety of customization opens and engine options, including chess variants and automatic output of results to .PGN files.

Because of its overall usefulness and flexibility, cutechess-CLI v1.0.0 will be used in this analysis in order to organize, execute, and record games between engines that are being tested.

4.3.3 Bayeselo

Bayeselo is a freeware command-line tool developed by Rémi Coulom in 2005. Bayeselo takes a sample of games between players and estimates the player's relative Elo and likelihood of superiority (discussed in Sections 3.4.2 and 3.4.3 respectively) which can be used to further analyze results.

Bayeselo has a number of features that make it an ideal analysis tool. Bayeselo accounts for chess's first-move advantage and behaves correctly when players have widely varying ratings. Additionally, Bayeselo has built-in support for chess game result files, .PGN, which makes it an ideal choice for analyzing chess games.

Because of its accuracy, LOS calculation, and ability to extract wins, losses, and players from .PGN files, Bayeselo will be used in this analysis. Bayeselo will be used in order to more precisely examine the differences between engine strength under different circumstances.

4.3.4 PGN-Extract

PGN-Extract is an open-source command-line tool developed by David J. Barnes. PGN-Extract allows for the manipulation and analysis of chess game result files (.PGN files). This manipulation can be done automatically using in game factors like game length or opening.

This analysis will use PGN-Extract v17-55 to split game results based on game length and number of pieces remaining. By analyzing games separately across these variables, it will be possible to better understand when and how engines make use of endgame memorized positions.

4.3.5 PolyGlot

PolyGlot is an open-source command-line interfaced tool developed by Fabien Letouzey. PolyGlot is an opening book manager and an adapter which allows communication between tools that rely on the normally incompatible Universal Chess Interface and Chess Engine Communication Protocols (which are discussed in more detail in Section 3.5).

PolyGlot supports engine configuration options, which are provided to it using a polygot.ini file. This feature allows UCI engines to be configured using the command line by invoking PolyGlot in the following way:

```
polygot polyglot.ini [-ec engine] [-ed enginedirectory]
```

This analysis will make use of PolyGlot v2.03 in order to translate between XBoard and UCI engines and to configure UCI engines using command-line arguments.

4.3.6 Nalimov Tablebases

The Nalimov Tablebases are the most commonly used 3-6 piece tablebases (discussed in further detail in Section 3.2.3.2). 5-piece Nalimov bases were completed in 1998 by Eugene Nalimov and offered many advantages in compression over other tablebases of the time. As a result of their early completion and efficiency, Nalimov tablebases have become the most widely supported engine tablebase format.

This analysis will use and test 5-piece Nalimov tablebases for engines that support them.

4.3.7 Sygyzy Tablebases

The Sygyzy tablebases, completed in 2013 by Ronald de Man, are also exhaustive endgame references for boards with 3-6 pieces. As a result of more modern compression techniques, the Sygyzy tablebases are smaller than the previously used Nalimov ones by a factor of around eight times (see Section 3.2.3.2 for more information on their sizes). As a result, some modern engines have migrated to using Sygyzy tablebases.

This analysis will use and test 5-piece Sygyzy tablebases where possible for engines that support them.

4.4 Overall Method

In order to answer the questions proposed in Section 4.1, the following three tests will be performed on an engine and a version of it that has been altered in some way. These tests are described in further detail in their individual sections: 4.5, 4.6, and 4.7 respectively:

- Time Control - An engine with no tablebase will play against an engine with a tablebase, at two different time controls.
- Tablebase Size - An engine with a complete 5-piece tablebase will be compared to one with a smaller partial tablebase.
- Drive Speed - An engine with its tablebase stored on an solid-state drive will play an engine with its tablebase stored on a traditional hard drive.

Each of these tests will be performed on a strong engine of around 3400 Elo, a medium-strength engine of around 3000 Elo, and a weaker engine of around 2600 Elo in order to understand how the results vary with an engine's overall playing strength.

Each test will involve STS suite comparisons and/or several hundred games engine games. The engine games played will be roughly split up as follows, to avoid biasing the results towards one or two engines:

- 50% head-to-head games between an engine and its modified counterpart.
- 25% games between the original engine and unmodified engines of similar skill, to establish a baseline for comparison.
- 25% games between the modified engine and unmodified engines of similar skill, which will be compared with its unmodified results.

These results will be broken down into games that lasted longer than 50 turns (per player) and games that lasted longer than 80 turns (per player) to better understand tablebase influence. The percentage of games that fall into each of these categories will also be given, as this information could be relevant for determining engine strength/ability with tablebases, as well as sample size. Games that before turn 50 are less likely to have been influenced by the presence of tablebases, whereas games that end after this point are much more likely to have been calculated using memorized tablebase information.

All results will be carefully examined using Bayeselo to estimate differences in playing strength and likelihood of superiorities between engine versions.

4.4.1 Constants

Except when otherwise noted, the following parameters will remain constant throughout testing:

- Memorized information will be stored on a 120GB Intel 530 SSD with a read latency of 80 microseconds. When a HDD is used, it will be a 7200 RPM WD Blue drive with a read latency of 6 Gb/s.
- An Intel Core i5 4670k CPU @ 3.40GHz will be used for all testing.
- A single thread will be given to each engine during testing.
- 10 seconds will be given to each engine for each game, to allow for more games to be played, unless otherwise stated.

- 256MB of DDR3 memory will be allocated for engine hashing using the UCI parameter “Hash=256”.
- 128MB of DDR3 memory will be used for tablebase hashing where possible using the UCI paremter “NalimovCache=128”.
- Default engine settings will be used when changes are not explicitly stated.

4.4.2 Engines Tested

The engines used in this analysis are shown in figures 3, 4 and 5. The engines in bold will be tested with and without tablebases, while those not in bold will be used as reference comparison engines.

Name	Tablebase Format	CCRL 40/40 Rating	Supported Protocol(s)
Aristarch 4.50	-	2597	XBoard/UCI
Amyan 1.72	-	2591	XBoard/UCI
Muse	-	2591	XBoard/UCI
N2 0.4	**Nalimov**	**2591**	**UCI**

Table 3: Weaker engines that will be used and their CCRL 40/40 ratings [8]

Name	Tablebase Format	CCRL 40/40 Rating	Supported Protocol(s)
Wasp 2.6	-	3047	UCI
iCE 3.0	-	3042	UCI
Spike 1.4	**Nalimov**	**3021**	**XBoard/UCI**
Spark 1.0	-	2973	UCI

Table 4: Medium-strength engines that will be used during analysis and their CCRL 40/40 ratings [8]

Name	Tablebase Format	CCRL 40/40 Rating	Supported Protocol(s)
Stockfish 9	**Sygyzy**	**3444**	**UCI**
Komodo 9	-	3315	UCI
Houdini 1.5a	-	3239	UCI

Table 5: Strong engines that will be used and their CCRL 40/40 ratings [8]

4.5 Test 1: Time Control

4.5.1 Test-Specific Goals

This test will analyze the impact of time on tablebase usefulness. It will compare performance gains when using tablebases during 10 second games to performance gains when using tablebases during 50 second games.

By better understanding the relationship between time and tablebase usefulness, engine developers and users will be able to make more informed choices about when to use tablebases. If tablebase performance was found to be more useful during shorter games, for instance, a user who wanted his/her engine to excel during these games would be much more inclined to use memorized information in his/her engine.

4.5.2 Variables

The following variables will be examined during this test:

- Time Control - Performance gains/losses for using a tablebase during short 10 second games will be compared to performance gains/losses during longer 50 second games.
- Engine Strength - Tests will be run on all three strengths of engine to find any correlations between engine strength and how time control affects performance.

4.5.3 Method

The following process will then be repeated for all three engine strength levels:

- 250 head-to-head games will be played between the engine with no tablebase and the same engine with a complete tablebase at a 10-seconds per-game, per-side time control.
- 100 additional head-to-head games will be played between the same engines at a 50 seconds per-game, per-side time control.

4.5.4 Results

4.5.4.1 Weak Engine

The results of the 250 head-to-head games played between weak engines with a 10-second time control are shown in Table 6. Tablebase use appears to significantly improve play, especially in the endgame.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
N2	N2_NoTB	-	100%	88	71	91	53%	+22	89%
N2	N2_NoTB	50 moves	96%	86	70	84	53%	+22	87%
N2	N2_NoTB	80 moves	54.4%	48	29	59	57%	+46	97%

Table 6: Results of 250 weak engine games played at 10s

The results of 100 head-to-head games between N2 using tablebases and its no-tablebase counterpart are shown in Table 7. Tablebase use does not appear to increase playing strength at this longer time control.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
N2	N2_NoTB	-	100%	14	16	70	49%	-6	41%
N2	N2_NoTB	50 moves	94%	14	16	65	48%	-8	36%
N2	N2_NoTB	80 moves	64%	7	12	45	46%	-18	27%

Table 7: Results of 100 weak engine games played at 50s

4.5.4.2 Medium-Strength Engine

The results of 250 head-to-head games played between medium-strength engines with a 10-second time control are shown in Table 8. Tablebase use appears to significantly strengthen overall play.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
Spike	Spike_NoTB	-	100%	86	68	96	54%	+24	90%
Spike	Spike_NoTB	50 moves	83.2%	83	58	67	56%	+40	97%
Spike	Spike_NoTB	80 moves	32%	30	19	31	57%	+40	89%

Table 8: Results of 250 medium-strength engine games played at 10s

The results of 100 head-to-head games between Spike using tablebases and its no-tablebase counterpart are shown in Table 9. Tablebase use appears to increase playing strength, albeit to a smaller degree.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
Spike	Spike_NoTB	-	100%	15	13	72	51%	+6	58%
Spike	Spike_NoTB	50 moves	73%	15	13	45	51%	+4	55%
Spike	Spike_NoTB	80 moves	25%	5	5	15	50%	-2	47%

Table 9: Results of 100 medium-strength engine games played at 50s

4.5.4.3 Strong Engine

The results of 250 head-to-head games played between strong engines with a 10-second time control are shown in Table 10. Tablebase use appears to increase playing strength, although not as significantly as it does for weaker engines.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
Stockfish	Stockfish_NoTB	-	100%	31	21	198	52%	+10	73%
Stockfish	Stockfish_NoTB	50 moves	88.4%	31	21	159	52%	+12	77%
Stockfish	Stockfish_NoTB	80 moves	30.4%	16	10	25	54%	+18	73%

Table 10: Results of 250 strong engine games played at 10s

The results of 100 head-to-head games between Stockfish using tablebases and its no-tablebase counterpart are shown in Table 11. Tablebase use appears to increase playing strength at a longer time control.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
Stockfish	Stockfish_NoTB	-	100%	12	4	84	54%	+18	76%
Stockfish	Stockfish_NoTB	50 moves	89%	12	4	73	54%	+20	76%
Stockfish	Stockfish_NoTB	80 moves	44%	9	4	31	56%	+20	70%

Table 11: Results of 100 strong engine games played at 50s

4.5.5 Analysis of Results

From the results found in Section 4.5.4, the following conclusions can be made about the variables outlined in Section 4.5.2:

- Tablebase use has a significant performance benefit across all tested levels of play when using a short time control. This gain can be expected to be approximately +20 Elo points for engines similar to those tested.
- In longer games, Tablebase use tends to help stronger engines more. Stockfish saw an 18-point gain on increased time settings, whereas Spike and N2 had changes of +6 and -6, respectively.

As a result of these statements, the following conclusions can be drawn about engine use and development:

- Tablebase use is important for optimal performance in short time control games. An engine that does not use them can expect to perform approximately 20 Elo points worse than an engine that does.
- Tablebase use only becomes important at higher playing strengths in long time control games.

4.6 Test 2: Tablebase Size

4.6.1 Test-Specific Goals

This test will analyze the relevance of tablebase size to engine performance. It will compare performance obtained when using complete tablebases to performance obtained using partial ones.

By determining the cost of using incomplete tablebases, engine devlopers and users will be able to make more informed choices about which tablebases to use. For example, if tests found that some tablebase files did not contribute noticably to engine performance, a user with only 100MB of memory on his/her phone could use only specific tablebases to maximize engine performance given his/her constraints.

4.6.2 Variables

The following variables will be examined during this test:

- Tablebase Completeness - An engine using complete 5-piece tablebases will be compared to an engine using partial 5-piece tablebases to analyze the impact of tablebase completeness on engine performance.
- Engine Strength - Tests will be run on all three strengths of engine to find any correlations between engine strength and how tablebase completeness affects performance.

4.6.3 Method

Sections B.2 and B.3 describe the process by which “trimmed” (slightly reduced tablebases with most functionality) and “stripped” (significantly reduced tablebases with only vital functionality remaining) tablebases will be obtained. The total sizes of these tablebases are shown in figure 12.

Name	Full Size	Trimmed Size	Stripped Size
Nalimov 5-Piece	7.05 GB	5.48 GB	481 MB
Sygyzy 5-Piece	938 MB	784 MB	93.3 MB

Table 12: Sizes of trimmed and stripped tablebases

The following process will then be repeated for all three engine strength levels in order to approximate the usefulness of "trimmed" and "stripped" tablebases:

- An engine with a full tablebase, an engine with a trimmed tablebase, and an engine with a stripped tablebase will be analyzed using the Strategic Test Suite three times, with their median score recorded.
- 250 head-to-head games will be played between the engine with a full tablebase and the same engine with a trimmed tablebase.
- 250 additional head-to-head games will be played between the engine with a full tablebase and the same engine with a stripped tablebase

4.6.4 Results

4.6.4.1 Weak Engine

STS results for the weak engine, N2, are shown below in Table 13:

Configuration	NumPos	BestCount	Score	Score(%)	Rating
N2	1500	511	6974	46.5%	1827
N2_TrimmedTB	1500	510	6964	46.4%	1825
N2_StrippedTB	1500	511	6974	46.5%	1827

Table 13: STS comparison between N2, N2_TrimmedTB, and N2_StrippedTB

The results of the 250 head-to-head games played between a weak engine using its full and trimmed tablebases are shown in Table 14. The trimmed tablebase appears to affect performance erratically, if at all.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
N2	N2_TrimmedTB	-	100%	87	85	78	50%	+2	54%
N2	N2_TrimmedTB	50 moves	96%	87	83	70	51%	+6	61%
N2	N2_TrimmedTB	80 moves	46.4%	35	44	37	46%	-20	22%

Table 14: 250 weak engine games played between complete and trimmed tablebases

Table 15 shows the results of the 250 head-to-head games played between a weak engine using its full and stripped tablebases. Interestingly, this test seems to indicate that the stripped tablebase might improve play, if it has any impact.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
N2	N2_StrippedTB	-	100%	78	91	81	47%	-18	17%
N2	N2_StrippedTB	50 moves	95.2%	77	86	75	48%	-12	24%
N2	N2_StrippedTB	80 moves	53.6%	44	42	48	51%	+4	56%

Table 15: 250 weak engine games played between complete and stripped tablebases

4.6.4.2 Medium-Strength Engine

STS results for the medium-strength engine, Spike, are shown below in Table 16:

Configuration	NumPos	BestCount	Score	Score(%)	Rating
Spike	1500	814	9941	66.3%	2708
Spike_TrimmedTB	1500	807	9877	65.8%	2689
Spike_StrippedTB	1500	816	9953	66.4%	2711

Table 16: STS comparison between Spike, Spike_TrimmedTB, and Spike_StrippedTB

The results of the 250 head-to-head games played between the medium-strength engine using its full and trimmed tablebases are shown in Table 17. This test shows a small performance boost for using trimmed tablebases.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
Spike	Spike_TrimTB	-	100%	67	76	107	48%	-12	26%
Spike	Spike_TrimTB	50 moves	81.6%	59	70	75	47%	-20	16%
Spike	Spike_TrimTB	80 moves	29.2%	20	21	32	49%	-8	41%

Table 17: 250 medium-strength engine games played between complete and trimmed tablebases

Table 18 shows the results of 250 head-to-head games played between the medium-strength test engine using its full and stripped tablebases. This test indicates a small performance boost for using full tablebases.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
Spike	Spike_StripTB	-	100%	76	66	108	52%	+12	76%
Spike	Spike_StripTB	50 moves	83.6%	67	58	84	52%	+14	75%
Spike	Spike_StripTB	80 moves	28.8%	19	20	33	49%	-4	45%

Table 18: 250 medium-strength engine games played between complete and stripped tablebases

4.6.4.3 Strong Engine

STS results for the strong engine, Stockfish, are shown below in Table 19:

Configuration	NumPos	BestCount	Score	Score(%)	Rating
Stockfish	1500	1062	12036	80.2%	3330
Stockfish_TrimmedTB	1500	1059	12034	80.2%	3329
Stockfish_StrippedTB	1500	1061	12043	80.3%	3332

Table 19: STS comparison between Stockfish, Stockfish_TrimmedTB, and Stockfish_StrippedTB

The results of the 250 head-to-head games played between Stockfish using its full and trimmed tablebases are shown in Table 20. This test shows no major difference between the tablebases.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
Stockfish	Stockfish_TrimTB	-	100%	23	24	205	49%	-2	42%
Stockfish	Stockfish_TrimTB	50 moves	81.6%	20	24	160	49%	-4	42%
Stockfish	Stockfish_TrimTB	80 moves	33.2%	14	14	55	50%	+0	50%

Table 20: 250 strong engine games played between complete and trimmed tablebases

Table 21 shows the results of 250 head-to-head games played between the strong test engine, Stockfish, using its full and stripped tablebases. This test shows a noticable improvement in performance when using the full tablebases.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
Stockfish	Stockfish_StripTB	-	100%	26	15	209	52%	+10	75%
Stockfish	Stockfish_StripTB	50 moves	81.2%	26	15	162	53%	+12	76%
Stockfish	Stockfish_StripTB	80 moves	32.8%	20	9	53	57%	+34	88%

Table 21: 250 strong engine games played between complete and stripped tablebases

4.6.5 Analysis of Results

From the results found in Section 4.6.4, the following conclusions can be made about the variables outlined in Section 4.6.2:

- The tablebase files removed in the trimming/stripping process did not impact STS scores.
- Performance remained roughly the same between trimmed and untrimmed tablebases across all engine strengths
- Performance took a small hit for using stripped tablebases, especially in the stronger engines.

As a result of these statements, the following conclusions can be drawn about engine use and development:

- When solving tactics puzzles, less tablebase information can be used with little performance loss, if any.
- Tablebases can be trimmed or stripped to use less disk space while maintaining their performance benefits.

4.7 Test 3: Drive Speed

4.7.1 Test-Specific Goals

This test will analyze the relevance of drive speed to tablebase performance. It will compare performance obtained when using tablebases on solid-state memory versus performance on slower HDD memory.

By determining the cost of using an engine on slower memory, engine developers and users will be able to make more informed decisions on when and where to use tablebases. For instance, if tests found a negligable difference between SSD and HDD tablebase speed, a user could potentially obtain a performance benefit from downloading larger tablebases for use on his/her slower but less expensive memory.

4.7.2 Variables

The following variables will be examined during this test:

- Drive Speed - An engine using tablebases stored on a SSD will be compared to the same engine using tablebases stored on a HDD to better understand how drive speed impacts the performance of memorized positions.
- Engine Strength - Tests will be run on all three strengths of engine to find any correlations between engine strength and how drive speed affects performance.

4.7.3 Method

The following process will be repeated for all three engine strength levels:

- The engine using SSD-using engine and its HDD-using counterpart will be tested using the Strategic Test Suite three times. The median result will be used in analysis.
- 250 head-to-head games will be played between the SSD-using engine and its HDD-using counterpart.
- 120 games will be played between the SSD-using engine and the similar-strength opponents outlined in Tables 3-5.
- 120 games will be played between the HDD-using counterpart and its similar-strength opponents.

4.7.4 Results

4.7.4.1 Weak Engine

STS results for the weak engine, N2, are shown below in Table 22:

Configuration	NumPos	BestCount	Score	Score(%)	Rating
N2	1500	511	6974	46.5%	1827
N2_SlowTB	1500	511	6969	46.5%	1826

Table 22: STS comparison between N2 and N2_SlowTB

The results of the 250 head-to-head games played between a weak engine using SSD and HDD tablebases are shown in Table 23. The faster tablebase access appears to noticeably improve overall play in this strength bracket.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
N2	N2_SlowTB	-	100%	98	73	79	55%	+34	96%
N2	N2_SlowTB	50 moves	96%	96	73	71	55%	+32	95%
N2	N2_SlowTB	80 moves	49.6%	48	37	39	54%	+30	88%

Table 23: 250 weak engine games played between SSD and HDD tablebases

Table 24 shows the results of the 240 games played between N2 and its similar engines. These results seem to indicate a performance loss when using the faster tablebases.

Player	Opponent	W	L	D	Score	Est. Elo	LOS
N2	Aristarch	19	13	8	56%	-	-
N2	Amyan	20	5	15	69%	-	-
N2	Muse	12	20	8	40%	-	-
N2_SlowTB	Aristarch	14	10	16	55%	-	-
N2_SlowTB	Amyan	20	11	9	61%	-	-
N2_SlowTB	Muse	19	9	12	62%	-	-
N2	N2_SlowTB	-	-	-	-	-45	11%

Table 24: 240 games played between N2 variants and their weak opponents

4.7.4.2 Medium-Strength Engine

STS results for the medium-strength engine, Spike, are shown below in Table 25:

Configuration	NumPos	BestCount	Score	Score(%)	Rating
Spike	1500	814	9941	66.3%	2708
Spike_SlowTB	1500	811	9889	65.9%	2692

Table 25: STS comparison between Spike and Spike_SlowTB

Table 26 shows the results of 250 head-to-head games played between a medium-strength engine using SSD and HDD tablebases. The faster tablebase access appears to have a significant effect on overall playing strength in most of the games.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
Spike	Spike_SlowTB	-	100%	87	56	107	56%	+38	98%
Spike	Spike_SlowTB	50 moves	85.2%	80	52	81	57%	+42	98%
Spike	Spike_SlowTB	80 moves	30.4%	24	21	31	52%	+10	61%

Table 26: 250 medium-strength engine games played between SSD and HDD tablebases

The results of the 240-game test between Spike and similar engines are shown in Table 27 below. In this test, the increase in tablebase access time seems to have had a negligable impact on Spike's performance.

Player	Opponent	W	L	D	Score	Est. Elo	LOS
Spike	Wasp	4	22	14	28%	-	-
Spike	iCE	6	25	9	26%	-	-
Spike	Spark	38	0	2	95%	-	-
Spike_SlowTB	Wasp	6	22	12	30%	-	-
Spike_SlowTB	iCE	4	25	11	24%	-	-
Spike_SlowTB	Spark	40	0	0	100%	-	-
Spike	Spike_SlowTB	-	-	-	-	-15	37%

Table 27: 240 games played between Spike variants and their medium-strength opponents

4.7.4.3 Strong Engine

STS results for the strong engine, Stockfish, are shown below in Table 28:

Configuration	NumPos	BestCount	Score	Score(%)	Rating
Stockfish	1500	1062	12036	80.2%	3330
Stockfish_SlowTB	1500	1061	12014	80.1%	3323

Table 28: STS comparison between Stockfish and Stockfish_SlowTB

Table 29 shows the results of 250 head-to-head games played between a strong engine using SSD and HDD tablebases. The difference in tablebase access time seems to have an exceedingly small effect on Stockfish's performance, if any.

Player	Opponent	Filter	% Games	W	L	D	Score	Est. Elo	LOS
Stockfish	Stockfish_SlowTB	-	100%	27	27	196	50%	+0	51%
Stockfish	Stockfish_SlowTB	50 moves	82%	27	27	151	50%	+0	51%
Stockfish	Stockfish_SlowTB	80 moves	39.6%	18	16	40	51%	+4	54%

Table 29: 250 strong engine games played between SSD and HDD tablebases

The results of the 240-game test between Stockfish and similar engines are shown in Table 30 below. In this test, the increase in tablebase access time seems to have had a strong impact on Stockfish's performance.

Player	Opponent	W	L	D	Score	Est. Elo	LOS
Stockfish	Komodo	35	1	24	78%	-	-
Stockfish	Houdini	56	1	3	96%	-	-
Stockfish_SlowTB	Komodo	29	8	23	68%	-	-
Stockfish_SlowTB	Houdini	47	3	10	87%	-	-
Stockfish	Stockfish_SlowTB	-	-	-	-	+103	98%

Table 30: 240 games played between Stockfish variants and their strong opponents

4.7.5 Analysis of Results

From the results found in Section 4.7.4, the following conclusions can be made about the variables outlined in Section 4.7.2:

- Tablebase access speed has a slight overall correlation with engine performance in real games. Tablebase speed has little to no impact on solving the Strategic Test Suite.
- Engine strength does not appear to have any relationship with tablebase access speed and relative performance gains/losses. Performance gains/losses are situational, if relevant.

As a result of these statements, the following conclusions can be drawn about engine use and development:

- When solving tactics puzzles, tablebases can be stored on slower mediums with little performance loss, if any.
- Tablebase speed has a highly situational effect on engine performance, and varies based on engine and opponent.

5 Conclusion

This report sought to gain quantitative insight on the pros and cons of using massive tables of memorized data in a difficult, large search-space game like chess. To this end, the analysis measured performance differences among chess engines over multiple variables.

In order to better understand the importance and impact of memorized positions in modern chess engines, each of the following factors was analyzed in depth:

- Influence of time control on tablebase importance
- Relationship between tablebase size and tablebase performance
- Importance of tablebase access time to engine performance

Through this analysis, the following conclusions were demonstrated and observed:

- Tablebase use is important for optimal performance in short time control games. An engine that does not use them can expect to perform approximately 20 Elo points worse than an engine that does.
- Tablebase use only becomes important at higher playing strengths in long time control games.
- When solving tactics puzzles, tablebases can be stored on slower mediums with little performance loss, if any.
- Tablebase speed has a highly situational effect on engine performance, and varies based on engine and opponent.
- When solving tactics puzzles, less tablebase information can be used with little performance loss, if any.
- Tablebases can be trimmed or stripped to use less disk space while maintaining their performance benefits.

These discovered results can be used to better tune and customize chess engines to meet the needs of specific applications. The results found can inform researchers, engine programmers, and engine users in the future. Although this information is relatively specific to chess, the techniques, tactics used, and general principles discovered can be applied to many other complex topics.

5.1 Future Work

Opening Books This report focused heavily on endgame tablebases. Further analysis can be done on the importance of opening books in modern strong and weak engines.

Deeper Testing on a Per-Engine Basis This report used a wide approach to engine testing in order to gain insight about engine trends. Future work could look very closely at a single engine's source code to discover the circumstances in which memorized positions have value.

A References

References

[1] "Endgame tablebases." [Online]. Available: https://chessprogramming.wikispaces.com/Endgame+Tablebases

Information about Endgame Tablebases, including a table of the sizes of commonly used ones

[2] "Lomonosov endgame tablebases." [Online]. Available: http://chessok.com/?page_id=27966

Information about the 7-men Lomonsov Tablebases

[3] "Xboard project history," 2000. [Online]. Available: http://tim-mann.org/history.html

Describes the origin of XBoard protocol

[4] "Can use of endgame tablebases weaken play?" 2003. [Online]. Available: http://horizonchess.com/FAQ/Winboard/weaktablebase.html

Discusses and analyzes the significance of the ways tablebase use can hurt engine performance

[5] "Uci protocol," 2004. [Online]. Available: http://wbec-ridderkerk.nl/html/UCIProtocol.html

Official documentation for the UCI protocol

[6] "Likelihood of superiority," 2009. [Online]. Available: http://www.talkchess.com/forum3/viewtopic.php?f=7&t=30624&sid=dfb9efd39a63f54340c94a422fb8fa22

Includes the cited formula for calculating LOS

[7] "Strategic test suite," 2009. [Online]. Available: https://sites.google.com/site/strategictestsuite/

Official webpage of the strategic test suite

[8] "Ccrl 40/40," 2018. [Online]. Available: http://www.computerchess.org.uk/ccrl/4040/

The most widely used and accurate computer chess comparison. Has information about most engines and their historical ratings

[9] V. Allis, "Searching for solutions in games and artificial intelligence," 1994. [Online]. Available: http://fragrieu.free.fr/SearchingForSolutions.pdf

Goes into great depth about the complexity of chess and its potential as an intractable problem

[10] A. Elo, "The rating of chess players, past and present," 1978.

Arpad Elo's book which describes and explains the Elo rating system

[11] D. S. et All., "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," 2018. [Online]. Available: https://arxiv.org/pdf/1712.01815.pdf

A widely popular paper that contains a good amount of insightful chess information

[12] G. C. Haworth, "Reference fallible endgame play," 2002. [Online]. Available: http://centaur.reading.ac.uk/4579/1/2002c_CoW7_H_Reference_Fallible_Endgame_Play.pdf

Goes over methods to improve tablebase navigation versus non-perfect players

[13] T. A. Marsland and P. G. Rushton, "Mechanisms for comparing chess programs," 1973. [Online]. Available: http://webdocs.cs.ualberta.ca/~tony/OldPapers/Marsland-Rushton-ACM73

One of the first papers to discuss testing of computer chess engines

[14] E. Nalimov, G. C. Haworth, and E. Heinz, "Space efficient indexing of chess endgame tables," 2000. [Online]. Available: http://centaur.reading.ac.uk/4562/1/2000c_ICGA_J_NHH_Space-Efficient_Indexing.pdf

One of the first papers and most in-depth papers on endgame tablebase compression

[15] J. Rollason, “Statistical minefields with version testing,” 2007. [Online]. Available: http://www.aifactory.co.uk/newsletter/2007_04_stat_minefields.htm

A publication that discusses modern engine testing

[16] A. Scotti, “Winboard/xboard protocol state diagram,” 2004. [Online]. Available: https://walkofmind.com/programming/chess/xboard.htm

Excellent illustration of the XBoard protocol and state machine

[17] C. E. Shannon, “Programming a computer for playing chess,” 1949. [Online]. Available: https://www.webcitation.org/5oFLE7Mgx

One of the first and most influential papers published in the creation of chess engines

[18] K. Thompson, “Retrograde analysis of certain endgames,” 1986. [Online]. Available: https://pdos.csail.mit.edu/~rsc/thompson86endgame.pdf

One of the first papers and researchers to investigate the construction of exhaustive endgame tablebases

B Appendix

B.1 Tablebase File Notation

Tablebase files are usually organized in the following format:

```
K<piece><piece><...>[v]K<piece><...>
```

With `piece` being replaced by a piece's letter and a `v` sometimes being added for clarity.

For example, a tablebase file containing information about a king and two pawns versus a lone king would be named similarly to:

```
KPPvK.rtbz
```

B.2 Trimming 5-Piece Tablebases

The goal of creating a "trimmed" 5-piece tablebase is to remove completely superfluous endgames that do not contribute to engine performance in order to save some drive space. The tablebases that will be trimmed are those that are either exceedingly unlikely to occur in a real game or those that are trivial to win for any competent player.

To this end, the following tablebase files will be removed to create a "trimmed" tablebase file (reference Section B.1 for information on what these labels mean):

- KBBB versus K - An endgame with 3 bishops of the same color is exceedingly unlikely to ever occur between two players trying to win, as it requires underpromotion of a pawn.
- KNNN versus K - Likewise, an endgame with 3 knights of the same color is unlikely to ever occur.
- KQQ versus anything - This matchup is generally trivial to win or calculate for the side with two queens, even against a single queen.
- KQR versus anything - This matchup is also trivial to calculate. Even against a queen this is an easy win for the Queen + Rook side.
- KQN versus anything except another queen - Against any piece lower than a queen, this matchup is an easy win.
- KQB versus anything except another queen - See above.
- KRR versus anything except a queen - See above.

This process results in a size reduction of roughly 20%. The ramifications of these reductions are analyzed and discussed in Section 4.6.

B.3 Stripping 5-Piece Tablebases

The goal of creating a "stripped" 5-piece tablebase is to remove all but the most common and useful 5-piece endgames to significantly reduce tablebase size while maintaining some of the benefits of tablebases.

To this end, the following 5-piece tablebase files will be kept from a "trimmed" tablebase in order to create a "stripped" tablebase file (reference Section B.1 for information on what these labels mean). All other 5-piece files will be discarded:

- KQP versus KQ - This matchup is difficult to play correctly and easily be drawn.
- KRP versus KR - Same as above.
- KPP versus KP - Same as above.
- KPP versus KR - This matchup can sometimes be won or drawn by the side with two pawns using proper planning.
- KRB versus KR - This matchup is usually a draw if played correctly, but tablebase use can lead to surprising wins.
- KRN versus KR - Same as above.

"Stripping" a tablebase as described results in a size reduction of roughly 90%. The ramifications of this massive reduction is analyzed and discussed in Section 4.6.

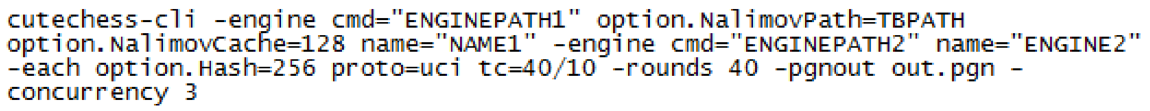
B.4 Using PGN-Extract

Using PGN-Extract to output the games which lasted for `X` or less moves from `in.pgn` to `out.pgn` is done as follows. This is done using a .bat file and is repeated for 50 and 80 moves:

```
pgn-extract -blX --output out.pgn in.pgn
```

B.5 Using Cutechess CLI

Cutechess CLI is used to organize games played between two or more engines. For this analysis, it was typically called using a .bat file as follows, with some slight modification based on the specific test:



```
cutechess-cli -engine cmd="ENGINEPATH1" option.NalimovPath=TBPATH
option.NalimovCache=128 name="NAME1" -engine cmd="ENGINEPATH2" name="ENGINE2"
-each option.Hash=256 proto=uci tc=40/10 -rounds 40 -pgnout out.pgn -
concurrency 3
```

Figure 5: The .bat file used to run Cutechess CLI

- `-engine` - indcates that following parameters describes a single engine. Two uses of this command are used to set up the engines
- `cmd=` - indicates the path to an engine's executable
- `name=` - the name an engine will be given in the output file
- `option.OPTIONNAME=` - sets an engine's option to the specified value. This is used to set up tablebases and hash size
- `-each` - indicates that following parameters describe both engines
- `proto=` - the protocol this engine uses
- `tc=` - the time control this engine will use
- `-rounds` - the number of games to play
- `-pgnout` - the output file for game results
- `-concurrency` - the number of games that can be played at once via multi-threading

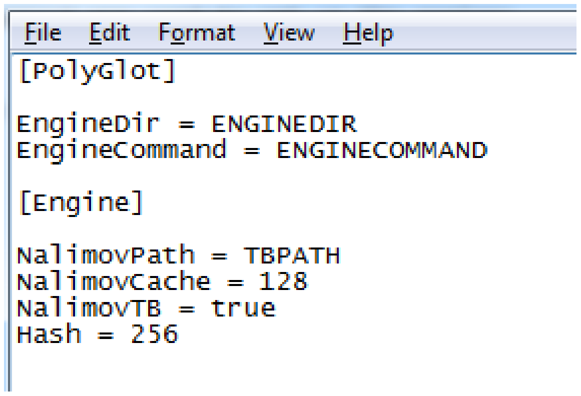
B.6 Using STS With Engine Options

PolyGlot is used in order to properly configure engines that are being run by the Strategic Test Suite. In this process, STS will run PolyGlot, which will properly configure and then run the appropriate engine.

The .bat file used to run PolyGlot using the Strategic Test Suite is as follows, where `sts.epd` contains the tests:

```
STS_Rating_v3 -f "sts.epd" -e "polyglot pg.ini" --proto uci -h 256 --getrating
```

The pg.ini is a PolyGlot initialization file and contains information about which engine to run and which settings to use. A standard one is shown below:



```
[PolyGlot]

EngineDir = ENGINEDIR
EngineCommand = ENGINECOMMAND

[Engine]

NalimovPath = TBPATH
NalimovCache = 128
NalimovTB = true
Hash = 256
```

Figure 6: A simple PolyGlot initialization file

B.7 Testing to Ensure Tablebases are Properly Configured

Unless an engine has a logging feature, it's difficult to tell when endgame tablebases are correctly loaded. There is no universally-supported way for an engine to inform a user of improper tablebase configuration. Because tablebase use is central to this project, the following test is used with all new engines and tablebases to ensure they had been properly configured and were working as intended:

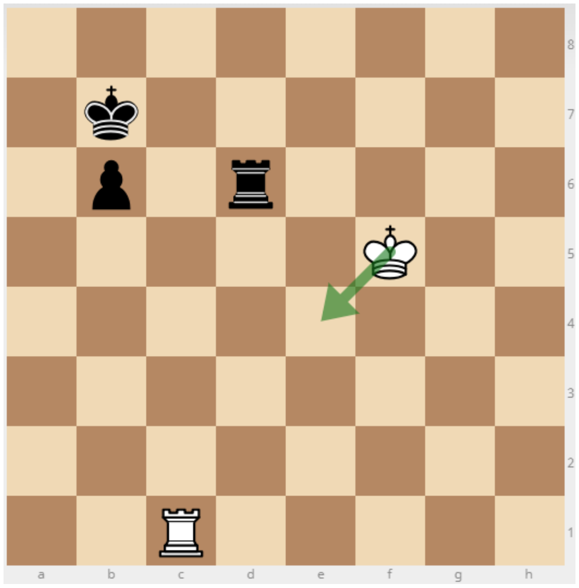
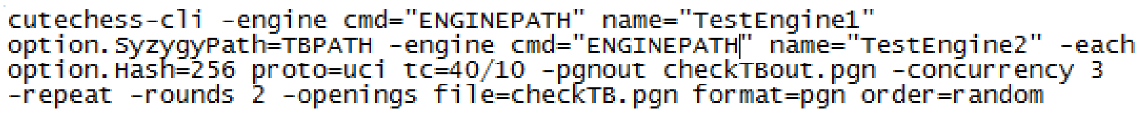


Figure 7: The tablebase test position. Ke4 leads to a draw in 15 moves, but all other positions lead to Black winning

The position shown above was used to test that engines were correctly accessing the 5-piece tablebases. When played according to a tablebase, Ke4 is the only good move. When an engine tries to play this position without a tablebase, however, it typically choses Ke5. As a result, a mis-configured engine can be immediately detected.

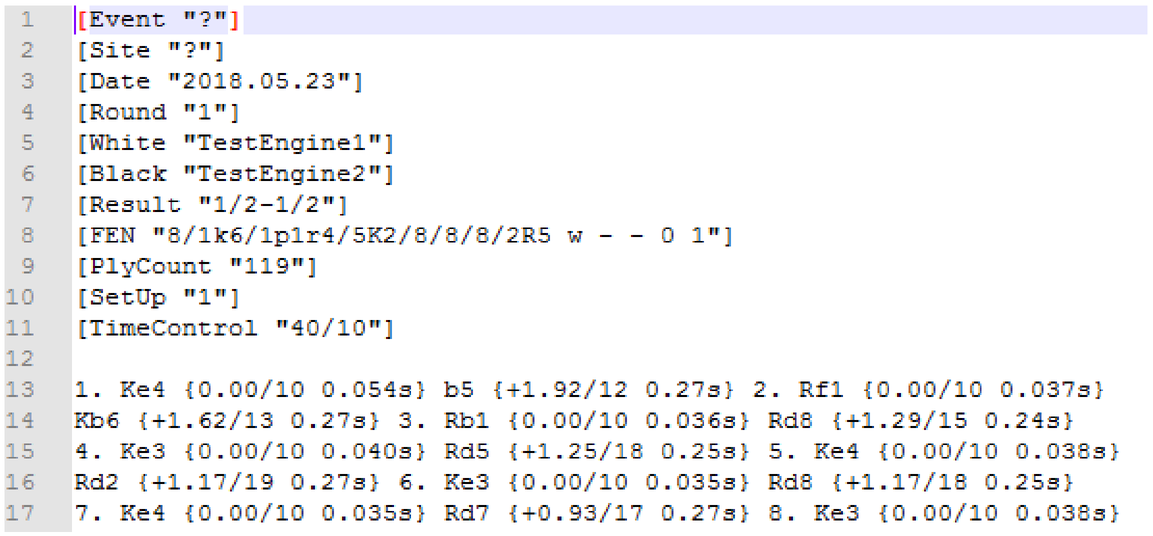
This test was automated using Cutechess CLI using a .bat file with the following one-line command. Refer to Section B.5 for more information about using Cutechess:



```
cutechess-cli -engine cmd="ENGINEPATH" name="TestEngine1"
option.SyzygyPath=TBPATH -engine cmd="ENGINEPATH" name="TestEngine2" -each
option.Hash=256 proto=uci tc=40/10 -pgnout checkTBout.pgn -concurrency 3
-repeat -rounds 2 -openings file=checkTB.pgn format=pgn order=random
```

Figure 8: The .bat file used to execute the cutechess tablebase test. checkTB.pgn contains the position to test

This command gives TestEngine1 the path to the tablebases, but does not give it to TestEngine2. Cutechess plays the same position from both sides and records the results. If the tablebases are correctly configured, TestEngine1 will tie as White (making the Ke4 move discussed above). If the tablebases are incorrectly configured, TestEngine1 will lose as White (making the mistake move Ke5).



```
[Event "?"]
[Site "?"]
[Date "2018.05.23"]
[Round "1"]
[White "TestEngine1"]
[Black "TestEngine2"]
[Result "1/2-1/2"]
[FEN "8/1k6/1p1r4/5K2/8/8/8/2R5 w - - 0 1"]
[PlyCount "119"]
[SetUp "1"]
[TimeControl "40/10"]

1. Ke4 {0.00/10 0.054s} b5 {+1.92/12 0.27s} 2. Rf1 {0.00/10 0.037s}
Kb6 {+1.62/13 0.27s} 3. Rb1 {0.00/10 0.036s} Rd8 {+1.29/15 0.24s}
4. Ke3 {0.00/10 0.040s} Rd5 {+1.25/18 0.25s} 5. Ke4 {0.00/10 0.038s}
Rd2 {+1.17/19 0.27s} 6. Ke3 {0.00/10 0.035s} Rd8 {+1.17/18 0.25s}
7. Ke4 {0.00/10 0.035s} Rd7 {+0.93/17 0.27s} 8. Ke3 {0.00/10 0.038s}
```

Figure 9: An output file of a successful test

The above picture shows the beginning of an output file after proper configuration. Note the tied result of "1/2-1/2" and the first move of Ke4.

B.8 Using Bayeselo

Information about W/L/D, Elo, and LOS were calculated using the following Bayeselo commands:

`readpgn input.pgn`

This command reads player names and game records out of a Portable Game Notation (.PGN) file.

The `elo` command opens Bayeselo's Elo-estimation interface.

The `mm` command will then compute maximium likelihood Elo ratings from the given data and print the amount of time it took.

The `exactdist` command then computes Elo intervals assuming exact opponent Elo ratings.

The `ratings` command then prints out a table of all players and their relative ratings.

The `los` command then prints a likelihood-of-superiority matrix, which can be used find an engine's LOS with any other one analyzed.

B.9 Glossary

50-Move Rule A player may claim a draw if no pawn movement or capture has been made in the last 50 moves (50 moves for each player). Some endgames have forced checkmate sequences that take more than 50 moves and are thus counted as draws due to this rule.

Solid-State Drive A solid-state drive is a nonvolatile memory that has a faster access time and lower latency than traditional platter-based hard disk drives. Because of this, solid-state drives are often recommended by engine developers and other computer users trying to get the fastest performance out of large, long-term memories.

B.10 Full Code

Note that these files refer to file locations as they were set up on my machine. This code could require modification to work on machines where the files are located in different places.
extract50.bat:

```
pgn-extract -bl50 --output match50.pgn in.pgn
```

extract80.bat:

```
pgn-extract -bl80 --output match80.pgn in.pgn
```

RunSTSRating_v3.bat:

```
STS_Rating_v3 -f "STS1-STS15_LAN.epd" -e "polyglot polyglot.
    ini" --proto uci -h 256 --getrating
```

polyglot_N2.ini:

```
[PolyGlot]

EngineDir = C:\Users\Chris\Desktop\SeniorProjectStuff\
    About2500\n2_x64
EngineCommand = n2_x64.exe

[Engine]

NalimovPath = C:\Users\Chris\Desktop\SeniorProjectStuff\
    Nalimov
```

```
NalimovCache = 128
Hash = 256
```

polyglot_N2_SlowTB.ini:

```
[PolyGlot]

EngineDir = C:\Users\Chris\Desktop\SeniorProjectStuff\
    About2500\n2_x64
EngineCommand = n2_x64.exe

[Engine]

NalimovPath = G:\Nalimov
NalimovCache = 128
Hash = 256
```

polyglot_N2_StrippedTB.ini:

```
[PolyGlot]

EngineDir = C:\Users\Chris\Desktop\SeniorProjectStuff\
    About2500\n2_x64
EngineCommand = n2_x64.exe

[Engine]

NalimovPath = C:\Users\Chris\Desktop\SeniorProjectStuff\
    Nalimov_stripped
NalimovCache = 128
Hash = 256
```

polyglot_N2_TrimmedTB.ini:

```
[PolyGlot]

EngineDir = C:\Users\Chris\Desktop\SeniorProjectStuff\
    About2500\n2_x64
EngineCommand = n2_x64.exe

[Engine]
```

```
NalimovPath = C:\Users\Chris\Desktop\SeniorProjectStuff\
    Nalimov_trimmed
NalimovCache = 128
Hash = 256
```

polyglot_Spike.ini:

```
[PolyGlot]

EngineDir = C:\Users\Chris\Desktop\SeniorProjectStuff\
    About3000\spike_14
EngineCommand = Spike1.4.exe

[Engine]

NalimovPath = C:\Users\Chris\Desktop\SeniorProjectStuff\
    Nalimov
NalimovCache = 128
Hash = 256
```

polyglot_Spike_SlowTB.ini:

```
[PolyGlot]

EngineDir = C:\Users\Chris\Desktop\SeniorProjectStuff\
    About3000\spike_14
EngineCommand = Spike1.4.exe

[Engine]

NalimovPath = G:\Nalimov
NalimovCache = 128
Hash = 256
```

polyglot_Spike_StrippedTB.ini:

```
[PolyGlot]

EngineDir = C:\Users\Chris\Desktop\SeniorProjectStuff\
    About3000\spike_14
EngineCommand = Spike1.4.exe
```

```
[Engine]

NalimovPath = C:\Users\Chris\Desktop\SeniorProjectStuff\
    Nalimov_stripped
NalimovCache = 128
Hash = 256
```

polyglot_Spike_TrimmedTB.ini:

```
[PolyGlot]

EngineDir = C:\Users\Chris\Desktop\SeniorProjectStuff\
    About3000\spike_14
EngineCommand = Spike1.4.exe

[Engine]

NalimovPath = C:\Users\Chris\Desktop\SeniorProjectStuff\
    Nalimov_trimmed
NalimovCache = 128
Hash = 256
```

```
[PolyGlot]

EngineDir = C:\Users\Chris\Desktop\SeniorProjectStuff\
    About3400\stockfish-9-win\Windows
EngineCommand = stockfish_9_x64.exe

[Engine]

SyzygyPath = C:\Users\Chris\Desktop\SeniorProjectStuff\
    syzygy
Hash = 256
```

polyglot_Stockfish_SlowTB.ini:

```
[PolyGlot]

EngineDir = C:\Users\Chris\Desktop\SeniorProjectStuff\
    About3400\stockfish-9-win\Windows
EngineCommand = stockfish_9_x64.exe
```

```
[Engine]

SyzygyPath = G:\syzygy
Hash = 256
```

polyglot_Stockfish_StrippedTB.ini:

```
[PolyGlot]

EngineDir = C:\Users\Chris\Desktop\SeniorProjectStuff\
    About3400\stockfish-9-win\Windows
EngineCommand = stockfish_9_x64.exe

[Engine]

SyzygyPath = C:\Users\Chris\Desktop\SeniorProjectStuff\
    syzygy_stripped
Hash = 256
```

polyglot_Stockfish_TrimmedTB.ini:

```
[PolyGlot]

EngineDir = C:\Users\Chris\Desktop\SeniorProjectStuff\
    About3400\stockfish-9-win\Windows
EngineCommand = stockfish_9_x64.exe

[Engine]

SyzygyPath = C:\Users\Chris\Desktop\SeniorProjectStuff\
    syzygy_trimmed
Hash = 256
```

checkTB.pgn:

```
[FEN "8/1k6/1p1r4/5K2/8/8/8/2R5 w - - 0 1"]
```

cutechess_checkN2.bat:

```
cutechess-cli -engine cmd="C:\Users\Chris\Desktop\
    SeniorProjectStuff\About2500\n2_x64\n2_x64.exe" name="
    TestEngine1" option.NalimovPath=C:\Users\Chris\Desktop\
    SeniorProjectStuff\Nalimov option.NalimovCache=128 -
    engine cmd="C:\Users\Chris\Desktop\SeniorProjectStuff\
    About2500\n2_x64\n2_x64.exe" name="TestEngine2" -each
    option.Hash=256 proto=uci tc=40/10 -pgnout checkTBout.pgn
     -concurrency 3 -repeat -rounds 2 -openings file=checkTB.
    pgn format=pgn order=random
```

cutechess_checkN2_slow.bat:

```
cutechess-cli -engine cmd="C:\Users\Chris\Desktop\
    SeniorProjectStuff\About2500\n2_x64\n2_x64.exe" name="
    TestEngine1" option.NalimovPath=G:\Nalimov option.
    NalimovCache=128 -engine cmd="C:\Users\Chris\Desktop\
    SeniorProjectStuff\About2500\n2_x64\n2_x64.exe" name="
    TestEngine2" -each option.Hash=256 proto=uci tc=40/10 -
    pgnout checkTBout.pgn -concurrency 3 -repeat -rounds 2 -
    openings file=checkTB.pgn format=pgn order=random
```

cutechess_checkSpike.bat:

```
cutechess-cli -engine cmd="C:\Users\Chris\Desktop\
    SeniorProjectStuff\About2500\n2_x64\n2_x64.exe" name="
    TestEngine1" option.NalimovPath=C:\Users\Chris\Desktop\
    SeniorProjectStuff\Nalimov option.NalimovCache=128 -
    engine cmd="C:\Users\Chris\Desktop\SeniorProjectStuff\
    About2500\n2_x64\n2_x64.exe" name="TestEngine2" -each
    option.Hash=256 proto=uci tc=40/10 -pgnout checkTBout.pgn
     -concurrency 3 -repeat -rounds 2 -openings file=checkTB.
    pgn format=pgn order=random
```

cutechess_checkSpike_slow.bat:

```
cutechess-cli -engine cmd="C:\Users\Chris\Desktop\
    SeniorProjectStuff\About3000\spike_14\Spike1.4.exe" name
    ="TestEngine1" option.NalimovPath=G:\Nalimov option.
    NalimovCache=128 -engine cmd="C:\Users\Chris\Desktop\
    SeniorProjectStuff\About3000\spike_14\Spike1.4.exe" name
    ="TestEngine2" -each option.Hash=256 proto=uci tc=40/10 -
    pgnout checkTBout.pgn -concurrency 3 -repeat -rounds 2 -
    openings file=checkTB.pgn format=pgn order=random
```

cutechess_checkStockfish.bat:

```
cutechess-cli -engine cmd="C:\Users\Chris\Desktop\
    SeniorProjectStuff\About3400\stockfish-9-win\Windows\
    stockfish_9_x64.exe" name="TestEngine1" option.SyzygyPath
    =C:\Users\Chris\Desktop\SeniorProjectStuff\syzygy -engine
     cmd="C:\Users\Chris\Desktop\SeniorProjectStuff\About3400
    \stockfish-9-win\Windows\stockfish_9_x64.exe" name="
    TestEngine2" -each option.Hash=256 proto=uci tc=40/10 -
    pgnout checkTBout.pgn -concurrency 3 -repeat -rounds 2 -
    openings file=checkTB.pgn format=pgn order=random
```

cutechess_checkStockfish_slow.bat:

```
cutechess-cli -engine cmd="C:\Users\Chris\Desktop\
    SeniorProjectStuff\About3400\stockfish-9-win\Windows\
    stockfish_9_x64.exe" name="TestEngine1" option.SyzygyPath
    =G:\syzygy -engine cmd="C:\Users\Chris\Desktop\
    SeniorProjectStuff\About3400\stockfish-9-win\Windows\
    stockfish_9_x64.exe" name="TestEngine2" -each option.Hash
    =256 proto=uci tc=40/10 -pgnout checkTBout.pgn -
    concurrency 3 -repeat -rounds 2 -openings file=checkTB.
    pgn format=pgn order=random
```

cutechess_Nalimov.bat:

```
cutechess-cli -engine cmd="C:\Users\Chris\Desktop\
    SeniorProjectStuff\About2500\n2_x64\n2_x64.exe" option.
    NalimovPath=C:\Users\Chris\Desktop\SeniorProjectStuff\
    Nalimov\3-4-5 option.NalimovCache=128 name="N2" -engine
    cmd="C:\Users\Chris\Desktop\SeniorProjectStuff\About2500\
    amyane\amyan.exe" name="Amyan" -each option.Hash=256
    proto=uci tc=40/10 -rounds 40 -pgnout N2_vs_Amyan.pgn -
    concurrency 3
```

This bat file was reused and modified for the following engines:

- N2, located at C:\Users\Chris\Desktop\SeniorProjectStuff\About2500\n2_x64\n2_x64.exe
- Muse, located at C:\Users\Chris\Desktop\SeniorProjectStuff\About2500\muse0953\muse64_095
- Aristarch, located at C:\Users\Chris\Desktop\SeniorProjectStuff\About2500\aristarch-engine\Aristarch.exe
- Amyan, located at C:\Users\Chris\Desktop\SeniorProjectStuff\About2500\amyane\amyan.exe

- Wasp, located at C:\Users\Chris\Desktop\SeniorProjectStuff\About3000\wasp_260\wasp260-x64.exe
- Spike, located at C:\Users\Chris\Desktop\SeniorProjectStuff\About3000\spike_14\Spike1.4.exe
- Spark, located at C:\Users\Chris\Desktop\SeniorProjectStuff\About3000\spark-1.0\spark-1.0-win64-mp.exe
- iCE, located at C:\Users\Chris\Desktop\SeniorProjectStuff\About3000\ice_full_v3_658\ice3-x64.exe

This bat file was reused and modified for the following tablebases:

- Normal Nalimov, located at C:\Users\Chris\Desktop\SeniorProjectStuff\Nalimov
- Trimmed Nalimov, located at C:\Users\Chris\Desktop\SeniorProjectStuff\Nalimov_trimmed
- Stripped Nalimov, located at C:\Users\Chris\Desktop\SeniorProjectStuff\Nalimov_stripped
- Slow Nalimov, located at G:\Nalimov

This bat file was reused and modified for the following output files:

- N2_SlowTB_vs_Amyan.pgn
- N2_SlowTB_vs_Aristarch.pgn
- N2_SlowTB_vs_Muse.pgn
- N2_vs_Amyan.pgn
- N2_vs_Aristarch.pgn
- N2_vs_Muse.pgn
- N2_vs_N2_NoTB.pgn
- N2_vs_N2_SlowTB.pgn
- N2_vs_N2_StrippedTB.pgn
- N2_vs_N2_TrimmedTB.pgn
- Long_N2_vs_N2_NoTB.pgn

- Spike_SlowTB_vs_iCE.pgn
- Spike_SlowTB_vs_Spark.pgn
- Spike_SlowTB_vs_Wasp.pgn
- Spike_vs_iCE.pgn
- Spike_vs_Spark.pgn
- Spike_vs_Wasp.pgn
- Spike_vs_Spike_NoTB.pgn
- Spike_vs_Spike_SlowTB.pgn
- Spike_vs_Spike_StrippedTB.pgn
- Spike_vs_Spike_TrimmedTB.pgn
- Long_Spike_vs_Spike_NoTB.pgn

cutechess_Syzygy.bat:

```
cutechess-cli -engine cmd="C:\Users\Chris\Desktop\
    SeniorProjectStuff\About3400\stockfish-9-win\Windows\
    stockfish_9_x64.exe" option.SyzygyPath=G:\syzygy name="
    Stockfish_SlowTB" -engine cmd="C:\Users\Chris\Desktop\
    SeniorProjectStuff\About3400\komodo-9_9dd577\Windows\
    komodo-9.02-64bit.exe" name="Komodo" -each option.Hash
    =256 proto=uci tc=40/10 -rounds 60 -pgnout
    Stockfish_SlowTB_vs_Komodo.pgn -concurrency 3
```

This bat file was reused and modified for the following engines:

- Houdini, located at C:\Users\Chris\Desktop\SeniorProjectStuff\About3400\Houdini_15a\Houdi
- Komodo, located at C:\Users\Chris\Desktop\SeniorProjectStuff\About3400\komodo-9_9dd577\Windows\komodo-9.02-64bit.exe
- Stockfish, located at C:\Users\Chris\Desktop\SeniorProjectStuff\About3400\stockfish-9-win\Windows\stockfish_9_x64.exe

This bat file was reused and modified for the following tablebases:

- Normal syzygy, located at C:\Users\Chris\Desktop\SeniorProjectStuff\syzygy
- Trimmed syzygy, located at C:\Users\Chris\Desktop\SeniorProjectStuff\syzygy_trimmed
- Stripped syzygy, located at C:\Users\Chris\Desktop\SeniorProjectStuff\syzygy_stripped
- Slow syzygy, located at G:\syzygy

This bat file was reused and modified for the following output files:

- Stockfish_SlowTB_vs_Houdini.pgn
- Stockfish_SlowTB_vs_Komodo.pgn
- Stockfish_vs_Houdini.pgn
- Stockfish_vs_Komodo.pgn
- Stockfish_vs_Stockfish_NoTB.pgn
- Stockfish_vs_Stockfish_SlowTB.pgn
- Stockfish_vs_Stockfish_StrippedTB.pgn
- Stockfish_vs_Stockfish_TrimmedTB.pgn
- Long_Stockfish_vs_Stockfish_NoTB.pgn