KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Engineering / Technology

First cycle, 15 credits

A Comparative Study of Parallel Search Techniques in a Chess Engine

Young Brothers Wait Concept or Lazy SMP?

ALEX SÄFSTRÖM
SIMON YLLMARK

Stockholm, Sweden, 2025

A Comparative Study of Parallel Search Techniques in a Chess Engine

Young Brothers Wait Concept or Lazy SMP?

ALEX SÄFSTRÖM

SIMON YLLMARK

Degree Programme in Computer Engineering
Bachelor's Programme in Information and Communication Technology
Date: January 27, 2025

Supervisor: Aws Al-Zarqawee
Examiner: Ki Won Sung
School of Electrical Engineering and Computer Science
Swedish title: En Jämförande Studie av Parallella Söktekniker i en Schackmotor
Swedish subtitle: Young Brothers Wait Concept eller Lazy SMP?

Abstract

An important aspect to consider when creating a chess engine is the choice of parallel search algorithm. Two of the most common approaches are Young Brothers Wait Concept (YBWC) and Lazy Symmetric Multiprocessing (SMP). During the research process, it became apparent that while quite a few papers have been written about Chess Engines (CEs) with one of the two algorithms, very little exists regarding comparing parallel search techniques of CEs head-to-head.

This project aims to determine how YBWC and Lazy SMP compare in terms of scalability, speedup, and playing strength. To accomplish this, the popular open source chess engine Crafty was modified to use Lazy SMP instead of YBWC. Both versions were then tasked with searching for the best move from 30 different positions 50 times each to determine the speedup from increasing thread-counts. They also played 30 matches against four engine opponents of different strengths to calculate the Elo rating of both versions of Crafty at different thread-counts.

The findings of this project indicate that neither algorithm consistently outperforms the other. While each algorithm excels in specific scenarios, their strengths and weaknesses depend on resource availability. These results suggest that YBWC performs better than Lazy SMP when utilising a large amount of computer resources. Conversely, Lazy SMP outperforms YBWC when operating with limited resources. YBWC's structured approach to organising resource utilisation gives it an advantage in resource-rich environments, but it becomes a drawback when resources are scarce. On the other hand, Lazy SMP's more chaotic, direct approach works well in low-resource situations, but becomes harder to manage as resource availability increases.

The lack of previous research could make the findings valuable for other chess engine developers when selecting chess algorithms, particularly when deciding between Lazy SMP and YBWC.

Keywords

Chess Engine, Young Brothers Wait Concept, Lazy SMP, Parallel Search, Scalability, Elo Rating

Sammanfattning

En viktig aspekt att ta hänsyn till när man skapar en schackmotor är valet av av parallellsökalgoritm. Två av de vanligaste metoderna är Young Brothers Wait Concept (YBWC) och Lazy Symmetric Multiprocessing (SMP). Under forskningsprocessen blev det uppenbart att trots att en del artiklar har skrivits om schackmotorer med en av de två algoritmerna, så finns det väldigt lite som direkt jämför parallella söktekniker för schackmotorer med varandra.

Detta projekt försöker avgöra hur YBWC och Lazy SMP står sig mot varandra gällande skalbarhet, hastighet och spelstyrka. För att åstadkomma detta användes den modifierades en populär schackmotor med öppen källkod vid namn Crafty för att använda Lazy SMP i stället för YBWC. Båda versionerna fick sedan söka efter det bästa draget från 30 olika positioner 50 gånger vardera för att avgöra hur mycket snabbare de blir med ökat antal trådar. De spelade även 30 matcher mot fyra schackmotorer av olika styrka för att beräkna Elo-värdet för de båda versionerna av Crafty vid olika antal trådar.

Resultaten av detta projekt visar att ingen av algoritmerna konsekvent överträffar den andra. Även om varje algoritm utmärker sig i specifika scenarier beror deras styrkor och svagheter på resurstillgången. Dessa resultat tyder på att YBWC presterar bättre än Lazy SMP när en stor mängd datorresurser utnyttjas. Omvänt presterar Lazy SMP bättre än YBWC när man arbetar med begränsade resurser. YBWC:s strukturerade sätt att organisera resursutnyttjandet ger den en fördel i resursrika miljöer, men blir en nackdel när resurserna är knappa. Lazy SMP:s mer kaotiska och direkta tillvägagångssätt fungerar å andra sidan bra i resurssvaga situationer, men blir svårare att hantera när resurstillgången ökar.

Avsaknaden av tidigare forskning kan göra resultaten värdefulla för andra schackmotorutvecklare när de väljer schackalgoritmer, särskilt när de väljer mellan Lazy SMP och YBWC.

Nyckelord

Schackmotor, Young Brothers Wait Concept, Lazy SMP, Parallelsökning, Skalbarhet, Elorating

Acknowledgements

We would like to thank our supervisor Aws Al-Zarqawee and our examiner Ki Won Sung for their guidance throughout the project.

Stockholm, January 2025
Alex Säfström Simon Yllmark

Figures

Tables

Listings

List of acronyms and abbreviations

CCRL	Computer Chess Rating Lists
CE	Chess Engine
CPU	Central Processing Unit
ER	Elo Rating
FIDE	The International Chess Federation
FOSS	Free and Open Source Software
GM	Grand Master
GUI	Graphical User Interface
Lazy SMP	Lazy Symmetric Multiprocessing
NN	Neural Network
PST	Piece-Square Table
SIMD	Single Instruction Multiple Data
SMT	Simultaneous Multithreading
UCI	Universal Chess Interface
YBWC	Young Brothers Wait Concept

Chapter 1

Introduction

Despite its age, chess continues to attract more players and is today more popular than ever [1]. Many players use computers to play against their opponents online, to practice against the computer itself, or to study and learn from previous games. When using a computer to play chess, a Chess Engine (CE) can be used to play against or to analyse the moves of a game, and is therefore an indispensable tool to these players. Historically, a large portion of the performance gains over time for CEs has come from the increased performance of the computer hardware they run on.

Due to the heat generated by increasing clock speeds of Central Processing Units (CPUs) and the difficulty of shrinking the size of transistors, much performance is today gained from having multiple cores in the same CPU. It is however not as easy to write a program that utilises multiple cores, and today CEs use a variety of techniques to achieve this. This study aims to modify an open-source, multi-threaded CE so that it can choose between two of the most popular techniques, Young Brothers Wait Concept (YBWC) and Lazy Symmetric Multiprocessing (Lazy SMP), to compare how the choice between the two affects the speedup, playing strength, and scalability of the engine.

1.1 Background

Chess can trace its history back to the game of Chaturanga, an Indian board game from at least 500 B.C. Chaturanga spread worldwide through travelling traders, becoming altered and adapted by locals as time went on. Among others, this led to the creation of Xiangqi in China, Shogi in Japan, and eventually early versions of chess in Europe. By the 15th century, the chess of the time could be considered relatively close to the modern game but would continue to have its rules standardised throughout Europe well into the 1800s [2].

During the latter half of the 19th century, the most prevalent theory transitioned from a very aggressive, attack-heavy style to one based on positional advantage. By the 1920s this would start to, in turn, be replaced by hypermodernism, in which control of the centre with pieces is favoured over direct occupation with pawns. There had been several world championships, but it was not until 1948 that they would be standardised by The International Chess Federation (FIDE) [3].

It was also around this time that interest in creating machines that could play chess would start to materialise, with Claude E. Shannon in 1950 publishing one of the first papers that described the challenges surrounding creating a machine that would be able to play "[...]a tolerably good game of chess[...]" [4].

Shannon explains that simple games can have an evaluation function that can calculate if any given position will lead to a win, loss, or draw and that a machine could be programmed to use this evaluation function with the positions from each possible move from the current position to always play perfectly. He gives Nim as an example of a game that has been solved like this and reasons that this cannot be done in practice with chess due to the humongous number of possible moves. Shannon calculates that with the average of around 30 legal moves from any position and the average game length of 40 moves before resignation, a computer would need to calculate 10^{120} positions and that this would take more than 10^{90} years even if one position could be calculated every picosecond (10^{-12} seconds).

He further points out that if one were to use a dictionary of all possible positions, that would require storing around 10^{43} unique positions. To solve this, he concludes that a chess program must use strategy to be able to play, suggesting the use of an approximate evaluation function that at least takes into account the material worth of pieces, their positioning, and their mobility to assign a score to any given position. The program should then generate a tree of all legal moves for a few moves ahead, evaluate the resulting positions at the bottom of the tree, and propagate the scores up the tree to the root by choosing the highest scores for white's turn and the lowest scores for black's turn, what would today be called a minimax search algorithm.

Around the same time, Alan Turing became the first person to publish a program that could play an entire game of chess, however, no computer at the time was strong enough to run it. Turing played a match against Alick Glennie in 1952 by calculating the algorithm's moves by hand but ended up losing after 29 moves [5].

The exponential increase in the processing power of computers as well as algorithmic improvements ensured that the playing strength of chess programs would continue to grow. In 1985, the reigning chess world champion Garry Kasparov was able to play against 32 of the strongest CEs in the world simultaneously and win against all of them [6, ~06:00]. Only 12 years later in 1997, would Kasparov be the first world champion to lose against a CE, IBM's Deep Blue, by $3\frac{1}{2} - 2\frac{1}{2}$ [7].

A CE can be used for other things than just recreational play. Many chess players use a CE as a tool for analysing their chess game to improve, which Garry Kasparov was one of the first Grand Masters (GMs) to do [8].

There are many CEs today such as Stockfish, Alpha-zero and many more. There are also many minor CEs developed by individuals. Developing a CE is not a new concept and throughout the years, many techniques have been developed that allow them to reach greater playing heights.

One of the most effective ways to improve the performance of a CE is the use of multithreading to execute its search in parallel. Two of the most common parallelisation techniques are YBWC and Lazy SMP, but not much work has been publicised about how the performance of the two compare. This thesis aims to create a CE that can operate using both strategies to quantify the strengths and weaknesses of both against each other.

1.2 Problem

This research project addresses the lack of scientific papers comparing YBWC and Lazy SMP head to head, even though many researchers have implemented the algorithms in their own CEs.

To mitigate the problem, a comparative study of the two parallelisation strategies was conducted, and an investigation was carried out to assess how well the two algorithms perform given the parameters of search speed, playing strength, and scalability. By conducting experiments, data was collected to compare the two algorithms in these aspects. Through the comparison of the performances of the two algorithms, it was anticipated that a conclusion could be drawn regarding whether one algorithm would be a better fit under certain circumstances.

1.2.1 Original problem and definition

The research aims to examine the operational properties of the two algorithms YBWC and Lazy SMP. The research focuses on how the two algorithms affect the CE's speedup and Elo Rating (ER). The research also aims to examine whether one algorithm scales better with multiple threads than the other.

1.2.2 Scientific and engineering issues

The main engineering experiences drawn from this project come from implementing the two parallel search algorithms. Such experiences include achieving efficient thread management, good load balancing, and managing and reducing synchronisation overhead.

More broadly for the developed program, further experiences also come from managing the implementation complexity and ensuring the robustness of the CE with thorough error handling and testing.

Scientifically, the project aims to deepen the knowledge of parallel algorithms. This knowledge was gained by getting insight into how the two parallelisation strategies explore the sizeable search space required by CEs to operate, and into how the two strategies scale with increasing thread-counts.

1.3 Purpose

This project aims to compare Lazy SMP and YBWC to determine which one is better in various aspects. It serves as a comparative study between the two algorithms. Due to the limited information available on their comparison, this research seeks to fill that gap. The hope is that the findings will be valuable to CE developers by providing empirical data on how well the two techniques perform under identical conditions and that it will contribute to ongoing investigations into efficient parallel computing strategies.

1.4 Goals

The main goal of this project is to make a comparative study of Lazy SMP and YBWC, this goal can then further be split into smaller sub-goals. Here is a list of sub-goals that should be completed:

1. Select an open-source CE to work with
2. Implement Lazy SMP and YBWC
3. Perform tests to measure the following parameters:
   (a) Speedup
   (b) Scalability
   (c) ER
4. Analyse the measured data and draw conclusions from the analysis

1.5 Research Methodology

For this project, quantitative research has been selected as the methodology. The philosophical assumption guiding this research is positivism, based on the belief that an objective description of the performance of the two parallel search techniques can be achieved through the creation and testing of hypotheses.

To accomplish this, the experimental research method will be utilised, as it allows for a fixed number of variables to be examined to see how changes affect outcomes. In this case, the main variables will be the parallel search technique employed and the number of threads allocated to the engine.

Given the positivistic philosophical assumption and the experimental research approach, a deductive research strategy is the most logical choice, as it involves confirming or refuting hypotheses and using the results to establish causality between the variables and their effects. The primary causality to be investigated is whether one parallel search algorithm offers a performance and/or scalability advantage over the other.

There are however also some qualitative aspects to the project. To begin, YBWC and Lazy SMP are not the only parallel search algorithms used by CEs, but they were deemed to be the two most popular alternatives as most multi-threaded engines found during preliminary research used one of the two techniques.

Another qualitative aspect is the literature study portion of the project (as can be seen in Chapter 3). Here, sources were chosen based on their relevancy to the parallelisation techniques and their findings.

Finally, the open-source CE that will be modified for this project was selected using qualitative requirements (which are listed in step 5 of Section 4.1).

1.5.1 Philosophical Assumption

It is believed that the choice of positivism as the philosophical assumption suits the research well. This is because it is based on testing hypotheses to either strengthen or weaken them through experimentation. It is worth mentioning that positivism has been recognised as a suitable method for testing the performances of information and communication technologies [9].

Positivism is characterised by the assumption that reality is objective and does not depend on either the observer or the instrument. This assumption is appropriate for research concerning information and communication technologies, where the data is considered valid in itself. In this context, the data is not merely an opinion that can be easily changed, but rather an observation of the truth.

Although realism was considered, it was concluded that conducting tests for the research would be essential, leading to the decision that positivism would be a more suitable approach. Realism mainly centres on understanding the data itself, including the conclusions that can be drawn from it and the insights that can be gained [9]. In contrast, this research focuses more on measuring performances with various variables.

1.5.2 Research Method

The experimental research method was chosen as it was believed to be best suited to the research. It was considered that the adjustment of variables would be central to the planned test experiments with Lazy SMP and YBWC. The most important variables would be the number of threads available and the algorithm employed.

Experimental studies focus on cause and effect; if one variable is changed while keeping the rest of the system intact, the effect can be observed. This method sets parameters for the investigation and examines how those changes affect the system [9].

The descriptive research method was rejected because the focus is not on asking, ”Why does the CE perform better when a variable is changed?” Instead, the interest lies in verifying or falsifying hypotheses, such as, ”Lazy SMP is more scalable than YBWC.” Investigating a deeper understanding of why the CE performs better when a variable is altered would be a subsequent step in the project.

The descriptive research method investigates what conclusions can be drawn from a set of data and what can be said about it. This method would be more suitable for research projects where more information about the actual data used would be of interest [9].

The applied research method was also rejected because the research does not aim to solve known technological problems for a CE, nor is it a continuation of someone's research that requires further examination of specific details.

The applied research method is appropriate when a researcher wishes to examine existing research and investigate certain details further [9].

1.5.3 Research Approach

A deductive research approach is deemed most suitable because most of the research will focus on proving or disproving hypotheses. The following is a list of hypotheses that will be examined:

- Lazy SMP will scale better than YBWC.
- Lazy SMP and YBWC will have similar ERs.

The deductive research approach primarily involves deducing what is true or false through tests with quantitative data [9].

The abductive research approach was rejected because the research does not focus on understanding the various reasons why the CE performs better or worse when different variables are adjusted. Instead, the interest lies in determining whether it performs better or worse compared to previous attempts. The abductive research approach combines elements of both inductive and deductive research, merging the verification or falsification of theories or hypotheses with the generation of new hypotheses while testing their validity [9].

1.6 Delimitation

1. The CE will not be written from scratch, but rather modified to support both Lazy SMP and YBWC. No other parallelisation technique will be considered.

2. This project is not concerned with achieving the utmost performance by deep-diving into every operational aspect of the CE and will as such not strive for perfection in every aspect. Decent performance using more basic techniques will be enough to compare the two parallel search algorithms as long as they are given the same prerequisites.

3. The use of different operating systems or hardware configurations will not be considered as a performance aspect in this research.

1.7 Benefits, Ethics, and Sustainability

The benefits gained from knowing how YBWC and Lazy SMP compare against each other will allow CE developers to make a more informed decision when choosing which parallelisation technique to use. This could lead to stronger and more efficient CEs that can achieve higher levels of play.

As CEs become more efficient, the energy consumption of the machines they run on to play at the same level of difficulty lessens. Efficient use of computing resources are important both for the performance of the program and the sustainability of using computers.

There is a potential dual-use concern with CEs. Although they are primarily used recreationally and for improving one's skill through play and analysis, they can also be used for cheating purposes. A stronger CE would likely be better at helping someone cheat in chess. This is not a great concern, as both YBWC and Lazy SMP are already publicly available, and robust cheating prevention and detection is in place in tournaments.

1.8 Structure of the thesis

Chapter 2 introduces the reader to the basics of chess, and to how a CE is structured.

Chapter 3 is the backbone of this report where other researchers' work in similar areas are examined and evaluated to give inspiration and improve the research conducted in this research project.

Chapter 4 presents the chosen method for conducting research in this project and discusses the validity of the method along with the reliability of the data measured.

Chapter 5 discusses the work that was performed, the experiments that were implemented in practice, and what equipment was used.

The results from the experiments and analysis of the data collected from them are presented in Chapter 6.

In Chapter 7, the results from the previous chapter are discussed.

Chapter 8 will contain a section for conclusions of the discussion in Chapter 7 and the results from Chapter 6, a section for limitations or shortcomings in this project, and a section that contains a discussion of what could have been improved in this research for future researchers to expand on the research written in this report.

Chapter 2

Background

Section 2.1 will introduce the reader to important techniques used by a CE. Section 2.2 will build upon this by introducing how a CE with these features can be parallelised with a focus on the two parallelisation techniques the report is concerned about comparing.

2.1 Features of a Chess Engine

A CE is program designed to calculate the best possible move for a given chessboard setup. It evaluates the board and assigns a numerical value ranging from negative to positive infinity. A positive score indicates an advantage for White, while a negative score favours Black.

What makes a CE remarkable is its ability to analyse many moves ahead, exploring countless possible outcomes to determine a winning strategy for either side.

A fully-featured CE can be quite complex with many moving parts. To make it more digestible, the main features are categorised into four main parts:

- Board Representation
  - This is required by the CE to know where all pieces are located and to move the pieces.
- Move Generation
  - This refers to how the CE creates every possible legal move for all pieces.

- Move Search
  - This specifes how the CE goes through all available moves to choose the most advantageous one.
- Evaluation
  - This is how the CE determines how good any given position is. A score is given to positions, which Move Search uses when picking what move to make.

Section 2.1.1 will describe how a chessboard can be represented in a CE, Section 2.1.2 will describe how moves are generated, Section 2.1.3 will describe how an engine searches for the optimal move to play, and Section 2.1.4 will describe how moves are scored to estimate how good they are.

2.1.1 Board Representation

2.1.1.1 Bitboards

The chessboards and pieces have been chosen to be represented using bitboards. A standard chessboard consists of 64 (8×8) squares, meaning that each square's occupancy can be represented by a 64-bit unsigned integer (a bitboard). It is necessary to differentiate between the different types of pieces and their respective players, meaning that at least 12 (6 pieces × 2 colours) bitboards are required. Three additional bitboards (representing all white pieces, all black pieces, and all pieces) can be derived from the mandatory 12 for convenience by using logical addition (bitwise OR) on the relevant bitboards [10, 11]. Figure 2.1 provides a visual representation of a bitboard.

Bitboards are convenient to use for a few reasons. The first reason is derived from the fact that most processors today are 64-bit, which means that it is very efficient to perform bitwise operations on them since the bitboards themselves are also 64-bits large [10, 11].

The second reason is that bitwise operations make it easy to isolate certain pieces without having to manually go through all the pieces and move the pieces, which will be useful when generating moves. Additionally, the small size of bitboards requires less memory to be allocated for them.

8	56	57	58	59	60	61	62	63
7	48	49	50	51	52	53	54	55
6	40	41	42	43	44	45	46	47
5	32	33	34	35	36	37	38	39
4	24	25	26	27	28	29	30	31
3	16	17	18	19	20	21	22	23
2	08	09	10	11	12	13	14	15
1	00	01	02	03	04	05	06	07
	A	B	C	D	E	F	G	H

(a) Bit Index

8	1	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
	A	B	C	D	E	F	G	H

(b) All Pieces

Figure 2.1: (a) shows how bitboards are indexed, with 00 being the least significant bit and 63 being the most significant bit. (b) shows the bitboard representing all pieces in their initial positions.

2.1.1.2 Alternatives

Since there is a bitboard for each type of piece it is considered a piece-centric solution to board representation. The alternative to this is board-centric approaches like two-dimensional arrays or Mailboxes. A solution using two-dimensional arrays would simply be eight by eight large integer array, where the integer encodes any piece sitting the square it represents. The problem with 2D arrays is that it takes multiple instructions to calculate the memory address by this type of indexing, which is an unnecessary performance penalty when generating moves [10].

Mailboxes is a common name for board-centric solutions that use a one-dimensional array or a list, the simplest of which works exactly like a 2D array except it is indexed $0 - 63$. Both this solution and 2D arrays are sensitive to wraps when generating moves, which means that an illegal move which would go out of bounds ends up on the opposite side of the board. These solutions must manually check every generated move to see if it contains an erroneous wrap. This problem also exists with bitboards, but can easily be solved, as will be discussed in Section 2.1.2. A mailbox solution to this would be to use a larger 120 (10×12) large array that has positions that would be outside the squares of a regular chessboard (two to the left and right, and one above and below). Moves that land in these inaccessible squares can then be ignored after reading the contents of that array position [10]. An example of how these are indexed is shown in Figure 2.2.

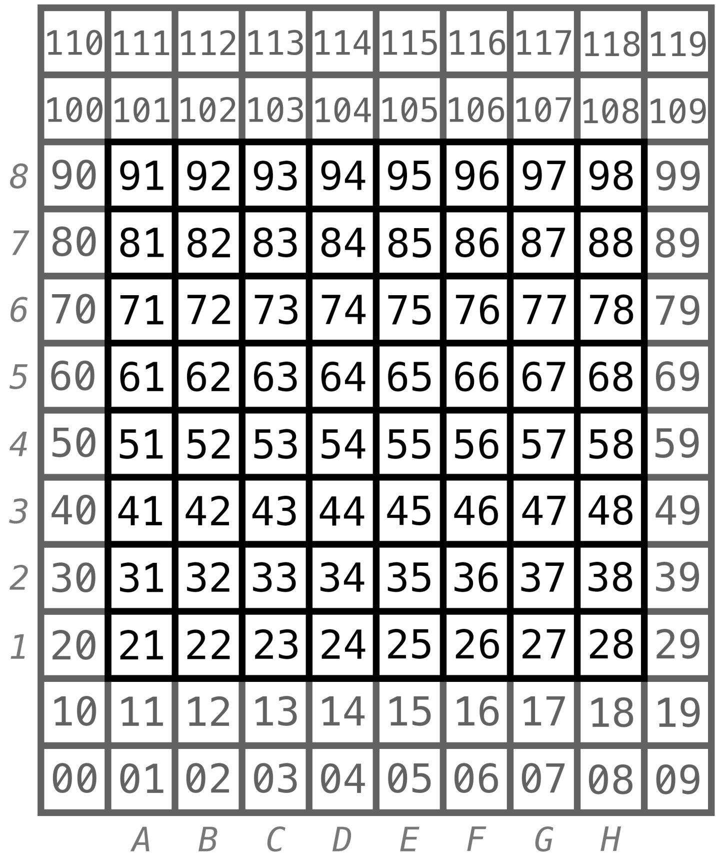


Figure 2.2: A visual representation of a 10×12 mailbox. Black indices correspond to regular squares, while grey are inaccessible..

Another mailbox solution is $0x88$ boards. These are a 128-long array where valid positions are represented by:

$$index \mod 16 < 8$$

and invalid positions are represented by:

$$index \mod 16 \geq 8$$

To simplify, the index can be written in hexadecimal and then the indices whose least significant nibble is less than eight are valid. Figure 2.3 shows how such a board is indexed. They are called $0x88$ boards because a position can be validated by using bitwise AND on the position index and the hexadecimal $0x88$, where valid results will be zero. $0x88$ ends up being faster than the other mailbox techniques because it can be determined whether a piece has landed on an invalid square in a single bitwise operation. Since half of the $0x88$ board is only required for validity checks, that half could also be used to store additional information about squares they shadow (i.e., additional information about $0x00$ can be stored in $0x08$, information about $0x32$ can be stored in $0x3a$, etc.). [10].

8	70	71	72	73	74	75	76	77	78	79	7a	7b	7c	7d	7e	7f
7	60	61	62	63	64	65	66	67	68	69	6a	6b	6c	6d	6e	6f
6	50	51	52	53	54	55	56	57	58	59	5a	5b	5c	5d	5e	5f
5	40	41	42	43	44	45	46	47	48	49	4a	4b	4c	4d	4e	4f
4	30	31	32	33	34	35	36	37	38	39	3a	3b	3c	3d	3e	3f
3	20	21	22	23	24	25	26	27	28	29	2a	2b	2c	2d	2e	2f
2	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
1	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
	A	B	C	D	E	F	G	H								

Figure 2.3: A visual representation of an $0x88$ board. Black indices correspond to regular squares, while grey are inaccessible.

Mailbox solutions typically store the type of piece as a number in the mailbox array. This means that you need to go through the array and compare the contents before you can start generating moves. To mitigate this, you can keep a list each for all white and black pieces where each entry describes the type of piece and its location on the board [10].

The main reasons to use bitboards are that they are more performant than mailboxes without needing additional techniques, they have simpler move generation, and they are used in most modern engines.

2.1.2 Move Generation

To generate moves with bitboards, you need to make a distinction between sliding pieces and non-sliding pieces. Non-sliding pieces are those that can move to a fixed number of possible positions; pawns, knights, and kings. Sliding pieces are those that can go indefinitely in a direction until reaching an obstacle; rooks, bishops, and queens. The fact that non-sliding pieces can not ”jump” past other pieces and keep going makes their move generation more complex, and will as such be described after non-sliding pieces.

2.1.2.1 Non-sliding pieces

For non-sliding pieces (pawns, knights, and kings) you can use special bitboards (sometimes called lookup tables) and bit-shifting to create new positions. Lookup tables are 64-bit unsigned integers that are used to isolate or remove the pieces on a rank or file from a bitboard by using bitwise AND on them [11]. Examples of lookup tables can be seen in Figure 2.4.

8	0	0	1	0	0	0	0	0
7	0	0	1	0	0	0	0	0
6	0	0	1	0	0	0	0	0
5	0	0	1	0	0	0	0	0
4	0	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0	0
2	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0
	A	B	C	D	E	F	G	H

(a) Isolation

8	1	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
	A	B	C	D	E	F	G	H

(b) Removal

Figure 2.4: (a) shows a lookup table that is used to keep only the pieces on the C-file. (b) shows a lookup table that keeps all pieces except those on the third rank.

First, two isolation lookup tables create a temporary bitboard containing only a single piece. Since the bitboard it came from is known, what type of piece it is and consequently how that piece moves is also known. Because the piece is isolated, it can be moved by simply shifting the entire bitboard by an amount corresponding to where it can go [11]. As an example, a piece will go one step forward with a ”«`8`” shift, a step right with ”»`1`”, and two steps backwards and one to the left with ”»17”.

To avoid wrapping (moving horizontally close to the edge and ending up on the other side), removal lookup tables will be used for moves that could lead to a wrap [11]. An example is the knight’s movement, which goes in an ”L” pattern. Two of the moves go right two steps and vertically one step, so when generating either of these moves removal lookup tables are always used for files G and H on the bitboard before shifting. This means that the bitboard will be all zeros if the knight was on the G or H files when generating these moves, but work normally when generating these moves from any other file. Another example is the king who can move one square in any direction. If it is standing on the A file, three moves would result in a wrap (up-left, left, and down-left), so when generating these moves a removal lookup table is always used for file A before shifting. Removal lookup tables are not needed for ranks, since when a bitboard is shifted so a piece goes beyond the first or eight ranks it will not wrap to the opposite rank, instead resulting in a bitboard of all zeros which is desired.

Figure 2.5 shows three instances of a knight trying to move forward two squares and right one square. It shows an error from not using lookup tables, how a lookup table corrects it, and how the lookup table does not affect the outcome of valid moves.

All the positions that the pieces could theoretically move to are given by this process, but in reality, fewer positions will almost always be available. This is because other pieces on these squares have not been taken into account so far. The same square as another piece of the same colour cannot be occupied by a piece, so these are removed by performing a bitwise AND on the generated positions and the inverse of all pieces of the same colour (own pieces are represented as zero, while everything else is represented as one).

2.1.2.2 Sliding pieces

The easiest way to handle sliding pieces (rook, bishops, and queens) is to do it the same way as non-sliding pieces and calculate one square at a time in each direction until the sliding piece can’t go any further [12].

This is quite inefficient though, as more possible moves can be generated from sliding pieces than non-sliding pieces. A pawn can move at most to four

$$\left(\begin{matrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{matrix}\right) \gg 17 = \begin{matrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\1&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{matrix}$$

$$\left(\begin{matrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{matrix} \;\&\; \begin{matrix}0&1&1&1&1&1&1&0\\0&1&1&1&1&1&1&0\\0&1&1&1&1&1&1&0\\0&1&1&1&1&1&1&0\\0&1&1&1&1&1&1&0\\0&1&1&1&1&1&1&0\\0&1&1&1&1&1&1&0\\0&1&1&1&1&1&1&0\end{matrix}\right) \gg 17 = \begin{matrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{matrix}$$

$$\left(\begin{matrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{matrix} \;\&\; \begin{matrix}0&1&1&1&1&1&1&0\\0&1&1&1&1&1&1&0\\0&1&1&1&1&1&1&0\\0&1&1&1&1&1&1&0\\0&1&1&1&1&1&1&0\\0&1&1&1&1&1&1&0\\0&1&1&1&1&1&1&0\\0&1&1&1&1&1&1&0\end{matrix}\right) \gg 17 = \begin{matrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{matrix}$$

Figure 2.5: Image showing how wrapping works with three examples of a knight trying to move forward two squares and right one square. The top-most has the knight on H8 without a lookup table, erroneously ending up on A6. Middle-most is on H8 *with* a lookup table, correctly not giving a valid position. Bottom-most is on G4 with a lookup table, correctly landing on H6.

different positions (counting diagonal attacks and en passant, which are very situational), knights can at most move to nine different positions (counting castling), and kings can at most move to ten different positions (counting long and short castling) while a bishop could theoretically move up to thirteen positions, a rook up to fourteen positions, and a queen up to twenty-seven positions. In addition, when generating moves like this with sliding pieces you must check if the piece can continue moving after every square while all non-sliding moves are generated at the same time with illegal moves being quickly removed after.

Instead of the simple solution described above, Magic Bitboards will be used. Rooks and bishops are allowed to move indefinitely in the orthogonal and diagonal directions respectively, until an edge of the chessboard or a blocking piece is reached. Since only pieces that are in the way can stop these pieces, 64 bitboards (one for each square) are created for rooks and bishops, isolating the squares of the pieces' attack patterns that can contain blocking pieces. A bitboard containing only the pieces that block the rook or bishop is obtained by performing a bitwise AND on the bitboard of all pieces. By removing the pieces that do not affect move generation, a reasonable precomputation of the bitboard containing possible moves for each combination of blocking pieces for every square can be achieved.

Magic Bitboards is a technique in which these solution bitboards are stored in a hashtable, using the bitboards that contain only the blocking pieces as indices to locate them [12]. Perfect hashing (hashing without collisions) is employed by Magic Bitboards, using a (magic) number that is ANDed with the bitboard of blocking pieces to create a unique index. This process is only necessary for rooks and bishops, as queens move like a combined rook and bishop. When looking up moves for queens, the process is thus performed twice: once to find rook moves from their position and once to find bishop moves.

An alternative to the use of Magic Bitboards is the performance of a PEXT lookup. This method is similar to Magic Bitboards in that a hash table is utilised to find all possible moves, but the Parallel bits EXTract CPU instruction is employed to generate the index. This approach is slightly faster than Magic Bitboards [12], but it was decided not to use it since it is not natively supported by all of the authors' computers.

2.1.2.3 Legality and Special Moves

The previous two subsections have outlined how the attack and movements are created in a pseudo-legal manner. This means that the moves and attacks are directed towards places that the pieces can reach, but do not consider whether the movement places the king in check. Any move that puts the king in check or maintains the king in check is not permitted, so any such moves must be removed.

The simplest solution for this is to retain all pseudo-legal moves and to detect and remove or invalidate them when generating moves for the opponent in the next ply. This approach has the advantage of not requiring the legality of every move to be calculated, as many will be pruned away by the search function, however, legality is still established for all un-pruned moves [13].

Pawns are special because they move and attack using different patterns. Therefore, any movement that lands on an occupied square regardless of colour must be removed and only attacks that land on squares occupied by the opposite colour are kept. They can additionally move two squares forward on their first turn, so this additional move must also generated for white pawns on the second rank and for black pawns on the seventh rank. Finally, they have a special attack, the *"en passant"*, which allows them to take a pawn by going diagonally behind it if the enemy pawn moved two squares during the previous turn. The easiest way to implement this is to have an en passant bitboard that marks the square that is passed when a pawn moves two squares and to include this bitboard along the opponents pieces when checking if pawn attacks are valid.

When a pawn reaches the final rank of the board (eight for white, one for black) it will be given a promotion. This can be solved with bitboards by replacing the single move that placed the pawn on the final rank with one of four new positions – one with a knight on that square, one with a rook, one with a bishop, and one with a queen.

Another special move is castling – where a king is moved two squares towards one of its rooks and the rook is moved to the square on the other side of the king. It can only be performed if the king and rook have not moved previously, there are no pieces between them, and none of the king, rook, or the squares between them are under attack. It is considered a non-sliding move despite involving a rook because there are only two outcomes (long castling with the A-file rook and short castling with the H-file rook). For pawns, it is only necessary to check their positions to see if they are allowed to move two squares because they cannot move backwards. Kings and rooks, however, can do this so a flag must be set that tells if they have moved yet or not. Isolating bitboards can be used to see if there are any pieces between the king and the rooks and stop when the bitboard of those isolated squares is not zero.

2.1.3 Move Search

When generating all moves from a position, the engine will go to the generated positions and start generating moves from this position. Doing this creates a tree of positions that will need to be searched for the move with the highest likelihood of leading to a victory. This section will describe methods used when searching through this game tree.

2.1.3.1 Negamax algorithm

Negamax is a simplified implementation of the minimax algorithm. The main idea behind minimax is that the terminal nodes of a game tree are evaluated and assigned scores. These scores are then propagated up the tree and the branch with the best score will be chosen as the best move. The scores are propagated up the tree by selecting the smallest score (minimising) and choosing the largest score (maximising) every other depth level. Here the white player will choose the largest score and the black player the smallest score. This works because the players want to play the score that is best for them and therefore worst for the opponent.

Minimax requires two almost identical separate functions for minimising and maximising and can therefore introduce bugs from accidentally using the wrong function or from needing to know if the player is black or white. Negamax simplifies this by only using one function and always maximising the engine regardless of colour. It works because a maximised score for black will be the same as the maximised score for white if the pieces are in the same positions but with their colours swapped. This means that the maximised scores from the next depth level can be negated and it will be equivalent to if they were minimised [14]. Figure 2.6 shows a small game tree where black minimises the score and white maximises the score.

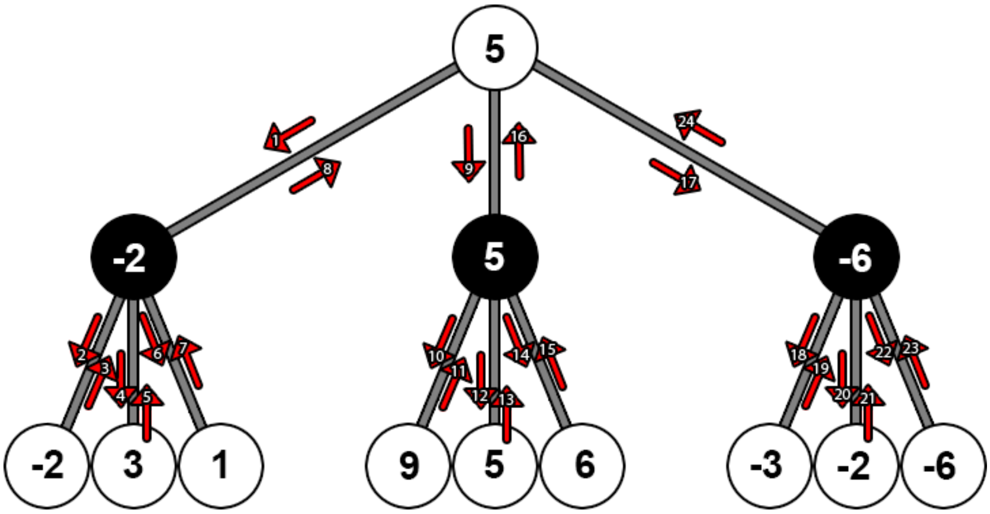


Figure 2.6: Visualisation of how minimax and negamax select the best move, created in Adobe Photoshop.

Pseudo-code describing the negamax logic can be seen in Listing 2.1.

Listing 2.1: Negamax Algorithm Pseudo-code

```
int negamax(int depth){
    if(depth == 0)
        return evaluate();

    int max_score = INT_MIN;
    generate_moves();
    while(move = get_next_move()){
        make_move(move);
        score = -negamax(depth - 1);
        unmake_move(move);
        if(score > max_score)
            max_score = score;
    }
    return max_score;
}
```

2.1.3.2 Alpha-beta pruning

Alpha-beta pruning is a minimax/negamax improvement that greatly reduces the number of moves needing to be evaluated by removing those that are guaranteed not to be played. It works by keeping track of two values, alpha which stores the minimum score the maximising player can get, and beta which stores the maximum score the minimising player can get.

If the maximising player encounters a score greater than the beta it will immediately stop searching since the minimising player will never choose that move. Conversely, if the minimising player finds a score lower than the alpha, the search will stop because the maximising player will not select it [15].

Figure 2.7 shows the same tree as in Figure 2.6 but now with alpha-beta pruning. The alpha cutoff when evaluating the second branch is -2 and there is no cutoff in this branch because no child node has a lower value than this. The alpha cutoff is then updated to 5 when the third branch is evaluated, and because the first node is less than this the remaining child nodes are cut off.

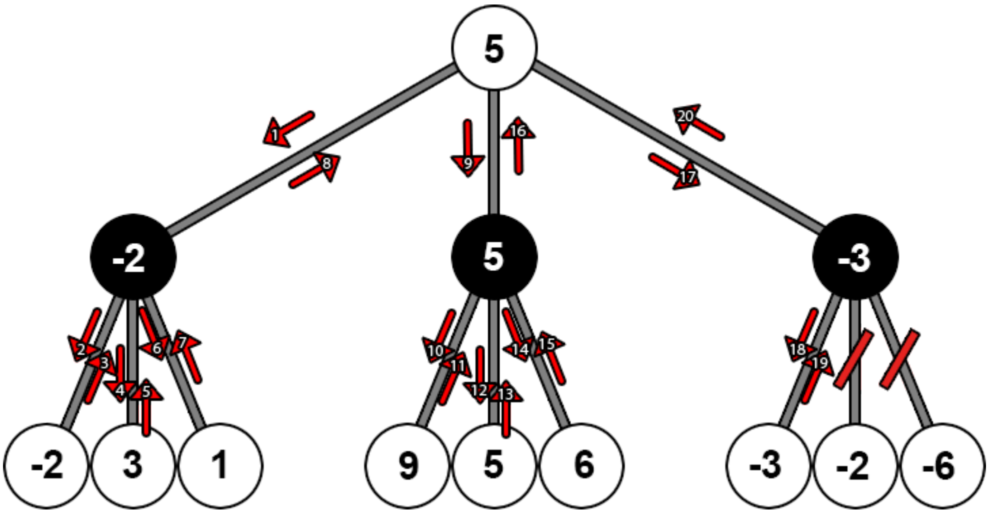


Figure 2.7: Visualisation of how alpha-beta pruning removes nodes from consideration in a minimax/negamax tree, created in Adobe Photoshop.

Alpha-beta pruning works best when the best moves are searched first. Michael Levin showed that the number of terminal nodes (T) generated to a certain depth (n) can be estimated with a fixed number of branches (b) from each node to be $T = b^n$ and that it can be reduced with alpha-beta pruning to at best $T = b^{\lceil n/2 \rceil} + b^{\lfloor n/2 \rfloor} - 1$ with perfect ordering [15].

Table 2.1 uses Levin's theorem to show how much the number of terminal nodes required to evaluate can be reduced assuming 30 branches from each node.

depth (n)	b^n	$b^{\lceil n/2 \rceil} + b^{\lfloor n/2 \rfloor} - 1$
0	1	1
1	30	30
2	900	59
3	27000	929
4	810000	1799
5	24300000	27899
6	729000000	53999
7	21870000000	836999
8	656100000000	1619999
9	19683000000000	25109999
10	590490000000000	48599999

Table 2.1: Number of terminal nodes to evaluate with and without alpha-beta pruning with perfect ordering and 30 branches per node (b) up to a depth of 10.

2.1.3.3 Transposition tables

When generating every possible move the same positions will inevitably repeat sooner or later. It would seem wasteful to recalculate how good the same position is repeatedly, so evaluated positions should be stored to quickly look up how good a position is when reencountered. The solution is to use a transpositions table – a hashtable that uses the positions as the key and stores the position score, the depth it was searched at, and the best following move [16].

With the score, the generation and evaluation of the children of the position can be skipped. With the depth, it can be determined whether a reevaluation of the position is necessary due to the presence of a sub-tree that is considered too small. With the best move, the children can be ordered during the reevaluation of the position when insufficient depth is present. This ensures that the previously best move is evaluated first, which often has a good chance of being identified as the best move again. Consequently, the rest of the sub-tree can be pruned.

2.1.4 Evaluation

The evaluation function analyses a board state to determine which player, White or Black has the advantage. This section will explore various technologies used to develop evaluation functions, explaining how they work and highlighting the tools and methods commonly employed in their creation.

2.1.4.1 Material

Assigning values to different chess pieces is crucial for helping a computer evaluate trades and assess the relative worth of each piece. This evaluation is a fundamental aspect of chess strategy, as the value of a piece is often relative to the position and the current state of the game [17].

To implement this, each chess piece is initially assigned a static value, which may adjust dynamically as the game progresses. A Piece-Square Table (PST) will be used to fine-tune the value of each piece based on its position on the board. Additionally, a float value will be introduced to represent the current game phase, which transitions linearly through three stages: Opening, Middlegame, and Endgame. This progression is unidirectional, moving from the Opening phase to the Endgame, and cannot be reversed.

2.1.4.2 Pawn Structure and centre control

In chess, controlling the center of the board with pawns is highly advantageous. Maintaining a connected pawn structure with minimal isolated pawns is also critical, as scattering pieces across the board without purpose can weaken a player's position [17].

Implementing this guideline requires careful programming. A system will be developed to evaluate the connectivity of pawns and assess their positioning using a PST. This table will classify squares as favourable, neutral, or unfavourable for pawn placement, guiding the program toward optimal pawn structures [18].

2.1.4.3 King safety

Evaluating king safety is essential for an accurate evaluation function, but it is challenging to implement. This difficulty arises from the fact that king safety is a relative concept, varying throughout the game rather than being a static condition [17].

Several factors can be considered to assess king safety more effectively:

1. Pawn Shield

   (a) Are there pawns positioned in front of the king, providing a protective barrier?

2. Pawn Storm

   (a) Are enemy pawns advancing to attack the pawns shielding the king?

3. Attack Units

   (a) How many enemy pieces are targeting the king or the pieces defending it?

2.1.4.4 Piece-Square Tables

A PST is a data structure that assigns values to each square on the chessboard for a specific piece. It evaluates whether a piece occupies a favourable, neutral, or unfavourable position. Additionally, it guides the CE by indicating that moving a piece to a particular square may confer a strategic advantage over the opponent. This mechanism facilitates the efficient development of minor pieces, a critical aspect of strong chess play.

The PST can be implemented as a byte array of size 64, where each index represents a square on the chessboard and contains a value corresponding to the strategic importance of that square for a given piece [18].

2.1.4.5 Machine learning

Developing an evaluation function based on Neural Network (NN) technology was initially considered but ultimately dismissed due to its complexity and the limited time available for implementation. Instead, a static evaluation function was chosen, as it presented a more straightforward approach.

If a NN-based evaluation function were to be implemented, several key challenges would need to be addressed. First, uncertainty exists regarding the sourcing of appropriate training data. Questions arise about how this data would be preprocessed and transformed into a suitable format for training. Additionally, considerations include the time required to train the model, the methods for testing its accuracy and reliability, and how to evaluate its impact on the overall performance of the CE [19, 20, 21, 22].

The CE Stockfish experienced a significant improvement in it's ER after transitioning from traditional static evaluation to a NN-based evaluation function known as an Efficiently Updatable Neural Network. This suggests that the most advanced evaluation functions today leverage neural network architectures for superior performance [13].

Since these kinds of NNs affect evaluation, they are a separate optimisation from parallelisation which affects move search.

2.2 Parallelisation

The core of this project handles parallelisation. Parallelisation is when a computer executes multiple tasks in parallel on separate processors that would otherwise be executed sequentially by a single processor. The ideal scenario for the CE would be Single Instruction Multiple Data (SIMD) scenarios where the data being processed does not have any dependencies of each other. Then multiple threads could be executed simultaneously which would increase the efficiency of the CE [23].

The two parallel search techniques handled in this study are the YBWC and Lazy SMP. The tree is being generated and evaluated as it is searched through, and the challenge comes from how multiple threads should work together to improve the performance without interfering with each other. Section 2.2.2 describes how YBWC works, and Section 2.2.3 how Lazy SMP works.

2.2.1 Cores and Threads

Two common terms used when discussing parallelisation are cores and threads. Although the two terms are often used interchangeably, there is a distinction to be made between the two. A core refers to a physical processing unit inside the CPU. Most CPUs made today contain more than one core, and are therefore known as multicore CPUs. Having multiple cores in the same CPU allows it to execute code in parallel by having different instructions being run at the same time on different cores [24].

A traditional computer program is executed completely sequentially, by running its code on a single core in the same order it is laid out. This is known as being single-threaded, as the entire process is run on only one software thread. However, some parts of the program may be completely independent of each other and could be run without affecting the outcome of the other. Because of this, many modern programs are multithreaded, meaning that a process executes parts of the program concurrently by dividing it into multiple threads [25, 26].

There can be thousands of threads wanting to execute at a given time on a modern computer and it is the responsibility of the operating system to switch threads in and out of context to allow them to execute. At any given time, the number of threads that can execute in parallel is defined by the number of cores of the CPU. Some CPUs support a feature called Simultaneous Multithreading (SMT), which allows a single physical core to execute multiple instructions at once with instruction pipelining. When active, this means that a single physical core appears as two or more logical cores to the operating system. SMT enables at least twice as many threads to be executed at the same time, at the cost of each thread running slightly slower than with SMT disabled [24].

When executing on multiple threads is mentioned throughout this thesis, it should be implicitly understood by the reader to mean that these threads are run their own respective physical or logical cores unless stated otherwise.

2.2.2 Young Brothers Wait Concept

The main idea behind YBWC comes directly from its name – other threads must not handle the younger brothers of the nodes in the search tree until their eldest brother has been evaluated. This means that when the search is started, the main thread will generate all moves from the root position, make the first of those moves, generate all moves for it, make the first of those moves, and repeat until maximum depth has been reached. The main thread is now at the leftmost terminal node, evaluates it, and unmakes that move. The first eldest brother has now been fully handled, so the other threads can handle the rest of the child nodes from that position. When all the children of this node have been evaluated, the best score from them will be assigned to this node and it will be finished, meaning that the younger brothers of it can be handled. This process is then repeated until the entire tree has been evaluated [27].

There are however more things of note than this. To begin, a thread gets to work on a sub-tree by asking a different thread if it has work it can give, and the asked thread either responds that it has nothing to share or with the lowest (closest to max depth) sub-tree it has available. The use of alpha-beta pruning would also mean that the amount of work a thread must do on a sub-tree is reduced or cut off entirely by the master thread that gave it that work, so the threads must be able to communicate about this for time not to be wasted on useless computations [27, 28].

Figure 2.8 shows a search tree in which the nodes circled in blue can be searched in parallel after their respective oldest brother nodes have been evaluated.

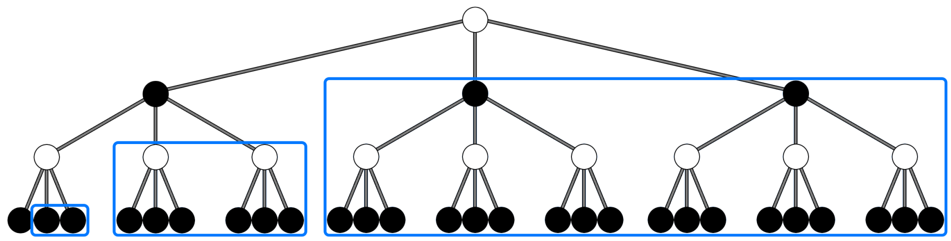


Figure 2.8: A search tree where the nodes circled in blue can be evaluated in parallel according to YBWC.

2.2.3 Lazy SMP

Lazy SMP is a relatively new parallelisation algorithm first described in 2013 by Daniel Homan on the Computer Chess Club Forum, inspired by a previous forum discussion about "lazy" ways to implement symmetric multiprocessing, and it is stated that it is quite simple to implement [29]. This algorithm is set apart from other parallelisation algorithms because the communication between the threads is minimal and reduced to only the transposition table (described in Section 2.1.3.3).

The idea behind this algorithm is to simply start the different threads simultaneously, offset from each other with a different root move ordering and/or different depths. The different threads will update the transposition table as they conduct their search, so when a thread encounters a position that has already been evaluated by another thread it will simply retrieve the score from the transposition table. The threads would likely try to evaluate the same positions at the same time before they were finished if all threads start with the same move ordering, so their initial positions are offset to reduce this risk. For the use of a shared transposition table to be efficient, it needs to be made lock-less to avoid the synchronisation overhead [23].

A problem that occurs with this parallelisation strategy is for the threads to know if they are evaluating a cutoff node or which of its children are cutoff nodes. Different kinds of technologies can be implemented to mitigate the reoccurring problem but it remains a flaw in the algorithm. [23]

Chapter 3

Literature Study

This chapter will describe some important reports that were read in preparation for this project. Section 3.1 will describe the results of some reports that studied YBWC, Section 3.2 will detail Lazy SMP reports, and Section 3.3 will depict papers on ER.

3.1 Young Brothers Wait Concept

3.1.1 Parallel Chess Searching and Bitboards

In the science paper "Parallel Chess Searching and Bitboards", the author David Ravn Rasmussen was experimenting with a CE to get a better understanding of how well YBWC works with multiple processors. The author found that YBWC is not very scalable, the speedup gained when adding many processors was not ideal. One thing that the author mentions is that the memory bandwidth in the experiments was not ideal. One could argue if the memory bandwidth was more ideal then the amount of data transferred during operation would have had a lesser impact on the measured speedup [30].

The results from their experiments were:
"Maximum speedup was 9.2 on 22 processors in the main experiment, 8.3 on 16 processors in the preliminary experiment and 9.4 on 12 processors in the truly dedicated 12 processors experiment".

The author argues that the main reason why YBWC is not very scalable is because there are too many nodes containing a cutoff node that are searched in parallel. The author also states that the size of the search tree grows linearly, with 10 processors the tree size doubles as compared to 1 and triple with 20 processors.

The author further notes that the speedup gained when adding more processors is sub-linear, meaning that the rate at which the tree grows is greater than the gained speedup. However, the least amount of time used is at 17 processors. The author then thought that the reason for this is that there are too many nodes containing a cutoff node that are searched in parallel.

3.1.2 Two-level Task Scheduling for Parallel Game Tree Search Based on Necessity

In their conference paper "Two-level Task Scheduling for Parallel Game Tree Search Based on Necessity," Akira Ura et al. aimed to improve speedup by minimising unnecessary task execution. They achieved this by classifying tasks into two priority classes: a lower class for tasks likely to be pruned by alpha-beta pruning, and a higher class for tasks less likely to be pruned.

While this approach resulted in improved speedup, the authors suggest that expanding the classification to three levels could yield even better results. However, they also caution that introducing a third class might increase inter-process communication, potentially diminishing the speedup gains [31].

3.1.3 GridChess: Combining Optimistic Pondering with the Young Brothers Wait Concept

In his paper "GridChess: Combining Optimistic Pondering with the Young Brothers Wait Concept", Kai Himstedt describes his implementation of a distributed YBWC search as well as optimistic pondering and how these affect the engine's ER [32].

The engine is based on Toga by Tomas Gaksch, with the YBWC implementation called Cluster Toga and the whole system using optimistic pondering being GridChess. Among other things, Himstedt measured how Cluster Toga running on different numbers of cores and clusters performed (without optimistic pondering) against Gaksch's Toga running on a single core after 70 games each with 20-minute timers and 5-second increments.

The results the author presented show that Cluster Toga running on a single cluster with a single core won against Gaksch's Toga 48.57% of the time, with Cluster Toga's Elo difference being -10 (95% confidence interval: [-67, +47]). With two cores on a single cluster, it won 55.00% of the time and has an Elo difference of +36 (interval: [-20, +92]). With four cores on a single cluster, it won 66.43% of the time with the Elo difference of +120 (interval: [+63, +120]).

This trend continues when using multiple clusters up until four clusters with four cores each, but further increasing to eight clusters with four cores each has the same win percentage of 72.86% and Elo difference of +172 (but with a slightly tighter interval: [+119, +232] vs [+115, +238]).

The author admits that the width of the confidence intervals makes it difficult to determine the actual Elo difference between the engines, likely due to the relatively low number of matches played, but points out that Cluster Toga consistently has a positive Elo difference with at least four cores and that this difference increases with more cores.

The author also explains that the number of games played and the number of opponents was limited by time constraints and the amount of access allowed to the computer cluster Cluster Toga ran on. He further explains that Gaksch's Toga was run on a separate computer that although uses a different CPU should have the same single-core performance as the ones Cluster Toga ran on, that the YBWC implementation in Cluster Toga has some slight inherent overheads, and that Gaksch's Toga was compiled with higher optimisation, but does not explicitly state that these are reasons to why Cluster Toga performed worse than Gaksch's Toga when single-threaded.

While this report does not present a time-based speedup from running on multiple cores (or clusters), it does show that the speedup achieved leads to better play. Assuming that the performance from running on multiple clusters is comparable to the performance from running on the same amount of cores locally, it would seem that scales well up to 16 cores with 32 cores resulting in a minimal difference.

3.1.4 A Fully Distributed Chess Program

In their paper "A Fully Distributed Chess Program", Rainer Feldmann et al. discuss how they parallelised a CE (ZUGZWANG) with YBWC and alpha-beta improvements [28]. The alpha-beta improvements they implemented include transposition tables (global, local, and distributed), local killer lists, local history tables, and iterative deepening.

In their solution, each processor will divide the work they are given into subproblems, and the processors not currently working will randomly ask another processor for a subproblem to work on. When a processor receives a subproblem, it in turn can also be subdivided and sent to processors asking for work. When a subproblem has been solved, the solution is sent back to the processor that initially gave away the subproblem.

The processors communicate with each other by sending a few kinds of messages to update the alpha-beta search windows. When the search window for a node changes, it will inform the processors with subproblems for this node of this with the "NEWBOUND" message. If the change of the search window leads to nodes being cut off immediately, a "CUTOFF" message will be sent to the affected processors instead so they stop immediately.

The authors use the 24 positions from the Bratko-Kompec test to measure the performance of ZUGZWANG [33]. Feldmann et al. present that the speedup from having 64 processors search to a depth of seven plies (a ply is one side's turn) from these positions is on average 34.77. The authors explain that this average is made higher by one of the positions that on its own had a speedup of 63.57. To account for this outlier position, they also measured this position to a depth of six plies which resulted in a speedup of 41.41 with the new average for all being 27.99.

It should be noted that Feldmann et al. did more measurements than just those discussed here, but a copy of this source that included the entire results chapter could not be found. Nonetheless, the results mentioned in the abstract and mainly discussed in the conclusion chapter were included in their entirety.

3.1.5 Summary

Table 3.1 summarises the pros and cons of the sources.

Author	Pros	Cons
David Rasmussen	YBWC speedup measurements on different cores, source code available of CE	Not ideal experiments and Could have tested other parallelisation strategies
Akira Ura	Implementation of YBWC was unique and explains how the order of executed tasks can affect results due to pruning	Done further testing with 3 priority classes and comparative testing with other strategies based on YBWC
Himstedt	Good Elo and win rate increase from multiple cores	Speedup not measured
Feldmann et al.	Good speedup for a high number of cores	A low number of tested positions makes the average results vulnerable to outliers

Table 3.1: A brief summary of the pros and cons of the YBWC sources in the literature study.

3.2 Lazy SMP

3.2.1 Piece By Piece: Building a Strong Chess Engine

In their paper "Piece By Piece: Building a Strong Chess Engine", Sander Vrzina studied how different techniques can be used to develop a competent CE, ending up with an engine playing at around 2400 Elo [13].

One of the techniques implemented was Lazy SMP for parallel search. Vrzina found that Lazy SMP was simple to implement, and measured a speedup of 40% when running on four cores. Unfortunately, Vrzina experienced some instability in Lazy SMP that led to the engine making unexpected blunders which they attribute to an unidentified error in either the alpha-beta pruning logic or in the transposition table.

Vrzina had initially tried implementing a different parallel search technique but was not satisfied with the speedup and so chose to use Lazy SMP. They expect an implementation without the error in their engine would eliminate the blunder moves and that greater speedups are possible. The author also ranks which techniques they thought would be most important to focus on if they were to continue work on the engine, and listed Lazy SMP as the top priority due to the potential speedup it offers.

3.2.2 A Complete Chess Engine Parallelized Using Lazy SMP

In his master thesis "A Complete Chess Engine Parallelized Using Lazy SMP", Emil Fredrik Østensen describes the techniques he used when creating a CE, focusing on using Lazy SMP to parallelise it [23]. Østensen describes how he avoids synchronisation overhead by using a lockless hashtable as the transposition table by including a checksum calculated by inserting thread in the entries.

Østensen managed to create an engine with an estimated Elo of 2238 and measured a speedup at depth 9 of 1.14 with two threads and 2.01 with four threads, the average number of nodes searched for the same number of positions being 78% more with two threads and 101% more with four threads, and the average number of nodes searched per second being 202% as many with two threads and 405% as many with four threads.

Østensen believes that the best way to improve upon Lazy SMP would be to reduce its randomness, and suggests two ways to potentially accomplish this. The first way would be to occasionally check the transposition table for moves in its history to avoid redundant searches. The second way would be to make the algorithm determine if it is currently in a CUT node or an ALL node before deciding the child node to explore first.

3.2.3 Evaluating Heuristic and Algorithmic Improvements for Alpha-Beta Search in a Chess Engine

In his bachelor's thesis "Evaluating Heuristic and Algorithmic Improvements for Alpha-Beta Search in a Chess Engine", Henrik Brange describes how he created a CE utilising MVV-LVA move ordering, a transposition table, iterative deepening, and Lazy SMP parallel search with measurements of how these techniques affect the execution time of the engine [34].

The Lazy SMP implementation was tested on four cores against a single-threaded version of the same engine, and when searching to depth 6 for 600 board positions the average execution time of Lazy SMP was 33.4% lower than the single-threaded version (a speedup of 1.51). Brange also measured the memory usage during iterative deepening and found that Lazy SMP required slightly over 20% more memory for the first two iterations, but only about 1% more for the third iteration.

Furthermore, the number of hits to the transposition table was measured for each depth, with Lazy SMP being at worst having 2.8 as many hits and at best 25 times as many. As the engine was made in Java, the author also recorded how often and how much time was spent by the JVM garbage collector and found that the Lazy SMP version spent 24.5% of its execution time on average on garbage collection as opposed to just 7.8% by the single-threaded equivalent.

The author lists three factors that could have affected the paper's validity. The first one is that the board positions chosen, although taken from real games, may not be representative of real play since most of them depict middle-game, but defends this choice because the middle game is the most complex and where the search tree's branching factor is the greatest.

The second is the dynamic optimisation made by the JVM which can make the performance (especially in early execution) vary, but this was counteracted by running the search 60 times without the transposition table before starting recording time.

The final factor is the fact that all implemented techniques are connected and affect each other's execution, which means that implementing different techniques can potentially have a great impact on the efficiency of others.

Finally, the author concludes that care should be taken when implementing Lazy SMP in Java to minimise the garbage created to reduce the amount of time taken up by garbage collection.

3.2.4 Branch and Bound Algorithm for Parallel Many-Core Architecture

In their paper “Branch and Bound Algorithm for Parallel Many-Core Architecture”, Kazuki Hazama and Hiroyuki Ebara discuss their solution to the travelling salesman problem using Lazy SMP [35].

The authors developed a Lazy SMP implementation in C++ using OpenMP and conducted experiments on five sets of 700 to 2000 cities on 8, 16, 32, 64, and 128 cores. The results show that after five runs the average time to search for the shortest path decreased with each increase in core count until 32 or 64 cores depending on the set of cities.

The authors explain that a reason for this is that the shared hash table they used in the solution only allows one thread to write at a time, which leads to the overhead from waiting for write permissions eventually becoming larger than the time saved from the increased core count. The largest average time decrease they found was for the “pr1002” data set when going from eight to sixteen cores, where eight cores on average finished in 4735.27 seconds and sixteen cores in 68.46 seconds – a speedup of 69.17.

It should be noted that the hash tables used as transposition tables in chess are typically lockless, allowing multiple simultaneous writes, as was previously discussed by Østensen (Section 3.2.2).

3.2.5 Summary

Table 3.2 summarises the pros and cons of the sources.

Author	Pros	Cons
Vrzina	Easy to implement, decent speedup	Blunders from instability, dependent on transposition table implementation
Østensen	Good speedup, minimal communication	Synchronisation overhead, large search overhead
Brange	Good speedup, high transposition table hit rate	Higher memory usage, much time spent on garbage collection
Hazama and Ebara	Scales quite high	Overhead from exclusive writes to hash table

Table 3.2: A brief summary of the pros and cons of the Lazy SMP sources in the literature study.

3.3 Elo

3.3.1 Chess Results Analysis Using Elo Measure with Machine Learning

In this study, Fadi Thabtah et al. tried to examine how good ERs are for measuring a player's strength. They begin by describing Elo as a measured player strength based on their past performance, if you win a lot of matches then your Elo increases.

The researchers found that ERs are surprisingly accurate for estimating how likely it is that a player would win or lose, a player with a higher ER is more likely to win against a player with a lesser. They found that the ER is not very accurate for measuring a player's strength but rather which player is more likely to win [36].

3.3.2 Chess Game Result Prediction System

In this science paper, Zheyuan Fan et al. try to estimate how good the ERs of chess players are at predicting the outcomes of matches. The prediction is more accurate if the ER differs more from player to player, if the difference is small then the outcome will be more unpredictable.

Though the source has not compared its data to other data from other sources to check the accuracy of the results, considering empirical evidence the data extracted in the source should be valid. Bear in mind that the source compares estimated outcomes with actual outcomes to estimate prediction accuracy.

"Chess itself is a rather unpredictable game, especially if two players are close in rating performance and there are tons of games where lower-rated players upset higher-rated players. Therefore, a success rate of 55.64 % given the three possible game outcomes is not a bad result." [37]

3.3.3 Summary

Table 3.3 summarises the pros and cons of the sources.

Author	Pros	Cons
Fadi Thabtah	ER explained Elo accuracy explained	goes further in detail about what Elo Rating is
Zhenyuan Fen	Elo unpredictability when small differences between players	Results need more comparison, give examples

Table 3.3: A brief summary of the pros and cons of the Elo sources in the literature study.

3.4 Speedup

In both the Lazy SMP (Section 3.2) and YBWC (Section 3.1) literature studies, it is shown that both algorithms exhibit increased speedup as more threads are added. This project adopts a similar methodology for measuring speedup, with a key modification: instead of relying solely on average values, multiple speedup measurements are taken to calculate both the median and the standard deviation from the median.

This approach provides a more robust evaluation of each algorithm's performance across multiple iterations, helping to identify which algorithm achieves higher and more consistent speedup. While previous studies primarily focused on calculating speedup as an average time value across iterations, this project incorporates those values into a broader statistical analysis by including standard deviation as a measure of variability.

3.5 Engines

This section contains a list of CEs that are Free and Open Source Software (FOSS). The CEs will be used in this project for measuring the ER for the handpicked CE. The only requirements for the opponents were that they are free to use, that they have a pre-compiled binary available for Linux, and that they have an ER in a similar range to that of the handpicked engine to avoid one dominating the other. All ER measurements presented in this list were taken from 40/15 matches (time control beginning at fifteen minutes where an additional fifteen minutes is awarded after every forty moves) performed by the Computer Chess Rating Lists (CCRL), one of the most trusted computer chess bodies.

1. Crafty

   - Crafty is a CE with an ER of 2953, according to the CCRL. Crafty was made primarily by Robert Hyatt, which began in the 1990s, and the final version was released in 2016.
   - This is the handpicked CE for this research project that will be modified and used in experiments. Crafty was picked because it is written in C which is a programming language where execution can be described in more detail than other programming languages.
   - There will be two versions of Crafty, the original version that implements YBWC and the other version that implements Lazy SMP. Both versions will initially have the same ER of 2953 because ER is an estimation of a player's strength and both engines will compete against other engines with known ER to refine the ER [38].

2. Clovis III

    - Clovis III is a CE with an ER of 2864, according to the CCRL. Clovis is being developed by Jonathan McDermid, with version III released on March 1st 2023.
    - It is a single-threaded Universal Chess Interface (UCI) engine written in C++ using many techniques, including bitboards, negamax, transposition tables, and PSTs.

3. Atlas 3.91

    - Atlas 3.91 is a CE with an ER of 2916, according to the CCRL. The developer of the engine, Andrés Manzanares Campillo, describes Atlas as a hobby project which is free to use according to CCRL.
    - The latest version of the engine is Atlas 4.35 but the older version have an ER closer to Crafty. Atlas 3.91 include UCI protocol. The developer included contact information on the website. The first version of Atlas was released in April 2004, and version 3.91 was released in January 2018.

4. Amoeba 3.4

    - Amoeba is a CE with an ER of 3043, according to the CCRL. It was developed by Richard Delorme, with the latest version of 3.4 released on December 16th 2021.
    - Amoeba is a multi-threaded UCI engine written in the D programming language.

5. Chess22k 1.14

    - Chess22k is a CE with an ER of 3117, according to the CCRL. It was developed by Sander Maassen vd Brink, with the latest version of 1.14 released on April 26th 2020.
    - Chess22k is a multithreaded UCI engine written in Java.

Since the main purpose of this report is to compare parallelisation techniques, it may be interesting to know what the opponents have implemented. However, this is not relevant to the tests because they will all run on a single thread. As such, only their ERs are required.

p

Chapter 4

Method

4.1 Research Process

Step 1: Preliminary Research

1. The research begins with searching for research papers on CEs from databases, for example, the KTH Library's database search engine and Google Scholar to get a baseline knowledge of the general architecture of CEs, and how the different parts affect it.
2. Zotero was used to organise and collect sources. This makes things easier later on to create a reference table for the sources used for this thesis report.

Step 2: Formulate a research question

1. Before the preliminary research had started, the intention was to create a CE and document ways in which its performance could be improved through known techniques. Through the preliminary research, it was discovered that similar work had been conducted many times, leading to the conclusion that improvements in this area would be challenging with the current research. The focus was shifted to a comparative study of two parallelisation strategies, YBWC and Lazy SMP. It was believed that transforming the research into a comparative study would contribute more to the field of Chess research, given that not many scientific papers compare the two algorithms.
2. Then, the question of which parameters should be used to measure the performance of the algorithms for comparison arose. Interest

was directed towards whether one parallel search algorithm would be faster, whether one would scale better to a higher thread-count, and whether one would result in stronger game-play.

Step 3: Literature Review

1. When sufficient knowledge had been gained from the preliminary research, a literature review was initiated on an area that was found to be interesting and under-explored: YBWC compared to Lazy SMP.
2. A comparison of both parallel search techniques in terms of efficiency and game performance was desired, so a literature review on ER was also initiated to gain a better understanding of how to quantify win probability.

Step 4: Research planning

1. Three aspects are to be measured: speedup, ER, and scalability.
2. Speedup shall be based on time-to-depth measurements from different thread-counts. Time-to-depth is a measurement utilised in many papers encountered that records the time taken for the engine to search a specified number of moves in advance from a given starting position, or in other words, the time required to reach a depth level in the search tree. These measurements will be taken from a large number of middle-game positions to gain a broad sample size while avoiding the early book moves that are typically handled by the Graphical User Interface (GUI).
3. ER will be estimated by allowing the CE to play a substantial number of matches against other CEs with a known ER using an automated matchmaking program. An ER for the CE will then be estimated based on statistics from its performance in previous matches and an example of an ER formula. The formulae for calculating Elo (4.1 and 4.2) [39, 38] require an initial rating to function. The initial ratings are taken from the CCRL, which the formulae 4.1 and 4.2 will then transform into a more refined rating.

$$ExpectedScore_A = 1/(1 + 10^{(Elo_B - Elo_A)/400}) \tag{4.1}$$

$$NewElo_A = Elo_A + K(Score_A - ExpectedScore_A) \tag{4.2}$$

4. Two types of scalability will be calculated; how speedup scales and how ER scales against increased thread-counts. Neither calculation requires bespoke measurements, instead using the measured data from the previous two items.
5. It has been concluded that the aspects that need to be documented for this research are the outcomes of the CE battles against other CEs, the ER of those opponents, and the time-to-depth recordings. These shall all be recorded for each combination of parallel search algorithm and thread-count.

Step 5: Implement the parallel search algorithms

1. The first step in this process is to select an open source CE to implement the algorithms in. A few criteria were put in place before one was selected.
   - Must be FOSS
   - Written in a language understood by the authors.
   - Preferably already having YBWC implemented, as Lazy SMP is broadly considered easier to implement.
   - Must already support transposition tables, preferably lock-less.
   - Should ideally use traditional evaluation (as opposed to neural networks), as the authors are more familiar with it.
   - Should ideally be written with readability in mind over performance, to ease comprehension of its inner workings and enable modification.
2. With this in mind, Crafty by Robert Hyatt was chosen, it was the only CE found that managed to meet all requirements.
3. After the CE has been selected, a version implementing Lazy SMP will be developed. In addition, tests will be developed to measure execution time and to automate matches against the opponent CEs.

Step 6: Execute experiments

1. What data to be measured is described in Section 4.3.
2. How the data shall be measured is described in Section 4.4.

Step 7: Analyse results

Step 8: Discuss results

4.2 Research Paradigm

As discussed in Section 1.5, a decision has been made to perform quantitative, positivistic research using an experimental research method and a deductive approach.

An experimental research strategy will be employed because the project aims to verify or falsify hypotheses. Experimental research is about changing dependent or independent variables to understand better how they would affect the outcome. This method helps the user to get a better understanding of the correlation between variables and the outcome.

The Ex Post Facto research strategy was considered but ultimately dismissed because it is best used when the researchers for some reason cannot manipulate the variables affecting the outcome, usually due to ethical concerns or when the requirements for reproducing some research are out of reach. It has been determined that neither of these poses a hindrance and that the data that can be gathered using the experimental strategy will be more aligned with the intended measurements.

As the experimental research strategy has been chosen exclusively, the use of experiments to collect data is the only natural choice. Experiments have been selected as the data collection method, as this aligns well with the experimental research strategy. While other options were available according to source [9], experiments appear to be the most valid choice, whereas the alternatives seemed to fall outside the scope of the research.

The data analysis will consist of both computational mathematics and statistics. Statistics will be used to summarise the collected data, identify patterns and anomalies, and determine probability distributions. Computational mathematics will be employed to analyse the efficiency of the algorithms, including time and space complexity, as well as for Elo calculation.

For quality assurance, the reliability, replicability, and validity of the method will be assessed. It is important to understand the extent to which the collected data deviates from run to run, the degree to which the method can be reused to obtain the same or similar results, and the level of trust that can be placed in the correctness of the measurements. These aspects are vital, as they directly affect the scientific credibility and impact of the findings. Furthermore, ethical considerations must be taken into account when collecting, analysing, and reporting the data, as well as regarding the use of open-source solutions as part of the engine.

4.3 Data Collection

There are two kinds of data of interest in this project: the measured speedup and the ER. A separate measurement for scalability is not required, as it is derived from the other measurements.

4.3.1 Speedup

Speedup will be measured as a relation between the time it took to execute a specific task with one thread compared to executing the same task with multiple threads. The experiments will measure speedup from 50 different middle-game chess positions up to depth 20, where the positions are handpicked GM matches. The timer will start when the search begins and stop when the search has finished at the final depth. The mathematical formula 4.3 will be used to measure the speedup:

$$Speedup_N = Time_1/Time_N \tag{4.3}$$

Where $Time_1$ is the time it took for one processor to finish the work, and $Time_N$ is the time it took N number of processors to finish the same work.

4.3.2 Elo Rating

ER can be measured as mathematical computation based on the CE's previous performances against other CEs with known ER. Previous measurements by the CCRL will be used to get an initial ER for the CE. The ER will be refined by letting the CE play against other CEs with cutechess-cli and the new ER will be calculated with formulas 4.1 and 4.2. The more the CE wins against other CEs the higher ER it gets [36, 37].

4.3.3 Scalability

Although there is no single formula for calculating scalability, it can be described by analysing a few other factors. With the speedup measurements for all thread-counts, regression analysis can be used to create a function that roughly describes what speedup can be expected for a given thread-count.

Furthermore, the efficiency of an increased thread-count can be calculated with 4.4. In this equation, N denotes the number of threads used and $Speedup$ is the result of equation 4.3. By calculating the efficiency of each thread-count from the experiments, a possible trend could be derived.

$$Efficiency_N = Speedup_N / N \tag{4.4}$$

The measurements here are taken from an identical problem size for all thread-counts, which means that strong scaling holds and Amdahl's law can be used to calculate the theoretical limit on speedup from the serial and parallel parts of the program. From this, since the actual speedup is known, an estimated size of the parallel portion of the program can be derived, as can be seen in Equation 4.5.

$$Speedup_N = 1/(s + (p/N)) \tag{4.5}$$

Where speedup is from formula 4.3, N is the number of threads used, s the part of the program that is sequential and p is the part of the program that can be executed in parallel. The condition must hold that $s + p = 1$ where from this condition p can be determined.

4.3.4 Ethics to method

A moral question worth considering is whether the results are being manipulated or if the findings do not contribute to chess research. It can be argued that these concerns do not present significant ethical dilemmas and are not worth taking into account.

4.3.5 Summary

The kinds of tests that will be executed in this research can be viewed in Table 4.1 along with how many times those tests will be run at minimum for each configuration. The CE configurations are shown in Table 4.2.

Section 5.2 goes into greater detail of the tests.

Test type	**Number of runs**
Time-to-depth	1500 (30 positions × 50 repetitions)
Engine competition	120 (4 opponents × 30 matches)

Table 4.1: Table showing the types of tests and how many times they will be run.

Algorithm	Number of threads
YBWC	1
YBWC	2
YBWC	4
YBWC	8
Lazy SMP	1
Lazy SMP	2
Lazy SMP	4
Lazy SMP	8

Table 4.2: Table showing each configuration of parallel search algorithm and number of threads to be used for the tests.

4.4 Experimental design and Planned Measurements

This research will investigate what parallelisation strategy, Lazy SMP or YBWC, produces the best results. For this, three parameters will be investigated: speedup, scalability and ER. To measure how these parameters change in different conditions, the variables of the parallel search algorithm (YBWC or Lazy SMP) used and the number of threads the engine is run on will be used.

To be able to compare the two parallelisation algorithms fairly, they shall both be implemented in the same CE. Doing this will remove the variables that would arise from comparing two entirely different engines. For this project, the open-source CE Crafty was chosen as the base to build off of.

Crafty is an engine originally developed by Robert Hyatt in 1994 and was one of the first engines to use bitmaps for board representation. Crafty was one of the most influential CEs of the 1990s and 2000s and was continuously developed until its last release, version 25.2 on October 29th 2016. Crafty was chosen for this project due to its simple design, well-commented code, and robust YBWC implementation that would allow us to focus on writing a forked version that instead implements Lazy SMP with relative ease.

Crafty uses the XBoard protocol for communication, which necessitates a frontend that supports both XBoard and UCI for us to be able to conduct tests against most modern engines. For this cutechess-cli was chosen, a very popular command-line frontend that supports both protocols, with many advanced options available.

To measure speedup and scalability, an experiment will be conducted where the engine has to search some number of moves ahead from a given middle game position. The experiment uses a large number of initial positions, and each combination of parallel search algorithm and thread-count will be run from every position. The experiment shall record the time it took to complete the search to the given depth. From this point, speedup and scalability can be derived, and the methodology for doing so is described in Section 4.3.

Before the experiments began, two hypotheses were formulated. The first hypothesis is that Lazy SMP is more scalable than YBWC due to YBWC having to spend time organising the work distribution between the threads. The second hypothesis is that the two algorithms will have similar ERs due to not having found any report during the pre-study claiming that Lazy SMP leading to significantly better or worse play than the alternatives.

The hypotheses will be verified or falsified through experiments. It is believed that by measuring speedup and scalability, it will be possible to estimate or conclude whether one algorithm is more, equally, or less scalable than the other. Additionally, the ER of both algorithms will be measured to estimate the extent of the ER boost gained by increasing the number of threads; this could also be described as the measurement of scalability for speedup and ER gained per thread-increase.

ER will be measured by letting the CE play against a given number of opponents of known ER a given number of times. Initially, the ER of both algorithms YBWC and Lazy SMP using one thread will be calculated, with the number of threads being doubled and a new ER being calculated for up to eight threads. With a known ER for each combination of parallel search algorithm and thread-count, it will be possible to derive the amount of ER gained by each thread-increase for both algorithms.

It is worth mentioning that the same static evaluation function will be used for both algorithms, meaning that when both algorithms examine a board state, they will assign it the same rating. This consistency enhances the accuracy of the comparison, as both algorithms will, in most cases, draw the same conclusions and evaluate a board state simultaneously. Furthermore, if testing is conducted on the same hardware and both algorithms face the same opponents, the comparative study between the two algorithms becomes more relevant, as they utilise the same or similar resources.

4.4.1 Hardware and Software to be used

Hardware: the experiments will be run on one of the authors' personal computers. The specifications are listed in Section 4.8.

Software:

- The source code of Crafty
- JetBrains CLion (IDE)
- GitHub (version control)
- cutechess-cli (automated matchmaking for local engines)
- Overleaf (report writing)
- Grammarly (grammar control for the report)
- Apple Numbers (spreadsheet data analysis)
- Zotero (reference management)

4.5 Assessing reliability and validity of the data collected

4.5.1 Validity of method

A quantitative method is presented in this report. It addresses how the choice of using YBWC or Lazy SMP affects the performance of a CE by measuring speedup, scalability, and ER for different thread-counts. The authors have determined these areas to be the most important concerning parallelisation of a CE, but other less central areas such as resource usage or performance on different operating systems could be of interest to some. Still, the chosen parameters cover the essential differences between the parallel search algorithms.

The chosen method should align well with existing metrics used to measure the performance of CEs. Throughout the pre-study, most calculations of speedup and scalability were done using data from time-to-depth measurements. Many have chosen to calculate ER as suggested by this method, from conducting matches against other CEs, but others have chosen to make looser estimates such as online tests. From a quantitative perspective, the chosen method should give a reasonable ER measurement, but this depends on how many matches can be played in the allotted time for the project.

4.5.2 Reliability of method

The method's reliability will be asserted by running small-scaled versions of the experiments to see how much variance there is in the results. The results will also be compared against previous works, such as those listed in Chapter 3, to see if they are in line with the characteristics of their results.

4.5.3 Data validity

The performance of YBWC and Lazy SMP will rely on both how they are implemented and the overall performance of the rest of the engine. Ideally, the entire engine would be perfectly implemented with all available improvements present, but this is unrealistic for many reasons – the most obvious of ones being the continued innovation in the strongest CEs on the market, the technical skill of the authors of this report, and the time limit imposed on this project.

The open-source CE Crafty will be the basis for the project, which should greatly reduce the amount of work required before the tests can begin. Furthermore, this should allow more time to be spent on assuring the correct implementation of the parallel search algorithms.

The engine developed for this project will as such not be able to compete against the likes of AlphaZero or Stockfish, but that is not its purpose. The two parallel search techniques will be part of an otherwise identical engine in the tests. There may exist advanced improvements that would benefit one technique over the other and would therefore affect the data collected if implemented. The engine developed here will not focus on such advanced techniques, and will instead act more like a baseline comparison between YBWC and Lazy SMP.

The data that will be measured are relevant to the phenomena being investigated. Calculating speedup and scalability requires time measurements, and are here recorded by letting the engine search a fixed number of moves ahead from a large number of middle game positions. Each combination of parallel search technique and the number of threads will search to the same depth and use the same positions to ensure the data generated by them are comparable with each other. Middle game positions are used because opening moves are typically handled by the GUI using a library, and all endgames with up to seven pieces have been solved with Syzygy tables [40]. It is therefore middlegame positions that best reflect the performance of an engine. Using a large number of positions further provides a wider sample size of data to analyse. Researchers from other projects have used a similar approach to measure the data speedup [13, 23, 35].

It is difficult to correctly assess the playing strength of a player, let alone a CE, but calculating the ER is today the most recognised and common way to give a score to the probability of a player's likelihood to win a match. This calculation requires win/loss/draw results from many matches to be accurate, but the data itself has a low risk of being inaccurate assuming the engines are allocated resources fairly.

4.5.4 Reliability of data

Other researchers that have conducted similar kinds of research have trusted their data to be objective but have still taken it with a grain of salt [23, 28, 30, 32]. The phenomenon that when executing the same task with the same equipment would yield the same results could happen but is not a given fact. To measure the reliability of the data measurements of standard deviation between execution to estimate how much deviation can be expected. The data can be assumed to be objective but the accuracy of the data can be taken with a grain of salt.

4.6 Planned Data Analysis

This research is quantitative, where the data analysis strategy will consist of both computational mathematics and statistics. The collected data will be visualised with figures with analysis and conclusions following shortly after.

This analysis method goes hand in hand with other researchers' approaches [31, 34, 35].

4.6.1 Data Analysis Technique

Regression analysis will be applied to the collected data. This will develop mathematical functions that estimate the changes in execution speed and playing strength for parallel search techniques using different thread-counts. From the collected data a mathematical function can be estimated to estimate how much speedup and ER is gained for the algorithms YBWC and Lazy SMP for every thread added [41]. From previous measurements between independent and dependent variables, future outcomes can be estimated.

The analysis can be used for calculating variances in statistical data [42, ch. 14]. The method will be used when calculating the differences in execution time between identical executions to estimate an average deviation in the collected data.

4.6.2 Software Tools

For creating figures and writing the report the online tool Overleaf will be used for this project. A spreadsheet tool will be used to organise the collected data, perform regression analysis, and identify the distribution of data.

4.7 Evaluation framework

During the pre-study session of this research, other researchers' work have been examined and presented in Chapter 2 of this report. The results of their research have been assessed but their methods have not been assessed as thoroughly. Even though the method part has not been examined in greater detail the other researchers' method of approach is similar to the method chosen in this project.

The research method can be summarised as a five-stage method:

1. Perform tests
2. Evaluate results
3. Plan new tests

4. Iterate steps 1-3
5. Result and analysis

The research method in itself is flexible and not detailed but should produce results. The validity of the results can be discussed because the method itself is not rigorous and detailed. There is in place an overarching plan of how the test should be measured and for what purpose. There is a risk for deviation from set research measurement goals in step 2 but if diligently evaluated that could be avoided or if an error occurs a step back in the process and re-evaluate the results.

4.8 System documentation

The tests were run on one of the authors' personal computers. The relevant specifications are listed below:

- CPU: AMD Ryzen 5 3600 (6 physical cores, 12 logical cores with simultaneous multithreading, 3.6 GHz, 32 MB L3 cache)
- RAM: Corsair Vengeance LPX 16 GB (2×8 GB dual-channel, 3200 MHz, DDR4)
- Operating System: Ubuntu 24.04.1 LTS (running under Windows Subsystem for Linux on Windows 11 Pro 23H2)

Chapter 5

Development

5.1 Implementation Details

This section details the process of implementing Lazy SMP, and how it what changes were required for it to work. The source code is available on GitHub.

5.1.1 Analysis

Implementing Lazy SMP began with downloading version 25.2 of Crafty from its official repository and going through its source code to become familiar with the design of the engine before making changes to it. The basic flow of the program can be described as follows:

1. Create the game tree.
2. Read the command line and configuration file for options that must be set before the engine is initialised.
   2.1 `Option()` (in `option.c`) identifies all valid options and calls the appropriate functions to configure the engine appropriately.
3. Initialise the chessboard along with all data structures required for the engine to function using `Initialize()` (in `init.c`)
4. Reread command line arguments and the configuration file for options that must be configured after initialisation.
   4.1 `Option()` identifies all valid options and calls the appropriate functions to configure the engine appropriately.

5. Enter the main loop of the program.
   - 5.1 Read text entered by the user.
     - 5.1.1 If `Option()` recognises an option from the text, that option will be configured.
     - 5.1.2 If no option is recognised, `InputMove()` (in `input.c`) is used to check the text for moves and extract them (if found).
   - 5.2 When a move has been input, `MakeMoveRoot()` (in `make.c`) is used to make that move. After the move has been made, the current player's turn is flipped (white to black, or vice versa).
   - 5.3 Check if the made move resulted in a draw by three-fold repetition, the 50-move rule, or by insufficient material before conducting a search.
   - 5.4 Call the `Iterate()` function (in `iterate.c`) to start the iterative deepening search.
     - 5.4.1 Initialise the variables necessary for the search. Most are set to a default value, but some will use the results of the previous search when available.
     - 5.4.2 If the `smpmt` option was set to a value greater than 1, it will create the specified number of threads using `pthread_create()`.
       - 5.4.2.1 When a thread is created it goes to the `ThreadInit()` function (in `thread.c`). All helper threads are initialised here, and when all are done they call the `ThreadWait()` function where they wait until a searching thread wants help.
     - 5.4.3 If the current search is not the first one, store the principle variation from the previous search in the hash table to make sure it is accessible.
     - 5.4.4 The `Search()` function in `search.c` is used to search one iteration.
       - 5.4.4.1 First determine if the match should end in a draw due to repetition or the 50-move rule before going further.
       - 5.4.4.2 Use the transposition table to see if the current position has already been encountered, and if so return the stored value.

5.4.4.3 If a Syzygy table exists, there are at most six pieces left, and the last move wasn't a capture, use the Syzygy table to find a response without searching.

5.4.4.4 If possible try to do a null-move search to get a cut-off without having to perform a full search.

5.4.4.5 If the current position is part of the principle variation but was not found in the transposition table, recursively call the `Search()` function again with a search depth that is two less.

5.4.4.6 When none of the previous checks are fruitful, perform a full search with the `SearchMoveList()` function with the `mode` parameter set to `serial`.

5.4.4.7 `SearchMoveList()` will initiate variables depending on if `mode` is set to `serial` or `parallel`.

5.4.4.8 Go through the move list

5.4.4.9 Make the move from the list using the `MakeMove()` function in `make.c`.

5.4.4.10 Determine if the move puts the opponent in check, as that will slightly change how the search is performed.

5.4.4.11 Use Futility Pruning and Late Move Pruning to reduce the size of the search tree.

5.4.4.12 Use Late Move Reductions to make the search depth smaller if it appears late in the move list, and is therefore considered likely to be poor.

5.4.4.13 Use the `SearchMove()` function to perform a Principle Variation Search using a smaller beta value to get a null window. The function recursively traverses the branches from the given position by calling the `Search()` function. If the final depth level has been reached (it is at a terminal node), the `Quiesce()` function (in quiesce.c) is called instead to perform a Quiescence Search where static evaluation is used to get a stand/pat score and it tries to generate captures and checks to improve upon it. `SearchMove()` will perform a second search with the original beta value if the value produced is between the alpha and beta values (and therefore not a null window).

5.4.4.14 Unmake the move and check its result. If it is a Fail-High it cancels the search and tells any sibling threads to stop searching. If it is In-Window it will update the alpha and beta values and continue. If it is Illegal it will just continue to the next move in the list.

5.4.4.15 Since the oldest brother node has been searched in this branch it will check if the current node is an appropriate splitting point with the `ThreadSplit()` function (in thread.c). If so, it will use the `Split()` function to choose an available thread to copy data from the parent thread to and tell it to start searching.

5.4.4.16 When all moves have been gone through, the best move will be returned to the parent thread if it is searching in parallel. If it is instead serial, the best move is returned, or MATE/DRAW if no legal move was found.

5.4.5 Check if the result from the iteration is either a Fail-Low or Fail-High, in which case Alpha or Beta respectively is updated for the next iteration.

5.4.6 When all iterations have finished the results of the search are printed according to the user settings.

5.4.7 If the `smp_nice` flag is set and `ponder` is not (for example during computer matches), all helper threads are killed. The resulting value of the search is then returned to the main method.

5.5 Check if the opponent is offering a draw and respond depending on the search results.

5.6 Check if Crafty has been checkmated or is in a stalemate, and if so end the game and respond accordingly.

5.7 Check if Crafty wants to offer a draw or resign depending on the search results.

5.8 Print the move chosen from the search.

5.9 Change to the opponent side to see if the chosen move leads to a draw. If it does, end the game and inform the opponent.

5.10 Save the ponder move of the principle variation so it can be used in the next search.

5.1.2 Implementing Lazy SMP

The programming began when sufficient knowledge of Crafty's inner workings had been acquired. The following subsections will describe the process for implementing Lazy SMP in Crafty.

5.1.2.1 Serialisation

The first change made to the source code was to make the search functions single-threaded. This is because the threads should perform their search independently from one another and communicate using the transposition table.

The serialisation was done by modifying `search.c` to remove all checks for `mode` being set to `parallel` and all checks for `CPUS` (a variable defined when compiling that tells Crafty how many threads it is allowed to use) being greater than one. No further change was necessary for serialisation as the remaining code will execute the same way as when the original program is run single-threaded.

5.1.2.2 First attempt

Unfortunately, the first attempt at writing a Lazy SMP implementation was unsuccessful. For the sake of completeness, however, the process will still be described.

The first attempt to implement Lazy SMP was based on using a customised waiting function in `thread.c` that all threads, including the main thread, start their search from simultaneously. In `thread.c`, this new waiting function was called instead of `ThreadWait()` after having been initialised in `ThreadInit()`. The function was also called by the main thread in `iterate.c` in place of the `Search()` call.

The wait function begins with entering an infinite `while` loop. The first operation in the loop is to use `pthread_barrier_wait()` to ensure the main thread is ready before data is copied from it to the helper threads. Since much of the development was done on a MacBook and because macOS does not include built-in `pthread_barrier` support, a custom implementation made by Aleksey Demakov called DarwinPthreadBarrier was used instead when compiled on Mac computers.

A new function based on CopyFromParent() was used to give the helper threads data needed to begin their searches. This function copies all the data that CopyFromParent() does, except for statistical counters as that would overwrite the counters of the helper threads. This function is only called when the ID of the thread is not zero because the main thread would never need to copy data from itself to itself.

A second pthread_barrier_wait() call is made so that the main thread does not start a search that could change its data before it is copied by the helpers. After the barrier, all threads start searching in parallel using the Search() function.

When a result has been returned to the main thread, it will return that value to iterate.c where the rest of the program continues as normal.

This version of the program compiled correctly, but would crash due to segmentation faults when being run with multiple threads. The cause of this is unknown but is in hindsight thought to be because some of the data copied to the helper threads not having been properly initialised before being copied.

Regardless of the reason why this attempt failed, the program was rolled back to the changes made by the serialisation and the Lazy SMP implementations of other engines were investigated more closely before making a second attempt.

5.1.2.3 Second attempt

The second and final attempt at implementing Lazy SMP takes inspiration from the Cheng CE. More specifically it is based on the pseudo-code presented by Cheng's author, Martin Sedlak, on the Computer Chess Club forums when describing Cheng's Lazy SMP implementation. The three pseudo-code functions detailed in Sedlak's forum post are presented in Listing 5.1 [43].

Listing 5.1: Cheng Lazy SMP Pseudo-code

```
IteratativeDeepening:
    synchronize smp threads (copy age, board,
      ↪ history, repetition list, multipv =>
      ↪ helpers)
    depth 1 with full width window on 1 thread
    loop (depth=2 .. max)
```

```
        AspirationLoop:
            (as usual)
            start helper threads( depth, alpha,
              ↪ beta )
            root( depth, alpha, beta)
            stop helper threads
            (rest as usual)
        end aspiration loop
    end id loop

starting helper threads:
    clear smp abort flag
    for each helper thread:
        copy rootmoves and minimum qs depth =>
          ↪ helper
        signal helper to start root search at
          ↪ current depth (add 1 for each even
          ↪ helper (assuming 0-based indexing)
          ↪ with aspiration alpha, beta bounds
          ↪ and wait until helper starts
          ↪ searching

aborting helper threads:
    set abort flag for each helper and wait for
      ↪ each to stop searching
```

Similar to the first attempt, a custom waiting loop is responsible for the majority of the implementation. This function is only called by the helper threads after they have been initialised. The function, called `LazyWait()`, first calls the `CalculateDepthOffset()` function to get an offset to how deep the threads will search. This function uses the ID of the calling thread to give an offset that is 0 for $1/2$, 1 for $1/4$, 2 for $1/8$, and so on. This distribution is based on that used by Østensen in his master thesis "A Complete Chess Engine Parallelized Using Lazy SMP" [23]. The offset is in place to coerce the threads into searching different paths of the search tree to better utilise the transposition table.

After having acquired the offset, `LazyWait()` enters an infinite `while` loop. The thread is then stuck in another loop where it continuously checks if the search has been terminated while it is stopped. Both cases are changed by the main thread, when terminated the thread will return and when no longer stopped it will start searching using `Search()` to a depth equal to the current iteration depth plus the offset. The thread will set itself to stopped after completing searching, at which point the infinite loop repeats.

Immediately before entering the iterative deepening loop in `iterate.c`, a new function called `CopyToHelpers()` is called by the main thread. This function is based on the existing `CopyFromParent()` function, but has been modified to take the `TREE` structure of the main thread and copy all the relevant data to all helper threads.

Also in `iterate.c`, the main thread will start and stop all helper threads immediately before and after beginning its own search respectively. This is only done if more than one thread is in use and if the current iteration is not the first to make sure that that there are root moves available to be searched. The helper threads are started with the `StartHelpers()` function that gives all helper threads the alpha, beta, and side to move. They are stopped with the `ThreadStopAll()` function, that works by setting the `stop` flag of all threads to 1.

There are a few additional changes done for this implementation. To begin with, the `TREE` structures of the helper threads are initiated by `ThreadInit()` in `thread.c`, instead of being copied from the the main thread when being assigned a branch to search.

Another change was to multiply the size of the `root_moves` array by the number of threads, so that each thread gets their own copy of all root moves. This was done because the array is sometimes rearranged during the search, and if multiple threads did this at the same time the data could be damaged.

A similar change was to make `failhi_delta` and `faillo_delta` arrays, so that each thread has its own value. If they all shared the same values, a fail low or fail high could update the alpha or beta to create a larger window than necessary.

5.2 Testing

5.2.1 Writing The Tests

Two python scripts were written in order to automate the testing process. The first script (which can be seen in Appendix A.1) is used to perform the time-to-depth tests. It should be placed in the same directory as the Crafty executable along with the 30 different GameXX.txt files containing the moves that set up the position to search from.

This script takes three parameters: the first GameXX.txt file to search, the last GameXX.txt file to search from, and the number of threads that Crafty should be run with. The chess moves in the GameXX.txt files were carefully selected by the authors from a file titled $Stockfish\_17\_64-bit\_4CPU.commented.[492].pgn.7z$, available on the CCRL's website. The reason the moves were hand-picked was to ensure that the GameXX files contained a diverse range of chess openings, with the goal of improving test results. Of these 30 complete games, the GameXX files contain the first 24 moves of each game so that Crafty starts searching from a middle-game position. All 30 GameXX files are listed in Appendix B.

It then executes crafty, tells it to use a 256 MB transposition table, a 64 MB pawn hash table, to not ponder, to search to the depth 25, to use at most 120 seconds to search, and how many threads to use before starting the search. When the script detects that the search has finished it saves the time it took to a log file and terminates Crafty. This process is repeated 50 times for each GameXX.txt file within the range GameXX.txt files specified by the user. The search depth of 25 was chosen because small scale testing with random positions had shown that this usually resulted in a time between 10-40 seconds. There was a worry that using too small a search depth could lead to small variances between runs having a large affect on the distributions of the measurements, as well as very short measurements potentially becoming inaccurate. Search depth 25 was found to be good trade-off between stable measurements and being fast enough to perform multiple times. The time limit of 120 seconds was chosen to skip a search in case the engine randomly crashed when an author was away from the testing computer, with 120 seconds being deemed high enough from small scale testing to not aborting searches that were still ongoing.

The second script (shown in Appendix A.2) is used to automate matches between two versions of crafty and the opponents. It requires that both versions of crafty have been configured in cutechess under the names ”crafty-ybwc” and ”crafty-lazy” and that all opponents have been configured with the names ”Amoeba”, ”Atlas”, ”Chess22k”, and ”Clovis” before starting. The script must be placed in the same directory as cutechess-cli. To use different engines than those used in this thesis, replace the names in the opponents array with those corresponding to the engines configured in cutechess. Crafty is tasked with playing 30 matches against each opponent (15 times as white and 15 times as black), with a time control of 0/3 (a three-minute timer without extensions). The matches are repeated with one thread, two threads, and four threads allocated to Crafty. It was initially planned to use a time control of 40/5 (a five-minute timer with five additional minutes added every 40 moves), but small-scale testing revealed that only five matches against each opponent could have been performed under the planned time frame for testing.

This script takes one commandline parameter: the number of threads that Crafty shall use.

5.2.2 Executing The Tests

Before running the tests, all non-essential background applications were closed to free up system resources. The Windows Task Manager and Core Temp (version 1.18.1) were used to monitor resource usage and CPU temperature respectively. Testing only began when Task Manager reported 0% CPU usage across all cores and Core Temp reported temperatures under 40°C. A bug in Windows sometimes caused the "CTF Loader" process (a Windows service responsible for handling some alternate user input methods [44]) to use around 25% of the available CPU utilisation and temperatures of over 50°C, but this was resolved by force-closing it with task manager and relaunching it by running `ctfmon.exe`.

The time-to-depth tests were run on 1, 2, 4, and 8 threads, while the Elo tests were run on 1, 2, and 4 threads. Nothing more was needed for the time-to-depth tests, but the Elo tests required more work to calculate the ERs. Two additional python scripts were as such written.

The first script (seen in Appendix A.3) goes through the logs created when playing the matches and calculates the number of won matches, the number of lost matches, the number of drawn matches, and the win rate of Crafty against each opponent for all thread-counts and writes it to a text file.

The second script (in Appendix A.4) also goes through the logs, but instead calculates an ER for Crafty based on the results from all matches for every thread-count. To calculate the Elo, an initial rating for Crafty, ratings for the opponents, and a K-factor are all required per formulae 4.1 and 4.2. The initial rating for Crafty and the opponents' ratings were taken from the CCRL, while the K-factor is a command-line parameter.

Chapter 6

Results and Analysis

This chapter will present the results and discussions surrounding them.

6.1 Major results

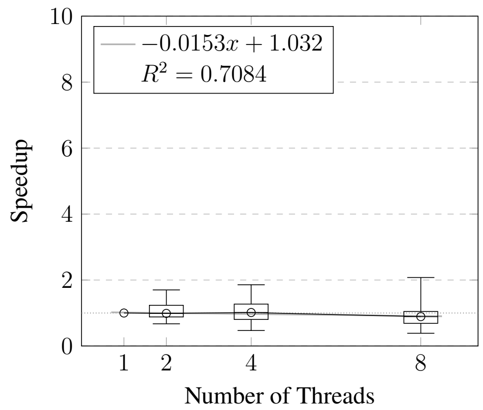
The major results are divided into two categories; those obtained from the time-to-depth tests (in Section 6.1.1) and those from the ER tests (in Section 6.1.2).

6.1.1 Time-to-Depth Test Results

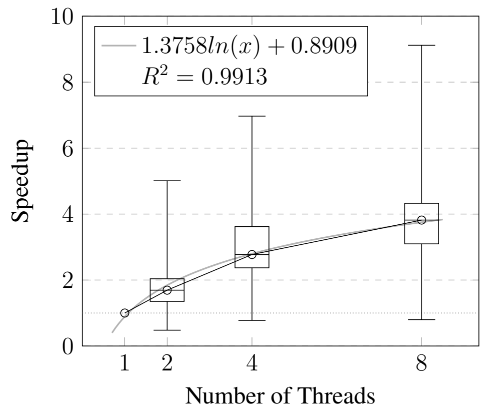
6.1.1.1 Speedup

To determine how much speedup is gained from utilising additional threads for both Lazy SMP and YBWC, both algorithms were tasked with finding the optimal move for 30 different positions 50 times each, while recording the time it took. After the tests, the median times for each position were calculated to exclude anomalous extreme times (which would have affected the average times).

Figure 6.1 depicts two box-and-whiskers diagrams for Lazy SMP and YBWC respectively. The diagrams use the median times for each of the 30 positions to show the speedup gained from running with additional threads. The single-threaded runs show no spread as those are considered the base times, and can't have a speedup against themselves. The subsequent thread counts are shown as the speedup compared to the base times for each position. As such, a value of two would correspond to being twice as fast and a value of 0.5 would be half as fast as the median single-threaded time for the same position. The same results are also presented numerically in Table 6.1 for Lazy SMP and in Table 6.2 for YBWC.



(a) Lazy SMP



(b) YBWC

Figure 6.1: Box-and-whiskers diagrams showing the change in median time-to-depth speedup as thread counts increase for Lazy SMP (a) and YBWC (b). The dotted line represents 1 (no change). Shows a trend line for the median.

The data in Figure 6.1 shows that the median speedup remains at around one regardless of thread count for Lazy SMP, though it dips slightly for eight threads. The largest speedup across the median measurements for all positions increases for each increase in thread count, from 1.700 with two threads to 1.857 with four and 2.076 with eight. In a similar pattern, the smallest speedup decreases with each thread count increase, with 0.674 with two threads, 0.470 with four, and 0.387 with eight. The trend line of the median is linear and has a R^2 score of 0.7084.

The data from YBWC shows that each increase in thread count leads to an improvement in median speedup, as well as maximum and minimum speedup (though all minimum speedup are still less than one). The graph shows that the improvements appear to be logarithmic, which coincides with the logarithmic trend line of the median that has a R^2 score of 0.9913.

When comparing the speedup data from Lazy SMP and YBWC, the latter shows a clear improvement with each doubling of thread counts while the former remains mostly stagnant. YBWC has a larger improvement over Lazy SMP for every quartile and thread count except for the minimum speedup with two threads (0.480 vs. 0.674).

The spread of the measured data increases with larger thread counts for both Lazy SMP and YBWC, but the extreme values are always closer to the median for Lazy SMP.

Threads	Min	Lower Quartile	Median	Upper Quartile	Max
1	1.000	1.000	1.000	1.000	1.000
2	0.674	0.881	0.989	1.235	1.700
4	0.470	0.806	1.015	1.271	1.857
8	0.387	0.689	0.891	1.046	2.076

Table 6.1: Table displaying the time-to-depth speedup data from Figure 6.1 for Lazy SMP.

Threads	Min	Lower Quartile	Median	Upper Quartile	Max
1	1.000	1.000	1.000	1.000	1.000
2	0.480	1.353	1.693	2.038	5.001
4	0.777	2.370	2.773	3.618	6.970
8	0.801	3.097	3.818	4.328	9.116

Table 6.2: Table displaying the time-to-depth speedup data from Figure 6.1 for YBWC.

6.1.1.2 Standard deviation

An important factor to consider when comparing YBWC and Lazy SMP is the standard deviation in the time taken to solve a chess problem. Does one parallelisation strategy consistently solve a specific problem in a shorter time span than the other? If one algorithm solves most problems more quickly, it would be considered more reliable.

Following the time-to-depth speedup test, the standard deviation of the measurements for each position was calculated. Figure 6.2 shows box-and-whiskers plots of the standard deviation calculations for both Lazy SMP (blue, top) and YBWC (red, bottom) grouped by thread counts. The same results are also presented numerically in Table 6.3 for Lazy SMP and in Table 6.4 for YBWC. The results from Figure 6.2 show that Lazy SMP and YBWC perform similarly when single-threaded, with Lazy SMP deviating a bit less than YBWC.

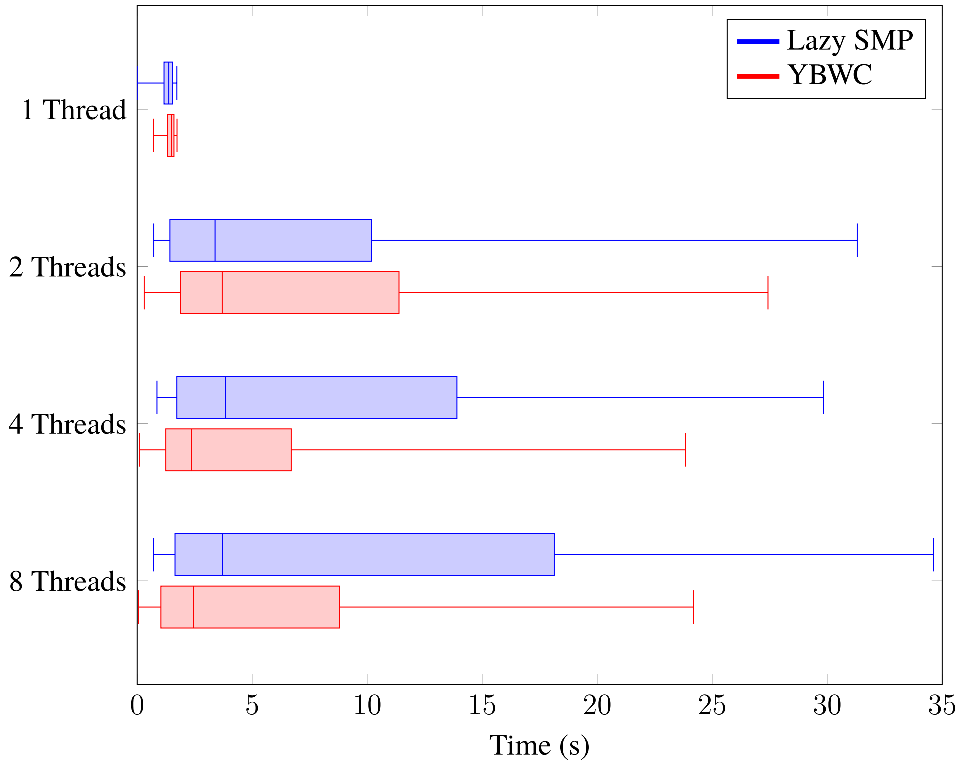


Figure 6.2: Box-and-whiskers diagram showing the standard deviations across all GameXX-file's measurements. Both the Lazy SMP and YBWC versions are shown for 1, 2, 4, and 8 threads.

It should be noted that these calculations are based on the times as they were reported by Crafty. This means that times under one minute have millisecond accuracy while those over one minute have single-second accuracy. This explains why the single-threaded Lazy SMP version has a minimum standard deviation of 0ms, as all runs for "Game6.txt" took 1m 10s. While the standard deviation for this position would have been less than 1s, it is unrealistic to suggest that there was no deviation. Therefore, if "Game6.txt" is excluded the new minimum for single-threaded Lazy SMP is 530ms by "Game16.txt".

With two threads, Lazy SMP produces a lower median deviation and lower and upper quartiles than YBWC. The lower and upper quartiles of both Lazy SMP and YBWC are a similar distance from the median as for the other algorithm. The minimum and maximum deviation of YBWC are lower than those of Lazy SMP.

The results show that for both four and eight threads, YBWC produces more stable results than Lazy SMP in terms of smaller extreme values, smaller lower and upper quartile distance from the median, as well as a smaller median.

Threads	Min	Lower Quartile	Median	Upper Quartile	Max
1	0ms	1168ms	1375ms	1531ms	1726ms
2	722ms	1424ms	3387ms	10196ms	31306ms
4	861ms	1725ms	3852ms	13902ms	29843ms
8	707ms	1646ms	3721ms	18133ms	34631ms

Table 6.3: Table displaying the data from Figure 6.2 for Lazy SMP.

Threads	Min	Lower Quartile	Median	Upper Quartile	Max
1	706ms	1320ms	1501ms	1600ms	1773ms
2	307ms	1895ms	3698ms	11384ms	27426ms
4	94ms	1245ms	2372ms	6968ms	23846ms
8	55ms	132ms	2450ms	8793ms	24182ms

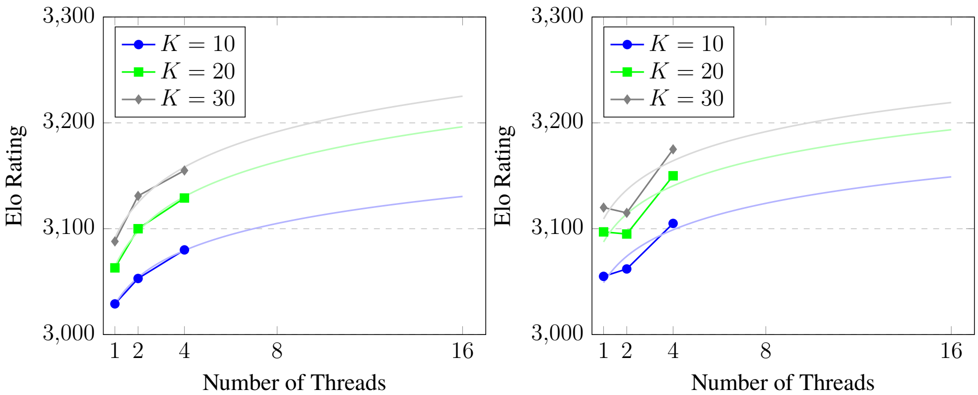
Table 6.4: Table displaying the data from Figure 6.2 for YBWC.

6.1.2 Elo Rating Test Results

This section will present the results of the ER tests. The results presented come from Crafty's matches against four different opponents. Both the regular version of Crafty 25.2 with the YBWC parallelisation and the modified Lazy SMP version were tested.

Figure 6.3 shows two graphs depicting the calculated ERs for Lazy SMP and YBWC for the three thread counts. The same ERs are also presented numerically in Table 6.5. The ERs have been calculated with three different K-factors (10, 20, 30) to show how this choice affects the rating. The K-factor determines how much a player's Elo should increase or decrease after a match. As such beginners typically use high K-factors, while more experienced players use lower K-factors.

This is seen in both graphs by how the plots are spaced out, with $K = 10$ giving the lowest Elo and $K = 30$ the highest. It is also seen in the YBWC graph in the use of two threads, where K-factor 10 gives a small Elo increase over one thread and the two higher K-factors give a small decrease. The Lazy SMP version, on the other hand, shows a clear increase in Elo with two threads for each K-factor.



(a) Lazy SMP

(b) YBWC

Figure 6.3: Plots showing how ERs change for Lazy SMP (a) and YBWC (b).

Threads	K=10	K=20	K=30
1	3029	3063	3088
2	3053	3100	3131
4	3080	3129	3155

(a) Lazy SMP Elo Ratings.

Threads	K=10	K=20	K=30
1	3055	3097	3120
2	3062	3095	3115
4	3105	3150	3175

(b) YBWC Elo Ratings.

Table 6.5: The ERs calculated with different K-factors as thread-counts increase for Lazy SMP (a) and YBWC (b).

Both graphs also have logarithmic functions that best fit the plots overlaid to visualise how additional threads are projected to impact the ERs. These functions were calculated in Apple Numbers by plotting the data points and selecting the ”logarithmic” trendline.

For Lazy SMP these functions and their associated R^2 values are:

- $K = 10 : y = 36.789 \times ln(x) + 3028.5, R^2 = 0.9988$
- $K = 20 : y = 47.609 \times ln(x) + 3064.3, R^2 = 0.9951$
- $K = 30 : y = 48.330 \times ln(x) + 3091.2, R^2 = 0.9739$

and for YBWC:

- $K = 10 : y = 36.067 \times ln(x) + 3049, R^2 = 0.8527$

- $K = 20 : y = 38.231 \times ln(x) + 3087.5, R^2 = 0.7217$
- $K = 30 : y = 39.674 \times ln(x) + 3109.2, R^2 = 0.6823$

Another way to visualise the results of the matches is shown in Figure 6.4. These graphs plot the win rate of the two versions of Crafty against the four opponents as thread-counts increase. The win rate is calculated as the average of all game outcomes against an opponent where winning is 1, losing is 0, and drawing is $\frac{1}{2}$.

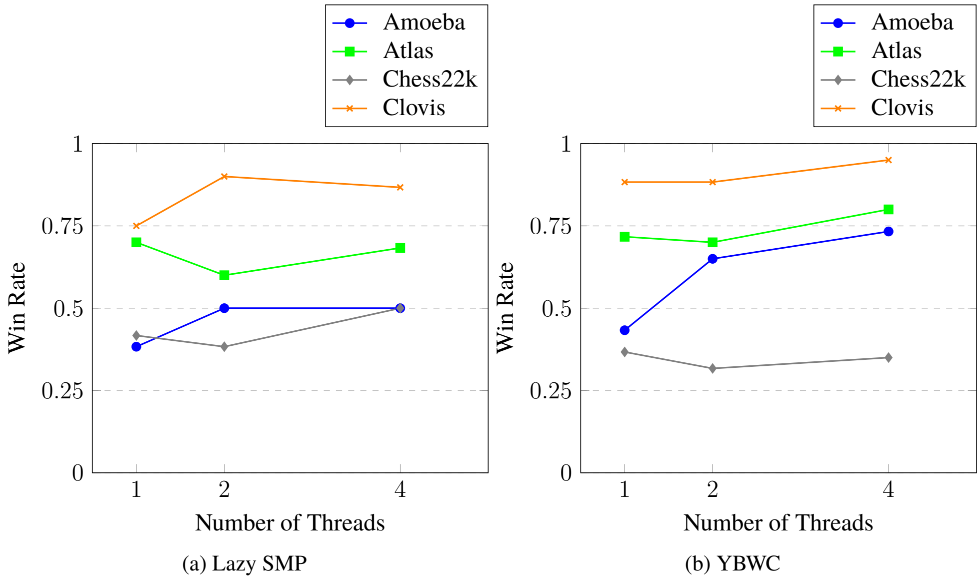


Figure 6.4: Plots showing how the win rates against four opponents change for Lazy SMP (a) and YBWC (b).

The figure shows that the win rates against all opponents are very similar between Lazy SMP and YBWC when single-threaded.

With two threads, Lazy SMP sees a sizably increased win rate against Clovis and Amoeba, a small decrease against Chess22k, and a sizable decrease against Atlas. YBWC has a large increase against Amoeba, no change against Clovis, a slight decrease against Atlas, and a small decrease against Chess22k.

Finally, with four threads Lazy SMP displays a sizable increase against Atlas (though still slightly less than single-threaded) and Chess22k, no change against Amoeba, and a slight decrease against Clovis. YBWC shows a sizable increase against Amoeba, Atlas, and Clovis, and a slight increase against Chess22k (though still less than single-threaded).

The results can further be visualised by the total number of wins in Figure 6.5, the number of losses in Figure 6.6, and the number of draws in Figure 6.7. The total number of wins and the win-rate differ in that the former is purely the number of times Crafty won while the latter is the percentage of games that resulted in a win. As such, drawn games count as a half win towards the win percentage, but not the number of wins.

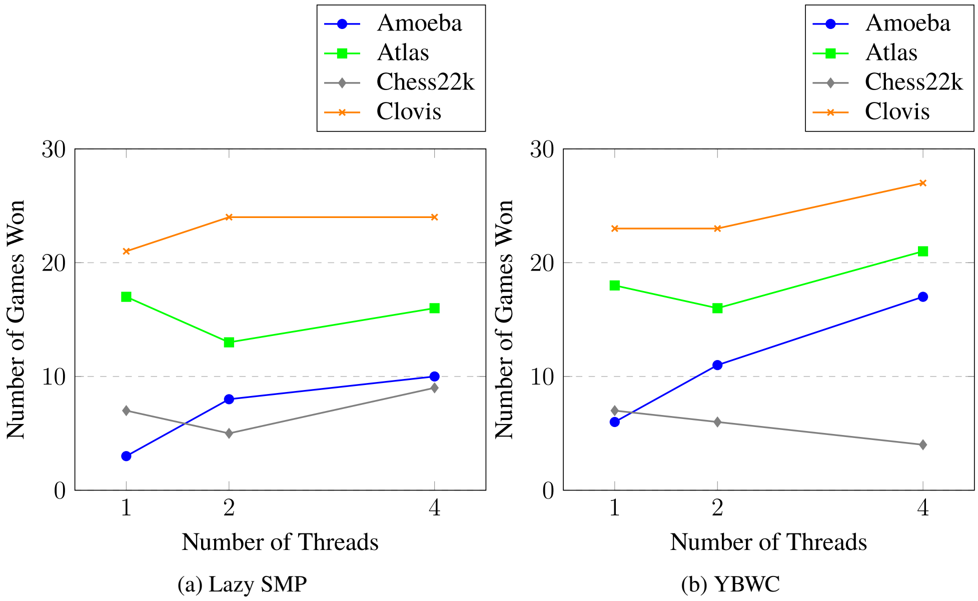
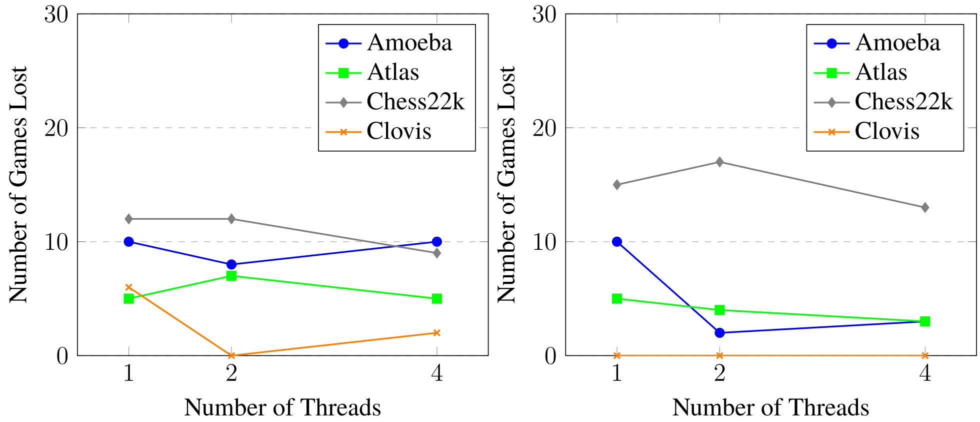


Figure 6.5: Plots showing how the number of wins against four opponents changes for Lazy SMP (a) and YBWC (b).



(a) Lazy SMP

(b) YBWC

Figure 6.6: Plots showing how the number of losses against four opponents changes for Lazy SMP (a) and YBWC (b).

6.2 Reliability Analysis

6.2.1 Speedup

Access to computer equipment with 16 cores would have been beneficial for the speedup tests, as it would better demonstrate how effectively the two algorithms utilise abundant computational resources. Such a setup could reveal performance characteristics that might not be apparent on systems with fewer available threads, especially regarding scalability and resource utilisation.

It is also worth noting that more detailed speedup tests, with thread-counts incrementally tested from 1, 2, 3, 4, ..., up to 16 threads, would have provided a more comprehensive dataset. This approach would offer a finer-grained understanding of how performance scales with the number of threads, compared to the limited data points from testing only 1, 2, 4, and 8 threads.

Another important consideration is the accuracy of the timing data reported by CE Crafty. Since Crafty reports time measurements with lower precision for durations exceeding one minute, any data points where the execution time exceeds this threshold should be excluded from the analysis. Including such data could lead to misleading conclusions, as the inaccuracies might distort the true performance characteristics of the algorithms.

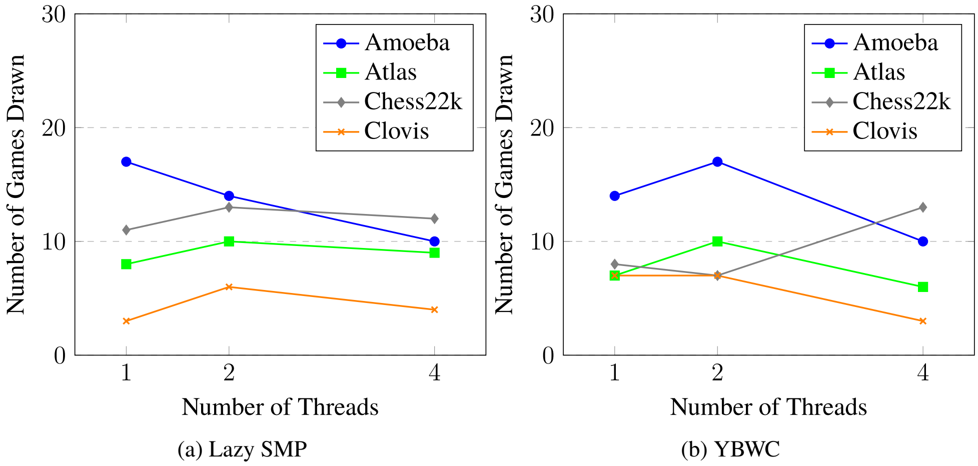


Figure 6.7: Plots showing how the number of draws against four opponents change for Lazy SMP (a) and YBWC (b).

6.2.2 Elo

The measurements of Elo gained with an increased number of threads for the two algorithms would have been more detailed if the project's computer equipment had supported the use of eight threads for a chess match. This would have provided four measurement points, an improvement over the current three.

6.2.3 Lazy SMP

Considering that the Lazy SMP algorithm was implemented by two novice programmers while the YBWC algorithm was developed by professionals, it is reasonable to assume that YBWC is more robustly implemented. This suggests that the Lazy SMP implementation may contain more flaws and be less fully optimised in comparison.

6.3 Validity Analysis

6.3.1 Elo

The Elo measurements in this project follow a standardised approach, using the same algorithm employed by organisations like FIDE and websites such as Chess.com. This alignment with common practice ensures that our method is reliable and produces comparable results.

6.3.2 Speedup

The method used for the speedup test in this project is straightforward: select a chess position, measure how quickly the algorithms solve it using x threads, and repeat the process y times to calculate the median. In this study y was set to 50, but increasing this number would have improved the accuracy of the measurements. Our analysis revealed that a larger sample size would have resulted in data points more closely following a normal distribution when graphed. This underestimation of the required number of tests affected the precision of the results

Chapter 7

Discussion

7.1 Elo

The results, as shown in Section 6.1.2 and illustrated in graph 6.3, indicate that the ERs achieved by the YBWC and Lazy SMP algorithms are roughly similar. While there are slight differences in Elo gains depending on the number of threads used, their final ERs are nearly identical. This suggests that both algorithms perform comparably.

As discussed in Section 3.3, predicting the winner whether a bot or human becomes increasingly difficult when the Elo difference is small. Consequently, if these two algorithms were to compete using the same number of threads (e.g., 1, 2, or 4 threads), the match would most likely result in a draw.

The calculated logarithmic functions all have very high R^2 values for Lazy SMP, and somewhat high for YBWC. A high R^2 value means that the function closely adheres to the data points it tries describe, and could be a good predictor of future values. Unfortunately, the logarithmic functions presented here only had three data points each to be calculated from and could be improved by calculating the ER for more thread-counts. Table 7.1 shows what the logarithmic functions predict the ER to be for higher thread-counts for both Lazy SMP and YBWC. The table shows that both algorithms are predicted to keep producing similar ERs for higher thread-counts, but the exact numbers should be taken with a grain of salt due to the functions being calculated from few data points.

Of the papers handled in the pre-study, only Himstedt measured Elo changes from increased core counts [32]. Himstedt's findings were that

Threads	K=10	K=20	K=30
8	3105	3163	3192
16	3131	3196	3225
32	3156	3229	3259
64	3181	3262	3292
128	3207	3295	3326
256	3233	3328	3359

(a) Predicted Lazy SMP Elo Ratings

Threads	K=10	K=20	K=30
8	3124	3167	3192
16	3149	3193	3219
32	3174	3220	3247
64	3199	3246	3274
128	3224	3273	3302
256	3249	3299	3329

(b) Predicted YBWC Elo Ratings

Table 7.1: The predicted ERs for Lazy SMP and YBWC using the logarithmic function calculated in Section 6.1.2.

their new implementation (CLUSTER TOGA) had an Elo difference of -10 compared to the single threaded original engine (TOGA) with one core, a difference of +36 with two cores, and a difference of +120 with four cores. Since these diffeneces are compared to the same opponent it can be assumed that the ER of CLUSTER TOGA increased by about 47 when going from one core to two cores, and by about 130 when going from one core to four cores.

This can be compared to the ERs calculated in this report, where Lazy SMP had an increase of 29, 37, or 43 (depending on K-value) when going from one thread to two threads, and an increase of 51, 66, or 67 when going from one to four threads. The YBWC implementation saw a difference of +7, -2, or -5 when going from one to two threads, and an increase of 50, 53, or 55 when going from one to four threads.

7.2 Speedup

The results presented in Section 6.1.1 in Figure 6.1 indicates that Lazy SMP produces more stable performance metrics than YBWC when utilising one or two threads. In contrast, YBWC demonstrates greater stability compared to Lazy SMP when running on four to eight threads. Stability in this context refers to a narrower range of time required to solve a chess problem, with fewer extreme outliers and a reduced standard deviation from the median.

This difference in performance could be attributed to YBWC's superior task delegation mechanism, which contrasts with Lazy SMP's random and unrestricted thread allocation. With many threads available, Lazy SMP's approach may lead to multiple threads working on the same subtrees, resulting in redundant processing and the need to abort some tasks. In comparison, YBWC likely minimises such collisions, making it more stable when a larger number of threads are available.

While YBWC may have a slower start compared to Lazy SMP, its structured thread management ensures better performance consistency as the number of threads increases. This makes YBWC particularly effective in scenarios with higher thread-counts, while Lazy SMP performs well in environments with fewer computational resources.

Among the papers handled in the pre-study, most calculated some sort of speedup. The most relevant comparison for YBWC being against Rasmussen with the dedicated 12 processor machine achieved a 1.889 speedup with two cores, 3.617 with four cores, and 6.727 with eight cores [30]. Rasmussen's findings are largely similar to those in the median measurements in the upper quartile of positions in this report (see Table 6.2). The other sources also measured speedup, but used processors with much higher core-counts that did not include data for the same (or close to the same) thread-counts used in this project.

For Lazy SMP, Vrzina measured a 40% (1.4) speedup with four cores [13], Østensen calculated speedups of 1.14 and 2.01 with two and four cores respectively [23], and Brange measured a 33.1% average reduction in execution time ($\sim$ 1.49 speedup) with four cores [34]. These findings are consistent with the results of this report, mostly fitting between the median measurements of the positions between the upper quartile and max (see Table 6.1).

It should be noted that the exact depth to search to, as well as the positions to search from are different between the reports in the prestudy themselves and this report. To truly get a direct comparison, the same positions and similar search depths must be used. This is because some positions may not be possible to search much faster even with additional thread counts, and because a position could become much more complicated to search at a larger depth than the one measured.

7.3 Summary of Elo and Speedup

While both Lazy SMP and YBWC yield roughly similar Elo results overall, their performance in solving chess problems varies significantly depending on the number of threads utilised. As shown in Section 6.1.2, Lazy SMP experiences a substantial Elo boost when moving from one to two threads. In contrast, YBWC appears to lose Elo in two out of three cases when transitioning from one to two threads.

These results suggest that YBWC's more advanced thread delegation algorithm may be redundant or even counterproductive when limited to just two threads. On the other hand, the absence of a dedicated thread delegation mechanism in Lazy SMP seems to have a positive impact on performance in this configuration. This outcome highlights the possibility that simpler parallelisation strategies may be more effective in scenarios with limited thread-counts, whereas more complex algorithms might only show their advantage as the number of available threads increases.

7.4 Parallelisation of both algorithms

With the use of Amdahl's formula (see Equation 4.5), it can be calculated which percentile of the algorithm is parallelisable and a maximal speedup. The percentile p can be estimated with an arbitrary value of $Speedup_N$ from tables 6.1 and 6.2 for both algorithms. For example, let us say the $Speedup_4 = 1.015$ for Lazy SMP and $Speedup_2 = 1.693$ for YBWC, where both values are the medians for both algorithms. To calculate the percentile of the algorithm that is parallelisable, the relation that the sequential part plus the parallelisable of the algorithm must be equal to 1:

$$s + p = 1 => s = 1 - p \tag{7.1}$$

where s is the sequential part of the algorithm and p is the parallelizable part. Then the percentile of the program that is parallelizable for Lazy SMP would be calculated as:

$1.015 = 1/(s+(p/4)) => 1.015 = 1/(1-p+(p/4)) = 1/(1-3p/4) => 1.015(1 - 3p/4) = 1 => (1.015 - 1) = (1.015) * 3p/4 => p = 4 * (1.015 - 1)/ * 3(1.015) = 0.0197044335 \approx 0.02 = 2\%$

The estimated percentile of the program that is parallelisable is 2%. For YBWC, an estimated percentile of the algorithm that is parallelisable would be calculated as:

$1.693 = 1/(s+(p/2)) => 1.693 = 1/(1-p+(p/2)) = 1/(1-p/2) => 1.693(1 - p/2) = 1 => 1.693 - 1 = 1.693p/2 => p = 2 * (1.693 - 1)/(1.693) = 0.8186650916 \approx 0.82 = 82\%$

According to Amdahl's formula (Equation 4.5), YBWC shows that a large portion of its program (82%) is parallelisable, whereas Lazy SMP shows only 2%. This result is somewhat surprising, as one would expect Lazy SMP to be more parallelisable than YBWC based on the way the algorithms are designed. The poor scalability of Lazy SMP could be attributed to its underperformance in the speedup test, which likely contributes to it's disappointing speedup results. This suggests that Lazy SMP may be unreliable in terms of scalability, with thread collisions potentially being the underlying cause. These collisions could hinder efficient parallelisation, leading to sub-optimal performance when scaling with more threads.

Then consider YBWC only because according to the results YBWC is much more scalable than Lazy SMP. It can be calculated with the formula (Equation 4.5) a maximal speedup for YBWC by letting the value N (the number of threads) go to infinity:

$\lim_{N\to\infty} Speedup_N = \lim_{N\to\infty} 1/((1-0.82)+(0.82/N)) = 1/(1-0.82) = 5.5555555...\approx 5.6 = 560\%$

The maximum speedup achievable by the YBWC algorithm is estimated to be approximately 560%. This implies that, in theory, YBWC can perform up to 5.6 times better with an infinite number of threads compared to a single thread. While a 5.6x speedup might seem significant, it is somewhat disappointing, as it highlights a diminishing return on performance with increasing resources.

The fact that infinite resources do not yield infinite performance suggests inherent scalability limitations within the algorithm. Given this limitation, it may be beneficial to calculate the performance-per-thread efficiency to optimise resource allocation. Such a calculation could help determine the optimal number of threads required to achieve a balance between performance gains and resource utilisation, thereby maximising overall efficiency.

7.5 How to improve Lazy SMP

The current design of Lazy SMP is functional, but it does not prevent thread collisions. An improvement to Lazy SMP would involve ensuring that all threads are assigned unique tasks, thereby eliminating the possibility of multiple threads attempting to execute the same task simultaneously.

One potential solution is to implement a system in which each thread randomly selects a unique task from the available pool. This could be achieved by assigning a unique identifier (for example, a number) to each node in the search tree. Before executing, each thread would be required to acquire a unique set of node identifiers, ensuring that no two threads work on the same task.

To implement this approach, threads would need a communication mechanism to coordinate task allocation and avoid conflicts. This could be accomplished through a shared data structure or a distributed task management system, where threads communicate their selected tasks and update the pool of available tasks in real-time. This sort of coordination would, however, go against the idea of Lazy SMP being lazy.

7.6 Implemented Lazy SMP vs YBWC

Is the Lazy SMP implementation by two novices comparable to the YBWC developed by professionals? An unofficial test was conducted midway through development where the two algorithms competed in chess matches during the development of Lazy SMP. Most matches that did not encounter bugs ended in draws, with some resulting in victories for YBWC. While YBWC may be more effectively implemented, the test results suggest that the Lazy SMP implementation is still competitive and not entirely outmatched.

Chapter 8

Conclusions and Future work

8.1 Conclusions

The project have meet the 4 objective that was set in section 1.4. The project have found an open source CE, an Lazy SMP and YBWC algorithm have been implemented, the test have been executed and analysis and conclusions have been drawn from the gathered data. Furthermore, the following conclusions can be drawn from the results of the tests.

The results of Chapter 6 indicate that Lazy SMP and YBWC produce similar ERs, making the choice between the two algorithms somewhat arbitrary. However, a notable trend is the preference for Lazy SMP, mainly due to its simplicity of implementation and comparable Elo performance, as confirmed by our research. Our findings also highlight that as the number of threads increases, the risk of thread collisions becomes more significant. This leads to computational time being wasted on aborting and restarting threads, rather than executing them efficiently. Therefore, for systems with a high number of threads, it becomes advantageous to utilise an algorithm that can intelligently delegate tasks to minimise thread collisions and optimise resource utilisation. Hence forth, the choice of algorithm depend on the specific circumstances where according to our research:

- For systems with one or two available threads, Lazy SMP demonstrates slightly better performance, likely due to its simpler structure and lower overhead.

- For systems with four or more available threads, YBWC tends to perform better, as its more sophisticated thread delegation helps mitigate the risk of thread collisions and makes better use of the available computational resources.

Thus, while Lazy SMP is preferable for small-scale parallelism, YBWC is more suitable for environments with a higher thread-count, where efficient task management becomes critical.

8.2 Limitations

There are a couple of limitations in this student project, below is a list where limitations are mentioned:

1. The CPU of the computer used for testing could have had a higher core-count
2. The Lazy SMP implementation was not done by professionals and had a time limit
3. There could have been more data points for the speedup test
4. There could have been more CEs for the Elo test and more matches
5. The positions and search depths used in time-to-depth tests are not standardised across reports
6. Excluded computers internal workings in the report

There are a couple of limitations for this project which is related to the fact that this is a student project. This project does not have any big funding from companies that want to invest in the project.

8.3 Contribution

This project has contributed to the chess community by presenting a scientific paper that compares the two most popular chess algorithms YBWC and Lazy SMP, for a parallelised CE. These findings are intended to help CE developers make informed decisions when choosing between the two algorithms. Additionally, the paper offers insights into the weaknesses of both algorithms and how they can be refined to better suit the specific needs of a CE under development.

8.4 Future work

It would be interesting to see what result one would get from the tests when using more threads. For example what results would Lazy SMP and YBWC get when using 16 threads or more.

8.4.1 What has been left undone?

The objectives of this project have been finalised, with the results presented, analysed, and discussed. However, the project did not include additional testing with a larger number of threads or testing with odd numbers of threads (with the exception of 1 thread, such as 3, 5, 7, etc.). The primary reason for this limitation was due to constraint of available time.

8.4.2 Next obvious things to be done

The most obvious next step would be to conduct tests with a larger number of threads to better understand the scalability of the algorithms. Additionally, a more granular approach to testing—where the thread-count increases incrementally by one (i.e., 1, 2, 3, 4, 5, ..., 16) instead of the current sequence of 1, 2, 4, and 8 would help fill gaps in the results and provide a more detailed performance profile. It was not done in this report because of time constraints, as the number of tests needing to be conducted would have increased drastically.

Another clear step would be to have a Lazy SMP implementation carried out by professionals, ensuring both algorithms are more comparable in terms of implementation. This would improve the comparative test results, as the two algorithms would be competing against each other at their best.

8.5 Reflections

Throughout the course of this project, many of the intended objectives have been successfully completed. One notable aspect of the project is that the initial idea evolved significantly over time. This evolution has had both positive and negative effects. On the positive side, the iterative ”learn and adapt” approach allowed for continuous improvements and refinements, ultimately enhancing the project’s quality and depth. However, the lack of a clear and well-defined direction at the outset was a challenge, as it proved to be time-consuming and introduced some inefficiencies in managing the project timeline.

References

[1] U.s. chess federation membership 2023. [Online]. Available: https://www.statista.com/statistics/1460028/membership-us-chess-federation/ [Page 1.]

[2] A. E. Soltis. Chess - history | britannica. [Online]. Available: https://www.britannica.com/topic/chess/History [Page 2.]

[3] C. Stapczynski. History of chess | who invented chess. [Online]. Available: https://www.chess.com/article/view/history-of-chess [Page 2.]

[4] C. E. Shannon, "Programming a computer for playing chess," in *Computer Chess Compendium*, D. Levy, Ed. Springer, pp. 2–13. ISBN 978-1-4757-1968-0. [Online]. Available: https://doi.org/10.1007/978-1-4757-1968-0_1 [Page 2.]

[5] L. Clark, "Remembering alan turing: from codebreaking to AI, turing made the world what it is today," section: tags. [Online]. Available: https://www.wired.com/story/turing-contributions/ [Page 3.]

[6] "The reconstruction of turing's "paper machine"." [Online]. Available: https://videolectures.net/turing100_kasparov_friedel_paper_machine/ [Page 3.]

[7] C. c. Team (CHESScom). Kasparov vs. deep blue | the match that changed history. [Online]. Available: https://www.chess.com/article/view/deep-blue-kasparov-chess [Page 3.]

[8] F. Friedel and G. Kasparov. Garry kasparov on how it all started. [Online]. Available: https://en.chessbase.com/post/garry-kasparov-on-how-it-all-started [Page 3.]

[9] A. Håkansson, "Portal of research methods and methodologies for research projects and degree projects." CSREA Press U.S.A, pp. 67–73. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-136960 [Pages 6, 7, and 47.]

[10] R. Hyatt, "Chess program board representations." [Online]. Available: https://craftychess.com/hyatt/boardrep.html [Pages 12, 13, 14, and 15.]

[11] P. Keller. Chess and bitboards. [Online]. Available: https://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/index.html [Pages 12, 16, and 17.]

[12] P. Bijl and A. P. Tiet, "Exploring modern chess engine architectures." [Pages 17 and 19.]

[13] S. Vrzina, "Piece by piece: Building a strong chess engine." [Online]. Available: https://www.cs.vu.nl/~wanf/theses/vrzina-bscthesis.pdf [Pages 19, 27, 35, 54, and 81.]

[14] V. Vuckovic, "Candidate moves method implementation in MiniMax search procedure of the achilles chess engine," in *2015 12th International Conference on Telecommunication in Modern Satellite, Cable and Broadcasting Services (SIKS)*. doi: 10.1109/ SKS.2015.7357795 pp. 314–317. [Online]. Available: https://ieeexplore.ieee.org/document/7357795 [Page 21.]

[15] D. J. Edwards and T. P. Hart, "The alpha-beta heuristic," accepted: 2004-10-04T14:39:10Z. [Online]. Available: https://dspace.mit.edu/handle/1721.1/6098 [Page 23.]

[16] D. M. Breuker, J. Uiterwijk, and J. van den Herik, "Information in transposition tables." [Page 24.]

[17] Evaluation - chessprogramming wiki. [Online]. Available: https://www.chessprogramming.org/Evaluation#Basic_Evaluation_Features [Page 25.]

[18] Simplified evaluation function - chessprogramming wiki. [Online]. Available: https://www.chessprogramming.org/Simplified_Evaluation_Function [Pages 25 and 26.]

[19] B. A. Abdelghani, J. Dari, and S. Banitaan, "Comparing traditional and deep learning approaches in developing chess AI engines,"

in *2023 3rd International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*. doi: 10.1109/ICECCME57830.2023.10252232 pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/10252232 [Page 26.]

[20] J. Madake, C. Deotale, G. Charde, and S. Bhatlawande, "CHESS AI: Machine learning and minimax based chess engine," in *2023 International Conference for Advancement in Technology (ICONAT)*. doi: 10.1109/ICONAT57137.2023.10080746 pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/10080746 [Page 26.]

[21] E. Vazquez-Fernandez and C. A. C. Coello, "An adaptive evolutionary algorithm based on tactical and positional chess problems to adjust the weights of a chess engine," in *2013 IEEE Congress on Evolutionary Computation*. IEEE. doi: 10.1109/CEC.2013.6557727. ISBN 978-1-4799-0454-9 978-1-4799-0453-2 978-1-4799-0451-8 978-1-4799-0452-5 pp. 1395–1402. [Online]. Available: http://ieeexplore.ieee.org/document/6557727/ [Page 26.]

[22] E. Vazquez-Fernandez, C. A. Coello Coello, and F. D. S. Troncoso, "An evolutionary algorithm coupled with the hooke-jeeves algorithm for tuning a chess evaluation function," in *2012 IEEE Congress on Evolutionary Computation*. IEEE. doi: 10.1109/CEC.2012.6252977. ISBN 978-1-4673-1509-8 978-1-4673-1510-4 978-1-4673-1508-1 pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/6252977/ [Page 26.]

[23] E. F. Østensen, "A complete chess engine parallelized using lazy SMP." [Online]. Available: https://www.semanticscholar.org/paper/A-Complete-Chess-Engine-Parallelized-Using-Lazy-SMP-Østensen/623825b7669ae053cdcacce69b51a6e8a927cbfb [Pages 27, 29, 30, 36, 54, 63, and 81.]

[24] A. Upadhyay. Two threads, one core: How simultaneous multithreading works under the hood. [Online]. Available: https://blog.codingconfessions.com/p/simultaneous-multithreading [Pages 27 and 28.]

[25] N. Troccoli and J. Cain, "CS110 lecture 13: Introduction to multithreading." [Online]. Available: https://web.stanford.edu/class/cs110/lectures/cs110-win2122-lecture-13.pdf [Page 28.]

[26] M. Flinders and I. Smalley. What is parallel computing? | IBM. [Online]. Available: https://www.ibm.com/think/topics/parallel-computing [Page 28.]

[27] "Parallel algorithms for machine intelligence and vision." [Online]. Available: https://link.springer.com/10.1007/978-1-4612-3390-9 [Pages 28 and 29.]

[28] R. Feldmann, P. Mysliwietz, and B. Monien, "A fully distributed chess program." [Online]. Available: https://www.semanticscholar.org/paper/A-Fully-Distributed-Chess-Program-Feldmann-Mysliwietz/6b75facdf4608cbd798092ec6eb5436b2209e361 [Pages 29, 34, and 54.]

[29] D. Homan. Lazy SMP, part 2 - TalkChess.com. [Online]. Available: https://www.talkchess.com/forum/viewtopic.php?t=46858 [Page 29.]

[30] D. R. Rasmussen, "Parallel chess searching and bitboards." [Online]. Available: https://www.semanticscholar.org/paper/Parallel-Chess-Searching-and-Bitboards-Rasmussen/969a4b67c5e361bd34998c6e5b40221730526ad5 [Pages 31, 54, and 81.]

[31] A. Ura, D. Yokoyama, and T. Chikayama, "Two-level task scheduling for parallel game tree search based on necessity," vol. 21, no. 1, pp. 17–25. doi: 10.2197/ipsjjip.21.17. [Online]. Available: https://www.jstage.jst.go.jp/article/ipsjjip/21/1/21_17/_article [Pages 32 and 55.]

[32] K. Himstedt, "GridChess: Combining optimistic pondering with the young brothers wait concept," vol. 35, no. 2, pp. 67–79. doi: 10.3233/ICG-2012-35202 Publisher: IOS Press. [Online]. Available: https://content.iospress.com/articles/icga-journal/icg35202 [Pages 32, 54, and 79.]

[33] D. Kopec and I. Bratko, "THE BRATKO-KOPEC EXPERIMENT: A COMPARISON OF HUMAN AND COMPUTER PERFORMANCE IN CHESS," in *Advances in Computer Chess*, ser. Pergamon Chess Series, M. R. B. Clarke, Ed. Pergamon, pp. 57–72. ISBN 978-0-08-026898-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780080268989500097 [Page 34.]

[34] H. Brange, "Evaluating heuristic and algorithmic improvements for alpha-beta search in a chess engine." [Online]. Available: http://lup.lub.lu.se/student-papers/record/9069249 [Pages 37, 55, and 81.]

[35] K. Hazama and H. Ebara, "Branch and bound algorithm for parallel many-core architecture," in *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*. doi: 10.1109/CANDARW.2018.00058 pp. 272–277. [Online]. Available: https://ieeexplore.ieee.org/document/8590912 [Pages 38, 54, and 55.]

[36] F. Thabtah, A. J. Padmavathy, and A. Pritchard, "Chess results analysis using elo measure with machine learning." doi: 10.1142/S0219649220500069 Publisher: World Scientific Publishing Company. [Online]. Available: https://www.worldscientific.com/worldscinet/jikm [Pages 39 and 48.]

[37] Z. Fan, Y. Kuang, and X. Lin, "Chess game result prediction system." [Online]. Available: https://cs229.stanford.edu/proj2013/FanKuangLin-ChessGameResultPredictionSystem.pdf [Pages 40 and 48.]

[38] R. Mittal. What is an ELO rating? [Online]. Available: https://medium.com/purple-theory/what-is-elo-rating-c4eb7a9061e0 [Pages 41 and 45.]

[39] G. Purohit. Elo rating algorithm. Section: DSA. [Online]. Available: https://www.geeksforgeeks.org/elo-rating-algorithm/ [Page 45.]

[40] N. Fiekas. KvK – syzygy endgame tablebases. [Online]. Available: https://syzygy-tables.info/ [Page 54.]

[41] D. L. Mohr, W. J. Wilson, and R. J. Freund, "Chapter 7 - linear regression," in *Statistical Methods (Fourth Edition)*, D. L. Mohr, W. J. Wilson, and R. J. Freund, Eds. Academic Press, pp. 301–349. ISBN 978-0-12-823043-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128230435000072 [Page 55.]

[42] G. Blom, J. Enger, G. Englund, J. Grandell, and L. Holst, *Sannolikhetsteori och statistikteori med tillämpningar*, 6th ed. Studentlitteratur. ISBN 978-91-44-12356-1 [Page 55.]

[43] M. Sedlak. Lazy SMP in cheng - TalkChess.com. [Online]. Available: https://www.talkchess.com/forum/viewtopic.php?t=55188 [Page 62.]

[44] ZarsuMcAwesome and DaveM121. What does the task in task manager called CTF loader actually do? [Online]. Available: https://answers.mi

crosoft.com/en-us/windows/forum/all/what-does-the-task-in-task-man
ager-called-ctf/629689ae-ba70-41de-9b4c-ad438a63e384 [Page 66.]

Appendix A

Testing Scripts

A.1 Time-to-depth

```python
import sys
import subprocess
import time
from colorama import Fore, Style


def run_crafty(first_pos, last_pos, nr_of_threads):
    # Open logfile and write the current date and time to the first new line
    log_file = open(
        (f"log_file_{first_pos}to{last_pos}_"
         f"{nr_of_threads}threads.txt"), "a")
    now = time.localtime()
    date_time_str = time.strftime("%Y-%m-%d %H:%M:%S", now)
    log_file.write(f"{date_time_str}\n")

    # Write how many threads are used to the log
    log_file.write(f"Searching with {nr_of_threads} thread(s)\n")

    # How many runs will need to be done
    final_run = (last_pos + 1 - first_pos) * 50
    completed_run = 0
```

```python

    # Go through all positions
    for pos_nr in range(first_pos, last_pos + 1):
        # Write what game-file is currently being
         ↪  run
        log_file.write(f"\nGame{pos_nr}.txt:\n")

        # Run Crafty 50 times
        for _ in range(50):
            print(f"{Fore.BLUE}Executing
             ↪  ./crafty{Style.RESET_ALL}")
            process = subprocess.Popen(
                [
                    "./crafty",
                    "ponder=off",
                    "hash=256m",
                    "hashp=64m",
                    "sd=25",
                    "st=120",
                    f"smpmt={nr_of_threads}",
                    f"read=Game{pos_nr}.txt",
                ],
                stdin=subprocess.PIPE,
                stdout=subprocess.PIPE,
                stderr=subprocess.PIPE,
                universal_newlines=True,
            )

            while True:
                output = process.stdout.readline()
                # Analyse output:
                # 'machine has' -> send go,
                # 'time used:' -> log time & send
                 ↪  end
                if output:
                    if "machine has" in output:
                        print(
```

```python
                        f"{Fore.BLUE}Sending 'go' to crafty{Style.RESET_ALL}")
                    process.stdin.write("go\n")
                    process.stdin.flush()
                elif "time used:" in output:
                    log_file.write(
                        f"\t{output.strip()}\n"
                    )
                    print(
                        f"{Fore.BLUE}Sending 'end' to crafty{Style.RESET_ALL}")
                    process.stdin.write("end\n")
                    process.stdin.flush()
                    print(
                        f"{Fore.BLUE}Crafty process is terminating{Style.RESET_ALL}"
                    )
                    break

        # Wait until Crafty successfully exits
        rc = process.poll()
        while rc != 0:
            print(
                f"{Fore.RED}Return code from Crafty is {rc}, waiting...{Style.RESET_ALL}"
            )
            time.sleep(0.5)
            rc = process.poll()
        print(
            f"{Fore.BLUE}Crafty process successfully terminated!{Style.RESET_ALL}"
        )
```

```
            # Print the progress so far
            completed_run += 1
            print(
                f"{Fore.CYAN}{completed_run} out of {final_run} done{Style.RESET_ALL}"
            )

        log_file.flush()

    # Close the log file
    log_file.close()

    # Print a green message when the loop is done
    print(f"{Fore.GREEN}Final run of crafty has been completed!{Style.RESET_ALL}")


if __name__ == "__main__":
    if len(sys.argv) != 4:
        print("Usage: python ttd_script.py <first_pos> <last_pos> <nr_of_threads>")
        sys.exit(1)

    first_pos = int(sys.argv[1])
    last_pos = int(sys.argv[2])
    nr_of_threads = int(sys.argv[3])

    run_crafty(first_pos, last_pos, nr_of_threads)
```

A.2 Cutechess Matchmaking

```python
import sys
import subprocess
import os
import select
from colorama import Fore, Style


def run_crafty(number_of_threads):
    crafty_types = ["lazy", "ybwc"]
    opponents = ["Amoeba", "Atlas", "Chess22k", "Clovis"]

    for crafty_type in crafty_types:
        for opponent in opponents:
            command = [
                "./cutechess-cli",
                "-engine",
                f"conf=crafty-{crafty_type}",
                f"option.cores={number_of_threads}",
                "-engine",
                f"conf={opponent}",
                "-each",
                "tc=0/180",
                "-games",
                "30",
                "-pgnout",
                f"{crafty_type}_blitz_{number_of_threads}core
            ]

            process = subprocess.Popen(
                command,
                stdin=subprocess.PIPE,
                stdout=subprocess.PIPE,
                stderr=subprocess.PIPE,
                universal_newlines=True,
```

```
            )

            # Use select to read stdout and stderr
            while True:
                reads = [process.stdout, process.stderr]
                ready_to_read, _, _ = select.select(reads, [], [])

                for stream in ready_to_read:
                    output = stream.readline()
                    if output:
                        if stream == process.stdout:
                            print(output.strip())
                            if "Finished match" in output:
                                break
                        elif stream == process.stderr:
                            print(
                                f"{Fore.RED}Error: {output.strip()}{Style.RESET_ALL}

                if process.poll() is not None:
                    break

            process.stdout.close()
            process.stderr.close()
            process.wait()


if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python elo_script.py <number_of_threads>")
        sys.exit(1)

    try:
```

```
        number_of_threads = int(sys.argv[1])
        run_crafty(number_of_threads)
    except ValueError:
        print("Please provide a valid integer for
         ↪  the number of threads.")
        sys.exit(1)

```

A.3 Win-rate

```python
import sys


def count_sequences_and_results(filename):
    sequence_count = 0
    crafty_white_count = 0
    crafty_black_count = 0
    white_results = {"wins": 0, "losses": 0, "draws": 0}
    black_results = {"wins": 0, "losses": 0, "draws": 0}
    total_results = {"wins": 0, "losses": 0, "draws": 0}

    with open(filename, "r") as file:
        lines = file.readlines()

        for i in range(len(lines) - 2):
            if (
                lines[i].startswith("[White ")
                and lines[i + 1].startswith("[Black ")
                and lines[i + 2].startswith("[Result ")
            ):
                sequence_count += 1

                # Check for "crafty" in the "[White " line
                if "crafty" in lines[i]:
                    crafty_white_count += 1
                    if "1-0" in lines[i + 2]:
                        white_results["wins"] += 1
                        total_results["wins"] += 1
                    elif "0-1" in lines[i + 2]:
                        white_results["losses"] += 1
```

```
                        total_results["losses"] += 1
                    elif "1/2-1/2" in lines[i + 2]:
                        white_results["draws"] += 1
                        total_results["draws"] += 1

                # Check for "crafty" in the "[Black " line
                if "crafty" in lines[i + 1]:
                    crafty_black_count += 1
                    if "1-0" in lines[i + 2]:
                        black_results["losses"] += 1
                        total_results["losses"] += 1
                    elif "0-1" in lines[i + 2]:
                        black_results["wins"] += 1
                        total_results["wins"] += 1
                    elif "1/2-1/2" in lines[i + 2]:
                        black_results["draws"] += 1
                        total_results["draws"] += 1

    return (
        sequence_count,
        crafty_white_count,
        crafty_black_count,
        white_results,
        black_results,
        total_results,
    )


def calculate_win_rate(results):
    total_games = results["wins"] + results["losses"] + results["draws"]
    if total_games == 0:
        return 0.0  # Avoid division by zero
    total_score = results["wins"] + (results["draws"] * 0.5)
```

```python
    return total_score / total_games


if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python script.py <algorithm>")
        sys.exit(1)

    algorithm = sys.argv[1]
    opponents = ["amoeba", "atlas", "chess22k", "clovis"]
    core_counts = [1, 2, 4]

    for core_count in core_counts:
        for opponent in opponents:
            filename = f"{algorithm}_{core_count}core_{opponent}.txt"

            (
                sequence_count,
                crafty_white_count,
                crafty_black_count,
                white_results,
                black_results,
                total_results,
            ) = count_sequences_and_results(filename)

            total_win_rate = calculate_win_rate(total_results)
            white_win_rate = calculate_win_rate(white_results)
            black_win_rate = calculate_win_rate(black_results)

            with open("results.txt", "a") as result_file:
                result_file.write(f"{filename}:")
```

```
                result_file.write(f"\n\tTotal Results:\n")
                result_file.write(f"\t\tMatches: {sequence_count}\n")
                result_file.write(f"\t\tWins: {total_results['wins']}\n")
                result_file.write(f"\t\tLosses: {total_results['losses']}\n")
                result_file.write(f"\t\tDraws: {total_results['draws']}\n")
                result_file.write(f"\t\tWin Rate: {total_win_rate:.3f}\n")

                result_file.write(f"\n\tWhite Results:\n")
                result_file.write(f"\t\tMatches: {crafty_white_count}\n")
                result_file.write(f"\t\tWins: {white_results['wins']}\n")
                result_file.write(f"\t\tLosses: {white_results['losses']}\n")
                result_file.write(f"\t\tDraws: {white_results['draws']}\n")
                result_file.write(f"\t\tWin Rate: {white_win_rate:.3f}\n")

                result_file.write(f"\n\tBlack Results:\n")
                result_file.write(f"\t\tMatches: {crafty_black_count}\n")
                result_file.write(f"\t\tWins: {black_results['wins']}\n")
                result_file.write(f"\t\tLosses: {black_results['losses']}\n")
                result_file.write(f"\t\tDraws: {black_results['draws']}\n")
                result_file.write(f"\t\tWin Rate: {black_win_rate:.3f}\n\n")

```

A.4 Elo Rating

```python
import sys


def calculate_elo(filename, crafty_rating, opponent_rating, k_factor):
    with open(filename, "r") as file:
        lines = file.readlines()

        for i in range(len(lines) - 2):
            if (
                lines[i].startswith("[White ")
                and lines[i + 1].startswith("[Black ")
                and lines[i + 2].startswith("[Result ")
            ):

                expected_score = 1 / (
                    1 + 10 ** ((opponent_rating - crafty_rating) / 400)
                )

                # Check for "crafty" in the "[White " line
                if "crafty" in lines[i]:
                    if "1-0" in lines[i + 2]:
                        actual_score = 1
                    elif "0-1" in lines[i + 2]:
                        actual_score = 0
                    elif "1/2-1/2" in lines[i + 2]:
                        actual_score = 0.5

                # Check for "crafty" in the "[Black " line
                if "crafty" in lines[i + 1]:
                    if "1-0" in lines[i + 2]:
                        actual_score = 0
```

```
                    elif "0-1" in lines[i + 2]:
                        actual_score = 1
                    elif "1/2-1/2" in lines[i + 2]:
                        actual_score = 0.5

                crafty_rating += k_factor * (actual_score - expected_score)

    return crafty_rating


if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: python elo_rating_script.py <algorithm> <k-factor>")
        sys.exit(1)

    algorithm = sys.argv[1]
    k_factor = int(sys.argv[2])
    opponents = ["amoeba", "atlas", "chess22k", "clovis"]
    opponent_ratings = [3043, 2916, 3117, 2864]
    core_counts = [1, 2, 4]

    for core_count in core_counts:
        crafty_elo = 2953

        for index in range(4):
            filename = (f"{algorithm}_{core_count}core_"
                        f"{opponents[index]}.txt")
            crafty_elo = calculate_elo(
                filename, crafty_elo, opponent_ratings[index], k_factor
            )

        with open("elo_results.txt", "a") as result_file:
```

```
                result_file.write(
                    f"Elo rating of {algorithm} with {core_count} core(s) using K-factor {k_factor}: {int(crafty_elo)}\n"
                )

    with open("elo_results.txt", "a") as result_file:
        result_file.write("\n")

```

Appendix B

GameXX.txt Files

B.1 Game1.txt

```
[Event "CCRL 40/15"]
[Site "CCRL"]
[Date "2024.09.08"]
[Round "943.4.981"]
[White "Stockfish 17 64-bit 4CPU"]
[Black "Dragon by Komodo 3.3 64-bit 4CPU"]
[Result "1/2-1/2"]
[ECO "C07"]
[Opening "French"]
[Variation "Tarrasch, open variation"]
[PlyCount "132"]
[WhiteElo "3639"]
[BlackElo "3626"]

1. e4 {+0.00/1 0s} e6 {+0.00/1 0s} 2. d4 {+0.00/1 0s} d5 {+0.00/1 0s} 3. Nd2
{+0.00/1 0s} c5 {+0.00/1 0s} 4. exd5 {+0.00/1 0s} Qxd5 {+0.00/1 0s} 5. Ngf3
{+0.00/1 0s} cxd4 {+0.00/1 0s} 6. Bc4 {+0.00/1 0s} Qd6 {+0.00/1 0s} 7. O-O
{+0.00/1 0s} Nf6 {+0.00/1 0s} 8. Nb3 {+0.00/1 0s} Nc6 {+0.00/1 0s} 9. Nbxd4
{+0.09/43 96s} Nxd4 {(Nxd4) -0.11/27 16s} 10. Nxd4 {(Nxd4) +0.05/37 16s} Be7
{(Be7) -0.12/29 18s} 11. c3 {(c3) +0.05/43 69s} O-O {(O-O) -0.08/28 19s} 12.
Re1 {(Re1) +0.03/38 16s} Qc7 {(Qc7) -0.06/32 61s} 13. Qe2 {(Bd3) +0.00/47 59s}
e5 {(e5) +0.00/31 22s} 14. Bb3 {(Bb3) +0.04/40 11s} exd4 {(exd4) +0.00/33 19s}
15. Qxe7 {(Qxe7) +0.04/44 17s} Bd7 {(Bd7) +0.00/36 17s} 16. Bd2 {(Bd2) +0.04/47
22s} Rae8 {(Rae8) +0.00/35 27s} 17. Qb4 {(Qb4) +0.04/47 18s} Ng4 {(Ng4)
+0.00/39 25s} 18. f4 {(f4) +0.02/45 16s} d3 {(d3) +0.00/40 25s} 19. Qc4 {(Rxe8)
+0.00/45 23s} Bc6 {(Bc6) +0.00/37 14s} 20. h3 {(h3) +0.05/41 19s} Qb6+ {(Qb6)
+0.00/39 14s} 21. Qd4 {(Qd4) +0.06/41 19s} Nf6 {(Nf6) +0.00/42 23s} 22. Be3
{(Qxb6) +0.08/43 17s} Nd5 {(Qb5) +0.00/39 14s} 23. Qxb6 {(Qxb6) +0.05/49 17s}
axb6 {(axb6) +0.00/41 14s} 24. Bxd5 {(Bxd5) +0.00/54 23s} Bxd5 {(Bxd5) +0.00/43
17s}
```

B.2 Game2.txt

```
[Event "CCRL 40/15"]
[Site "CCRL"]
[Date "2024.09.07"]
[Round "943.4.972"]
[White "Alexandria 7.0.0 64-bit 4CPU"]
[Black "Stockfish 17 64-bit 4CPU"]
[Result "1/2-1/2"]
[ECO "C88"]
[Opening "Ruy Lopez"]
[Variation "closed, 7...O-O"]
[PlyCount "130"]
[WhiteElo "3600"]
[BlackElo "3639"]

1. e4 {+0.00/1 0s} e5 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} Nc6 {+0.00/1 0s} 3. Bb5
{+0.00/1 0s} a6 {+0.00/1 0s} 4. Ba4 {+0.00/1 0s} Nf6 {+0.00/1 0s} 5. O-O
{+0.00/1 0s} Be7 {+0.00/1 0s} 6. Re1 {+0.00/1 0s} b5 {+0.00/1 0s} 7. Bb3
{+0.00/1 0s} O-O {+0.00/1 0s} 8. d3 {+0.00/1 0s} d6 {+0.00/1 0s} 9. a4
{+0.40/34 44s} Bd7 {(Na5) +0.00/44 157s} 10. c3 {(Bd2) +0.00/35 48s} Na5 {(Na5)
+0.00/48 12s} 11. Bc2 {(Ba2) +0.23/36 30s} c5 {(c5) +0.00/46 22s} 12. Nbd2
{(Nbd2) +0.00/39 24s} Qc7 {(Qc7) +0.00/38 28s} 13. Nf1 {(Nf1) +0.00/34 21s}
Rfb8 {(Rfb8) +0.01/42 55s} 14. axb5 {(axb5) +0.11/34 29s} axb5 {(axb5) +0.00/43
17s} 15. Ne3 {(h3) +0.00/39 38s} Bf8 {(Bd8) +0.01/36 17s} 16. h3 {(h3) +0.00/39
26s} Nc4 {(Nc4) +0.00/41 27s} 17. Rxa8 {(Rxa8) +0.00/37 17s} Nxe3 {(Nxe3)
+0.00/39 17s} 18. Bxe3 {(Bxe3) +0.00/38 24s} Rxa8 {(Rxa8) +0.00/36 22s} 19. d4
{(d4) +0.00/41 22s} h6 {(h6) +0.00/36 22s} 20. Nd2 {(Qe2) +0.00/39 20s} Ra2
{(Qa5) +0.00/48 24s} 21. Qb1 {(Qb1) +0.00/42 27s} Ra6 {(Ra6) +0.00/32 18s} 22.
Nf1 {(Nf1) +0.00/45 26s} g6 {(Qa7) +0.05/37 24s} 23. Nh2 {(Qd1) +0.00/38 30s}
Bg7 {(Bg7) +0.13/35 17s} 24. Nf3 {(Nf3) +0.00/40 57s} Kh7 {(Nh7) +0.04/41 79s}
```

B.3 Game3.txt

```
[Event "CCRL 40/15"]
[Site "CCRL"]
[Date "2024.09.07"]
[Round "943.4.974"]
[White "Berserk 13 64-bit 4CPU"]
[Black "Stockfish 17 64-bit 4CPU"]
[Result "1/2-1/2"]
[ECO "E11"]
[Opening "Bogo-Indian defence, Gruenfeld variation"]
[PlyCount "145"]
[WhiteElo "3614"]
[BlackElo "3639"]

1. d4 {+0.00/1 0s} Nf6 {+0.00/1 0s} 2. c4 {+0.00/1 0s} e6 {+0.00/1 0s} 3. Nf3
{+0.00/1 0s} Bb4+ {+0.00/1 0s} 4. Nbd2 {+0.00/1 0s} O-O {+0.00/1 0s} 5. a3
{+0.00/1 0s} Be7 {+0.00/1 0s} 6. e4 {+0.00/1 0s} d5 {+0.00/1 0s} 7. e5 {+0.00/1
0s} Nfd7 {+0.00/1 0s} 8. b4 {+0.00/1 0s} a5 {+0.00/1 0s} 9. Rb1 {+0.25/33 21s}
axb4 {(axb4) -0.04/44 63s} 10. axb4 {(axb4) +0.12/38 24s} b6 {(b6) -0.01/38
29s} 11. cxd5 {(b5) +0.24/37 19s} exd5 {(exd5) +0.00/39 21s} 12. Qb3 {(Qb3)
+0.17/43 60s} c6 {(c6) +0.00/53 27s} 13. b5 {(b5) +0.11/45 48s} Re8 {(Re8)
+0.00/44 25s} 14. bxc6 {(bxc6) +0.02/41 26s} Nxc6 {(Nxc6) +0.00/44 27s} 15. Bb5
{(Bb5) +0.01/40 12s} Ndb8 {(Ndb8) -0.01/39 24s} 16. O-O {(O-O) +0.28/37 15s}
Bf5 {(Bf5) -0.07/43 20s} 17. Rb2 {(Bd3) +0.03/39 33s} Na5 {(Na5) +0.00/36 25s}
18. Qe3 {(Qe3) +0.02/41 18s} Nbc6 {(Nbc6) +0.00/45 18s} 19. h3 {(Bd3) +0.08/41
22s} Qc8 {(Bd7) +0.00/39 26s} 20. Qf4 {(Qf4) +0.02/33 16s} h6 {(h6) +0.00/40
33s} 21. Rd1 {(Rd1) +0.05/34 17s} Bg6 {(Bg6) +0.02/36 19s} 22. Qg3 {(Kh2)
+0.00/38 29s} Rd8 {(Bf8) +0.09/37 30s} 23. Kh2 {(Re1) +0.00/40 66s} Bf8 {(Bf8)
+0.02/43 62s} 24. Bf1 {(Be2) -0.01/40 75s} Nb4 {(Qc7) +0.19/35 28s}
```

B.4 Game4txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.08”]
[Round ”943.4.976”]
[White ”Caissa 1.20 64-bit 4CPU”]
[Black ”Stockfish 17 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”B98”]
[Opening ”Sicilian”]
[Variation ”Najdorf variation”]
[PlyCount ”122”]
[WhiteElo ”3607”]
[BlackElo ”3639”]

1. e4 {+0.00/1 0s} c5 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} d6 {+0.00/1 0s} 3. d4
{+0.00/1 0s} Nf6 {+0.00/1 0s} 4. Nc3 {+0.00/1 0s} cxd4 {+0.00/1 0s} 5. Nxd4
{+0.00/1 0s} a6 {+0.00/1 0s} 6. Bg5 {+0.00/1 0s} e6 {+0.00/1 0s} 7. f4 {+0.00/1
0s} Be7 {+0.00/1 0s} 8. Qf3 {+0.00/1 0s} Qc7 {+0.00/1 0s} 9. O-O-O {+0.23/32
15s} h6 {(Nbd7) -0.40/41 60s} 10. Bh4 {(Bh4) +0.79/34 25s} Nbd7 {(Nbd7)
-0.32/35 18s} 11. Be2 {(Be2) +0.47/37 23s} b5 {(b5) -0.23/48 27s} 12. Bxf6
{(Bxf6) +0.43/39 30s} Nxf6 {(Nxf6) -0.24/44 21s} 13. e5 {(e5) +0.39/39 30s} Bb7
{(Bb7) -0.27/45 22s} 14. Qg3 {(Qg3) +0.37/40 18s} dxe5 {(dxe5) -0.21/44 27s}
15. fxe5 {(fxe5) +0.38/38 5s} Ne4 {(Ne4) -0.20/42 25s} 16. Nxe6 {(Nxe6)
+0.28/42 28s} Qxc3 {(Qxc3) -0.17/43 20s} 17. Nxg7+ {(Nxg7) +0.26/41 17s} Kf8
{(Kf8) -0.19/28 0s} 18. Ne6+ {(Ne6) +0.23/35 5s} Ke8 {(Ke8) -0.16/42 21s} 19.
Qxc3 {(Qxc3) +0.29/42 18s} Nxc3 {(Nxc3) -0.17/45 21s} 20. Nc7+ {(Nc7) +0.44/40
4s} Kf8 {(Kf8) -0.19/24 0s} 21. bxc3 {(bxc3) +0.31/38 5s} Rc8 {(Ba3) -0.18/53
104s} 22. Nd5 {(Nd5) +0.26/38 7s} Ba3+ {(Ba3) -0.15/38 13s} 23. Kb1 {(Kb1)
+0.31/37 4s} Bxd5 {(Bxd5) -0.17/41 19s} 24. Rxd5 {(Rxd5) +0.38/37 14s} Rg8
{(Rg8) -0.16/43 18s}
```

B.5 Game5.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.16”]
[Round ”944.3.957”]
[White ”Stockfish 17 64-bit 4CPU”]
[Black ”Dragon by Komodo 3.3 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”A15”]
[Opening ”English opening”]
[PlyCount ”137”]
[WhiteElo ”3639”]
[BlackElo ”3626”]

1. Nf3 {+0.00/1 0s} Nf6 {+0.00/1 0s} 2. c4 {+0.00/1 0s} g6 {+0.00/1 0s} 3. g3
{+0.00/1 0s} Bg7 {+0.00/1 0s} 4. Bg2 {+0.00/1 0s} O-O {+0.00/1 0s} 5. O-O
{+0.00/1 0s} d6 {+0.00/1 0s} 6. d4 {+0.00/1 0s} Nbd7 {+0.00/1 0s} 7. Qc2
{+0.00/1 0s} e5 {+0.00/1 0s} 8. Rd1 {+0.00/1 0s} Qe7 {+0.00/1 0s} 9. Nc3
{+0.42/38 52s} Re8 {(c6) -0.44/30 39s} 10. e4 {(e4) +0.47/37 26s} exd4 {(exd4)
-0.42/31 32s} 11. Nxd4 {(Nxd4) +0.44/36 16s} Ne5 {(Nc5) -0.39/29 22s} 12. Bg5
{(Bf4) +0.39/43 159s} c6 {(c6) -0.33/31 23s} 13. b3 {(b3) +0.40/39 18s} h6
{(h6) -0.33/31 38s} 14. Bf4 {(Be3) +0.18/45 229s} Nh7 {(Nh7) -0.32/31 58s} 15.
Be3 {(Be3) +0.17/41 26s} Bg4 {(Bg4) -0.23/32 20s} 16. Rf1 {(Rf1) +0.16/39 13s}
c5 {(c5) -0.18/31 19s} 17. Nde2 {(Nde2) +0.15/40 18s} Nf3+ {(Nf3) -0.20/31 20s}
18. Kh1 {(Kh1) +0.17/37 11s} Bxc3 {(Bxc3) -0.18/32 27s} 19. Qxc3 {(Qxc3)
+0.16/39 11s} Nhg5 {(Nhg5) -0.22/32 15s} 20. Qc2 {(Qc2) +0.12/38 17s} Nxe4
{(Nxe4) -0.16/31 20s} 21. h3 {(h3) +0.17/39 16s} Bf5 {(Bf5) -0.11/31 13s} 22.
Qb2 {(Qb2) +0.12/38 15s} Neg5 {(Neg5) -0.17/34 39s} 23. Nf4 {(Nf4) +0.12/38
15s} Be4 {(Be4) -0.06/34 43s} 24. Rad1 {(Rad1) +0.11/40 21s} Ne6 {(Ne6)
-0.10/34 17s}
```

B.6 Game6.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.08”]
[Round ”943.4.993”]
[White ”Stockfish 17 64-bit 4CPU”]
[Black ”Obsidian 13.0 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”C11”]
[Opening ”French”]
[Variation ”Burn variation”]
[PlyCount ”121”]
[WhiteElo ”3639”]
[BlackElo ”3612”]

1. e4 {+0.00/1 0s} e6 {+0.00/1 0s} 2. d4 {+0.00/1 0s} d5 {+0.00/1 0s} 3. Nc3
{+0.00/1 0s} Nf6 {+0.00/1 0s} 4. Bg5 {+0.00/1 0s} dxe4 {+0.00/1 0s} 5. Nxe4
{+0.00/1 0s} Be7 {+0.00/1 0s} 6. Bxf6 {+0.00/1 0s} Bxf6 {+0.00/1 0s} 7. Nf3
{+0.00/1 0s} Nd7 {+0.00/1 0s} 8. Bd3 {+0.00/1 0s} c5 {+0.00/1 0s} 9. Nxf6+
{+0.18/41 61s} Nxf6 {(Nxf6) -0.20/34 46s} 10. dxc5 {(dxc5) +0.18/38 26s} Qa5+
{(Qa5) -0.26/34 33s} 11. Qd2 {(Qd2) +0.19/39 17s} Qxc5 {(Qxc5) -0.26/34 15s}
12. O-O-O {(O-O-O) +0.20/41 24s} O-O {(O-O) -0.27/34 20s} 13. Qg5 {(Qg5)
+0.22/39 31s} Nd5 {(Qxg5) -0.25/35 34s} 14. Qh4 {(Qh4) +0.17/39 63s} h6 {(h6)
-0.28/35 43s} 15. Qd4 {(Qd4) +0.15/36 18s} Qxd4 {(Qxd4) -0.25/36 72s} 16. Nxd4
{(Nxd4) +0.09/37 23s} Nf6 {(Nf6) -0.25/34 18s} 17. Be2 {(Be2) +0.17/43 22s} Rb8
{(Rb8) -0.24/33 16s} 18. Rhe1 {(Rhe1) +0.13/44 33s} b6 {(b6) -0.28/36 25s} 19.
Nc6 {(Nc6) +0.13/42 24s} Rb7 {(Rb7) -0.28/36 14s} 20. g3 {(Ne5) +0.11/44 128s}
Rc7 {(Rc7) -0.23/37 14s} 21. Ne5 {(Ne5) +0.06/37 14s} Bb7 {(Bb7) -0.32/38 40s}
22. Rd4 {(Rd4) +0.09/42 25s} Ne4 {(Bd5) -0.32/37 73s} 23. f4 {(f4) +0.21/38
44s} Rfc8 {(Rfc8) -0.36/33 22s} 24. c3 {(c3) +0.18/35 10s} g6 {(Kf8) -0.34/36
18s}
```

B.7 Game7.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.08”]
[Round ”943.5.1”]
[White ”Stockfish 17 64-bit 4CPU”]
[Black ”RubiChess 20240817 64-bit 4CPU”]
[Result ”1-0”]
[ECO ”C11”]
[Opening ”French”]
[Variation ”Burn variation”]
[PlyCount ”162”]
[WhiteElo ”3639”]
[BlackElo ”3600”]

1. e4 {+0.00/1 0s} e6 {+0.00/1 0s} 2. d4 {+0.00/1 0s} d5 {+0.00/1 0s} 3. Nc3
{+0.00/1 0s} Nf6 {+0.00/1 0s} 4. Bg5 {+0.00/1 0s} dxe4 {+0.00/1 0s} 5. Nxe4
{+0.00/1 0s} Be7 {+0.00/1 0s} 6. Bxf6 {+0.00/1 0s} gxf6 {+0.00/1 0s} 7. Nf3
{+0.00/1 0s} b6 {+0.00/1 0s} 8. Bc4 {+0.00/1 0s} Bb7 {+0.00/1 0s} 9. Qe2
{+0.73/33 29s} c6 {(Nd7) -0.59/27 33s} 10. O-O {(O-O-O) +0.76/35 19s} O-O
{(O-O) -0.69/29 44s} 11. Bb3 {(Rad1) +0.76/43 221s} Kh8 {(Kh8) -0.60/27 18s}
12. c4 {(Rad1) +0.54/39 45s} Rg8 {(Rg8) -0.53/30 75s} 13. Rad1 {(Ng3) +0.58/37
23s} Na6 {(Na6) -0.55/26 14s} 14. Ng3 {(Ng3) +0.58/44 123s} Qf8 {(Qf8) -0.65/33
38s} 15. d5 {(d5) +0.58/42 12s} cxd5 {(cxd5) -0.56/29 11s} 16. cxd5 {(cxd5)
+0.65/43 24s} Nc5 {(Nc5) -0.76/33 10s} 17. dxe6 {(dxe6) +0.61/41 16s} Nxb3
{(Nxb3) -0.71/30 10s} 18. axb3 {(axb3) +0.56/40 18s} fxe6 {(fxe6) -0.64/27 12s}
19. Qxe6 {(Qxe6) +0.56/42 21s} Bc5 {(Bc5) -0.60/29 10s} 20. Nh4 {(Rd7) +0.53/39
19s} Re8 {(Re8) -0.53/29 10s} 21. Qc4 {(Qc4) +0.53/39 12s} Re5 {(Re5) -0.48/28
8s} 22. Qf4 {(b4) +0.58/41 19s} Reg5 {(Reg5) -0.49/29 11s} 23. Nhf5 {(Nhf5)
+0.51/38 15s} Bc8 {(a5) -0.54/28 14s} 24. Kh1 {(Kh1) +0.86/37 18s} a5 {(a5)
-0.87/31 53s}
```

B.8 Game8.txt

```
[Event "CCRL 40/15"]
[Site "CCRL"]
[Date "2024.09.08"]
[Round "943.5.7"]
[White "Stockfish 17 64-bit 4CPU"]
[Black "Stockfish 20230613 64-bit 4CPU"]
[Result "1/2-1/2"]
[ECO "C19"]
[Opening "French"]
[Variation "Winawer, advance, poisoned pawn variation"]
[PlyCount "143"]
[WhiteElo "3639"]
[BlackElo "3641"]

1. e4 {+0.00/1 0s} e6 {+0.00/1 0s} 2. d4 {+0.00/1 0s} d5 {+0.00/1 0s} 3. Nc3
{+0.00/1 0s} Bb4 {+0.00/1 0s} 4. e5 {+0.00/1 0s} c5 {+0.00/1 0s} 5. a3 {+0.00/1
0s} Bxc3+ {+0.00/1 0s} 6. bxc3 {+0.00/1 0s} Ne7 {+0.00/1 0s} 7. Qg4 {+0.00/1
0s} O-O {+0.00/1 0s} 8. Bd3 {+0.00/1 0s} f5 {+0.00/1 0s} 9. exf6 {+0.68/40 35s}
Rxf6 {(Rxf6) -0.58/39 37s} 10. Bg5 {(Bg5) +0.71/38 23s} Rf7 {(Rf7) -0.63/38
21s} 11. Qh5 {(Qh5) +0.72/39 28s} g6 {(g6) -0.64/37 19s} 12. Qd1 {(Qd1)
+0.71/37 19s} Qa5 {(Nbc6) -0.65/42 36s} 13. Nf3 {(Nf3) +0.67/39 77s} c4 {(c4)
-0.65/39 22s} 14. Be2 {(Be2) +0.80/34 17s} Nbc6 {(Nbc6) -0.65/35 18s} 15. Bd2
{(Bd2) +0.71/32 18s} Nf5 {(Nf5) -0.67/33 15s} 16. O-O {(O-O) +0.80/38 84s} Bd7
{(Bd7) -0.63/35 16s} 17. a4 {(a4) +0.69/34 33s} Raf8 {(h6) -0.61/35 17s} 18.
Qc1 {(Rb1) +0.80/32 20s} Re7 {(h6) -0.60/35 76s} 19. Bg5 {(Bg5) +0.70/42 154s}
Qxc3 {(Qxc3) -0.55/35 13s} 20. Bxe7 {(Bxe7) +0.64/38 11s} Ncxe7 {(Ncxe7)
-0.44/33 15s} 21. Rd1 {(Rd1) +0.59/42 24s} Nc6 {(Nc6) -0.62/42 37s} 22. Qd2
{(Qd2) +0.61/33 20s} Qxd2 {(Qxd2) -0.60/39 14s} 23. Rxd2 {(Rxd2) +0.56/35 14s}
Nd6 {(Nd6) -0.59/38 16s} 24. c3 {(c3) +0.70/36 20s} Ne4 {(Ne4) -0.44/36 16s}
```

B.9 Game9.txt

```
[Event "CCRL 40/15"]
[Site "CCRL"]
[Date "2024.09.08"]
[Round "943.5.14"]
[White "Berserk 13 64-bit 4CPU"]
[Black "Stockfish 17 64-bit 4CPU"]
[Result "1/2-1/2"]
[ECO "C41"]
[Opening "Philidor"]
[Variation "exchange variation"]
[PlyCount "138"]
[WhiteElo "3614"]
[BlackElo "3639"]

1. e4 {+0.00/1 0s} e5 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} d6 {+0.00/1 0s} 3. d4
{+0.00/1 0s} exd4 {+0.00/1 0s} 4. Nxd4 {+0.00/1 0s} Nf6 {+0.00/1 0s} 5. Nc3
{+0.00/1 0s} Be7 {+0.00/1 0s} 6. g3 {+0.00/1 0s} Nc6 {+0.00/1 0s} 7. Bg2
{+0.00/1 0s} Bg4 {+0.00/1 0s} 8. Nde2 {+0.00/1 0s} Qc8 {+0.00/1 0s} 9. h3
{+0.98/30 14s} Bxe2 {(Bxe2) -1.08/39 60s} 10. Qxe2 {(Qxe2) +1.00/34 14s} Nd4
{(Nd4) -1.05/37 20s} 11. Qd2 {(Qd2) +0.96/34 17s} Ne6 {(Ne6) -0.96/36 20s} 12.
f4 {(f4) +1.10/35 27s} c6 {(c6) -0.95/32 20s} 13. Qf2 {(b3) +1.03/34 20s} O-O
{(O-O) -0.97/39 113s} 14. Be3 {(Be3) +1.05/35 19s} b5 {(b5) -0.94/31 14s} 15.
O-O {(O-O) +0.99/33 13s} b4 {(Qc7) -0.93/33 20s} 16. Ne2 {(Na4) +1.07/35 23s}
d5 {(d5) -0.80/36 18s} 17. e5 {(e5) +0.69/42 104s} Ne4 {(Ne4) -0.89/40 32s} 18.
Bxe4 {(Bxe4) +0.77/39 14s} dxe4 {(dxe4) -0.91/35 16s} 19. Qg2 {(Qg2) +0.83/39
22s} Qa6 {(Qa6) -0.96/38 27s} 20. Rae1 {(Rae1) +0.88/35 14s} Qc4 {(Qc4)
-0.95/38 49s} 21. Kh2 {(Kh2) +0.89/38 20s} Rfe8 {(Qxc2) -0.97/37 50s} 22. Nc1
{(Nc1) +0.90/34 17s} Rad8 {(Rad8) -0.95/32 11s} 23. Bg1 {(Bg1) +0.98/36 14s}
Bc5 {(Bc5) -0.90/35 18s} 24. Rxe4 {(Rxe4) +0.85/35 25s} Bxg1+ {(Bxg1) -0.89/34
17s}
```

B.10 Game10.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.09”]
[Round ”943.5.15”]
[White ”Stockfish 17 64-bit 4CPU”]
[Black ”Caissa 1.20 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”C42”]
[Opening ”Petrov”]
[Variation ”classical attack, Marshall variation”]
[PlyCount ”144”]
[WhiteElo ”3639”]
[BlackElo ”3607”]

1. e4 {+0.00/1 0s} e5 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} Nf6 {+0.00/1 0s} 3. Nxe5
{+0.00/1 0s} d6 {+0.00/1 0s} 4. Nf3 {+0.00/1 0s} Nxe4 {+0.00/1 0s} 5. d4
{+0.00/1 0s} d5 {+0.00/1 0s} 6. Bd3 {+0.00/1 0s} Bd6 {+0.00/1 0s} 7. O-O
{+0.00/1 0s} O-O {+0.00/1 0s} 8. c4 {+0.00/1 0s} c6 {+0.00/1 0s} 9. Re1
{+0.28/38 45s} Bf5 {(Bf5) -0.35/37 23s} 10. Qb3 {(Qb3) +0.16/39 67s} Na6 {(Qd7)
-0.32/38 15s} 11. cxd5 {(cxd5) +0.30/35 21s} cxd5 {(cxd5) -0.66/38 28s} 12. Nc3
{(Nc3) +0.29/36 25s} Nb4 {(Be6) -0.58/38 19s} 13. Bxe4 {(Bxe4) +0.34/34 18s}
dxe4 {(dxe4) -0.61/38 14s} 14. Bg5 {(Bg5) +0.33/40 30s} Qd7 {(Qb6) -0.54/40
40s} 15. Nxe4 {(Nxe4) +0.50/35 19s} Be6 {(Rae8) -0.67/41 21s} 16. Qa3 {(Qa3)
+0.47/39 25s} f6 {(f6) -0.58/39 25s} 17. Bd2 {(Bd2) +0.41/42 23s} a5 {(a5)
-0.58/35 4s} 18. Nxd6 {(Nxd6) +0.44/39 17s} Qxd6 {(Qxd6) -0.54/37 5s} 19. Bxb4
{(Bxb4) +0.40/39 24s} axb4 {(axb4) -0.46/36 7s} 20. Qd3 {(Qd3) +0.37/39 23s}
Bf7 {(Bf7) -0.53/37 18s} 21. Nh4 {(Nh4) +0.41/40 24s} g6 {(g6) -0.50/38 18s}
22. Nf3 {(Nf3) +0.36/40 28s} Ra5 {(Ra5) -0.49/37 16s} 23. h3 {(h3) +0.37/36
24s} Kg7 {(Kg7) -0.59/33 17s} 24. b3 {(b3) +0.38/37 28s} Rc8 {(Rfa8) -0.51/36
17s}
```

B.11 Game11.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.09”]
[Round ”943.5.19”]
[White ”Stockfish 17 64-bit 4CPU”]
[Black ”Chess System Tal 2.00 EAS 64-bit 4CPU”]
[Result ”1-0”]
[ECO ”C42”]
[Opening ”Petrov”]
[Variation ”classical attack, Mason variation”]
[PlyCount ”127”]
[WhiteElo ”3639”]
[BlackElo ”3577”]

1. e4 {+0.00/1 0s} e5 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} Nf6 {+0.00/1 0s} 3. Nxe5
{+0.00/1 0s} d6 {+0.00/1 0s} 4. Nf3 {+0.00/1 0s} Nxe4 {+0.00/1 0s} 5. d4
{+0.00/1 0s} d5 {+0.00/1 0s} 6. Bd3 {+0.00/1 0s} Be7 {+0.00/1 0s} 7. O-O
{+0.00/1 0s} O-O {+0.00/1 0s} 8. c4 {+0.00/1 0s} Nc6 {+0.00/1 0s} 9. cxd5
{+0.39/36 26s} Qxd5 {(Qxd5) -0.79/30 34s} 10. Nc3 {(Nc3) +0.54/37 21s} Nxc3
{(Nxc3) -0.76/31 26s} 11. bxc3 {(bxc3) +0.49/39 25s} Bg4 {(Bf5) -0.82/29 40s}
12. Rb1 {(Rb1) +0.46/34 19s} Rab8 {(Rab8) -0.73/29 19s} 13. Re1 {(Re1) +0.44/37
30s} Bxf3 {(Bxf3) -0.91/29 25s} 14. gxf3 {(Qxf3) +0.43/37 28s} Qd7 {(a6)
-0.77/23 30s} 15. d5 {(d5) +0.74/34 26s} Na5 {(Na5) -1.12/24 21s} 16. c4 {(c4)
+0.76/34 19s} Bc5 {(b6) -1.08/25 21s} 17. Kg2 {(Qc2) +0.87/35 19s} Rbe8 {(Rbe8)
-0.57/24 17s} 18. Bb2 {(Bb2) +0.81/35 32s} Rxe1 {(Rxe1) -1.16/27 27s} 19. Qxe1
{(Qxe1) +0.80/39 22s} b6 {(b6) -1.33/28 20s} 20. Qd2 {(Qc3) +0.88/36 22s} f6
{(f5) -1.48/26 35s} 21. Qc2 {(Qc2) +1.03/39 26s} f5 {(f5) -1.39/29 16s} 22. Re1
{(Re1) +1.16/38 21s} Nb7 {(Nb7) -1.46/29 18s} 23. Re6 {(Re6) +1.26/33 19s} Qf7
{(Qf7) -1.50/31 22s} 24. Qc3 {(Qc3) +1.38/34 19s} Nd6 {(Nd6) -1.62/32 21s}
```

B.12 Game12.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.09”]
[Round ”943.5.21”]
[White ”Stockfish 17 64-bit 4CPU”]
[Black ”Dragon by Komodo 3.3 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”C42”]
[Opening ”Petrov”]
[Variation ”classical attack, Mason variation”]
[PlyCount ”122”]
[WhiteElo ”3639”]
[BlackElo ”3626”]

1. e4 {+0.00/1 0s} e5 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} Nf6 {+0.00/1 0s} 3. Nxe5
{+0.00/1 0s} d6 {+0.00/1 0s} 4. Nf3 {+0.00/1 0s} Nxe4 {+0.00/1 0s} 5. d4
{+0.00/1 0s} d5 {+0.00/1 0s} 6. Bd3 {+0.00/1 0s} Be7 {+0.00/1 0s} 7. O-O
{+0.00/1 0s} O-O {+0.00/1 0s} 8. Re1 {+0.00/1 0s} Nc6 {+0.00/1 0s} 9. Bxe4
{+0.22/45 136s} dxe4 {(dxe4) -0.27/31 23s} 10. Rxe4 {(Rxe4) +0.19/36 16s} Bf5
{(Bf5) -0.27/33 45s} 11. Re3 {(Re1) +0.27/43 101s} Nb4 {(Nb4) -0.32/34 29s} 12.
Re2 {(Re2) +0.25/41 26s} c5 {(Bg4) -0.27/31 20s} 13. c3 {(c3) +0.28/34 18s} Nd3
{(Nd3) -0.27/31 19s} 14. Nbd2 {(Re3) +0.30/41 60s} Re8 {(Re8) -0.22/30 27s} 15.
Nf1 {(Nf1) +0.33/36 33s} Qd7 {(Qd7) -0.32/31 42s} 16. a4 {(h3) +0.28/32 14s}
Bf6 {(Rad8) -0.21/29 20s} 17. Be3 {(Be3) +0.32/39 41s} c4 {(b6) -0.25/30 38s}
18. Ne1 {(b3) +0.42/31 11s} b5 {(h6) -0.33/32 92s} 19. axb5 {(Ng3) +0.37/38
29s} Qxb5 {(Qxb5) -0.25/32 16s} 20. Qa4 {(Qa4) +0.36/40 34s} a6 {(Qxa4)
-0.22/36 18s} 21. Nxd3 {(Nxd3) +0.32/41 22s} Bxd3 {(Bxd3) -0.15/35 31s} 22.
Qxb5 {(Qxb5) +0.30/35 14s} axb5 {(axb5) -0.16/35 16s} 23. Ree1 {(Ree1) +0.28/40
16s} h5 {(h5) -0.13/36 14s} 24. h3 {(h3) +0.27/39 22s} Be7 {(Ra4) -0.16/35 16s}
```

B.13 Game13.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.19”]
[Round ”944.4.79”]
[White ”Stockfish 17 64-bit 4CPU”]
[Black ”Ethereal 14.25 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”B93”]
[Opening ”Sicilian”]
[Variation ”Najdorf, 6.f4”]
[PlyCount ”121”]
[WhiteElo ”3639”]
[BlackElo ”3598”]

1. e4 {+0.00/1 0s} c5 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} d6 {+0.00/1 0s} 3. d4
{+0.00/1 0s} cxd4 {+0.00/1 0s} 4. Nxd4 {+0.00/1 0s} Nf6 {+0.00/1 0s} 5. Nc3
{+0.00/1 0s} a6 {+0.00/1 0s} 6. f4 {+0.00/1 0s} e5 {+0.00/1 0s} 7. Nf3 {+0.00/1
0s} Nbd7 {+0.00/1 0s} 8. a4 {+0.00/1 0s} Be7 {+0.00/1 0s} 9. Bc4 {+0.13/40 36s}
Qa5 {(O-O) +0.01/30 15s} 10. Qe2 {+0.25/35 18s} b5 {(b5) +0.04/33 28s} 11. Ba2
{+0.15/39 35s} bxa4 {(bxa4) +0.05/33 16s} 12. O-O {+0.09/38 28s} O-O {(O-O)
+0.12/30 16s} 13. Kh1 {+0.06/37 26s} Qc7 {(exf4) +0.12/33 22s} 14. g4 {(Nd5)
+0.05/34 34s} a5 {(Bb7) +0.00/33 20s} 15. g5 {(g5) +0.57/37 24s} Ba6 {(Ba6)
-0.08/33 16s} 16. Qd1 {+0.60/36 18s} Nxe4 {(Nh5) -0.26/37 85s} 17. Nxe4 {(Nxe4)
+0.56/35 23s} Bxf1 {(Bxf1) -0.09/36 21s} 18. Qxf1 {(Qxf1) +0.53/35 18s} Qxc2
{(Qxc2) -0.13/33 21s} 19. Bd5 {(Bd5) +0.52/37 48s} Rab8 {(Nb6) -0.30/36 40s}
20. Be3 {(Be3) +0.28/47 67s} exf4 {(exf4) -0.26/36 21s} 21. Bd4 {(Bd4) +0.28/39
17s} a3 {(a3) -0.16/35 12s} 22. bxa3 {+0.23/40 22s} Nc5 {(Nc5) -0.22/35 11s}
23. Rc1 {(Rc1) +0.30/40 24s} Qd3 {(Qd3) -0.18/36 13s} 24. Qxd3 {(Qxd3) +0.26/42
55s} Nxd3 {(Nxd3) -0.22/35 10s}
```

B.14 Game14.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.09”]
[Round ”943.5.28”]
[White ”Lizard 10.5 64-bit 4CPU”]
[Black ”Stockfish 17 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”C44”]
[Opening ”Ponziani”]
[Variation ”Steinitz variation”]
[PlyCount ”157”]
[WhiteElo ”3599”]
[BlackElo ”3639”]

1. e4 {+0.00/1 0s} e5 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} Nc6 {+0.00/1 0s} 3. c3
{+0.00/1 0s} d5 {+0.00/1 0s} 4. Qa4 {+0.00/1 0s} f6 {+0.00/1 0s} 5. Bb5
{+0.00/1 0s} Ne7 {+0.00/1 0s} 6. exd5 {+0.00/1 0s} Qxd5 {+0.00/1 0s} 7. O-O
{+0.00/1 0s} e4 {+0.00/1 0s} 8. Ne1 {+0.00/1 0s} Bf5 {+0.00/1 0s} 9. f3
{-0.57/28 36s} exf3 {(exf3) +0.41/37 44s} 10. Rxf3 {(Nxf3) -0.57/30 31s} a6
{(a6) +0.59/35 25s} 11. Bf1 {(Bf1) -0.48/29 22s} Qd7 {(Bd7) +0.65/42 149s} 12.
d4 {(d4) -0.54/29 21s} Nd5 {(Nd5) +0.62/36 18s} 13. Nd2 {(Bd3) -0.48/29 29s}
O-O-O {(O-O-O) +0.70/33 17s} 14. Nc4 {(Nc4) -0.42/29 23s} h5 {(g5) +0.62/38
29s} 15. Nc2 {(Nc2) -0.33/28 43s} Be4 {(Re8) +0.63/42 122s} 16. Rf2 {(Rf2)
-0.37/27 22s} h4 {(h4) +0.59/37 16s} 17. N2e3 {(N2e3) -0.50/31 35s} h3 {(h3)
+0.64/37 23s} 18. g3 {(Nxd5) -0.37/31 37s} Kb8 {(Kb8) +0.59/35 29s} 19. Nxd5
{(Nxd5) -0.60/32 66s} Bxd5 {(Bxd5) +0.55/35 17s} 20. Qd1 {(Bd2) -0.70/31 59s}
Bxc4 {(Ne7) +0.81/37 57s} 21. Bxc4 {(Bxc4) -0.63/29 14s} Bc5 {(Bc5) +0.78/36
26s} 22. Be3 {(Be3) -0.54/28 18s} Ne5 {(Ne5) +0.82/34 19s} 23. Be2 {(Be2)
-0.74/27 21s} Bb6 {(Bb6) +0.84/35 13s} 24. Qc2 {(Qc2) -0.92/25 14s} Rde8 {(Nf7)
+0.82/33 16s}
```

B.15 Game15.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.09”]
[Round ”943.5.33”]
[White ”Stockfish 17 64-bit 4CPU”]
[Black ”Obsidian 13.0 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”C47”]
[Opening ”Four knights”]
[Variation ”Belgrade gambit”]
[PlyCount ”167”]
[WhiteElo ”3639”]
[BlackElo ”3612”]

1. e4 {+0.00/1 0s} e5 {+0.00/1 0s} 2. Nc3 {+0.00/1 0s} Nf6 {+0.00/1 0s} 3. Nf3
{+0.00/1 0s} Nc6 {+0.00/1 0s} 4. d4 {+0.00/1 0s} exd4 {+0.00/1 0s} 5. Nd5
{+0.00/1 0s} Be7 {+0.00/1 0s} 6. Bc4 {+0.00/1 0s} O-O {+0.00/1 0s} 7. O-O
{+0.00/1 0s} Nxe4 {+0.00/1 0s} 8. Nxd4 {+0.00/1 0s} Bc5 {+0.00/1 0s} 9. c3
{-0.13/44 36s} Ne7 {(Ne7) +0.05/33 41s} 10. Re1 {(Qd3) -0.11/43 27s} Nxd5
{(Nxd5) +0.14/34 16s} 11. Bxd5 {(Bxd5) -0.13/40 18s} Nf6 {(Nf6) +0.22/36 22s}
12. Bg5 {(Bg5) -0.18/39 19s} c6 {(c6) +0.26/36 17s} 13. Bb3 {(Bb3) -0.10/38
26s} d5 {(d5) +0.28/37 15s} 14. Bc2 {(Bc2) -0.06/38 22s} Qd6 {(Qd6) +0.12/36
29s} 15. Qd2 {(Qf3) -0.04/39 20s} Ne4 {(Ne4) +0.27/39 16s} 16. Bxe4 {(Bxe4)
-0.03/37 18s} dxe4 {(dxe4) +0.27/42 16s} 17. Rxe4 {(Rxe4) -0.05/37 22s} Qd5
{(Qd5) +0.27/42 17s} 18. Ree1 {(Rae1) -0.06/34 21s} Bxd4 {(Bxd4) +0.36/36 30s}
19. cxd4 {(cxd4) +0.00/31 23s} f6 {(f6) +0.25/37 26s} 20. Be3 {(Bf4) -0.05/32
27s} Rf7 {(Rd8) +0.18/38 97s} 21. a4 {(f3) -0.09/34 25s} a5 {(a5) +0.21/38 44s}
22. b4 {(b4) -0.15/34 23s} Be6 {(Be6) +0.16/40 35s} 23. h3 {(bxa5) -0.11/38
30s} Rd7 {(Rd7) +0.20/37 22s} 24. Rab1 {(bxa5) -0.10/38 35s} axb4 {(Qa2)
+0.21/42 73s}
```

B.16 Game16.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.09”]
[Round ”943.5.36”]
[White ”PlentyChess 2.1.0 64-bit 4CPU”]
[Black ”Stockfish 17 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”C53”]
[Opening ”Giuoco Piano”]
[Variation ”Bird's attack”]
[PlyCount ”122”]
[WhiteElo ”3607”]
[BlackElo ”3639”]

1. e4 {+0.00/1 0s} e5 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} Nc6 {+0.00/1 0s} 3. Bc4
{+0.00/1 0s} Bc5 {+0.00/1 0s} 4. c3 {+0.00/1 0s} Nf6 {+0.00/1 0s} 5. b4
{+0.00/1 0s} Bb6 {+0.00/1 0s} 6. d3 {+0.00/1 0s} d6 {+0.00/1 0s} 7. a4 {+0.00/1
0s} a5 {+0.00/1 0s} 8. b5 {+0.00/1 0s} Ne7 {+0.00/1 0s} 9. Nbd2 {+0.03/37 74s}
O-O {(O-O) +0.00/49 78s} 10. O-O {(O-O) +0.01/38 35s} Ng6 {(Ng6) +0.00/50 28s}
11. d4 {(d4) +0.01/39 31s} Bg4 {(Bg4) +0.00/52 26s} 12. Qc2 {(Qc2) +0.00/41
32s} Qd7 {(Qd7) +0.00/57 29s} 13. Re1 {(dxe5) +0.00/42 75s} Rfe8 {(Rae8)
+0.00/49 29s} 14. h3 {(Bb2) -0.04/35 19s} Bxf3 {(Bxf3) +0.05/39 21s} 15. Nxf3
{(Nxf3) +0.00/37 19s} exd4 {(exd4) +0.00/50 21s} 16. Bb2 {(cxd4) +0.00/39 33s}
dxc3 {(dxc3) +0.08/37 24s} 17. Qxc3 {(Qxc3) +0.00/39 20s} c6 {(c6) +0.03/38
32s} 18. bxc6 {(Qc2) +0.00/41 43s} bxc6 {(bxc6) +0.04/47 34s} 19. Qc2 {(Qc2)
+0.00/42 18s} d5 {(Ne5) +0.00/51 30s} 20. Rad1 {(Rad1) +0.00/40 21s} Qc7 {(Qc7)
+0.00/47 24s} 21. Bxf6 {(Bxf6) +0.00/42 20s} gxf6 {(gxf6) +0.00/50 21s} 22. Bb3
{(Ba2) +0.00/41 16s} Nf4 {(Nf4) +0.00/43 35s} 23. Kf1 {(Kf1) +0.00/43 35s} Qc8
{(Rab8) +0.00/60 23s} 24. exd5 {(exd5) +0.00/39 17s} Rxe1+ {(Rxe1) +0.00/48
20s}
```

B.17 Game17.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.09”]
[Round ”943.5.41”]
[White ”Stockfish 17 64-bit 4CPU”]
[Black ”RubiChess 20240817 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”C55”]
[Opening ”Two knights defence (Modern bishop's opening)”]
[PlyCount ”149”]
[WhiteElo ”3639”]
[BlackElo ”3600”]

1. e4 {+0.00/1 0s} e5 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} Nc6 {+0.00/1 0s} 3. Bc4
{+0.00/1 0s} Nf6 {+0.00/1 0s} 4. d3 {+0.00/1 0s} Be7 {+0.00/1 0s} 5. O-O
{+0.00/1 0s} O-O {+0.00/1 0s} 6. Re1 {+0.00/1 0s} d6 {+0.00/1 0s} 7. a4
{+0.00/1 0s} Na5 {+0.00/1 0s} 8. Ba2 {+0.00/1 0s} c5 {+0.00/1 0s} 9. c3
{+0.20/44 143s} Nc6 {(Nc6) -0.26/23 9s} 10. Na3 {(Na3) +0.18/40 40s} h6 {(a6)
-0.32/26 13s} 11. Bd2 {(Bd2) +0.22/38 27s} a6 {(a6) -0.32/26 10s} 12. h3 {(h3)
+0.18/36 26s} Bd7 {(Re8) -0.26/26 13s} 13. Nc4 {(Nc4) +0.19/35 46s} Be6 {(b5)
-0.21/27 15s} 14. b4 {(b4) +0.13/43 197s} b5 {(b5) -0.19/28 13s} 15. Ne3 {(Ne3)
+0.09/36 9s} cxb4 {(cxb4) -0.18/25 9s} 16. Bxe6 {(Bxe6) +0.07/43 50s} fxe6
{(fxe6) -0.19/25 11s} 17. axb5 {(axb5) +0.06/39 13s} axb5 {(axb5) -0.17/26 10s}
18. Rxa8 {(Rxa8) +0.04/39 20s} Qxa8 {(Qxa8) -0.16/27 14s} 19. Qb3 {(Qb3)
+0.03/38 12s} Qc8 {(Qc8) -0.15/28 16s} 20. cxb4 {(cxb4) +0.07/42 71s} Kh7
{(Kh7) -0.16/28 14s} 21. Rc1 {(Qc3) +0.08/41 12s} Qd7 {(Qd7) -0.12/26 26s} 22.
Qb2 {(Qb2) +0.06/35 22s} Rc8 {(Rc8) -0.11/26 12s} 23. Ra1 {(Bc3) +0.04/42 41s}
Nh5 {(Rf8) -0.08/30 26s} 24. g3 {(g3) +0.00/51 10s} Rf8 {(Rf8) -0.09/33 33s}
```

B.18 Game18.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.20”]
[Round ”944.4.109”]
[White ”Stockfish 17 64-bit 4CPU”]
[Black ”Berserk 13 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”A65”]
[Opening ”Benoni”]
[Variation ”6.e4”]
[PlyCount ”179”]
[WhiteElo ”3639”]
[BlackElo ”3614”]

1. d4 {+0.00/1 0s} Nf6 {+0.00/1 0s} 2. c4 {+0.00/1 0s} c5 {+0.00/1 0s} 3. d5
{+0.00/1 0s} e6 {+0.00/1 0s} 4. Nc3 {+0.00/1 0s} exd5 {+0.00/1 0s} 5. cxd5
{+0.00/1 0s} d6 {+0.00/1 0s} 6. e4 {+0.00/1 0s} g6 {+0.00/1 0s} 7. Nge2
{+0.00/1 0s} Bg7 {+0.00/1 0s} 8. Ng3 {+0.00/1 0s} O-O {+0.00/1 0s} 9. Be2
{+0.62/37 40s} b6 {(Re8) -0.68/36 87s} 10. h4 {(a4) +0.75/33 23s} h5 {(h5)
-0.53/33 21s} 11. e5 {(Bg5) +0.76/35 44s} dxe5 {(dxe5) -0.52/29 14s} 12. Bg5
{(Bg5) +0.83/34 18s} Nbd7 {(Nbd7) -0.49/37 83s} 13. a4 {(O-O) +0.72/48 293s}
Bb7 {(Bb7) -0.49/32 12s} 14. O-O {(O-O) +0.65/38 32s} Qc7 {(Qc7) -0.55/38 96s}
15. Qd2 {(Re1) +0.68/40 23s} Rae8 {(a6) -0.50/34 21s} 16. Nb5 {(Rad1) +0.83/30
12s} Qb8 {(Qb8) -0.40/30 11s} 17. d6 {(d6) +0.75/31 17s} Nh7 {(e4) -0.37/35
49s} 18. Be7 {(Be7) +0.93/32 13s} Bf6 {(Bf6) -0.33/33 10s} 19. Bxf8 {(Bxf8)
+0.88/33 17s} Rxf8 {(Rxf8) -0.62/34 44s} 20. Bc4 {(Bc4) +0.89/38 94s} Bxh4
{(Bxh4) -0.62/34 12s} 21. Nc7 {(Nc7) +0.89/35 35s} Bg5 {(Bg5) -0.56/34 13s} 22.
Qd3 {(Qd3) +0.78/31 15s} Kg7 {(Kg7) -0.58/36 13s} 23. Rad1 {(Rad1) +0.79/32
17s} Bd8 {(Bd8) -0.60/37 24s} 24. Rfe1 {(Rfe1) +0.85/38 55s} Qc8 {(Qc8)
-0.57/36 21s}
```

B.19 Game19.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.09”]
[Round ”943.5.49”]
[White ”Stockfish 17 64-bit 4CPU”]
[Black ”Torch v3 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”C77”]
[Opening ”Ruy Lopez”]
[Variation ”Treybal (Bayreuth) variation (exchange var. deferred)”]
[PlyCount ”142”]
[WhiteElo ”3639”]
[BlackElo ”3630”]

1. e4 {+0.00/1 0s} e5 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} Nc6 {+0.00/1 0s} 3. Bb5
{+0.00/1 0s} a6 {+0.00/1 0s} 4. Ba4 {+0.00/1 0s} Nf6 {+0.00/1 0s} 5. Bxc6
{+0.00/1 0s} dxc6 {+0.00/1 0s} 6. d3 {+0.00/1 0s} Bd6 {+0.00/1 0s} 7. Nbd2
{+0.00/1 0s} Be6 {+0.00/1 0s} 8. b3 {+0.00/1 0s} Nd7 {+0.00/1 0s} 9. O-O
{-0.19/50 102s} Qe7 {(Qe7) +0.37/37 24s} 10. Rb1 {(Nc4) -0.23/41 61s} O-O-O
{(b6) +0.34/42 102s} 11. Nc4 {(Nc4) -0.19/37 13s} Bxc4 {(Bxc4) +0.30/39 22s}
12. bxc4 {(bxc4) -0.23/38 22s} b6 {(b6) +0.27/40 35s} 13. c3 {(c3) -0.26/42
42s} Nb8 {(c5) +0.24/41 30s} 14. a4 {(a4) -0.32/35 37s} a5 {(a5) +0.31/35 15s}
15. Qb3 {(Re1) -0.37/34 31s} g6 {(g6) +0.30/38 53s} 16. h3 {(Re1) -0.22/44
117s} f6 {(f6) +0.26/40 76s} 17. Be3 {(Re1) -0.18/37 14s} c5 {(c5) +0.40/34
12s} 18. Qb5 {(Qb5) -0.22/36 15s} Rhe8 {(Rhe8) +0.19/41 55s} 19. Nd2 {(Rfd1)
-0.19/40 23s} f5 {(Qd7) +0.37/36 12s} 20. f3 {(f3) -0.14/36 14s} Qd7 {(Qd7)
+0.40/39 13s} 21. Rfd1 {(Nb3) -0.15/37 15s} Be7 {(Be7) +0.33/41 34s} 22. Nb3
{(Nb3) -0.11/37 14s} Bh4 {(Qc6) +0.29/43 38s} 23. Bf2 {(Bf2) -0.12/39 16s}
Bxf2+ {(Bxf2) +0.31/38 17s} 24. Kxf2 {(Kxf2) -0.09/39 15s} Re6 {(Re6) +0.23/41
58s}
```

B.20 Game20.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.09”]
[Round ”943.5.52”]
[White ”Alexandria 7.0.0 64-bit 4CPU”]
[Black ”Stockfish 17 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”C88”]
[Opening ”Ruy Lopez”]
[Variation ”closed, 7...O-O”]
[PlyCount ”186”]
[WhiteElo ”3600”]
[BlackElo ”3639”]

1. e4 {+0.00/1 0s} e5 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} Nc6 {+0.00/1 0s} 3. Bb5
{+0.00/1 0s} a6 {+0.00/1 0s} 4. Ba4 {+0.00/1 0s} Nf6 {+0.00/1 0s} 5. O-O
{+0.00/1 0s} Be7 {+0.00/1 0s} 6. Re1 {+0.00/1 0s} b5 {+0.00/1 0s} 7. Bb3
{+0.00/1 0s} O-O {+0.00/1 0s} 8. h3 {+0.00/1 0s} Bb7 {+0.00/1 0s} 9. d3
{+0.60/31 21s} d5 {(d5) -0.15/44 74s} 10. exd5 {(exd5) +0.32/35 31s} Nxd5
{(Nxd5) -0.10/39 29s} 11. a4 {(Nxe5) +0.40/37 25s} Nd4 {(Nd4) -0.10/42 28s} 12.
Nbd2 {(Nbd2) +0.37/37 38s} Nxb3 {(Nxb3) -0.12/43 24s} 13. Nxb3 {(Nxb3) +0.52/33
20s} Nb4 {(f6) -0.03/40 22s} 14. Nxe5 {(Nxe5) +0.51/36 23s} Qd5 {(Qd5) +0.00/39
20s} 15. Nf3 {(f3) +0.34/36 22s} Qd7 {(Qd7) -0.08/40 30s} 16. Re3 {(Re3)
+0.36/38 21s} Nd5 {(Nd5) -0.04/39 20s} 17. Re5 {(Re5) +0.28/37 19s} Bd6 {(Bd6)
+0.00/41 30s} 18. Re1 {(Re1) +0.30/37 20s} Nb4 {(Nb4) -0.04/47 38s} 19. Ne5
{(Ne5) +0.32/38 28s} Qf5 {(Qf5) -0.06/41 19s} 20. Nd4 {(Nd4) +0.26/41 33s} Qf6
{(Qf6) -0.05/41 19s} 21. Ndf3 {(Ndf3) +0.38/39 28s} Bxe5 {(Bxe5) +0.00/39 19s}
22. Nxe5 {(Nxe5) +0.38/38 15s} Rae8 {(Rae8) +0.00/43 21s} 23. d4 {(d4) +0.33/40
47s} Qf5 {(Qf5) -0.03/42 19s} 24. Ra3 {(Ra3) +0.40/40 21s} f6 {(f6) +0.00/43
18s}
```

B.21 Game21.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.09”]
[Round ”943.5.54”]
[White ”Berserk 13 64-bit 4CPU”]
[Black ”Stockfish 17 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”C88”]
[Opening ”Ruy Lopez”]
[Variation ”closed, anti-Marshall 8.a4”]
[PlyCount ”122”]
[WhiteElo ”3614”]
[BlackElo ”3639”]

1. e4 {+0.00/1 0s} e5 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} Nc6 {+0.00/1 0s} 3. Bb5
{+0.00/1 0s} a6 {+0.00/1 0s} 4. Ba4 {+0.00/1 0s} Nf6 {+0.00/1 0s} 5. O-O
{+0.00/1 0s} Be7 {+0.00/1 0s} 6. Re1 {+0.00/1 0s} b5 {+0.00/1 0s} 7. Bb3
{+0.00/1 0s} O-O {+0.00/1 0s} 8. a4 {+0.00/1 0s} b4 {+0.00/1 0s} 9. d4
{+0.22/37 65s} d6 {(d6) -0.10/39 45s} 10. dxe5 {(dxe5) +0.28/33 15s} Nxe5
{(Nxe5) -0.08/36 19s} 11. Nxe5 {(Nxe5) +0.19/40 95s} dxe5 {(dxe5) -0.07/39 17s}
12. Qe2 {(Qe2) +0.11/40 80s} a5 {(a5) -0.06/38 33s} 13. Bc4 {(Rd1) +0.25/38
66s} h6 {(Bc5) -0.02/39 52s} 14. Rd1 {(Rd1) +0.22/33 17s} Bd6 {(Bd6) -0.02/45
27s} 15. Nd2 {(Nd2) +0.20/32 16s} Bb7 {(Bb7) +0.00/58 27s} 16. Bd3 {(b3)
+0.12/38 26s} Nd7 {(Qe7) +0.00/43 18s} 17. Nc4 {(Nc4) +0.01/37 18s} Nc5 {(Nc5)
+0.02/53 29s} 18. Be3 {(Be3) +0.00/39 11s} Nxd3 {(Nxd3) +0.00/58 44s} 19. cxd3
{(Qxd3) +0.02/41 29s} Re8 {(Qf6) +0.00/60 24s} 20. Rac1 {(f3) +0.14/34 32s} Bf8
{(Bf8) +0.00/36 20s} 21. f3 {(f3) +0.02/38 62s} Ra6 {(Bc8) +0.00/51 37s} 22. h3
{(b3) +0.01/35 10s} Bc8 {(Bc8) +0.00/57 24s} 23. b3 {(g4) +0.01/40 11s} Rg6
{(Be6) +0.00/56 47s} 24. Kh1 {(Kh1) +0.00/36 10s} c5 {(Qh4) +0.00/59 28s}
```

B.22 Game22.txt

```
[Event "CCRL 40/15"]
[Site "CCRL"]
[Date "2024.09.10"]
[Round "943.5.79"]
[White "Stockfish 17 64-bit 4CPU"]
[Black "rofChade 3.1 64-bit 4CPU"]
[Result "1-0"]
[ECO "E92"]
[Opening "King's Indian"]
[Variation "Gligoric-Taimanov system"]
[PlyCount "179"]
[WhiteElo "3639"]
[BlackElo "3571"]

1. c4 {+0.00/1 0s} Nf6 {+0.00/1 0s} 2. Nc3 {+0.00/1 0s} g6 {+0.00/1 0s} 3. d4
{+0.00/1 0s} Bg7 {+0.00/1 0s} 4. Nf3 {+0.00/1 0s} O-O {+0.00/1 0s} 5. e4
{+0.00/1 0s} d6 {+0.00/1 0s} 6. Be2 {+0.00/1 0s} e5 {+0.00/1 0s} 7. Be3
{+0.00/1 0s} Ng4 {+0.00/1 0s} 8. Bg5 {+0.00/1 0s} f6 {+0.00/1 0s} 9. Bh4
{+0.54/38 26s} g5 {(g5) -0.72/26 36s} 10. Bg3 {(h3) +0.59/37 18s} Nh6 {(Nh6)
-0.75/24 15s} 11. h3 {(h3) +0.46/43 73s} Nc6 {(Nc6) -0.73/26 31s} 12. d5
{(dxe5) +0.42/39 30s} Ne7 {(Ne7) -0.56/23 19s} 13. Bh2 {(Qd2) +0.47/41 57s} Ng6
{(Ng6) -0.60/22 34s} 14. g4 {(g4) +0.40/39 49s} Nf7 {(c5) -0.48/19 12s} 15. b4
{(Nd2) +0.47/37 58s} h5 {(Bd7) -0.25/23 40s} 16. Nd2 {(Rg1) +0.48/35 38s} Nf4
{(Nf4) -0.33/23 39s} 17. Bg1 {(Bg1) +0.47/32 12s} c5 {(c5) -0.39/23 50s} 18.
Rb1 {(bxc5) +0.47/34 38s} Qc7 {(Qc7) -0.38/23 43s} 19. Nb5 {(bxc5) +0.46/34
23s} Qe7 {(Qb6) -0.46/23 23s} 20. f3 {(bxc5) +0.49/37 33s} b6 {(Bd7) -0.40/24
35s} 21. Nc3 {(Nc3) +0.46/38 70s} cxb4 {(cxb4) -0.42/23 16s} 22. Rxb4 {(Rxb4)
+0.47/39 14s} Nd8 {(Nd8) -0.44/24 12s} 23. Nf1 {(Nf1) +0.48/42 39s} Nb7 {(Nb7)
-0.38/22 13s} 24. Ng3 {(Ng3) +0.53/33 12s} h4 {(Nc5) -0.47/25 28s}
```

B.23 Game23.txt

```
[Event "CCRL 40/15"]
[Site "CCRL"]
[Date "2024.09.10"]
[Round "943.5.81"]
[White "Stockfish 17 64-bit 4CPU"]
[Black "RubiChess 20240817 64-bit 4CPU"]
[Result "1-0"]
[ECO "D35"]
[Opening "QGD"]
[Variation "exchange, positional line"]
[PlyCount "176"]
[WhiteElo "3639"]
[BlackElo "3600"]

1. d4 {+0.00/1 0s} Nf6 {+0.00/1 0s} 2. c4 {+0.00/1 0s} e6 {+0.00/1 0s} 3. Nc3
{+0.00/1 0s} d5 {+0.00/1 0s} 4. cxd5 {+0.00/1 0s} exd5 {+0.00/1 0s} 5. Bg5
{+0.00/1 0s} Be7 {+0.00/1 0s} 6. e3 {+0.00/1 0s} c6 {+0.00/1 0s} 7. Bd3
{+0.00/1 0s} Ne4 {+0.00/1 0s} 8. Bf4 {+0.00/1 0s} Bf5 {+0.00/1 0s} 9. Bxe4
{+0.76/36 28s} dxe4 {(dxe4) -0.79/28 57s} 10. Nge2 {(Nge2) +0.84/34 20s} g5
{(Bg6) -0.76/30 33s} 11. Be5 {(Be5) +0.90/38 26s} O-O {(O-O) -0.71/27 10s} 12.
Ng3 {(Ng3) +0.82/39 28s} Bg6 {(Bg6) -0.74/26 10s} 13. Qb3 {(Ngxe4) +0.86/37
21s} Nd7 {(Nd7) -0.81/27 10s} 14. Qxb7 {(Qxb7) +0.86/40 20s} Rc8 {(Rc8)
-0.68/27 11s} 15. Nge2 {(Nge2) +0.79/38 31s} Nxe5 {(Nxe5) -0.75/26 13s} 16.
dxe5 {(dxe5) +0.83/35 17s} Rc7 {(Rc7) -0.79/28 13s} 17. Qa6 {(Qa6) +0.77/36
26s} Qb8 {(Qb8) -0.76/28 15s} 18. Rb1 {(Rb1) +0.85/42 98s} Rb7 {(Rb7) -0.70/28
18s} 19. Nd4 {(Nd4) +0.89/37 14s} Rb6 {(Rb6) -0.89/29 13s} 20. Qa4 {(Qa4)
+0.87/37 20s} Bb4 {(Rc8) -0.76/28 14s} 21. Rc1 {+0.70/41 47s} c5 {(c5) -0.98/30
20s} 22. Nc6 {(Nc6) +0.63/40 25s} Qb7 {(Qb7) -0.90/27 10s} 23. Nxb4 {(Nxb4)
+0.60/36 20s} cxb4 {(cxb4) -0.84/26 9s} 24. Ne2 {(Ne2) +0.54/36 24s} Ra6 {(Ra6)
-0.80/29 11s}
```

B.24 Game24.txt

```
[Event "CCRL 40/15"]
[Site "CCRL"]
[Date "2024.09.11"]
[Round "943.5.98"]
[White "Clover 7.0 64-bit 4CPU"]
[Black "Stockfish 17 64-bit 4CPU"]
[Result "1/2-1/2"]
[ECO "D15"]
[Opening "QGD Slav"]
[Variation "4.Nc3"]
[PlyCount "132"]
[WhiteElo "3592"]
[BlackElo "3639"]

1. d4 {+0.00/1 0s} d5 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} Nf6 {+0.00/1 0s} 3. c4
{+0.00/1 0s} c6 {+0.00/1 0s} 4. Nc3 {+0.00/1 0s} a6 {+0.00/1 0s} 5. c5 {+0.00/1
0s} Nbd7 {+0.00/1 0s} 6. h3 {+0.00/1 0s} Qc7 {+0.00/1 0s} 7. Bg5 {+0.00/1 0s}
h6 {+0.00/1 0s} 8. Bh4 {+0.00/1 0s} Nh5 {+0.00/1 0s} 9. g4 {+0.22/34 15s} g5
{(g5) -0.54/41 43s} 10. gxh5 {(gxh5) +0.19/38 56s} gxh4 {(gxh4) -0.50/36 19s}
11. Rg1 {(e4) +0.06/39 23s} Rh7 {(Rh7) -0.09/36 20s} 12. Nxh4 {(Nxh4) +0.10/40
23s} e5 {(Qf4) +0.00/49 28s} 13. e4 {(e4) +0.17/33 22s} Nf6 {(Nf6) +0.00/46
26s} 14. exd5 {(exd5) +0.11/34 24s} Nxd5 {(Nxd5) +0.00/45 21s} 15. Nxd5 {(Nxd5)
+0.00/39 30s} cxd5 {(cxd5) +0.00/44 19s} 16. Be2 {(Be2) +0.01/37 17s} Qe7
{(Rg7) +0.00/42 28s} 17. Nf3 {(Nf3) +0.05/37 24s} exd4 {(exd4) +0.00/49 21s}
18. Qxd4 {(Qxd4) +0.04/37 25s} Qxc5 {(Qxc5) +0.00/54 19s} 19. Qxc5 {(Qxc5)
+0.00/41 26s} Bxc5 {(Bxc5) +0.00/49 25s} 20. Rg8+ {(Rg8) +0.00/42 51s} Bf8
{(Bf8) +0.00/58 19s} 21. Rc1 {(Rc1) +0.00/45 17s} Ke7 {(Ke7) +0.00/57 28s} 22.
Kd1 {(Kd1) +0.00/45 17s} Be6 {(Be6) +0.00/53 28s} 23. Rc7+ {(Nd4) +0.00/44 19s}
Kd6 {(Kd6) +0.00/59 35s} 24. Rxb7 {(Rxb7) +0.00/44 18s} Rg7 {(Rg7) +0.00/57
19s}
```

B.25 Game25.txt

```
[Event "CCRL 40/15"]
[Site "CCRL"]
[Date "2024.09.11"]
[Round "943.5.109"]
[White "Stockfish 17 64-bit 4CPU"]
[Black "Lizard 10.5 64-bit 4CPU"]
[Result "1/2-1/2"]
[ECO "D25"]
[Opening "QGA, Janowsky-Larsen variation"]
[PlyCount "207"]
[WhiteElo "3639"]
[BlackElo "3599"]

1. d4 {+0.00/1 0s} d5 {+0.00/1 0s} 2. c4 {+0.00/1 0s} dxc4 {+0.00/1 0s} 3. Nf3
{+0.00/1 0s} Nf6 {+0.00/1 0s} 4. e3 {+0.00/1 0s} Bg4 {+0.00/1 0s} 5. Bxc4
{+0.00/1 0s} e6 {+0.00/1 0s} 6. Nc3 {+0.00/1 0s} Nbd7 {+0.00/1 0s} 7. O-O
{+0.00/1 0s} Be7 {+0.00/1 0s} 8. Be2 {+0.00/1 0s} O-O {+0.00/1 0s} 9. Qb3
{+0.54/39 36s} Nb6 {(Nb6) -0.49/33 125s} 10. Rd1 {(Rd1) +0.49/36 20s} Bh5 {(a5)
-0.51/30 20s} 11. e4 {(e4) +0.50/33 26s} Bg6 {(Bg6) -0.65/31 25s} 12. Bd3
{(Bd3) +0.62/35 20s} Bh5 {(Bh5) -0.59/30 21s} 13. Be3 {(Be3) +0.41/43 300s}
Bxf3 {(Ng4) -0.40/29 21s} 14. gxf3 {(gxf3) +0.49/32 11s} Nh5 {(Nh5) -0.32/28
19s} 15. f4 {(f4) +0.50/36 43s} g5 {(g5) -0.57/28 21s} 16. f5 {(f5) +0.53/41
81s} exf5 {(exf5) -0.55/29 23s} 17. exf5 {(exf5) +0.51/36 15s} Bd6 {(Bd6)
-0.46/28 22s} 18. Kh1 {(Kh1) +0.48/35 17s} Kh8 {(Kh8) -0.63/30 54s} 19. a4
{(Rg1) +0.45/40 151s} Nf4 {(g4) -0.44/27 27s} 20. Be4 {(Be4) +0.64/28 7s} Rb8
{(Nd7) -0.46/28 21s} 21. Rg1 {(Rg1) +0.71/29 13s} Nd7 {(Nd7) -0.59/27 22s} 22.
Bc2 {(Bc2) +0.53/38 58s} Nh3 {(Nh3) -0.55/28 28s} 23. Rg3 {(Rg3) +0.54/35 3s}
Bxg3 {(g4) -0.46/29 28s} 24. hxg3 {(hxg3) +0.56/27 2s} g4 {(Nf6) -0.61/27 21s}
```

B.26 Game26.txt

```
[Event "CCRL 40/15"]
[Site "CCRL"]
[Date "2024.09.12"]
[Round "943.5.132"]
[White "Alexandria 7.0.0 64-bit 4CPU"]
[Black "Stockfish 17 64-bit 4CPU"]
[Result "1/2-1/2"]
[ECO "D97"]
[Opening "Gruenfeld"]
[Variation "Russian, Alekhine (Hungarian) variation"]
[PlyCount "148"]
[WhiteElo "3600"]
[BlackElo "3639"]

1. d4 {+0.00/1 0s} Nf6 {+0.00/1 0s} 2. c4 {+0.00/1 0s} g6 {+0.00/1 0s} 3. Nc3
{+0.00/1 0s} d5 {+0.00/1 0s} 4. Nf3 {+0.00/1 0s} Bg7 {+0.00/1 0s} 5. Qb3
{+0.00/1 0s} dxc4 {+0.00/1 0s} 6. Qxc4 {+0.00/1 0s} O-O {+0.00/1 0s} 7. e4
{+0.00/1 0s} a6 {+0.00/1 0s} 8. Be2 {+0.00/1 0s} b5 {+0.00/1 0s} 9. Qb3
{+0.86/32 19s} c5 {(c5) -0.03/41 58s} 10. dxc5 {(dxc5) +0.93/34 21s} Be6 {(Bb7)
-0.03/39 21s} 11. Qc2 {(Qc2) +0.78/34 26s} Nbd7 {(Nbd7) +0.00/41 20s} 12. Be3
{(Be3) +0.50/35 31s} Rc8 {(Rc8) -0.04/41 27s} 13. Rd1 {(Rd1) +0.58/37 30s} Qa5
{(b4) -0.03/44 26s} 14. a3 {(a3) +0.68/36 21s} b4 {(b4) +0.00/50 27s} 15. axb4
{(axb4) +0.84/37 21s} Qxb4 {(Qxb4) -0.02/46 47s} 16. Nd4 {(Nd4) +0.74/38 40s}
Nxc5 {(Nxc5) +0.00/46 18s} 17. O-O {(O-O) +0.64/37 24s} Ng4 {(Ng4) +0.00/56
31s} 18. Bxg4 {(Bxg4) +0.38/37 22s} Bxg4 {(Bxg4) +0.00/48 18s} 19. f3 {(f3)
+0.65/34 18s} Be6 {(Be6) +0.00/47 29s} 20. Nd5 {(Nd5) +0.44/34 19s} Bxd5 {(Qb7)
+0.00/45 20s} 21. exd5 {(exd5) +0.65/36 28s} Bxd4 {(Bxd4) +0.00/46 18s} 22.
Rxd4 {(Rxd4) +0.42/34 20s} Qb7 {(Qb7) +0.00/55 18s} 23. Qd2 {(Qe2) +0.19/38
77s} Qb5 {(Nb3) +0.00/64 26s} 24. Qb4 {(Qb4) +0.22/34 14s} Rfd8 {(Rfd8)
+0.00/53 37s}
```

B.27 Game27.txt

```
[Event "CCRL 40/15"]
[Site "CCRL"]
[Date "2024.09.12"]
[Round "943.5.136"]
[White "Caissa 1.20 64-bit 4CPU"]
[Black "Stockfish 17 64-bit 4CPU"]
[Result "1/2-1/2"]
[ECO "E16"]
[Opening "Queen's Indian"]
[Variation "Riumin variation"]
[PlyCount "154"]
[WhiteElo "3607"]
[BlackElo "3639"]

1. d4 {+0.00/1 0s} Nf6 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} e6 {+0.00/1 0s} 3. c4
{+0.00/1 0s} b6 {+0.00/1 0s} 4. g3 {+0.00/1 0s} Bb7 {+0.00/1 0s} 5. Bg2
{+0.00/1 0s} Bb4+ {+0.00/1 0s} 6. Bd2 {+0.00/1 0s} Be7 {+0.00/1 0s} 7. Nc3
{+0.00/1 0s} O-O {+0.00/1 0s} 8. O-O {+0.00/1 0s} Na6 {+0.00/1 0s} 9. Rc1
{+0.61/32 27s} d5 {(Ne4) -0.54/39 64s} 10. cxd5 {(cxd5) +0.54/35 33s} Nxd5
{(Nxd5) -0.61/36 29s} 11. Nxd5 {(Nxd5) +0.56/33 15s} exd5 {(exd5) -0.54/34 19s}
12. Qc2 {(Qa4) +0.54/34 24s} c5 {(c5) -0.51/43 129s} 13. Rfd1 {(Rfd1) +0.59/35
26s} Bf6 {(Re8) -0.48/35 14s} 14. Bc3 {(Bc3) +0.64/34 21s} Rc8 {(Rc8) -0.50/33
18s} 15. e3 {(h4) +0.51/34 22s} Qe7 {(c4) -0.56/36 67s} 16. dxc5 {(dxc5)
+0.58/35 16s} Bxc3 {(bxc5) -0.44/34 20s} 17. Qxc3 {(Qxc3) +0.61/37 18s} Rxc5
{(Rxc5) -0.43/35 21s} 18. Qd2 {(Qd2) +0.59/42 44s} Rfc8 {(Rfc8) -0.45/35 21s}
19. Nd4 {(Nd4) +0.56/44 17s} Rxc1 {(Rxc1) -0.39/35 24s} 20. Rxc1 {(Rxc1)
+0.56/37 10s} Rxc1+ {(Rxc1) -0.39/40 15s} 21. Qxc1 {(Qxc1) +0.56/39 4s} Qd7
{(Nc5) -0.37/43 47s} 22. Bf3 {(h4) +0.56/45 50s} Nc5 {(h6) -0.37/46 15s} 23. h4
{(h4) +0.56/49 33s} Nd3 {(h6) -0.36/45 25s} 24. Qc3 {(Qd2) +0.56/42 23s} Ne5
{(Ne5) -0.36/39 20s}
```

B.28 Game28.txt

```
[Event "CCRL 40/15"]
[Site "CCRL"]
[Date "2024.09.20"]
[Round "944.4.110"]
[White "Berserk 13 64-bit 4CPU"]
[Black "Stockfish 17 64-bit 4CPU"]
[Result "1/2-1/2"]
[ECO "A65"]
[Opening "Benoni"]
[Variation "6.e4"]
[PlyCount "139"]
[WhiteElo "3614"]
[BlackElo "3639"]

1. d4 {+0.00/1 0s} Nf6 {+0.00/1 0s} 2. c4 {+0.00/1 0s} c5 {+0.00/1 0s} 3. d5
{+0.00/1 0s} e6 {+0.00/1 0s} 4. Nc3 {+0.00/1 0s} exd5 {+0.00/1 0s} 5. cxd5
{+0.00/1 0s} d6 {+0.00/1 0s} 6. e4 {+0.00/1 0s} g6 {+0.00/1 0s} 7. Nge2
{+0.00/1 0s} Bg7 {+0.00/1 0s} 8. Ng3 {+0.00/1 0s} O-O {+0.00/1 0s} 9. Be2
{+0.64/31 12s} a6 {(b6) -0.59/42 115s} 10. a4 {(a4) +0.66/30 13s} Re8 {(Nbd7)
-0.60/37 30s} 11. O-O {(O-O) +0.72/30 15s} Nbd7 {(Nbd7) -0.54/33 19s} 12. h3
{(h3) +0.67/32 21s} h5 {(Qc7) -0.61/37 40s} 13. Bg5 {(Bg5) +0.75/31 22s} Qb6
{(Qb6) -0.61/35 17s} 14. Qd2 {(Qd2) +0.77/36 38s} Nh7 {(Nh7) -0.72/38 57s} 15.
Bh6 {(Bh6) +0.81/34 19s} h4 {(h4) -0.67/37 34s} 16. Bxg7 {(Bxg7) +0.79/34 12s}
Kxg7 {(Kxg7) -0.61/35 16s} 17. a5 {(a5) +0.80/36 33s} Qd8 {(Qd8) -0.60/35 21s}
18. Nh1 {(Nh1) +0.74/37 33s} Rb8 {(Rb8) -0.59/34 20s} 19. f3 {(f3) +0.71/35
19s} f5 {(f5) -0.51/38 112s} 20. Nf2 {(Nf2) +0.61/32 14s} b5 {(b5) -0.48/34
16s} 21. axb6 {(axb6) +0.58/33 16s} Rxb6 {(Rxb6) -0.48/35 14s} 22. Ra3 {(Ra4)
+0.57/37 80s} Qf6 {(Qf6) -0.45/37 20s} 23. Re1 {(Ra4) +0.53/29 12s} Ng5 {(Ng5)
-0.39/33 35s} 24. Bf1 {(Qc1) +0.62/32 33s} Nf7 {(Nf7) -0.37/30 16s}
```

B.29 Game29.txt

```
[Event "CCRL 40/15"]
[Site "CCRL"]
[Date "2024.09.20"]
[Round "944.4.103"]
[White "Stockfish 17 64-bit 4CPU"]
[Black "Stockfish 20230613 64-bit 4CPU"]
[Result "1/2-1/2"]
[ECO "A48"]
[Opening "King's Indian"]
[Variation "London system"]
[PlyCount "122"]
[WhiteElo "3639"]
[BlackElo "3641"]

1. d4 {+0.00/1 0s} Nf6 {+0.00/1 0s} 2. Nf3 {+0.00/1 0s} g6 {+0.00/1 0s} 3. Bf4
{+0.00/1 0s} Bg7 {+0.00/1 0s} 4. e3 {+0.00/1 0s} d6 {+0.00/1 0s} 5. h3 {+0.00/1
0s} O-O {+0.00/1 0s} 6. Be2 {+0.00/1 0s} c5 {+0.00/1 0s} 7. c3 {+0.00/1 0s}
cxd4 {+0.00/1 0s} 8. cxd4 {+0.00/1 0s} Qb6 {+0.00/1 0s} 9. Qb3 {-0.21/40 35s}
Qxb3 {(Qxb3) +0.22/42 99s} 10. axb3 {(axb3) -0.09/38 18s} Nd5 {(Nd5) +0.25/43
10s} 11. Bg5 {(Bg5) -0.18/40 38s} Nc6 {(Nc6) +0.23/44 18s} 12. Nc3 {(Nc3)
-0.23/37 26s} Ndb4 {(Ndb4) +0.26/41 18s} 13. O-O {(O-O) -0.22/36 19s} h6 {(h6)
+0.27/38 14s} 14. Bh4 {(Bh4) -0.24/35 18s} g5 {(g5) +0.26/41 16s} 15. Bg3
{(Bg3) -0.24/38 19s} Bd7 {(Bd7) +0.27/35 15s} 16. Ra4 {(Rfd1) -0.24/37 28s} a5
{(a5) +0.17/32 25s} 17. Raa1 {(h4) -0.27/44 60s} f5 {(b6) +0.22/40 16s} 18.
Bc4+ {(Bc4) -0.19/41 25s} Kh8 {(Kh8) +0.21/42 14s} 19. Na4 {(Na4) -0.13/42 25s}
Rab8 {(Rab8) +0.03/53 103s} 20. Nb6 {(Nb6) -0.04/39 17s} Be8 {(Be8) +0.02/44
16s} 21. d5 {(d5) -0.06/42 26s} Na7 {(Na7) +0.02/43 15s} 22. Rfd1 {(Rxa5)
-0.09/47 86s} Nc8 {(Nc8) +0.02/48 15s} 23. Nxc8 {(Nxc8) -0.04/43 14s} Rxc8
{(Rxc8) +0.03/49 13s} 24. Rxa5 {(Rxa5) -0.05/42 16s} b5 {(b5) +0.03/48 14s}
```

B.30 Game30.txt

```
[Event ”CCRL 40/15”]
[Site ”CCRL”]
[Date ”2024.09.20”]
[Round ”944.4.96”]
[White ”rofChade 3.1 64-bit 4CPU”]
[Black ”Stockfish 17 64-bit 4CPU”]
[Result ”1/2-1/2”]
[ECO ”A20”]
[Opening ”English opening”]
[PlyCount ”166”]
[WhiteElo ”3571”]
[BlackElo ”3639”]

1. c4 {+0.00/1 0s} e5 {+0.00/1 0s} 2. g3 {+0.00/1 0s} Nc6 {+0.00/1 0s} 3. Bg2
{+0.00/1 0s} f5 {+0.00/1 0s} 4. d3 {+0.00/1 0s} Nf6 {+0.00/1 0s} 5. Nc3
{+0.00/1 0s} Bb4 {+0.00/1 0s} 6. Bd2 {+0.00/1 0s} O-O {+0.00/1 0s} 7. Nf3
{+0.00/1 0s} d6 {+0.00/1 0s} 8. O-O {+0.00/1 0s} Bxc3 {+0.00/1 0s} 9. Bxc3
{+0.42/21 7s} Qe8 {(Qe8) -0.41/39 62s} 10. e3 {(e3) +0.66/23 27s} a5 {(a5)
-0.43/35 25s} 11. Nd2 {(Nd2) +0.65/24 34s} h6 {(Nb4) -0.37/38 48s} 12. a3 {(a3)
+0.50/27 69s} a4 {(a4) -0.36/37 18s} 13. Qe2 {(b4) +0.57/26 35s} Bd7 {(Bd7)
-0.23/41 85s} 14. f4 {(f4) +0.50/27 37s} Qg6 {(exf4) -0.22/35 13s} 15. Nf3
{(Nf3) +0.66/28 28s} Qh5 {(Rae8) -0.16/37 19s} 16. Rae1 {(Rae1) +0.63/29 26s}
e4 {(e4) -0.13/39 19s} 17. Bxf6 {(Bxf6) +0.75/28 13s} gxf6 {(gxf6) -0.08/39
16s} 18. Nd4 {(Nd4) +0.67/29 19s} Qxe2 {(Qxe2) -0.05/43 22s} 19. Rxe2 {(Rxe2)
+0.69/28 14s} Nxd4 {(Rfe8) -0.03/47 29s} 20. exd4 {(exd4) +0.48/27 13s} d5
{(d5) -0.02/47 52s} 21. Rc2 {(Rc2) +0.53/27 12s} dxc4 {(dxc4) -0.02/42 27s} 22.
dxe4 {(dxe4) +0.40/28 26s} Rad8 {(Rad8) +0.00/52 17s} 23. Re1 {(Rd1) +0.94/30
22s} fxe4 {(fxe4) +0.00/46 23s} 24. Rxc4 {(Rxc4) +0.44/29 25s} f5 {(f5)
-0.02/41 18s}
```

TRITA-EECS-EX- 2023:0000

€€€€ For DIVA €€€€

{
”Author1”: { ”Last name”: ”Säfström”,
”First name”: ”Alex”,
”Local User Id”: ”u102sc3t”,
”E-mail”: ”alexsaf@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science”,
}
},
”Author2”: { ”Last name”: ”Yllmark”,
”First name”: ”Simon”,
”Local User Id”: ”u11pj47x”,
”E-mail”: ”syllmark@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science”,
}
},
”Cycle”: ”1”,
”Course code”: ”II42X/II43X”,
”Credits”: ”15.0”,
”Degree1”: {”Educational program”: ”Degree Programme in Computer Engineering”
,”programcode”: ”TIDAB”
,”Degree”: ”Bachelors degree”
,”subjectArea”: ”Technology”
},
”Degree2”: {”Educational program”: ”Bachelor’s Programme in Information and Communication Technology”
,”programcode”: ”TCOMK”
,”Degree”: ”Bachelors degree”
,”SubjectArea”: ”Technology”
},
”Title”: {
”Main title”: ”A Comparative Study of Parallel Search Techniques in a Chess Engine”,
”Subtitle”: ”Young Brothers Wait Concept or Lazy SMP?”,
”Language”: ”eng” },
”Alternative title”: {
”Main title”: ”En Jämförande Studie av Parallella Söktekniker i en Schackmotor”,
”Subtitle”: ”Young Brothers Wait Concept eller Lazy SMP?”,
”Language”: ”swe”
},
”Supervisor1”: { ”Last name”: ”Al-Zarqawee”,
”First name”: ”Aws”,
”Local User Id”: ”u1z4d131”,
”E-mail”: ”awsj@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science”,
”L2”: ”Computer Science” }
},
”Examiner1”: { ”Last name”: ”Sung”,
”First name”: ”Ki Won”,
”Local User Id”: ”u1mktu3y”,
”E-mail”: ”sungkw@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science”,
”L2”: ”Computer Science” }
},
”National Subject Categories”: ”10201, 10206”,
”Other information”: {”Year”: ”2025”, ”Number of pages”: ”1,127”},
”Copyrightleft”: ”copyright”,
”Series”: { ”Title of series”: ”TRITA-EECS-EX” , ”No. in series”: ”2023:0000” },
”Opponents”: { ”Name”: ”Jens Jonsson & Viktor Salomonsson”},
”Presentation”: { ”Date”: ”2025-01-1 09:00”
,”Language”:”eng”
,”Room”: ”via Zoom https://kth-se.zoom.us/j/5680136250”
,”Address”: ”Isafjordsgatan 22 (Kistagången 16)”
,”City”: ”Stockholm” },
”Number of lang instances”: ”2”,
”Abstract[eng]”: €€€€

An important aspect to consider when creating a chess engine is the choice of parallel search algorithm. Two of the most common approaches are Young Brothers Wait Concept (YBWC) and Lazy Symmetric Multiprocessing (SMP). During the research process, it became apparent that while quite a few papers have been written about \glspl{CE} with one of the two algorithms, very little exists regarding comparing parallel search techniques of \glspl{CE} head-to-head.
\\

This project aims to determine how YBWC and Lazy SMP compare in terms of scalability, speedup, and playing strength. To accomplish this, the popular open source chess engine Crafty was modified to use Lazy SMP instead of YBWC. Both versions were then tasked with searching for the best move from 30 different positions 50 times each to determine the speedup from increasing thread-counts. They also played 30 matches against four engine opponents of different strengths to calculate the Elo rating of both versions of Crafty at different thread-counts.

\\

The findings of this project indicate that neither algorithm consistently outperforms the other. While each algorithm excels in specific scenarios, their strengths and weaknesses depend on resource availability.
These results suggest that YBWC performs better than Lazy SMP when utilising a large amount of computer resources. Conversely, Lazy SMP outperforms YBWC when operating with limited resources. 'YBWCs structured approach to organising resource utilisation gives it an advantage in resource-rich environments, but it becomes a drawback when resources are scarce. On the other hand, Lazy 'SMPs more chaotic, direct approach works well in low-resource situations, but becomes harder to manage as resource availability increases.
\\

The lack of previous research could make the findings valuable for other chess engine developers when selecting chess algorithms, particularly when deciding between Lazy SMP and YBWC.

€€€€,
"Keywords[eng]": €€€€
Chess Engine, Young Brothers Wait Concept, Lazy SMP, Parallel Search, Scalability, Elo Rating €€€€,
"Abstract[swe]": €€€€

En viktig aspekt att ta hänsyn till när man skapar en schackmotor är valet av av parallellsökalgoritm. Två av de vanligaste metoderna är Young Brothers Wait Concept (YBWC) och Lazy Symmetric Multiprocessing (SMP). Under forskningsprocessen blev det uppenbart att trots att en del artiklar har skrivits om schackmotorer med en av de två algoritmerna, så finns det väldigt lite som direkt jämför parallella söktekniker för schackmotorer med varandra.
\\

Detta projekt försöker avgöra hur YBWC och Lazy SMP står sig mot varandra gällande skalbarhet, hastighet och spelstyrka. För att åstadkomma detta användes den modifierades en populär schackmotor med öppen källkod vid namn Crafty för att använda Lazy SMP i stället för YBWC. Båda versionerna fick sedan söka efter det bästa draget från 30 olika positioner 50 gånger vardera för att avgöra hur mycket snabbare de blir med ökat antal trådar. De spelade även 30 matcher mot fyra schackmotorer av olika styrka för att beräkna Elo-värdet för de båda versionerna av Crafty vid olika antal trådar.
\\

Resultaten av detta projekt visar att ingen av algoritmerna konsekvent överträffar den andra. Även om varje algoritm utmärker sig i specifika scenarier beror deras styrkor och svagheter på resurstillgången. Dessa resultat tyder på att YBWC presterar bättre än Lazy SMP när en stor mängd datorresurser utnyttjas. Omvänt presterar Lazy SMP bättre än YBWC när man arbetar med begränsade resurser. YBWC:s strukturerade sätt att organisera resursutnyttjandet ger den en fördel i resursrika miljöer, men blir en nackdel när resurserna är knappa. Lazy SMP:s mer kaotiska och direkta tillvägagångssätt fungerar å andra sidan bra i resurssvaga situationer, men blir svårare att hantera när resurstillgången ökar.
\\

Avsaknaden av tidigare forskning kan göra resultaten värdefulla för andra schackmotorutvecklare när de väljer schackalgoritmer, särskilt när de väljer mellan Lazy SMP och YBWC.

€€€€,
"Keywords[swe]": €€€€
Schackmotor, Young Brothers Wait Concept, Lazy SMP, Parallelsökning, Skalbarhet, Elorating €€€€,
}

acronyms.tex

```
%%% Local Variables:
%%% mode: latex
%%% TeX-master: t
%%% End:
% The following command is used with glossaries-extra
\setabbreviationstyle[acronym]{long-short}
% The form of the entries in this file is \newacronym{label}{acronym}{phrase}
%                                       or \newacronym[options]{label}{acronym}{phrase}
% see "User Manual for glossaries.sty" for the  details about the options, one example is shown below
% note the specification of the long form plural in the line below
\newacronym[longplural={Debugging Information Entities}]{DIE}{DIE}{Debugging Information Entity}
%
% The following example also uses options
\newacronym[shortplural={OSes}, firstplural={operating systems (OSes)}]{OS}{OS}{operating system}

% note the use of a non-breaking dash in long text for the following acronym
\newacronym{IQL}{IQL}{Independent -QLearning}

% example of putting in a trademark on first expansion
\newacronym[first={NVIDIA OpenSHMEM Library (NVSHMEM\texttrademark)}]{NVSHMEM}{NVSHMEM}{NVIDIA
     ↪ OpenSHMEM Library}

\newacronym{KTH}{KTH}{KTH Royal Institute of Technology}

\newacronym{CB}{CB}{Chess-Bot}
\newacronym{CE}{CE}{Chess Engine}
\newacronym{Eval}{Eval}{Evaluation Function}
\newacronym{Neg}{Negamax}{Negamax Algorithm}
\newacronym{MT}{MT}{Multi-Threading}
\newacronym{SP}{SP}{Sequential Processing}
\newacronym{GM}{GM}{Grand Master}
\newacronym{UCI}{UCI}{Universal Chess Interface}
\newacronym{GUI}{GUI}{Graphical User Interface}
\newacronym{Am}{Am}{Amdahl's law}
\newacronym{BS}{BS}{Board State}
\newacronym{SIMD}{SIMD}{Single Instruction Multiple Data}
\newacronym{TT}{TT}{Transposition Tables}
\newacronym{Bb}{Bb}{Bitboards}
\newacronym{ST}{ST}{Search Tree}
\newacronym{EF}{EF}{Evaluation Function}
\newacronym{ML}{ML}{Machine Learning}
\newacronym{SEF}{SEF}{Static Evaluation Function}
\newacronym{NN}{NN}{Neural Network}
\newacronym{SMP}{SMP}{Symmetric Multiprocessing}
\newacronym{LSMP}{Lazy SMP}{Lazy Symmetric Multiprocessing}
\newacronym{YBWC}{YBWC}{Young Brothers Wait Concept}
\newacronym{ER}{ER}{Elo Rating}
\newacronym{Sped}{Sped}{Speedup}
\newacronym{Scal}{Scal}{Scalability}
\newacronym{UML}{UML}{Unified Modelling Language}
\newacronym{IDE}{IDE}{Integrated Development Environment}
\newacronym{FOSS}{FOSS}{Free and Open Source Software}
\newacronym{CCRL}{CCRL}{Computer Chess Rating Lists}
\newacronym{FIDE}{FIDE}{The International Chess Federation}
\newacronym{SMT}{SMT}{Simultaneous Multithreading}
\newacronym{CPU}{CPU}{Central Processing Unit}
\newacronym{PST}{PST}{Piece-Square Table}
```