A Complete Chess Engine Parallelized Using Lazy SMP

Emil Fredrik Østensen

Thesis submitted for the degree of
Master in programming and networks
60 credits

Department of informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2016

A Complete Chess Engine Parallelized Using Lazy SMP

Emil Fredrik Østensen

A Complete Chess Engine Parallelized Using Lazy SMP

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

The aim of the thesis was to build a complete chess engine and parallelize it using the lazy SMP algorithm. The chess engine implemented in the thesis ended up with an estimated ELO rating of 2238. The lazy SMP algorithm was successfully implemented, doubling the engines search speed using 4 threads on a multicore processor.

Another aim of this thesis was to act as a starting compendium for aspiring chess programmers. In chapter *2 Components of a Chess Engine* many of the techniques and algorithms used in modern chess engines are discussed and presented in a digestible way, in a try to make the topics understandable. The information there is presented in chronological order in relation to how a chess engine can be programmed.

Contents

List of Figures

List of Tables

Preface

First of, I want to say that chess programming is awesome. It is both satisfying and bizarre to program a chess engine, giving it simple rules to play by, and then having it completely own you in the game afterwards. Chess programming is a hands on task which quickly gives a visible result. I recommend everyone who have been thinking of programming a chess engine to do so.

If you want to play the chess engine built in this thesis, an instruction of how it can be attained and run can be found in appendix A.

Many people have helped getting me through this thesis. I would like to thank my supervisor, Arne Maus. Your guidance truly helped me with this thesis, both the scholarly part as well as giving me the pushes I needed to finish.

I would like to thank my mother, who has always been there for me, giving me encouragement and love. My father, who taught me to believe in myself and voice my opinions. My sister, the most reflected person I know, your dedication has served as an inspiration for me in every part of my life.

I would also like to thank my extended family and friends who have been patient with me while I have been writing this thesis.

Thanks to the gang at Assa. You guys are great and I will remember the time spent with you at the university with great fondness.

My grateful thanks also to Camilla and my cousin Øyvind for their help in the finishing stages of the thesis.

Chapter 1

Introduction

This chapter will first give a brief presentation of the history of chess computers. It starts by giving the background of the game of chess and how, even in pre-computer days, people tried to build chess playing machines. Following this is the early days of chess computers starting in the 1950s with Claude Shannon and Alan Turing. It will present how the cold war was fought on computers playing the game of chess, and how machines came to outplay even the strongest human players in the game.

The background and motivation of this thesis is then given. Which states that research on parallel chess algorithms has stagnated, but that a new algorithm, lazy SMP, has shown promise. It also makes a point of how, while there is a lot of information available for aspiring chess programmers, the information regarding chess engines can be scattered and very specific, which can make it hard to see the whole picture when implementing a chess engine.

The chapter then details the problem statement of the thesis, explaining that focus will be given on implementing the lazy SMP algorithm as well as acting as a compendium for aspiring chess programmers.

The limitations of the thesis is discussed. The main limitations regard optimization of the engine and how a versatile enough test set can be hard to produce.

The research method section gives a quick explenation of why the *design paradigm* by the ACM education board is used in this thesis.

The main contributions of the thesis is given, explaining how the research presented in this thesis contributes to give an overview of the theoretical background of designing and implementing a chess engine, as well as contributing to a better understanding of how and why the lazy SMP algorithm works.

The chapter ends with an outline of the rest of thesis.

1.1 The History of Chess Computers

The game of Chess is an ancient game with roots so far back that it is hard to tell where it originated. It is believed to have come to Europe at the end of the first millennium and can be traced back to an ancient indian game from around 500 AD called 'Chaturanga'[18]. During the middle ages the rules of chess were developed towards modern chess. Even today there are still changes made to chess, although these are quite small and mostly concern time limits or small modifications to the draw-rules.

A chess board has 64 squares and 32 pieces, and while the board and pieces may not seem so complex, the number of variations in chess is huge. If we construct a game tree of chess, assuming a game length of 80 and an average of 35 possible moves per position we will get the enormous number of 10^{123} leaf nodes[41]! This is so huge it is unimaginable, for comparison there are an estimated 10^{80} atoms in the universe. Today's computers have no chance iterating through the whole game tree to find the perfect move. Chess is therefore an interesting problem to try to solve, as it has to be solved by giving the machine some sort of intelligence. The machine has to assume a best move based on limited data. A game like chess, a game with well defined rules - but too big to be solved, can be considered the first step toward artificial intelligence.

1.1.1 The Beginning of Chess Playing Machines

The most known pre-computer chess machine was *Chess Automata*, also known as the *Chess Turk*[25]. It had its origins in the late 1700 and was shaped like a big box, with a chess board on top. A mechanical upper body of a human moved pieces. The Chess Turk traveled around Europe and America and played games to awe and admiration of people watching. Some big names that it played against and beat were Napoleon Bonaparte and Benjamin Franklin. It also played the chess legend André Danican Philidor, but lost this game. Its creator opened the machine before and after the game, which showed an intricate design of mechanical wheels.

The Chess Turk was unsurprisingly fake. It was operated by a strong human chess player who hid in a hidden compartment in the box when it was opened, and operated the human upper body during the game. It does, however, show that an interest for such a machine has existed for a long time.

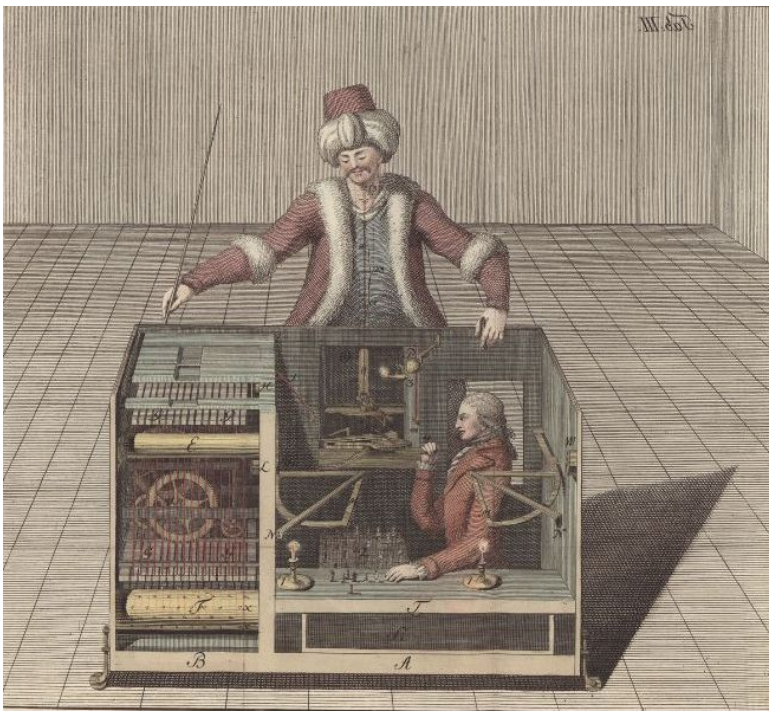


Figure 1.1: From a book that tried to explain the illusions behind the Kempelen chess playing automaton (known as the Turk) after making reconstructions of the device.[36]

The first real mechanical chess machine did not exist until 1890, when Torres y Quevedo built a machine that played the winning side of King and Rook vs King endgame, always ending in a mate[25]. Simple as this may seem, it paved the road for future chess machines, showing that it is indeed possible to make a chess playing machine. However, it wasn't until the late 1940s, when computers saw the light of day that we could start building real chess playing machines.

1.1.2 The Age of Computers

Chess computers as we know them today had their beginning in 1949, with Claude Shannon presenting his article *Programming a computer for playing chess*[41]. In this article Claude Shannon describes his idea of how two different types of chess computers, that he referred to as Type A and Type B, could be programmed:

The Type A chess computer was based on a minimax algorithm, where every possible move is calculated down to a predefined ply [1]. The positions it led to would then be evaluated and given a score by the computer. The algorithm would then alternate sides as it went up one level (down one

[1] **Ply** - the number of half-moves made; in chess you don't consider it a move until both black and white has moved a piece, so if white moves a piece and black moves a piece it is considered to be 1 move or 2 ply.

ply), and choose the move that gives the best evaluated position for that sides color based on the values given in the previous level. Basically, you assume the opponent is always making the best possible move (defined by the evaluation of your chess computer) and choose your best move based on that.

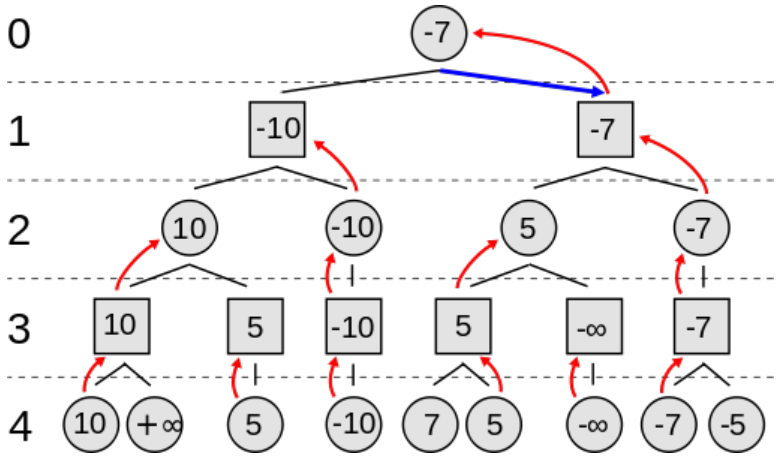


Figure 1.2: Minimax example. The red arrows show the player trying to play the 'best' move by chosing the best evaluation at each level. Odd levels are picking the smallest evaluation it can find (minimizing), while even levels ar picking the highest (maximizing).

The Type B chess computer was more strategic. Instead of calculating all moves down to a specific ply it would choose certain candidate moves and only calculate these based on some kind of chess knowledge that would have to be programmed into the computer. The number of candidate moves for a position should become fewer the deeper the ply, for example: At ply 1 the engine considers all moves of a position, at ply 2 only half of the moves are considered, at ply 3 only 1/4 of the moves are considered and so on. This way, it could calculate much deeper than the brute force Type A chess computer, since the growth of positions is not exponential. You could think of Type B chess computers as a more human approach, where only a few candidate moves are chosen to calculate and evaluate deeper variations.

Claude Shannon favored the Type B chess computers. His opinion was that the best chess engines would have to be programmed like this in order to improve the calculation-efficiency[41]. This way, only the supposedly best moves were considered, and lots of processing time would be saved not having to calculate bad variations.

In 1951, independently of Claude Shannon, Alan Turing wrote the worlds first Simulation of a chess computer[45]. It was written before he had a computer that could run it, so he did the calculations by hand. This chess computer was close to Shannon's Type A, however, it only considered captures and ran until the position was dead (no more captures)[32]. Alan Turing said about his algorithm that "Numerous criticisms of the machines

play may be made"[45], and it was easily defeated by an amateur chess player in the only match ever recorded.

1.1.3 The First Chess Computer

The first known computer that could play a complete game of chess came in 1957. It was made by Alex Bernstein and some of his colleagues at IBM and is popularly referred to as the *Bernstein program*[32]. It calculated moves to a depth of 4 ply, but as to not take too long to make a move it generated a max of 7 moves in each position. The move generation used several steps to generate moves and stopped when the allocated memory of 7 moves was reached[3]. The move generation worked like this:

1. Am I in check? If yes:
   (a) Try capturing the checking piece
   (b) Generate interposing moves
   (c) Move the King away
2. Look at all exchanges; do I win them? Should I move the piece away?
3. Castling
4. Develop minor pieces
5. Move Rooks to open files
6. Find possible critical squares created by Pawn-chains, try to put pieces on these
7. Pawn moves
8. Any other moves

This routine was then repeated (as the opponent) for the 7 positions created at ply 1, then repeated (as yourself again) for the 49 possible positions created at ply 2, and lastly repeated (as the opponent again) for the 343 position created at ply 3. In total the move generation could create up to 2401 positions at ply 4. The positions were then evaluated and the move was chosen using a minimax algorithm. Given the limitations of considering only 7 moves, this falls under the category of Shannon's Type B chess computer.

The machine used 8 minutes to make a move[3]. If the number of moves considered at each ply was raised to 8 it would have to evaluate up to 4096 positions and Bernstein reported that the chess computer would use about 15 minutes per move instead. If the depth was set 1 ply deeper (with 7 moves considered) it would have to evaluate up to 16807 positions and Bernstein reported that this took around 6½ hours. Time was the limit for the Bernstein program, and that is why the team chose to consider 7 moves per position and only calculate to a depth of 4 ply.

1.1.4 The Interest in Chess Computers Grow

Although papers and magazines wrote about the Bernstein Chess Computer when it first came out[26, p. 28], the interest in chess computers was still rather small. Computers playing chess (and playing it rather poorly) did not really get much hype. In 1966, however, a 4 game match between a chess computer written by Alan Kotok at MIT and reworked by John McCarthy at Stanford University and a soviet chess computer written by Vladimir Arlazarov, Georgy Adelson-Velsky, Alexander Bitman (a soviet master), Anatoly Uskov and Alexander Zhitovsky took place[26, p. 55]. It is one of the first computer vs. computer matches recorded and it fueled interest for chess computers as it built up under the cold war tension. Soviet won the match with a score of 2-0-2 (win-loss-draw), even though the Soviet computer was slower and used the more primitive Shannon's Type A-algorithm in comparison to Kotok-McCarthys Type B-algorithm[5].

After this, the number of chess computers exploded, and the strength of the chess computers grew substantially. Already in 1967 a chess computer written by Richard Greenblatt known as *Mac Hack Six* became an honorary member of the United States Chess Federation and achieved a rating of 1400 over the course of four months with a score of 3-12-3[13]. The interest fueled the starting of the North American Computer Chess championship in 1970, and later (at the request of the Soviet creators of a chess computer named *Kassia*) the World Computer Chess Championship in 1974. *Kassia* won the first championship with a score of 4-0-0, however, it avoided a match against the favorite, the American *Chess 4.0*, because *Chess 4.0* dropped a game in round 2 of the championship. An exhibition match was held after the championship, in which *Kassia* and *Chess 4.0* drew against each other.

While America used the loss against Kassia and the Soviet Union to get better funding for its research, the Soviet Union stopped making progress in the chess computer area. The Soviet Union felt that the limited computer power they had should not be used on games. Hence, this was the only chess championship won by the Soviet Union and people soon lost interest in the cold war rivalry in chess computers. A new rivalry soon took its place.

1.1.5 Man versus Machine

During the 1960s computers were far off from winning a game vs. a human chess master, but some thought the advancements of computers were strong enough to beat capable humans within a few years. In 1968 at a "machine intelligence workshop" at Edinburgh University the then Scottish chess champion, and somewhat of an expert within the field of artificial intelligence and chess computers, David Levy, came in talk with John McCarthy, one of the creators of the Kotok/McCarthy program from the 1966 match discussed earlier. John McCarthy believed that computers would be able to beat David Levy within 10 years at tournament time controls, but David Levy was skeptical. They made a bet of £250 that no

chess program would beat David Levy in a 2 series match at tournament time controls within 1978[27, p. 68]. This would be known as the famous Levy bet.

Later, many more would join David Levy's bet. By 1978 the pot had grown to £1250 and David Levy had become an international master. A new goal for chess computers had risen and during the mid 1970s many of the best chess computers were written with the specific purpose of beating David Levy.

Although the chess computers came *somewhat close* by 1978, most notably *chess 4.7* managed to draw a game vs. Levy after he made an early mistake, they did not really stand a chance. However, after the bet was over, Levy felt "that it would be a rather pity to remove the goal that chess programmers everywhere had been aiming for"[27, p. 102]. Together with *Omni magazine* David Levy offered a 5000$ prize to the first chess computer that could beat him in a match with regular time controls. And so, the joint quest of chess programmers around the world, *to beat David Levy,* continued.

During the 1980s the interest in computer chess continued to explode. Previously, few people had had access to the computer-power that was required to make a good chess engine, but as David Levy noted; the computers now became so cheap that almost everyone could buy a machine. That, coupled with the enormous amount of chess computer literature that started to exist, made the development easier.[27, p. 114]

David Levy knew how to play against computers. He usually played openings that led to closed positions, where chess computers have a disadvantage due to its limited depth in calculations, and one can argue that his extensive knowledge in computer chess made him a better opponent than many other players in his rating range. In 1989 *Deep Thought* finally won against David Levy to claim the prize of $5000. Over 2 decades had gone since the original bet of £250 was made, but the computers eventually beat the international master.

After this the chess computers next goal quickly became Garry Kasparov, the World Chess Champion at the time and without question one of the best chess players to ever have lived.

Garry Kasparov was a tough opponent. In 1985, in Hamburg, he played and beat 32(!) computers at a simul, which is an exhibition in which one player plays against multiple opponents simultaneously. In 1989, he also played the program that previously won against David Levy, *Deep Thought*, beating the computer twice in a two-game match. In 1996 Kasparov Beat *Deep Blue* in a six-game match, but not without some trouble. He actually lost the first game, but quickly recovered and he ended with a score of 3-2-1, beating *Deep Blue* with a score of 4-2. However, with the loss he showed some signs of weakness.

The year after, in 1997 an updated version of *Deep Blue* beat Kasparov in a six-game match 3,5 - 2,5. The game was tied after 5 games, and Kasparov tried to use one of David Levy's tricks - to play an opening which resulted in a closed midgame. *Mistake!* As black, he tried to use the Caro-Kann defense against the computer, which usually allows black to develop somewhat undistracted and have a good Pawn structure during the midgame. However, Kasparov did not usually play Caro-Kann and in move 7 he made a fatal and well known mistake allowing *Deep Blue* to gain a serious advantage. Kasparov resigned after only 19 moves!

The funny thing is; *Deep Blue* was still using its opening book[2] when Kasparov made his mistake[11, p. 127]. It did not even calculate the line it used! If Kasparov had not felt so threatened by the computer and instead of Caro-Kann used one of his more well known lines the game might have ended up with a completely different result.

With its focus on the alpha beta algorithm one can say that *Deep Blue* was a chess Computer of Shannon's Type A, albeit a bit more advanced. Chess computers took another direction entirely than Claude Shannon foresaw in 1948.

1.1.6 Shannon's Type A versus Shannon's Type B Chess Computers

Claude Shannon really hit the hammer on the nail with his article *Programming a computer for playing chess*. Even though the article was released over half a century ago, the ideas presented in that article is still prevalent in today's chess engines and the two types of chess engines that Claude Shannon described is still used to classify engines today, although in a bit broader sense then what Claude Shannon used; Type A being *the bruteforce search to a selected depth*, Type B being the more selective *only calculate a few variations, but explore them thoroughly*. Claude Shannon was convinced that a Type B program would need to be created for computers to play well, however, all the best chess computers today use algorithms that closely resembles Shannon's Type A. How did Claude Shannon guess so wrong? There are multiple reasons for this.

The biggest one, is that the minimax algorithm got some major improvements after Shannon wrote his article. Computer scientist quickly realized that the game tree in the minimax algorithm could be easily pruned. The algorithm got the same result, but without having to calculate all possible moves down to a certain ply, only a selected few. The most notable pruning technique is the Alpha Beta algorithm. Alpha Beta was independently invented a number of times[33] and was among other used in the Kotok-McCarthy program from 1966.

[2] **Opening Book** - a file of predetermined moves a chess engine have, to help it in the opening; it is loaded at startup and contains moves/replies to various depths in the opening.

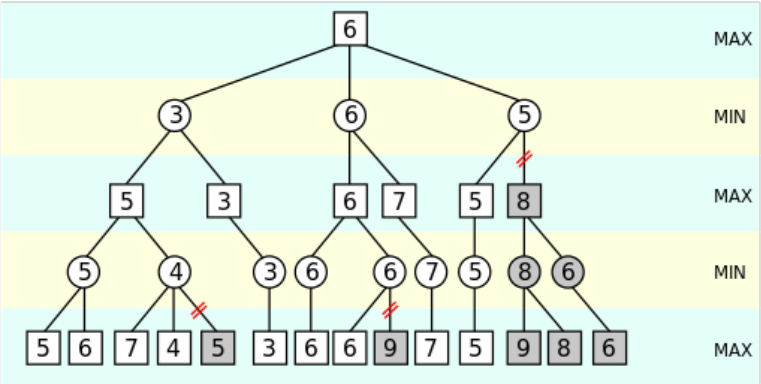


Figure 1.3: Alpha beta example. The greyed out areas do not have to be evaluated, and therefore do not have to be generated at all[22].

A formula for optimal pruning was discovered by Michael Levin[8]. Below are the results of how many leaf-nodes that must be examined compared to number of actual leaf-nodes with optimal alpha beta search given 40 moves per position.

ply	b^n	$b^{\lceil n/2 \rceil} + b^{\lceil n/2 \rceil} - 1$
0	1	1
1	40	40
2	1,600	79
3	64,000	1,639
4	2,560,000	3,199
5	102,400,000	65,569
6	4,096,000,000	127,999
7	163,840,000,000	2,623,999
8	6,553,600,000,000	5,119,999

Table 1.1: Optimal alpha beta-pruning given 40 moves per position.

As can be seen in 1.1 one can save extreme amounts of leaf-nodes examined when using optimal search. However, optimal search is quite difficult to achieve and it is actually one of the things today's top chess engines still improve upon. Engines have multiple different move-heuristics to achieve near optimal search.

Another reason is the effect of Moore's law. A loose definition of Moore's law (enough to explain the effect evolving CPUs had on chess engines), is that every 1-2 years the number of transistors on integrated circuits doubles[21]. Intel's first processor, the Intel 4004 had 2300 transistors, in 2012 it released its 3rd generation Intel core processor with over 1,4 billion transistors[20]; in effect, the computing power of PCs has grown exponentially. This has made Shannon's Type A much more viable than it

was in Claude Shannons days. Moores law also has another effect which is worth mentioning; the price of computers has gone down significantly. Today, everyone can afford a computer, everyone can program a chess-playing machine. This favors Shannon's Type A algorithms, why?

To write a good Shannon's Type B algorithm you are required to have a lot more chess knowledge than to write a good Shannon's Type A algorithm. One really needs to be an expert within both chess *and* programming to make an effective Type B chess computer. This severely limits the number of people that can make advancements in the area and thus has limited the progress of Type B chess computers, while the Type A chess computers thrive. The most notable try to make a Type B chess algorithm came from Mikhail Botvinnik. Not only was he a former World Chess Champion and teacher of Karpov and Kasparov (which both became World Chess Champions themselves), he also had a doctorate in electrical engineering. Botvinnik wrote multiple books on how to make a chess computer and he used advanced tactical problems to test out his algorithms. His tries, however, somewhat failed. While the chess computer was great at solving difficult problems by exploiting subtle weaknesses in the enemy position, its move generation had multiple flaws. The computer would miss simple tactics, and therefore be dropping pieces or even be checkmated out of the blue[6]. Sadly, he never got to fix these problems. There is no way of knowing where the Shannon's Type B computers would have been now if Botvinnik used more time to make his chess computer better.

1.1.7 Today's Chess Computers

Deep Blue, the chess computer that beat Garry Kasparov in 1997, was a specialized computer built for solving chess. What is used by people around the world today is not *chess computers* anymore, it is *chess engines*. Like most other computer programs it is no longer desired to have a specialized computer made for solving chess, hence chess engines are software programs that run on standard PCs. They no longer use specialized hardware, they are just programs calculating moves while communicating to a GUI using a communication protocol (2 examples of such protocols are the UCI and CEC). The chess computer championship that started in the 1970s and uses specialized hardware still exists, but it is nowhere near as popular as it was. Other tournaments exists now, like the *World Chess Software Championship*, which runs chess engines up against each other on standardized hardware. Even though these new engines calculate less positions in a second than *Deep Blue* did[19], they would have no trouble beating it. And neither Garry Kasparov or today's world chess champion Magnus Carlsen stand a chance against them. The smarter search as well as a better evaluation is what makes these engines better than *Deep Blue*.

The search, which mostly consists of better pruning, has been improved by ingenious people from all around the world, who invent better and better

search-algorithms. The smarter evaluation, however, comes mostly from testing. Testing what value each of the pieces should have, testing where the pieces stand the best, etc. Not much help from a chess master is necessary (it helps, of course) beyond the basic chess principles which most club-level players know.

Today, there is two top chess engines in the world, *Komodo* and *Stockfish* (with an honorary mention to *Houdini*). While these two play at almost the same level, surpassing each other on the leaderboards after they come with a big update, they have two very different styles of play.

Komodo can be called the slow thinker. It focuses a lot of its time on evaluating positions and does not go as deep as Stockfish. Its strong points can be said to be the middlegame, at more closed positions. It is more analytical, but that is not so strange, considering it is written by the chess grandmaster Larry Kaufmann.

Stockfish is more of a double-edged sword. It is an open source project with dozens of contributers. It focuses a lot on the pruning part of its algorithms, trying to get as deep as possible. Sometimes it makes small blunders where it thinks it has a winning move, only to later find out it did not get the compensation it thought it would get. A clear example of this can be taken from TCEC 2015 (Top Chess Engine Championship) where in game 46 of the final reaches deeper plies than Komodo in move 56-62, thinking it makes good moves, while komodo more correctly says the moves are losing[7].

Considering its creators, the difference in style is not really that strange. Komodo is created by a chess grand master and focuses more on the evaluation, creating more of a 'thinking' machine. Stockfish is created as an open source project and focuses more on the algorithmic part of the engine.

1.2 Background and Motivation

The playing strength of chess engines has moved well beyond even the strongest human players. While some of this may be attributed to computers getting more computing power each year, the progress can also be attributed to the improvement of chess playing algorithms. Since the clock speed of CPUs has stopped increasing the last years there has been a general shift toward parallelism in the computer world[44], this is also true for chess engines.

Parallelization of chess engines has been a hot topic to study since the 1980s[29], and while early research gave some good result, the progress halted in the 1990s. One example of this is that most of the top chess engines use the YBWC-algorithm for parallelization, an algorithm introduced in 1989[40]. In recent years, however, a new algorithm has shown some promise. In 2013 Daniel Homan introduced, in a chess engine

forum, a new parallelization concept he coined Lazy SMP. In January 2016 Stockfish implemented the algorithm in its release of Stockfish 7.

During the background research for this thesis not a single paper or thesis that gives a thorough examination of the Lazy SMP algorithm was found. Even though there exist a good amount of resources and information on how to implement the lazy SMP algorithm on the internet, and one can look up the raw results of Stockfish's cluster testing on *its* implementation of the lazy SMP algorithm, the information is scattered and not user friendly. This thesis aims to take the first step toward a better understanding on how and why the lazy SMP algorithm works.

Another point to be made is about the information of how a general chess engine works. Even though this information exists to a much bigger degree than the specifics of the lazy SMP algorithm, and some of the best engines are open source, the information is compiled in a way that makes it hard for aspiring chess programmers to start programming a new chess engine. The information is online and available for anyone who wants to create a chess engine, but it is hard to know where to begin. Multiple sources explains different parts of a chess engine, applying it to different specific programming languages and, in general, being very case specific on how various components are implemented.

1.3 Problem Statement

This thesis will aim toward compiling a lot of the information that exist on chess engines into a more digestible source. It will function as a compendium on how to create a chess engine entirely from scratch. The thesis will present the most popular concepts and algorithms used in modern chess engines.

Some topics will be explored more thoroughly than others and focus will be given on how the engine is parallelized using the Lazy SMP algorithm. The thesis will explain how Lazy SMP came to be, how it can be implemented in a chess engine, and will also give explanations on how and why it works.

1.4 Limitations

All the tests performed will be done using the chess engine implemented during this thesis. The chess engine does not perform on the level of the best chess engines in the world. It has not been the scope of this thesis to perfect the evaluation metrics the engine uses. Neither has there been much focus on optimizing the search heuristics or optimize the algorithms to run on the hardware tested.

The test that reports ELO rating gives only a rough guesstimate and is not to be considered reliable.

The lazy SMP test is performed on a test set of a thousand different chess positions. This is hopefully versatile enough to catch the most common game scenarios, but even though the positions are from various phases of the game and have multiple different traits, obviously not all situation the engine may encounter have been tested.

There has not been put an emphasis on making the engine find the optimal or close to the optimal moves in the positions tested. The engine has not been tweaked for this work, and only the comparative speed and positions explored is presented in the results.

1.5 Research Method

The research in this thesis will be based on the paper *Computing as a discipline* from 1989 by the ACM education board. The paper defines 3 paradigms that can be applied to computer research; *theory, abstraction and design*.

- **The theory paradigm** is rooted in mathematics. The research process consists of defining an object of study, making a theorem of possible outcomes, proving/disproving the theorem and interpreting the result.

- **The abstraction paradigm** is rooted in the experimental scientific methods. The process consists of forming a hypothesis, constructing a model and making predictions, collecting data on the model and finally analyze the result.

- **The design paradigm** is rooted in engineering. Its process consists of stating requirements, stating specifications, design and implementation of the system and finally testing the system.

The aim is to investigate how a chess engine can be improved on multicore systems using the lazy SMP algorithm. The lazy SMP algorithm is, as explained later in the thesis, based on some random factors of the structure of the game tree and implementation. The *theory* paradigm is therefore out of the question, because the random factors makes it very hard to precisely make a theorem of possible outcomes. The *abstraction* paradigm could be used in some extent to model a chess game tree and collecting data, but the size of the problem and the multiple different nuances of a chess game makes modeling a thorough enough model to be relevant for the task a huge job. The *design* paradigm was found to be the most fitting methodology to research the problem of this thesis. The aim is to generalize the problem to a set of requirements and specification, then implement and test it. This will hopefully lead to some valuable information and can be used to further the understanding of the lazy SMP algorithm.

1.6 Main Contribution

In *1.2 Background and Motivation* two main motivations of writing this thesis is given. One is the lack of a basic overview of the theoretical information behind the design and implementation of a complete chess engine. The other is the lack of information concerning the implementation and theoretical knowledge of the lazy SMP algorithm. With this in mind, the main contributions of the thesis are as follows.

- The thesis has compiled an overview of the basic components of a chess engine, explaining the theory behind how and why most parts of a chess engine work. This information is presented in a digestible way, so that future researchers can get a basic knowledge-base concerning the implementation and theory of chess engine algorithms.
- The lazy SMP algorithm has been researched and explained. The suggested lazy SMP design is implemented, tested, and is giving a speedup of the alpha beta algorithm using threads on a multicore CPU.

1.7 Outline

The rest of the thesis is organized as follows:

Chapter 2 will detail how a normal chess engine is built. It will present multiple variations of the general components that is normally used to build a complete chess engine.

Chapter 3 will focus on the more limited area of research that is how to parallelize a chess engine, mainly the alpha beta algorithm. It will introduce the concept of perfect and near perfect game trees, the traits of these and how the knowledge of these traits can be used when trying to parallelize a chess engine.

Chapter 4 will present the design and implementation of the engine built in this thesis. Giving the specifics of how the engine is implemented, as well as reasoning on why it was done this way.

Chapter 5 will present the results of the tests performed in the thesis. It will give reasons for what tests were performed and why these tests where chosen. It will present graphs of the result with quick explanations of what they show.

Chapter 6 will present a discussion of the results presented in the previous chapter.

Finally, Chapter 7 will present The conclusions that can be drawn from the results. It will give a quick summary of the entire thesis and then present future work that can be done on this branch of research.

1.8 Summary

This chapter first gave a brief presentation of the history of chess computers. Where it was detailed how computer chess took it first steps in the 1950s, to become the best chess player in the world in the end of the 20th century.

Following this was a section regarding the background and motivation of the thesis. Which explains how lazy SMP was a breath of fresh air in the research field of parallelizing chess engines, a field which has stagnated since the early 1990s.

The problem statement of the thesis was given, where it was explained that the thesis can act as a compendium, giving an overview of all information needed to implement a modern chess engine. It will focus on parallelizing the engine using the lazy SMP algorithm, an algorithm that to the author's knowledge there exists no previous papers about.

The research method in this thesis will be based on the *design* paradigm by the ACM education board. The paradigm focuses on the implementation and testing of the research problem.

The main contributions of this thesis is the theoretical overview given by the paper as well as the contribution towards the understanding of the lazy SMP algorithm.

Chapter 2

Components of a Chess Engine

This chapter will describe and explain the different components of a typical chess engine. To be able to play a complete game of chess versus other engines or humans an engine normally includes the following components.

An internal board representation A chess engine must have a way of internally representing the chessboard as well as where all the pieces on the chessboard are placed.

A way to evaluate a position A chess engine should have a way to evaluate any given position, to give it a single score. The unit of measure for this score is usually represented in centipawns. A centipawn is 1/100th of a Pawn, so 100 centipawns is equal to 1 Pawn. A negative centipawn score means that the opponent has an advantage, a positive centipawn score is an advantage to the side to move.

A way to guess the best move A chess engine must have a way of guessing the best move of a structure. This guess is usually based on an alpha beta search over a given position, where it uses the evaluation score as weights on the leaf nodes of the search tree to guess the best possible move.

A way to communicate with both humans and other chess engines To be able to play versus humans or other chess engines an engine must have a way to communicate. A typical chess engine will implement a communication protocol which it uses to talk to a GUI. This allows both humans and other engines to play with it.

This chapter will go over every aspect of what is needed to implement a modern chess engine.

Before outlining the components of a chess engine an explanation of the algebraic notation is given. Algebraic notation is a way to write down the

moves of a game of chess in a simple way that can be understood by the reader. This notation is used throughout the thesis and a quick explanation of its basic is needed to read the rest of the paper.

In section *2.2 Board Representation in a Computer Program* multiple different ways of how to implement a chess engine's internal board representation is outlined. An emphasis is given on how moves of the board is generated on each of these board representations.

The next section details and gives an example of how one evaluates a given position of chess. The evaluation is used in the alpha beta search, where the leaf nodes of the search are evaluated in order to search the game tree. The section gives some examples of what metrics can be used to find out if a position is good or bad. It returns a single number which can then be compared to another positions evaluation. The section also details how endgame tablebases can be used to enhance the evaluation of endgame positions.

The chapter goes on to explain how the search function works, so the engine can select a best possible move. With a thorough example of the alpha beta algorithm.

The engines communicate using a communication protocol, usually to some kind of GUI displaying the board. This chapter will explain how the UCI protocol works, and how the GUI communicates with the engine and tells it what to do. Each of the most relevant commands are explained, with given expected responses by the engine.

The chapter goes into more detail on how the alpha beta algorithm can be improved. The alpha beta algorithm works best if the best moves (meaning the one that will give a leaf node with the highest possible score for the side to move) are searched first. And so section *2.6 Improving the alpha beta algorithm* gives detailed information of various heuristics that can be used to better guess what move should be searched first.

2.1 Algebraic Notation

The algebraic notation is the most common way to write and record moves in a game of chess. A chessboard is made of 64 squares and in the algebraic notation each square has its own unique coordinate pair. The horizontal coordinates are named with letter a-h and is referred to as files. The vertical coordinates are numbered 1-8 and is referred to as ranks. For the white player the bottom left square is a1 and the top right square is h8.

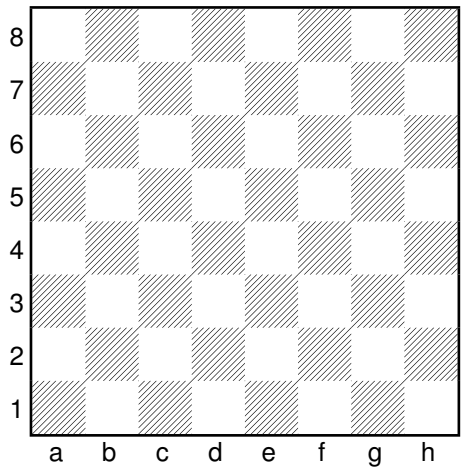


Figure 2.1: Empty chessboard showing the a-h and 1-8 coordinates.

Each piece has a short 1 letter name starting with the first character of the piece, with the exception of Knights(N) because it has the same first letter as Kings. They can also be named by using a picture of their piece:

- K / ♔- King
- Q / ♕- Queen
- R / ♖- Rook
- B / ♗- Bishop
- N / ♘- Knights

Pawns are not named. When writing a move you first start with the name of the piece, then say which square the piece is moved to. You don't say the start square. Example, the Spanish opening:

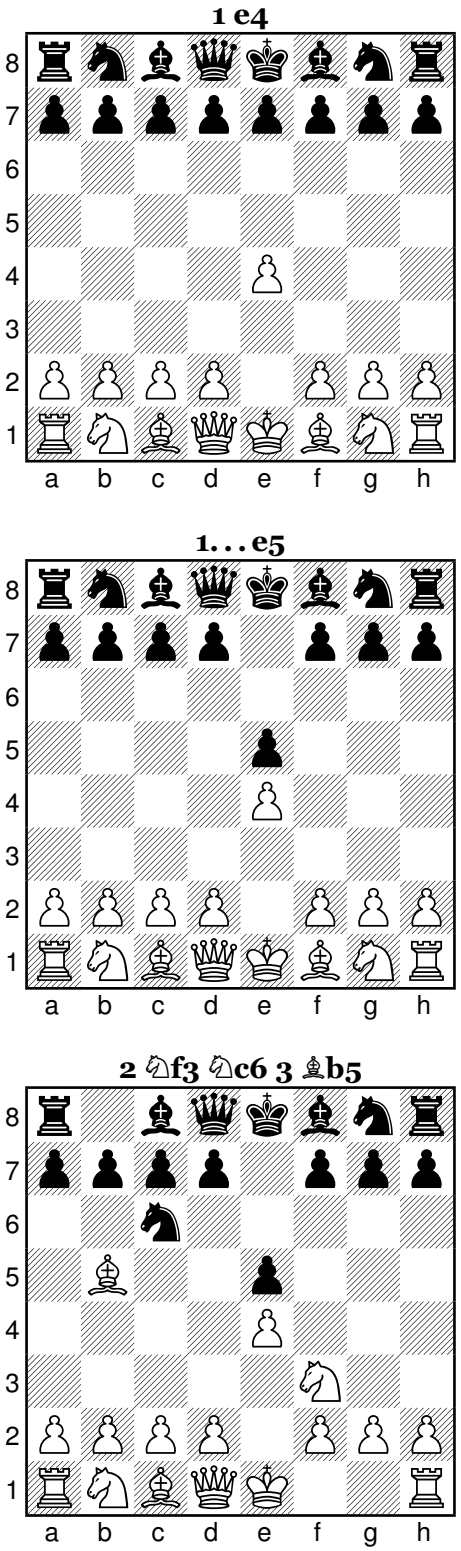


Figure 2.2: Example of the algebraic notation, the Spanish opening.

There are various exceptions to this rule, which will not be named here as they are not important to explain how a chess engine works.

2.2 Board Representation in a Computer Program

A chess engine needs some sort of internal board representation to be able to play. The board representation is used by a chess engine to generate moves as well as to evaluate positions.

The board representation in a chess engine contains information on where all the pieces are placed on the board. It also needs to contain some extra information, such as; if castling is allowed, if a Pawn can be captured en passant in the current move, info about the 50-move draw rule and 3 folds repetition moves.

Board representation may seem like a trivial task. Even entry level programmers will have little problem creating an internal representation of a chess board that can be used to generate moves and evaluate positions. However, as with most other parts of a chess engine one have to consider both speed and memory when selecting a board representation.

In the early days of computers memory was sparse and speed was slow when compared to today's standards. However, we still face the same issues today as they did when computer programming was in its infant stage: To make the chess engine faster, so it gets better. The search algorithm, explained more thoroughly in section *2.4 The Search Function, Selecting the Best Move*, is of course very important. Due to the cache, the fast memory on the CPU, another important thing is memory usage. Generally with less memory usage we get less cache misses, where the computer has to search fetch data from RAM instead of using the faster CPU cache. Hence the memory usage of the board structure is also important.

This section will give an overview of multiple different ways a chess engine can be programmed to represent the board of a chess game. The implementation is directly related to how the moves are generated, and this is the main reason many different representations exist. A quick explanation of how the moves are generated for each representation is naturally included.

2.2.1 The Naive Board Representation

A chess board has 64 squares and there are only 12 different types of pieces. The simplest solution would be an array of size 64 which contains 13 different enums, 6 enums for the white side and 6 enums for the black side, plus 1 enum for an empty square. With the opening position of chess, a solution can look something like the following example.

```
enum {P, R, B, N, K, Q, p, r, b, n, k, q, -} squares;

int board[64] = {
    R, N, B, Q, K, B, N, R,
    P, P, P, P, P, P, P, P,
    -, -, -, -, -, -, -, -,
    -, -, -, -, -, -, -, -,
    -, -, -, -, -, -, -, -,
    -, -, -, -, -, -, -, -,
    p, p, p, p, p, p, p, p,
    r, n, b, q, k, b, n, r
};
```

Figure 2.3: In this code example the Capital characters are white pieces (so it is mirrored upside down), the lower case characters are black pieces, the empty squares are given the name '-'. Since the enums only go to a size of thirteen the integers in the array can be represented using 4 bits.

In the code example the board is mirrored upside down so that the square a1 is at index 0 - board[0] points to R, a white Rook. The mirrored board may look a bit off at first, but to make a1 index 0 is quite logical and the standard way of drawing arrays places index 0 as the leftmost index. The position is correct and it does not matter to a computer where the coordinates are, as long as the program is consistent.

The naive board representation may seem okay, and will work, but as earlier mentioned; the chessboard representation is used by the engine to generate moves. This means all possible moves of a game tree, depending on the size and the depth the engine searches at this leads to millions of moves to be generated. The speed of the move generation is therefore essential. There is practically no end to how deep you can generate a chess game tree, the faster you can make the chess engine generate the moves - the more positions it gets to evaluate. With the board represented above, it will take unnecessary time and calculations to generate moves. Following is an example of the generation of Knight moves using the naive board representation.

Knight Move Generation Example

Let's say a white Knight is placed at index 36 in the naive board representation. To generate all the moves of this Knight the (somewhat simplified) code snippet below would find all possible moves.

```
nIndex = 36; //the Knight-index

if(!whitePiece(board[nIndex + 6]))
addMove()
if(!whitePiece(board[nIndex + 10]))
addMove()
if(!whitePiece(board[nIndex + 15]))
addMove()
if(!whitePiece(board[nIndex + 17]))
addMove()
if(!whitePiece(board[nIndex - 6]))
addMove()
if(!whitePiece(board[nIndex - 10]))
addMove()
if(!whitePiece(board[nIndex - 15]))
addMove()
if(!whitePiece(board[nIndex - 17]))
addMove()

int whitePiece (int value) {
    return value < p;
}
```

Figure 2.4: Knight move generation

The function whitePiece(int) returns true if the piece is white. addMove() is called to add the move checked, it should probably take some kind of move structure as an argument but it is omitted for simplification.

	a	b	c	d	e	f	g	h
1	0	1	2	3	4	5	6	7
2	8	9	10	11	12	13	14	15
3	16	17	18	19	20	21	22	23
4	24	25	26	27	28	29	30	31
5	32	33	34	35	36	37	38	39
6	40	41	42	43	44	45	46	47
7	48	49	50	51	52	53	54	55
8	56	57	58	59	60	61	62	63

Table 2.1: Picture showing the 'from square' marked in red, and the 'to square' marked in blue for all Knight moves

This may seem quite simple and looks correct. But what if the piece is placed on index 31? The code above would add illegal Knight moves, for example the Knight could jump from index 31 all the way across the board to index 41. In fact, you would have to add exceptions checking the index for everyone of these addMove calls, suddenly the code is messy and quite

complex. The next two board representation examples are quite similar to the naive solution but includes quite elegant fixes for this problem.

	a	b	c	d	e	f	g	h
1	0	1	2	3	4	5	6	7
2	8	9	10	11	12	13	14	15
3	16	17	18	19	20	21	22	23
4	24	25	26	27	28	29	30	31
5	32	33	34	35	36	37	38	39
6	40	41	42	43	44	45	46	47
7	48	49	50	51	52	53	54	55
8	56	57	58	59	60	61	62	63

Table 2.2: Erroneous moves generated

2.2.2 The 120-board

The 120-board or 10x12-board[42] is a mailbox representation that fixes the problems the move generation experienced in the naive board representation. In computer chess 'mailbox representation' used as a general term for board representations that are implemented as arrays, in a way that they look like a chess board, shown in the following example. The 120-board consists of an array that's 120 squares big and can be visualised like the following example.

```
int 120board[120] = {
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
    -1,  R,  N,  B,  Q,  K,  B,  N,  R, -1,
    -1,  P,  P,  P,  P,  P,  P,  P,  P, -1,
    -1,  -,  -,  -,  -,  -,  -,  -,  -, -1,
    -1,  -,  -,  -,  -,  -,  -,  -,  -, -1,
    -1,  -,  -,  -,  -,  -,  -,  -,  -, -1,
    -1,  -,  -,  -,  -,  -,  -,  -,  -, -1,
    -1,  p,  p,  p,  p,  p,  p,  p,  p, -1,
    -1,  r,  n,  b,  q,  k,  b,  n,  r, -1,
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
};
```

Figure 2.5: Here is the values of the 120 board from the starting position

```
int 120to64[120] = {
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
    -1,  0,  1,  2,  3,  4,  5,  6,  7, -1,
    -1,  8,  9, 10, 11, 12, 13, 14, 15, -1,
    -1, 16, 17, 18, 19, 20, 21, 22, 23, -1,
    -1, 24, 25, 26, 27, 28, 29, 30, 31, -1,
    -1, 32, 33, 34, 35, 36, 37, 38, 39, -1,
    -1, 40, 41, 42, 43, 44, 45, 46, 47, -1,
    -1, 48, 49, 50, 51, 52, 53, 54, 55, -1,
    -1, 56, 57, 58, 59, 60, 61, 62, 63, -1,
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
};
```

Figure 2.6: A simple array to convert to the naive representation

To make it more simple to understand there is also a converter array included, used to convert the indexes to an array of size 64. The advantages of the 120-board representation are the -1 values. The -1s are off-board values which are used to easily find the edge of the board, so that pieces do not move there. The reason the top and bottom edges are 2 rows big is for the Knights, the sides does not need that as the left side is just a continuation of the right side. Following is a new Knight move generation example, now using the 120 board representation.

Knight Move Generation Example

```
//the Knight-index (the Knight is at the 31 value in the 120to64 array
    above)
nIndex = 64to120[31];

if(onBoard(board[nIndex + 6])  && !whitePiece(board[nIndex + 6]))
addMove()
if(onBoard(board[nIndex + 10]) && !whitePiece(board[nIndex + 10]))
addMove()
if(onBoard(board[nIndex + 15]) && !whitePiece(board[nIndex + 15]))
addMove()
if(onBoard(board[nIndex + 17]) && !whitePiece(board[nIndex + 17]))
addMove()
if(onBoard(board[nIndex - 6])  && !whitePiece(board[nIndex - 6]))
addMove()
if(onBoard(board[nIndex - 10]) && !whitePiece(board[nIndex - 10]))
addMove()
if(onBoard(board[nIndex - 15]) && !whitePiece(board[nIndex - 15]))
addMove()
if(onBoard(board[nIndex - 17]) && !whitePiece(board[nIndex - 17]))
addMove()

int onBoard(int value) {
    return value != -1;
}
```

Figure 2.7: Simplified code to show the now correct Knight move generation of a 120 board

It is the same simple code as in the failed naive solution, only with 1 exception; now we also check if the square the Knight jumps to is off the board. Note that we could not do this offboard check for the naive solution because 31 + 10 = 41 which is in fact on the board.

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109
110	111	112	113	114	115	116	117	118	119

Table 2.3: With the 120 board one can correctly tell if the pieces moves off the board

2.2.3 The 0x88-board Representation

0x88[4] is another mailbox representation. This board representation is also used to easily calculate off-board positions, albeit in a bit different way. While the 120-board uses an array with an 'off-board'-value in the board, the 0x88 board is made in such a way that you can use the bitwise AND operation to check if the square is on the board or not. The 0x88-board is a 128 sized array, but only half of the array is used to represent the board, the other half is garbage - used only to offset the board so that the bitwise operation succeeds. A 0x88 board can be visualised like the following example.

```
int 0x88board[128] = {              /*GARBAGE BITS:*/
    R, N, B, Q, K, B, N, R, -1, -1, -1, -1, -1, -1, -1, -1,
    P, P, P, P, P, P, P, P, -1, -1, -1, -1, -1, -1, -1, -1,
    -, -, -, -, -, -, -, -, -1, -1, -1, -1, -1, -1, -1, -1,
    -, -, -, -, -, -, -, -, -1, -1, -1, -1, -1, -1, -1, -1,
    -, -, -, -, -, -, -, -, -1, -1, -1, -1, -1, -1, -1, -1,
    -, -, -, -, -, -, -, -, -1, -1, -1, -1, -1, -1, -1, -1,
    p, p, p, p, p, p, p, p, -1, -1, -1, -1, -1, -1, -1, -1,
    r, n, b, q, k, b, n, r, -1, -1, -1, -1, -1, -1, -1, -1
};
```

Figure 2.8: an array showing the 0x88 board, the value of the garbage bits are newer accessed, as the indexed are bitwise ANDed with 0x88 to find if the index is off the board

```
int 88to64[128] = {                       /*GARBAGE BITS*/
     0,  1,  2,  3,  4,  5,  6,  7, -1, -1, -1, -1, -1, -1, -1, -1,
     8,  9, 10, 11, 12, 13, 14, 15, -1, -1, -1, -1, -1, -1, -1, -1,
    16, 17, 18, 19, 20, 21, 22, 23, -1, -1, -1, -1, -1, -1, -1, -1,
    24, 25, 26, 27, 28, 29, 30, 31, -1, -1, -1, -1, -1, -1, -1, -1,
    32, 33, 34, 35, 36, 37, 38, 39, -1, -1, -1, -1, -1, -1, -1, -1,
    40, 41, 42, 43, 44, 45, 46, 47, -1, -1, -1, -1, -1, -1, -1, -1,
    48, 49, 50, 51, 52, 53, 54, 55, -1, -1, -1, -1, -1, -1, -1, -1,
    56, 57, 58, 59, 60, 61, 62, 63, -1, -1, -1, -1, -1, -1, -1, -1
};
```

Figure 2.9: An array converting from 0x88 board to the naive board representation

The advantages of the 0x88-board representation are much the same as the 120-board. We can easily find the edges of the board. Following is another Knight move generation example, now using the 0x88 board.

Knight Move Generation Example

```
//the Knight-index (the Knight is at the 31 value in the 88to64 array above)
nIndex = 64to88[31];

if(onBoard(nIndex + 6)  && !whitePiece(board[nIndex + 6]))
addMove()
if(onBoard(nIndex + 10) && !whitePiece(board[nIndex + 10]))
addMove()
if(onBoard(nIndex + 15) && !whitePiece(board[nIndex + 15]))
addMove()
if(onBoard(nIndex + 17) && !whitePiece(board[nIndex + 17]))
addMove()
if(onBoard(nIndex - 6)  && !whitePiece(board[nIndex - 6]))
addMove()
if(onBoard(nIndex - 10) && !whitePiece(board[nIndex - 10]))
addMove()
if(onBoard(nIndex - 15) && !whitePiece(board[nIndex - 15]))
addMove()
if(onBoard(nIndex - 17) && !whitePiece(board[nIndex - 17]))
addMove()

int onBoard(int index) {
    return !(index & 0x88);
}
```

Figure 2.10: Correct Knight move generation using 0x88 board representation

Just as simple and maybe even more elegant than the 120-board solution. In the onBoard function, the number we want to check is bitwise AND'ed with 0x88, if the AND operation returns 0, we know for sure we are inside the board and onboard returns TRUE. As explained, while the -1 in 120-board representation is a value used to check if the input is valid, in the 0x88-board its only garbage used to offset the actual board and are never accessed to check the value, you can see that the onBoard function above does not access the array (it uses only the index).

2.2.4 Bitboards

Bitboard board representation[4] is an example of a piece-centric board representation. Here, each type of piece is represented in a different 64bit value usually contained in an array. One array index hold the 64bit value for white Pawns, one for black Pawns, one for white Knights and so on. Bitboards are used to generate moves very fast, as this can be done using bitwise operations.

Pawns from opening position

```
1
2     int64 whitePawns = 0x00FF000000000000; //The hex value
3
4     Which in bits is this:
5     0000000011111111000000000000000000000000000000000000000000000000
6
7     Visualised as a board (now with a1 as commonly shown in the lower left
       corner):
8     0 0 0 0 0 0 0 0
9     0 0 0 0 0 0 0 0
10    0 0 0 0 0 0 0 0
11    0 0 0 0 0 0 0 0
12    0 0 0 0 0 0 0 0
13    0 0 0 0 0 0 0 0
14    1 1 1 1 1 1 1 1
15    0 0 0 0 0 0 0 0
16
```

Figure 2.11: Bitboard example, white Pawns. Each type of piece has its own 64-bit value

Here it can be seen that all the bits at the white Pawns starting position are ticked off, while below all the bits at the white Knights starting position is ticked off. A bitboard only contains information of a single type of piece, hence why bitboards are piece centric.

Knights from opening position

```
1  int64 whiteKnights = 0x4200000000000000; //The hex value
2
3  Visualised as a board:
4  0 0 0 0 0 0 0 0
5  0 0 0 0 0 0 0 0
6  0 0 0 0 0 0 0 0
7  0 0 0 0 0 0 0 0
8  0 0 0 0 0 0 0 0
9  0 0 0 0 0 0 0 0
10 0 0 0 0 0 0 0 0
11 0 1 0 0 0 0 1 0
```

Figure 2.12: Another bitboard example, now with white Knights

Bitboards generate moves very efficiently. Now that 64 bit CPU's have become the standard a full bitboard can fit in a single register in the CPU, which makes the bitwise operations even faster. As of now, bit-boards is the standard board representation method used by all top engines in the world, for example Houdini, Komodo and Stockfish.

While bitboards are very fast when calculating moves, they are not very practical when it comes down to finding out what specific piece is on a specific square. The information is spread out onto different 64 bit integers - it is not very practical to access one and one bit to find info on the various pieces. Therefore it is practical to include another board, one that is not

piece-centric, just for the engine to be able to access a specific square faster. For this the naive board representation is perfect as you do not generate the moves, you only use it to access information of a specific square.

Knight Move Generation Example

The way bitboard move generation works is that there is a lookup-table at each square for all the different pieces on the board. The table contains possible moves for all positions on the board.

Lets assume we start from the opening position:

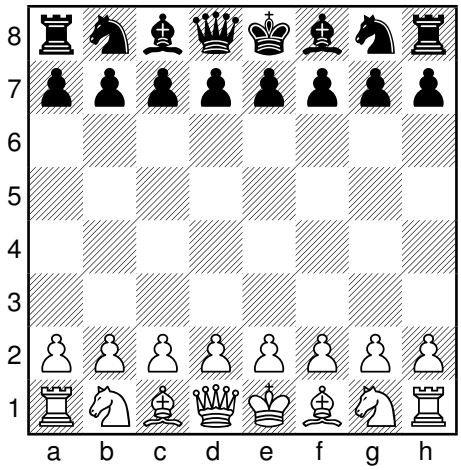


Figure 2.13: Opening Position

To generate moves for the Knight on b1 the engine must first create a target bitmap, which contains all the squares on the board the Knight can move to, ie. squares that does not contain white pieces. Then do a lookup of b2 in a lookup-table for Knight moves, this lookup-table is generated when first initializing the engine on startup. Finally the engine AND these two bitmaps together to get all possible Knight moves.

```
1. squares allowed  2. KnightMoves[b1]:  3. ~whitePieces
~whitePieces:                             & KnightMoves[b1]:
1 1 1 1 1 1 1 1     0 0 0 0 0 0 0 0     0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1     0 0 0 0 0 0 0 0     0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1     0 0 0 0 0 0 0 0     0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1  &  0 0 0 0 0 0 0 0  =  0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1     0 0 0 0 0 0 0 0     0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1     1 0 1 0 0 0 0 0     1 0 1 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
```

Figure 2.14: Moves for the Knight on b1 generated

The result is a bitboard that has all the possible Knight moves, Na3 and Nc3, ticked off.

Sliding Piece Move Generation

With sliding pieces the move generation gets a bit more advanced. There are 2 popular ways to generate moves for sliding pieces, rotated bitboards[4], and magic bitboards[10]. This thesis will focus on the magic bitboards move generation. Magic bitboards are a perfect hashing technique that contains bitboardmasks for all possible moves of a sliding for each square, depending on which squares has a piece that stops them on the way. To generate moves for a Rook the engine must AND all pieces of the board with the Rook moves for the square of the Rook to get occupied squares. This is multiplied with a magic number, explained below, and then bitshifted to get an index in the Rook move array. It is the same as with the Knight, but because the sliding pieces can be stopped we add one additional step and access the Rook moves in a double array instead.

1. ((allPieces & RookMask[a1])*magicNumber) » magicShiftRook[a1]
=
arrayIndex

2. RookMoves[square][arrayIndex] & whitePieces
=
move-bitboard for the Rook

1 ♘c3 e5

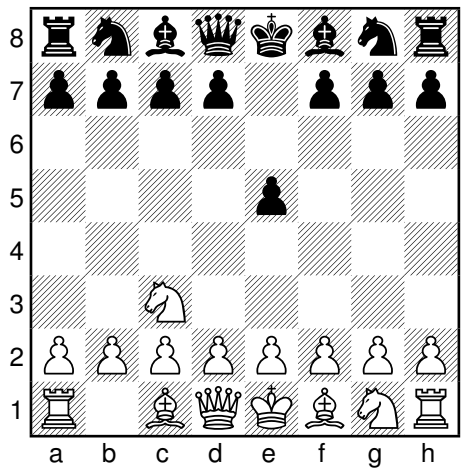


Figure 2.15: Position for the magic bitboard Rook example

```
1. allPieces        2. RookMask[a1]:    3. occupancy
1 1 1 1 1 1 1 1     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1     1 0 0 0 0 0 0 0     1 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 1 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     1 0 0 0 0 0 0 0     0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0     1 0 0 0 0 0 0 0     0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 1 1 1 1 1 1     0 1 1 1 1 1 1 1     0 0 1 1 1 1 1 1


4. occupancy        5. RookMagic[a1]:
1 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  SOME 0 0 0
0 0 0 0 0 0 0 0  *  0 0  MAGIC  0 0   >> magicShift[a1] = arrayIndex
0 0 0 0 0 0 0 0     0 0  BITS 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 1 1 1 1 1 1     0 0 0 0 0 0 0 0

6. RookMoves[a1][arrayIndex]
                    7. ~whitePieces     8. moves
0 0 0 0 0 0 0 0     1 1 1 1 1 1 1 1     0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0     1 1 1 1 1 1 1 1     0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0     1 1 1 1 1 1 1 1     0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0  &  1 1 1 1 1 1 1 1  =  0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0     1 1 1 1 1 1 1 1     0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0     1 1 0 1 1 1 1 1     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 1 1 0 0 0 0 0     0 1 0 0 0 0 0 0     0 1 0 0 0 0 0 0
```

Figure 2.16: Moves for the Rook on a1 generated

The magic numbers have to be generated in advance (not when initializing the engine) and is done in a bruteforce way. The reason to use the magic multiplier is to make the lookup-table more dense and take less space. If one had infinite memory one could make the lookup directly on the index of the occupancy in step 3 instead of generating an array index by multiplying with a magic number.

Magic number In computer chess what is called a magic number simply refers to a pre-generated, and therefore hardcoded, number that is used to generate a smaller, but still unique, index value to look up possible moves for a sliding piece.

As explained in the Knight move generation example, for non-sliding pieces this is pretty easy, and one only needs 64 different bitboards containing the moves for the piece on each square.For sliding pieces, however, this is not enough. A Rook, for example, has multiple different move-bitboards depending on how the other pieces on the board block it. The following example shows how three different occupations on the file of the Rook ultimately should lead to the same move-bitboard for the Rook.

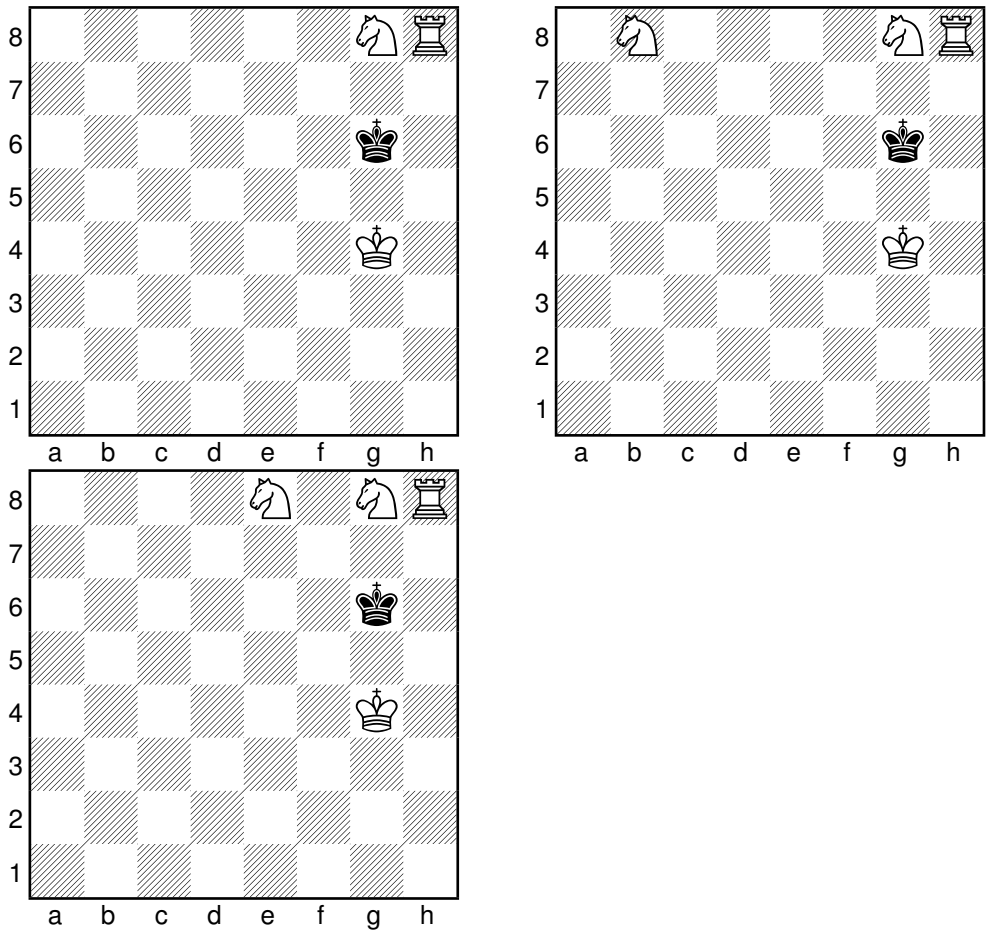


Figure 2.17: Three different positions, with different occupancy, which has the same move-bitboard for the white Rook on h8

```
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
```

Figure 2.18: The move-bitboard for the Rook, note that g8 is still ticked off. This is because the move-bitboard is later ANDed with the bitboard containing ~whitePieces, when finding the move-bitboard we do not seperate between black and white pieces until the end

Applying the magic numbers acts as a surjective function (a many to one relation), where each of the states of the board that should give the same move-bitboard (that was the case in the previous example) will in fact point to a place in the array that contains the specific move-bitboard.

The magic numbers are square specific (meaning that we need 64 of them for the Rooks, and another 64 for the Bishops) and are generated in this way: For each of the 64 indexes of chess squares, generate a random 64-bit number. For each of the relevant occupancy mask for this square multiply it by the random number and bitshift it to get a random index that fits to the pre-decided size of the move-bitboard. If the array index is not taken (or is taken by one of the others with the same move-bitboard) for every occupancy mask, this is a new magic number.

Perfect hashing of move-bitboards Since all indexes of the move-bitboard array (where index is the bithshifted bitboard-occupancy*magic number) are known at initialization, and everyone is either unique or is to point to the same move-bitboard, the move-bitboard array acts as a perfect hashing hash table where no collisions occur.

2.3 Evaluation of a Position

For the minimax algorithm to work a chess engine needs a way to compare different positions. The most common way this is done by chess engines is by having a function which evaluates a given position and gives it a single score. This evaluation is used to search for an optimal move at a given position. Humans have studied chess for hundreds of years and have already built up a good understanding of what makes a chess position good or bad. Most chess engines employ this knowledge to be able to evaluate a position. Chess is not solved yet, there does not exist a clearcut answer of what makes some positions better than others. Therefore, the evaluation function is heuristic in nature, and is the piece of code that requires the most understanding of chess. What makes a chess position good? That is a question every chess-player, even grandmasters, ask and study. The single score given by a chess engine is usually negative if the opposition has the better position and positive if the side to move as the best position.

Example of what an engine could use to score points in a position:

- Material count
  - An engine scores the material count of each side up against eachoter
- Advanced Knight positions
  - Since Knights moves a shorter distance than the Bishop or Rook it can count positively if it is close to 'the action' on the board, or negatively if it is far away from the other pieces.
- Rook-connectivity
  - Connected Rooks is usually a good thing, and can count positively

- Piece mobility (Rooks on open files, Bishops on open diagonals)
  - pieces with many available moves counts positively
- King safety
  - In the early to midgame King safety is crucial to avoid checks that can be detrimental to your position, a safe King, tucked behind Pawns counts positively. On the other hand, a King without any support, or far up the board counts negatively.
- Pawn structure
  - Passed Pawn, connected Pawns and advanced Pawns counts positively
  - Double Pawns, backward Pawns, isolated Pawns counts negatively

The positive and negative scores are added up to a single score of the position. People with some knowledge of chess can see that these things are like taken out of a chess book explaining how to evaluate imbalances in a chess position. What the various items are evaluated in proportion to each other and the value of the pieces is usually done with trial and error.

Bitboards makes it very easy to evaluate some of these metrics. To find if the Rook is on an open file, simply AND a bitboard mask marking the file of the Rook with the bitboard of all the pieces. To find if a Pawn is isolated AND the masked files of each side of the Pawn with the Pawn bitboard of the same color.

The imbalances are evaluated differently in different phases of the game. Standard chess textbooks generally divide the game of chess into 3 phases; opening, midgame and endgame. For chess engines the opening can be handled by using an opening database, containing most frequently used/best answers to a move. The midgame and endgame is evaluated according to the metrics given above, but for the endgame there are some differences to the evaluation.

These can be some of the differences:

- An active King, as opposed to a safe King can count positively
- Rooks *behind* passed Pawns can count positively
- Knights in closed position (positisons with many Pawns) can count positively
- Bishops can count positively in open positions

To avoid a point in the game where the engine suddenly evaluates the position differently, engines usually implements some kind of tapered

evaluation, a gradual transition from the midgame to the endgame. An example of tapered evaluation can be found in crafty[15]:

```
phase = Min(62, TotalPieces(white, occupied) + TotalPieces(black,
    occupied));
score = ((tree->score_mg * phase) + (tree->score_eg * (62 - phase))) / 62;
```

Figure 2.19: How tapered evaluation is done in crafty[15]

The phase is a simple int with a maximum score of 62, the fewer pieces on the board, the more the chess engine uses the endgame score (tree->score_eg) instead of the midgame score (tree->score_mg). The engine always evaluates both the endgame and the midgame score for every position, and puts more and more weight on the endgame evaluation as the game passes and pieces are captured.

2.3.1 Endgame Tablebases

There also exist endgame tablebases for position with 6 pieces or less. One of these is called the Syzygy Bases, created by Ronal de Man[28]. They work by containing all possible positions for 6 pieces or less on the board, with information on if the position is a win, draw or loss. The syzygy tables also has a second table containing a move for the position and a count for moves down to zero, meaning the number of moves before the 50 repetition count is reset. By using the moves in the position the engine will always win if the syzygy tables says the position is a win, but it can make the engine play unnatural, for example sacrificing a Queen, to play a Rook versus King endgame because the Queen sacrifice was the fastest way to reset the 50 move counter.

2.4 The Search Function, Selecting the Best Move

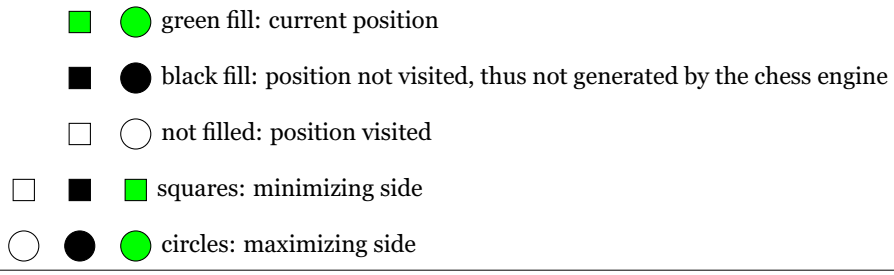
To get to what the evaluator thinks is a good position, the engine has to make the right moves on the board. This is done by the move-selector part of the engine. The prevalent way to search for the best moves among chess engines is some variation of the alpha beta algorithm. Usually with some metrics to make the branches of the tree well-ordered. Meaning that the engine tries to explore the branches that creates the most optimal pruning first.

It is the alpha beta algorithm that uses all of the other parts of the program. It makes and takes moves on the board given, to move up and down the game tree, and when it comes to a leaf-node, evaluates the position and uses this evaluation to choose the most optimal move.

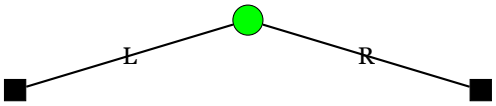
2.4.1 Example of how an Engine Searches a Position

Here is an example of how the alpha beta algorithm communicates with the rest of the program to choose the best possible move. It is simplified to get an easy overview of how an engine can work.

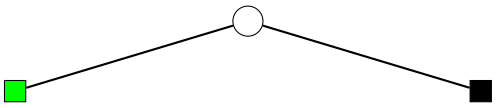
The example begins after the search function has been given a position. The search is to be done to a ply of 4. In this example there are only two possible moves by each side in all positions (which is unlikely, but we will use it for the sake of an easy example).



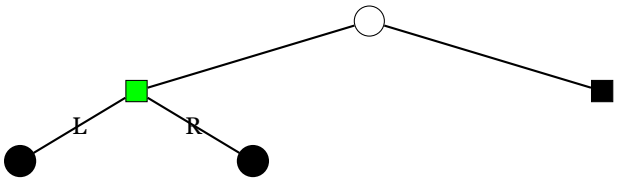
(a) The move-selector starts with the current board, here represented by a green circle. The algorithm is at ply 0 on the maximizing side.



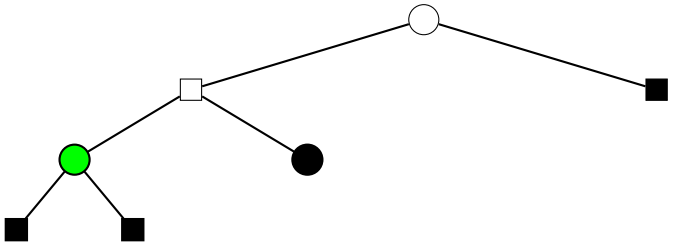
(b) The move-selector calls the move-generator which return a list of possible moves. In this example there are only two moves called L and R.



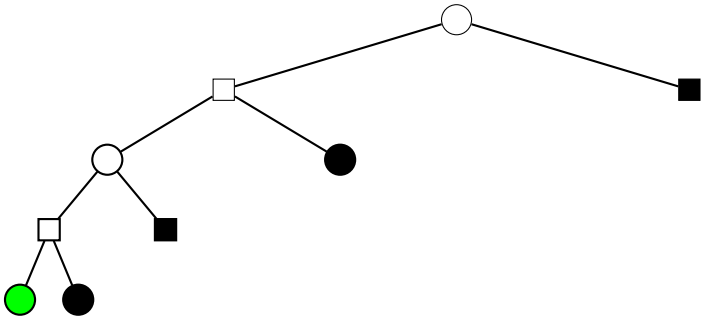
(c) The move-selector now calls the make-move function of the board, with L as argument. The algorithm has now made a half-move and the current board is the leftmost node. The algorithm is at ply 1, and now on the minimizing side.



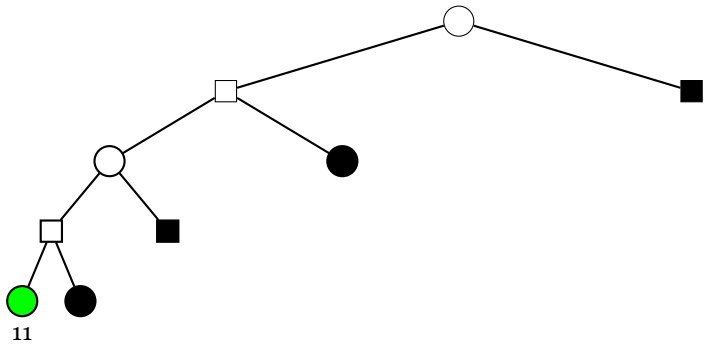
(d) The move-selector once again calls the make-move function of the board. It returns two possible moves, these are obviously not the same moves from the previous position (its blacks turn) but again, for simplicity we call them L and R.



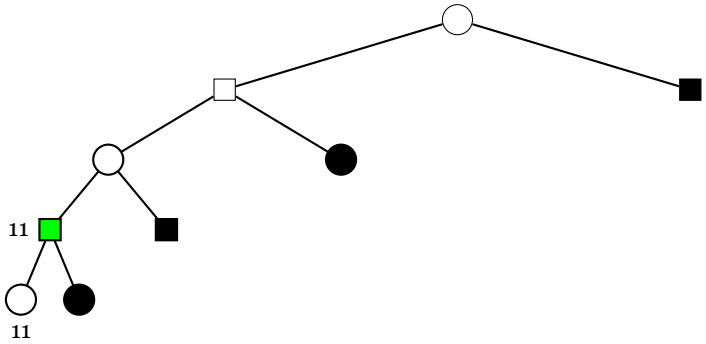
(e) Once again the move-selector calls make-move, with the new L as argument. We are now at ply 2. To speed up the illustration the move-selector has also called the move-generator for moves. The figure omits the move names L and R for simplicity



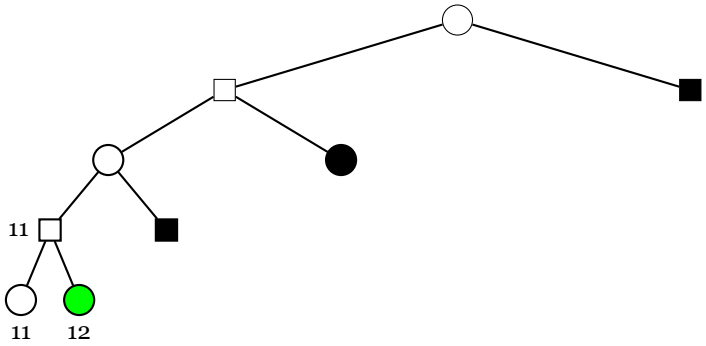
(f) We speed it up a bit more. The move-selector has now called make-move L, generate-moves, and make-move L again. We are now at ply 4. The depth that the this alpha beta-search was to search. Now we can evaluate the position for the first time.



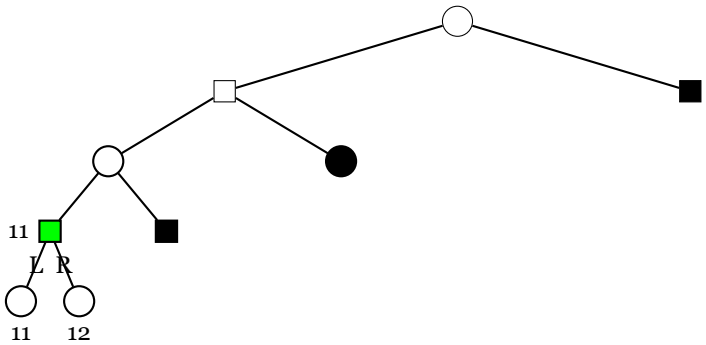
(g) The move-selector calls the evaluate-position function. It returns the value of the position in centipawns. In our example it returns 11.



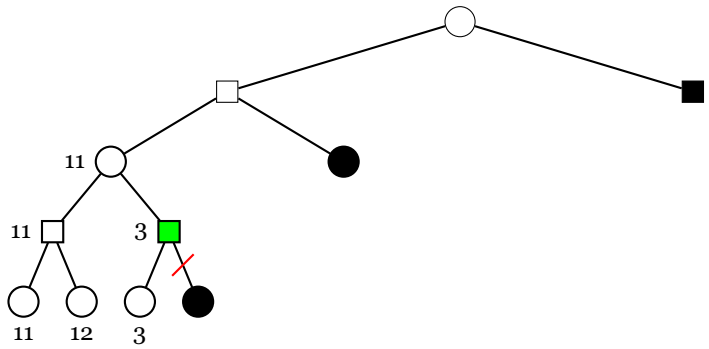
*(h) The algorithm now calls **unmake-move** to move up a level. We are at ply 3 again. 11 is saved as the minimum value this node can make.*



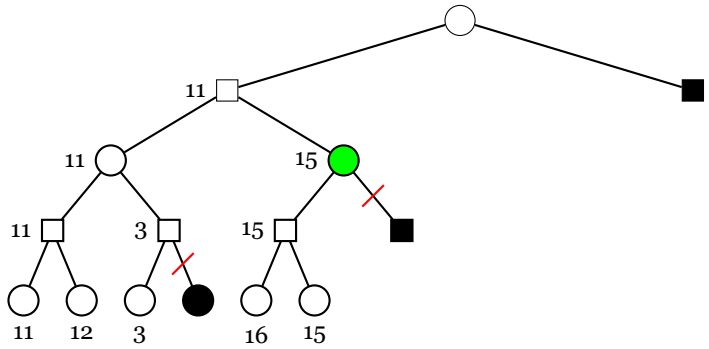
(i) The algorithm now calls make-move with the move R as argument and we move down a level, to ply 4 again. This poisition is then evaluated to a value of 12.



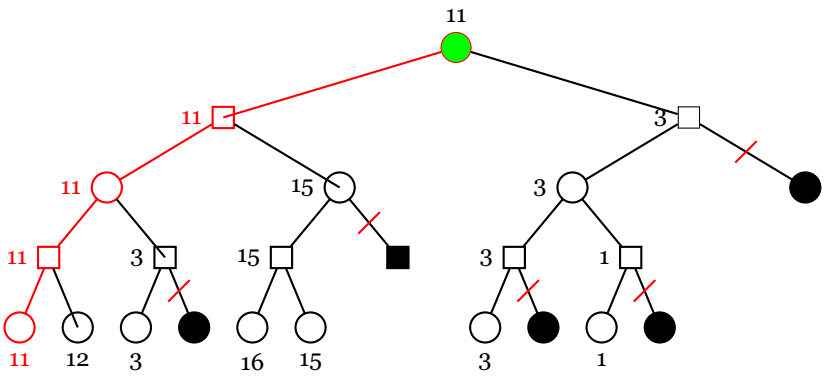
(j) We then unmake the move again, and move up a level, to ply 3. Since we now are on the minimizing side, the value 12 is discarded, as this value is bigger than 11.



(k) There were no more moves to explore in the previous branch, so we once again unmake the move and save the value found, 11. To speed things up a bit, in this drawing the move-selector has called make-move with R as argument, Generated the moves of this position, then called make-move with L as argument, evaluated the position to a score of 3, and then unmade the last move. Now, since the minimizing side can make a move with 3 as a score, the maximizing node above the current node would never choose this branch. It is therefore unnecessary to explore more in this branch, we get a cutoff.



(l) Again we skip a bit ahead. We move up the tree and down a branch that needs exploring. Two of the leafnodes evaluated to 16 and 15 respectively, this is reported upwards the tree as we the algorithm unmake moves and goes back up the tree to the current node. Now we can prune an even bigger branch to the right of the current node. As the minimizing side would never choose this way to go down (it would always choose 11 over 15, and as the current node is a maximizing node its value would be 15 or above).



(m) We skip to the end. The move-selector has moved down the right branch from the top, and some more pruning has occurred. The result saved as the best sequence of moves is drawn with the red line. This path is called the principal variation. And the engine saves this path. The path is reported to the GUI, as well as saved. Used for iterative deepening of the tree, a concept explained in 2.6 Improving the alpha beta algorithm

Figure 2.20: Complete Alpha Beta Example

2.5 The Communicator

To be able to play with the engine, it needs to be able to communicate with the other player. Fortunately, there are standards defined that makes engines able to talk to an interface, so that it can play with humans as well as other engines.

Two common protocols used to talk with a chess interface is the Chess Engine Communication Protocol (CECP) and the Universal Chess Interface protocol (UCI). Both work with most chess interfaces. There are some differences between the different communication protocols, but in the end it is about personal preference to choose one. The engine in this thesis will implement the UCI protocol.

2.5.1 The Universal Chess Interface

UCI is a communication protocol that defines communication between a chess interface and a chess engine. The communication is handled over the standard input (stdin) and standard output (stdout) channels. It was developed in 2000 by Rudolf Huber and Stefan Meyer-Kahlen, and the specifications are available free to download on Stefan Meyer-Kahlens web page shredderchess[30]. The extensive documentation of the protocol can be found there, but here is a rough explanation the most important commands.

Interface commands

- **uci**
  Used to ask the engine if it is capable with the uci protocol.

Engine answer: **uciok**

- **position startpos|fen (moves ..)**
  Gives the engine a position either as a FEN-string or just from the startposition, it can be followed by moves that has been made.

- **go (options)**
  Tells the engine to start searching the position, usually followed by some options to tell how much time the engine has left or the depth the engine should search at. In a game it is up to the engine to decide how much time it will use of what is left.

- **stop**
  Tells the engine to stop search immediately. The engine should answer with what move it has found as the best.
  Engine answer: **bestmove ...**

- **quit**
  Tells the engine to quit the program.

2.6 Improving the alpha beta algorithm

The alpha beta algorithm can be significantly improved from a standard "search to a selected depth"-method. There are various techniques to improve cutoffs as well as simply improving some of the limitations of the algorithm. Many of them relate to the ordering of moves to be searched, since the alpha beta algorithm gets the most cutoffs if the best branch is searched first. Some of the improvements may make the search less reliable, for example by pruning to much. With some optimization techniques it is only through extensive testing one can find if the optimization is worth it or not.

2.6.1 Iterative Deepening

An effective way to achieve a good move ordering is to use iteratively deepening searches. The way iterative deepening works is that the engine starts a search at a low depth, stores the best line of the search, and explore these moves first when searching at a deeper depth. This is done iteratively and highly improves the move-ordering, and thus the efficiency of the program. It can seem a bit backwards to search the same position again and again, but since the minimax algorithm is exponential to the order of 35, the engine saves so much time by improving the move ordering that only a small percent extra cutoff in the next depth will save the engine time. Fortunately enough the iterative deepening deepening also coincides with another restriction in chess, time. The engine can easily control time management by trying to search to a level deeper and just stop mid search if the time runs out, it still has the result from the shallower search.

2.6.2 Aspiration Window

Based on the results from the iterative deepening, the engine can further improve cutoffs by limiting the search space to a value close to the search-value obtained in previous searches[23]. The engine can force cutoffs by guessing that the value of a best move by setting the alpha and beta values before the search begins. This window of moves is called an aspiration window.

The aspiration window is based on guessing, and the engine can therefore end up with a best move that is outside off the window selected. If this happens it will have to do an entire research of the tree, which might be costly. The size of the window has to consider the cost of a research versus the time gained from a narrow window.

2.6.3 Capture Heuristic

Captures often cause a significant change in a position. Many chess engines therefore make sure to search the capture moves first to see if this improves the position or causes a cutoff. The ordering of the captures can for example be done in a mvvlva-fashion, most valuable victim - least valuable attacker. This means that one always captures the most valuable piece of the opposition first. If a Pawn can capture a Queen this is likely to be the best move and should therefore be searched first.

A more advanced capture heuristic is the Static Exchange Evaluation. It works by applying a function that explores all captures, the moves are then ordered by which exchanges appears to give the highest score. Robert Hyatt, the creator of Crafty suggested that in his engine the SEE is not significantly better than mvvlva but saves about 5% of time used searching in Crafty[16].

2.6.4 Killer Move and History Heuristic

Killer moves and history heuristic exploit the fact that chess positions only change marginally by each move made. This means that moves that caused a beta cutoff or improved the alpha score of one position may very well cause a cutoff or improve alpha in another position. If a quiet move causes a beta cutoff the engine saves this move as a killer move. The moves saved should then have a higher priority in other moves of the search, such that they are one of the first to be searched. The reason only quiet moves are saved are because the capture heuristic usually has higher priority. History moves works much the same as killer moves, but here the engine saves quiet moves that improved the alpha of the search.

2.6.5 Transposition Table/Hash Table

By the nature of chess, what is usually referred to as a game tree, is really a directed acyclic graph. Since two different move sequences can end up in

the same position, the alpha beta algorithm can often end up with a position that it has already explored. This is especially true for an engine which explores positions with iterative deepening. Info about the best moves, the cutoffs etc. can significantly improve move ordering. The transposition table is basically a big hash table which contains info about previously explored positions.

Depending on the size the transposition table can fill up quickly and the table often ends up storing positions not used and not likely (or impossible, for example after a Pawn has been captured) to be explored again. This can make the engine slow and not be able to perform efficiently, since the transposition table only consists of false hits, thus making it slower than it would be without a transposition table. It is important to have some techniques to evaluate if a entry is valuable for the program or can be replaced. To find out if the entry is valuable it can be given a score based on one the following criteria. If an entry is close to a leaf node it is more likely to come up again and be searched multiple times, saving the program a lot of time. If an entry is searched to great depth there is done a lot of work on this branch, if the engine hits this branch again it can therefore save lots of time by saving the previous result from the position. The criteria are rather conflicting so some tweaking is usually done on the engine to find an acceptable median. There should also be a check on age to make sure that positions searched a long time ago is replaced. This is usually done by considering the ply of the current position versus the ply of the searched position.

2.6.6 Quiescence Search

Since the alpha beta algorithm searches to one specific depth it is prone to be a victim of what is popularly named the horizon effect. The evaluation function does not make any moves and can not tell if a piece is hanging or not. Say the engine has searched to a depth of 7 and finds out its a whole Knight ahead of the opposition and happily makes the move. What if it were to search to a depth of 8 and find out that it loses a Queen in the next move? Not such a good move to make after all.

To limit the horizon effect the engine therefore should only evaluate quiet positions, position where no captures can be made. Quiescence search simply makes all captures possible in a position before it is evaluated. The ordering of the captures can be the same as explained in *Capture Heuristic*.

2.6.7 Search Extensions

Search extensions is a way to make the alpha beta algorithm search interesting positions more deeply. If the alpha beta search encounters what is believed to be an interesting position it simply decides to search this position deeper than the others. What qualifies as an interesting position is up to the engine to decide. The obvious drawback of a search extension

is that more nodes have to be searched before the algorithm can return and it is therefore wise to use search extensions sparsely. A good use of search extensions is an 'in check'-extension. If the King of the side to move in a position encountered is in check, the alpha beta algorithm searches one ply deeper. Since there usually is a limited number of moves that can be done when the King is in check the branching factor of this subtree is smaller than others.

2.6.8 Null Move Heuristic

The null move pruning technique was described by the programmers of the chess engine *Kaissa* in 1975[1]. It is used as a way to limit search space of the game tree. The idea is that if the side to move passes a move, which is not actually allowed in chess, and the opponent still does not improve the beta score, the alpha beta algorithm can probably assume that this branch will be cut off. The null move search is done on a minimal search window where alpha = beta-1. And is usually done in a shallower tree than the actual depth of the position.

There exist positions where passing a move actually benefits the side to move, so called zugzwang position, and if such a position is encountered it is detrimental to the engine to cut of this branch thinking it has the stronger position. It can be wise not to use the null move in endgame positions, where zugzwang is most often encountered. Or have some kind of zugzwang check to verify that the position is not in zugzwang before doing null move pruning.

2.7 Summary

This chapter presented most parts that is necessary to build a modern chess engine from scratch. It gave a detailed view of how one can solve the issue of representing the board, giving three examples of one can implement and generate moves on each of the board representation, to avoid unnecessary and slow checks that one would need if implementing what is named in this thesis as *the naive board representation*.

It then explained the concept of evaluating a position, giving the engine a metric it can use to create a minimax game tree with weighted leaf nodes, which it can apply an alpha beta algorithm on.

The chapter gave a detailed example of exactly how the alpha beta algorithm is applied for the engine to guess a best possible move.

It was explained that an engine needs a communication protocol to be able to play with other chess engines. One of these communication protocols, the UCI protocol, was then presented with an explanation of the most common commands the UI gives, and the expected outcome/replies of the engine when it gives these commands.

The chapter then gave multiple ways of improving the alpha beta algorithm. It was explained that if the engine searches the 'best' moves first it will get the most cutoffs, the improvements of the alpha beta algorithm is mostly heuristics to better the engines guess of what is the best move in a given position.

Chapter 3

Parallelization of a chess engine

This chapter will present research and techniques used to parallelize a chess engine.

It will begin by presenting the challenges met when trying to parallelize a chess engine using the alpha beta algorithm. The challenges consist of different types of overhead a multithreaded alpha beta search experiences. The overhead is centered around the pruning of the game tree. The sequential algorithm has intermediate results from pervious searches that it can use to better prune the game tree, a multithreaded algorithm must weigh the cost of sharing of this information up against the cost of less pruning.

The chapter then presents some characteristics of a perfectly ordered game tree and how this information can be used when creating a parallel search algorithm.

After this four different alpha beta parallelization algorithms are presented, noting benefits and drawbacks of each of them. The newly found lazy SMP algorithm, the one used to parallelize the engine in this thesis, will be given a more detailed explanation.

For the purpose of simplicity this chapter will use the word *thread* exclusively, meaning an independently executed block of code, but in the topics presented this can be interchangeably switched with processors run on a distributed system, allthough the cost of information-sharing will be higher.

We define that the tree is ordered so that the sequential alpha beta algorithm will always search nodes from left to right in the game tree. The left-most branch of a node is by this definition always searched first and the right-most node is searched last.

3.1 Overheads Experienced in Parallelization of a Chess Engine

The alpha beta algorithm can be said to be sequential in nature. The search uses the intermediate findings to limit the remainder of the search. If the left-most child of a node causes a cutoff, a search over the rest of the nodes children is avoided completely. The heuristics explained in *2.6 Improving the alpha beta algorithm* make this very likely to happen.

Optimizing the alpha beta algorithm on multi-core architectures using parallelization has shown to be challenging. There are multiple algorithms both suggested and used, but they all have some drawbacks, be it bad scalability or insufficient speedup.

Jonathan Schaeffer noted obstacles the parallel alpha beta search has to cope with[40]:

- Communication overhead
- Synchronization overhead
- Search overhead

Communication overhead The communication overhead comes as a result of threads sharing information. Depending on the implementation this can be different types of communication, for example messaging between a master and its slaves. A sequential alpha beta algorithm regularly updates the bounds that limit the search window of further search. If this information is to be shared to all threads working below that node it can cause what Schaeffer dubbed communication overhead. Shared memory can also lead to *synchronization overhead*, with threads waiting for locks to acquire data. In a lockless shared memory environment the communication overhead can be almost eliminated.

Synchronization overhead Synchronization overhead happens when thread has to idle before it can continue working or get new work after its job is finished. This can be a result of various different issues. For example waiting for a lock, which is directly related to the communication overhead. It can also be other things that prevent a thread from working; waiting for other threads to finish their search at a specified depth before it can move on in a iteratively deepening search. The amount of idling can be reduced by making the implementation include more concurrent work, however this will probably lead to more *search overhead*.

Search overhead Search overhead happens because of the larger tree explored in a parallel algorithm. Since the sequential algorithm always has the results of all previous searches it can highly prune the search tree. This is harder to do when multiple threads search the tree in

parallel. Ultimately, more nodes are searched and this reduces the effect of parallelism. Schaeffer gave the following formula to calculate the cost of search overhead:

$$search\ overhead = \frac{paralell\ tree\ size - sequential\ tree\ size}{sequeantial\ tree\ size}$$

The search overhead is strongly related to the communication overhead, without any information-sharing between threads, redundant or unnecessary search of branches is hard to avoid.

3.2 Characteristics of a Minimal Alpha Beta Tree

A minimal alpha beta tree is what is produced when the alpha beta search is done on a perfectly ordered game tree, meaning that the leafnodes to the left always holds a higher value than the ones to the right. In the paper *'An Analysis of Alpha-beta Pruning'*, Donald Knuth and Ronald Moore observed that a minimal alpha beta tree consists of three types of nodes and named them 1, 2 and 3[24]. This paper will use the terminology used by among others Robert Hyatt[17], where the names 1, 2 and 3 are dubbed PV-, CUT- and ALL-nodes.

Knuth and Moore observed that nodes in a perfectly ordered game tree searched using the alpha beta algorithm share one of the three following characteristics.

- PV-nodes: All children of a PV-node must be explored, the left-most child is a PV-node, the rest are CUT-nodes. The root node is a PV-node.
- CUT-nodes: Only the left-most child of a CUT-node has to be explored. This child is an ALL-node.
- ALL-nodes: All children of an ALL-node must be explored. All children of an ALL-node is a CUT-node.

This gives us some ideas of how a parallel alpha beta algorithm can be implemented. The thing to note here is that *all* the children of PV-nodes and ALL-nodes have to be searched, while only a single child of the CUT-node has to be searched.

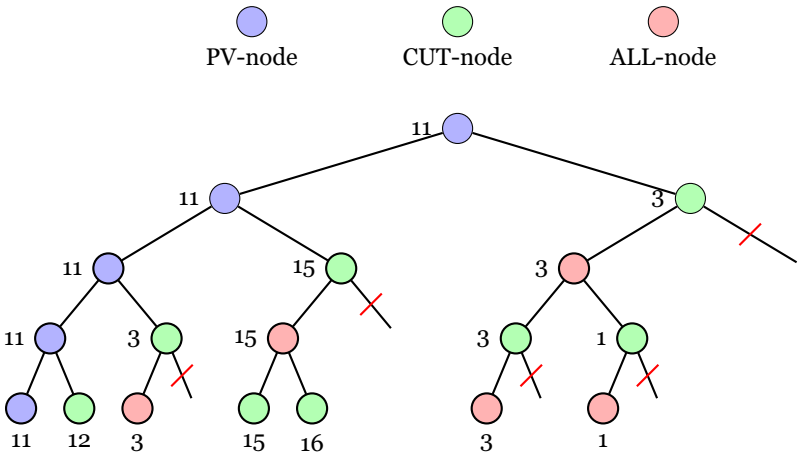


Figure 3.1: Node types in a minimal alpha beta tree.

3.3 Principal Variation Splitting (PVS)

Principal variation splitting, abbreviated PVS, is a simple multiprocessing algorithm first proposed by T. Anthony Marsland and Murray Campbell in their paper *'Parallel Search of Strongly Ordered Game Tree'*[29]. The idea behind the algorithm is that no cutoffs in a successor of a node can happen before at least one lowest acceptable bound (alpha value) is known at the root node of that branch. Before any parallelization happens in PVS the leftmost branch of the full game tree is explored.

The algorithm in itself is very simple. A main thread is given the rootnode. It walks down the left-most branch until a leaf-node is found. The main thread then evaluates the leaf-node and returns with the result to the father of the leaf node. When backing up, the tree it first updates the alpha-score of the current node, before it distributes the rest of the nodes children to idling threads. When all threads have finished searching their branches, giving the result to the main thread, the main thread can return to the previous ply with a new best score. The main thread then distributes the rest of the children of this node, and so on, all the way up to the rootnode. The algorithm can be further improved by doing the parallel search in a null window, where beta = alpha + 1, and only if the search fails low, meaning the branch returned alpha, the window is adjusted.

The PVS approach has several issues, as noted by Robert Hyatt[17]. One is the issue of scalability. Since the algorithm only splits the PV-nodes, and chess positions have an average of about 35 moves in each position the algorithm can not scale beyond this number of threads. Another issue is load balancing. Each thread is given one child of the PV-line to search. The corresponding subtrees of these childs can be vastly different in regards to size. This leads to synchronization overhead as threads have to wait for the others to be done with their sub nodes before the main thread can move up the tree and create more concurrent work.

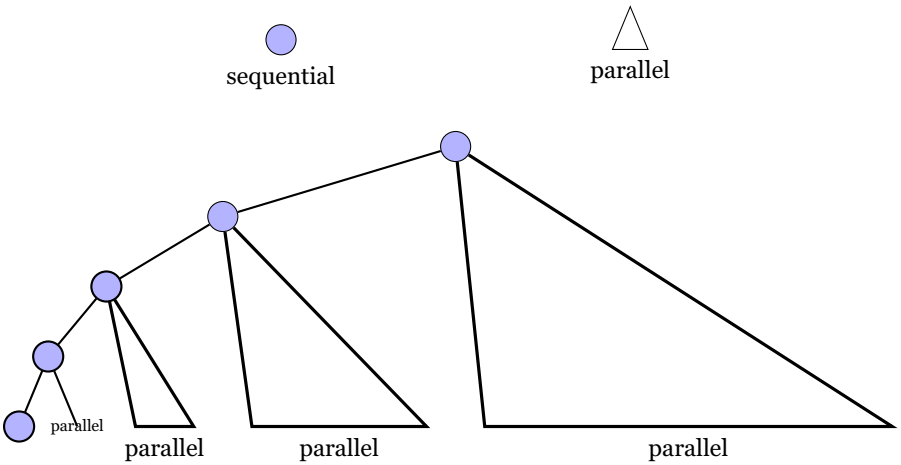


Figure 3.2: Principal variation split, a single thread searches down the leftmost path, evaluating the first leaf-node. As it returns upwards it updates the nodes alpha value and splits the remaining children to threads.

3.4 Young Brothers Wait Concept (YBWC)

Young brothers wait concept is a parallel algorithm proposed by Roger Feldmann. He presented an engine playing with the algorithm in his paper from 1991, *'A Fully Distributed Chess Program'*[9].

YBWC is a further improvement of the PVS algorithm. The concept of the algorithm is: The eldest son of any node must be completely evaluated before the young brothers of that node may be transmitted. The idea is much the same as PVS, that the search overhead is reduced dramatically by allowing the leftmost branch of a node to be completely explored before the other nodes can be searched. YBWC is even more strict than PVS in that every new node must explore their eldest son (meaning the left-most node) first, there is no distinguishing between the PV-nodes and the rest. In Feldmanns implementation, with the exception of starting the search at the rootnode, threads get jobs by what he dubbed work-stealing. When a thread completes its search of the eldest son, it adds the rest of the children to a queue. That way all idle threads can go to the queue and 'steal' work.

3.5 Dynamic Tree Splitting (DTS)

Dynamic tree splitting is an algorithm invented by Robert Hyatt and his Cray Blitz team[17]. It was implemented in his engine in the 1994 ACM computer chess event. Despite the impressive speed-up achieved in Cray Blitz this algorithm has not gained popularity among the top chess engines of today. A reason for this could be that the algorithm is rather complex, and even Robert Hyatt does not have the full implementation in his new engine Crafty[35].

The idea behind the algorithm is that a thread should never be idle, and splits should only happen at PV- or ALL-nodes, these splitpoints are a guesstimate. All threads start as idle and are queued to get work from a list of active splitpoints. This list starts with only the root-node, thus only one thread gets work at the beginning. After the first thread has searched the left-most path it pushes the nodes as splitpoints and the idle threads can start searching them. Now comes the more complex part: Immediately after a thread is finished and there are no more jobs left on the active splitpoint-list it broadcasts this to the other threads. The working threads share their active search trees to the idle thread. This thread then searches the trees for a a suitable splitpoint, hopefully an ALL-node, and pushes this to the queue. If a splitpoint causes a cutoff this is immediately shared to the threads working on this node and the search of the nodes below those threads are stopped.

3.6 Lazy SMP

Lazy SMP is a newly discovered parallel algorithm where threads have minimal to no communication between them except a shared hash table. It was first described by Daniel Homan in a computer chess forum[14]. He experimented with implementing parallelization in his engine, and to his surprise a very simple implementation gave him good results with about 1,6 times speedup with 4 threads on 4 cores. His implementation has been tweaked further and in January of 2016 Stockfish 7 was released with lazy SMP, replacing their previous YBWC-implementation[43].

The idea behind the algorithm is quite simple. With the use of a shared hash-table, let all threads search the full game tree and race to the finish line. The hash table makes sure that positions fully explored previously by any threads do not have to be re-explored. Some effort is put into nudging the threads to search down slightly different paths. For example; in Daniel Homans implementation he let half the threads search at depth+1, they also had local history and killer tables, which may make the threads evaluate children differently and thus search down sub-trees in a different order.

Lazy SMP is very different from the other search algorithms. By using a lockless hash-table it avoids having any communication overhead or synchronization overhead. It trades this by having a larger search overhead.

3.6.1 Search Overhead in Lazy SMP

Because there is close to no communication or synchronization overhead in lazy SMP, the main challenge of the algorithm is to reduce search overhead. Lazy SMP does not know if another thread has *started* its search down the branch of a node if that search is not completed (if the search is completed this is reflected in the hash-table). This causes search redundancy between threads that searches at the same depth.

One can completely avoid this type of search redundancy by making all threads search at different depths. This will not completely solve the problem of search overhead because it causes a different problem; that the threads does not get any value from iterative deepening. The threads will not get the benefits of a better move-ordering and aspiration window that a single thread will, because searches in shallower depths has not yet finished.

If one consider the properties of a perfectly ordered game tree, one can quickly make the connection that making threads search down the same sub-tree is not always a bad thing. If the current node is in fact a CUT-node, there is no benefit of searching down a different path, as this causes unnecessary work that is avoided in a single threaded implementation. The benefit of searching down different paths is only there if the thread is currently in an ALL-node. The game tree of a chess engine is not perfectly ordered, and there can therefore be some benefit of searching down all viable paths in every node, because the beta cutoff can occur in the sub-branch that is not necessarily the first one to be explored.

3.6.2 Lazy SMP, Private Heuristics

Since threads only share the information in the hash table, each thread has its own private killer move and history heuristic. By happenstance this actually benefits the lazy SMP algorithm. If a position encountered already exist in the hash-table it will not search down this path and will therefore not update its history and killer values as it otherwise would if the branch was searched normally. This causes the threads heuristics to be slightly inferior to the heuristics of the thread that actually explored the branch and updated the hash-table. Because of this the threads are very likely to choose slightly different paths when searching the game tree. Both threads searches close the optimal path, so they can benefit from each others result.

3.7 Summary

The different overheads of a parallel chess engine can be categorized to three different types; *communication overhead*, a result of information-sharing between threads, *synchronization overhead*, a result of threads idling - either waiting for specific resources such as locks or waiting for other threads to finish their work, and *search overhead*, which happens because the game tree searched is bigger in a multithreaded algorithm.

The characteristics of a minimal alpha beta tree can help give an idea on how to implement a parallel alpha beta algorithm. All nodes of a minimax game tree can be put into three categories, PV- CUT- and ALL-nodes. If work is split by making threads search down unique sub-branches, unnecessary work is only done if threads are split at a CUT-node. Since the chess engines game tree is not perfectly ordered, guessing what is a CUT-node is hard.

PVS, YBWC and DTS are all algorithms that make an attempt at reducing the splitpoints at CUT-nodes. DTS was the most successful, but is so complex that it is not popular among chess engines today.

Lazy SMP has another approach entirely, where threads do not share to the other threads what branches they are currently searching. However, if a thread comes to a node that is fully explored it can use the results directly, as the results are shared in a hash table.

Chapter 4

Design and Implementation

This chapter gives the design and implementation choices of the chess engine built in this thesis. As per the *design paradigm* formalized by the ACM board of education, it will first state the requirements of the chess engine. The requirements are a rough description of what the engine has to do to achieve the goals of this thesis.

Following are the specifications of the engine. While the requirements tells what the engine has to do, the sepcifications details how we plan to do it. The specifications gives some criterias that the implementation of the engine has to follow to be able to meet the requirements.

After this, a more specific design and implementation of the engine is presented. Here, it is discussed what the design and implementation choices of the engine were. On each of the implementation-decisions some reasoning is given as to why it was decided to implement the engine this way.

The last specification of the engine is a couple of debugging functions. Some of these are accompanied with a more detailed explanation of how they where used to find bugs.

4.1 Requirements

The aim of the thesis is to implement a complete chess engine and investigate how this engine can be improved by using threads on a multicore system by using the parallel algorithm lazy SMP.

The means of which to achieve this goal were decided to be the *design paradigm* formalized by the ACM board of education explained in *1.5 Research Method*. The first step of this research method is to state requirements of the program. To be able to achieve the goal of this thesis, the engine implemented has the following requirements.

A complete understanding of all the rules The chess engine should know all the rules of chess. It should never make an illegal move. It must

know all the current FIDE draw rules, this also include time limits - of which it should have a way of determining how much time it can use per move.

Play decent chess The engine should be somewhat good at chess, at least equivalent of a strong club player with an estimated 2100 rating.

Communicate with a UI The engine should be able to play with both humans and other engines, this should be done through the means of a chess UI.

Take advantage of multicore CPUs The engine should have a way of spawning threads and utilize these in a way that makes it play better chess.

4.2 Specifications

To fulfill the given requirements the engine has the following specifications.

A board representation, with corresponding move generation The chess engine should have a way of internally representing the board. The board structure implemented must be able to efficiently generate moves so that the engine can quickly search through a game tree.

A move structure The moves generated should be saved in a structure containing all the information needed to make and unmake the move on the corresponding board structure. The move structure has to be small, no more than 32 bits long, so that the engine keeps memory usage low while traversing the game tree.

An evaluation function The evaluation function of the engine should take a board structure and return a single score representing the evaluation of the position. This evaluation is to be used to weigh the leafnodes of the game tree. The evaluation function should differentiate between early- and midgame using tapered evaluation.

Alpha beta search function The engine is to be implemented with an alpha beta search function. To limit the search space and enhance the pruning capabilities of the algorithm the alpha beta algorithm must have the following improvements.

- Iterative deepening
- Aspiration window
- Capture heuristics
- Killer move and history heuristics

- Quiescence search
- Search extensions when the King is in check
- Null move pruning

While searching it should call a function to check if theres time left in the search. This call should happen every 2048 nodes searched, so it does not spend too much time checking if there is time left, but still exits before the time runs out.

Thread safe hash table The engine should have a hash table that contains the score of positions already searched. Since the engine utilizes multiple threads that access this table it needs to be thread safe.

UCI shell The engine should have a UCI shell. One thread, independent of the other threads and not used in the search, is set to communicate with a UI using the UCI protocol. The communication is to be done through the standard communication ports (stdin and stdout). This thread gets the amount of time left in the game by the UI. To make sure the engine never run out of time, it should always set the time limit of the search to 1/30 of the total game time left.

Lazy SMP algorithm The engine is to be parallelized using the lazy SMP algorithm.

Debugging functions To make certain that the program contains a minimal amount of bugs it should be equipped with debugging functions that picks up if the engine is incorrectly implemented. The debugging functions have to be able to do the following.

- Make sure that the engine is implemented in a way that it does not make an illegal move
- Assert that the input of the most vital functions are correct
- Make sure that evaluation is the same for both sides of the board, always. The engine should play equally good given black and white pieces.

4.2.1 Programming Language and Frameworks

The engine will be implemented using the C programming language. It was quickly decided that the implementation should be done on a programming language the author is comfortable with and has experience using. The choice therefore came down to either C or Java. The choice tipped in C's favor because it performs better than Java on most applications[34] and

the author has previous experience of implementing a chess engine using the C language.

The threads will be implemented using POSIX threads, the standardized C language threads programming interface. This decision was made based on the authors previous experience using POSIX threads. No other alternatives were considered.

The GCC compiler is used to compile the program. By defining a single compiler for the program some compiler-specific optimizations can be used.

4.3 Boardrep Representation

The engine in this thesis will use the bitboard board representation. The bitboard board representation is in many ways more complex, and is therefore a source of more errors, than the 120-board or 0x88 board representations. While there are some reports of a 2x speedup when switching from a 0x88 board representation to a bitboard board representation[12], Tord Romstad, one of the authors of Stockfish, claims that board representation barely matters and is more a matter of taste[39]. Besides speed there are other advantages to the bitboard board representation. When evaluating a position some metrics are faster and easier to calculate when having bitboards available. One example of this can be 'Rooks on open files' and is explained more detailed in section *4.5 Evaluation*.

There are some things the engine needs to do that the bitboard board representation does somewhat poorly. Since it is a piece-centric board representation it is inefficient to use when trying to find what piece (if any) stands on a specific square. To simplify this, the engine also has a naive board representation used only to extract what piece stands on a specific square. For example, this makes it faster to find out if any pieces are captured when making a move, so that the engine can remove that piece from the corresponding bitboard.

4.3.1 Move Generation

There are mainly two ways to generate moves from bitboards, the rotated bitboard fashion and the magic bitboard fashion. Early on in the thesis it was decided to focus on only one of these ways to generate moves in bitboards, the magic bitboards. When the choice of board representation fell on bitboards, it was natural to use magic bitboards to generate moves.

The engine will generate moves in a magic bitboard fashion, utilizing the perfect hashing move generation technique explained in *2.2.4 Bitboards* (note that these hash tables contains bitboards of moves and are not the same as the hash table used in the alpha beta search).

The move generation will be pseudo legal, meaning it will not check if the side to move's King is put in check by making the move. One of the reasons for doing it this way is that the move generation will have less special cases. To still adhere to the requirement of never playing an illegal move, an extra check will be implemented in the make-move function; if the side to move's King is in check after the move has been made it will unmake the move on the board and return false, making the engine ignore the move. Because of the alpha beta pruning many of the moves generated will never be explored, so it can possibly save time to not do the *'King in check'*-check when generating the moves.

4.4 Move Structure

The engines move structure will contain the following information.

- *From square* - a 6 bit index (0-63) of where the piece moves from.
- *To square* - a 6 bit index (0-63) of where the piece moves to.
- *Piece moved* - 4 bit representing the piece moving.
- *Captured piece* 4 bit representing the captured piece (0 is no piece).
- *Promoted piece* - 4 bit representing the piece promoted to (0 is no promotion).
- *En Passat bit* - ticked off if the piece was captured en passant.
- *Castling bit* - ticked off if the move was a castling move.

This information will be packed into a 32 bit integer to take up as little space as possible. Following is a visualization of the contents of the move-integer.

```
0000 0000 0000 0000 0000 0011 1111 -> From square
0000 0000 0000 0000 1111 1100 0000 -> To square
0000 0000 0000 1111 0000 0000 0000 -> Piece moved
0000 0000 1111 0000 0000 0000 0000 -> Captured piece
0000 1111 0000 0000 0000 0000 0000 -> Promoted piece
0001 0000 0000 0000 0000 0000 0000 -> En passant bit
0010 0000 0000 0000 0000 0000 0000 -> Castling bit
```

Figure 4.1: A visualization of the move structure, the ticked off bits represent what bits are reserved for each piece of information

4.5 Evaluation

To adhere to the requirement of playing decent chess, the engine will need a sufficient evaluation beyond just a simple material count. It will not implement any opening or endgame tables.

The evaluation function will give every position it evaluates a single centipawn score. The score is calculated by weighing two different positional scores against each other, one midgame score and one endgame score, this is explained more thoroughly in *4.5.4 Tapered Evaluation*. The evaluation metrics presented here is not taken from any source and is based on the authors somewhat limited understanding of chess. It was not the focus of the thesis to perfect the engines evaluation function.

The score is calculated piece by piece and added up to a single score. White pieces gets a positive score, black pieces gets a negative score. The pieces are scored this way.

- Base material score
- + Piece-specific tablebase lookup, of the piece's position on the board
- + Piece-specific tweak

4.5.1 Base Material Score

This is the base material score (in centipawns)

- Pawns 100
- Knights 325
- Bishops 325
- Rooks 500
- Queens 1000
- King 20000

The King score is more than all pieces combined and can be seen as infinite (even with 10 Queens on the board).

4.5.2 Tablebase Lookup

A tablebase lookup is added to the base material score of the piece. If the piece stands on a square considered good a positive score is added to the base material score, if it is on what is considered a bad square the score added is negative while a neutral square gets a score of 0. Following is the tablebases used to look up this score.

```
const int mirror[64] = {
    56  ,  57  ,  58  ,  59  ,  60  ,  61  ,  62  ,  63  ,
    48  ,  49  ,  50  ,  51  ,  52  ,  53  ,  54  ,  55  ,
    40  ,  41  ,  42  ,  43  ,  44  ,  45  ,  46  ,  47  ,
    32  ,  33  ,  34  ,  35  ,  36  ,  37  ,  38  ,  39  ,
    24  ,  25  ,  26  ,  27  ,  28  ,  29  ,  30  ,  31  ,
    16  ,  17  ,  18  ,  19  ,  20  ,  21  ,  22  ,  23  ,
    8   ,  9   ,  10  ,  11  ,  12  ,  13  ,  14  ,  15  ,
```

```
    0   ,   1   ,   2   ,   3   ,   4   ,   5   ,   6   ,   7
};

const int PAWN_SQ_VAL[64] = {
    0   ,   0   ,   0   ,   0   ,   0   ,   0   ,   0   ,   0   ,
    10  ,   10  ,   0   ,   -10 ,   -10 ,   0   ,   10  ,   10  ,
    5   ,   0   ,   0   ,   5   ,   5   ,   0   ,   0   ,   5   ,
    0   ,   0   ,   10  ,   20  ,   20  ,   10  ,   0   ,   0   ,
    5   ,   5   ,   5   ,   10  ,   10  ,   5   ,   5   ,   5   ,
    10  ,   10  ,   10  ,   20  ,   20  ,   10  ,   10  ,   10  ,
    20  ,   20  ,   20  ,   30  ,   30  ,   20  ,   20  ,   20  ,
    0   ,   0   ,   0   ,   0   ,   0   ,   0   ,   0   ,   0
};

const int KNIGHT_SQ_VAL[64] = {
    0   ,   -10 ,   5   ,   0   ,   0   ,   5   ,   -10 ,   0   ,
    0   ,   0   ,   0   ,   5   ,   5   ,   0   ,   0   ,   0   ,
    0   ,   0   ,   10  ,   10  ,   10  ,   10  ,   0   ,   0   ,
    0   ,   5   ,   10  ,   20  ,   20  ,   10  ,   5   ,   0   ,
    5   ,   10  ,   15  ,   20  ,   20  ,   15  ,   10  ,   5   ,
    10  ,   15  ,   15  ,   25  ,   25  ,   15  ,   15  ,   10  ,
    0   ,   0   ,   5   ,   10  ,   10  ,   5   ,   0   ,   0   ,
    0   ,   0   ,   0   ,   0   ,   0   ,   0   ,   0   ,   0
};

const int BISHOP_SQ_VAL[64] = {
    -5  ,   0   ,   -10 ,   0   ,   0   ,   -10 ,   0   ,   -5  ,
    0   ,   20  ,   0   ,   15  ,   15  ,   0   ,   20  ,   0   ,
    0   ,   0   ,   15  ,   10  ,   10  ,   15  ,   5   ,   0   ,
    0   ,   10  ,   0   ,   15  ,   15  ,   10  ,   5   ,   5   ,
    0   ,   10  ,   0   ,   15  ,   15  ,   10  ,   10  ,   0   ,
    0   ,   0   ,   15  ,   5   ,   5   ,   15  ,   0   ,   0   ,
    0   ,   15  ,   0   ,   10  ,   10  ,   0   ,   15  ,   0   ,
    -5  ,   0   ,   0   ,   0   ,   0   ,   0   ,   0   ,   -5
};

const int ROOK_SQ_VAL[64] = {
    0   ,   0   ,   5   ,   10  ,   10  ,   5   ,   0   ,   0   ,
    0   ,   0   ,   5   ,   10  ,   10  ,   5   ,   0   ,   0   ,
    0   ,   0   ,   5   ,   10  ,   10  ,   5   ,   0   ,   0   ,
    0   ,   0   ,   5   ,   10  ,   10  ,   5   ,   0   ,   0   ,
    0   ,   0   ,   5   ,   10  ,   10  ,   5   ,   0   ,   0   ,
    0   ,   0   ,   5   ,   10  ,   10  ,   5   ,   0   ,   0   ,
    25  ,   25  ,   25  ,   25  ,   25  ,   25  ,   25  ,   25  ,
    0   ,   0   ,   5   ,   10  ,   10  ,   5   ,   0   ,   0
};

const int QUEEN_SQ_VAL[64] = {
    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
};

const int KING_SQ_VAL_END[64] = {
    -50 ,   -10 ,   0   ,   0   ,   0   ,   0   ,   -10 ,   -50 ,
    -10,    0   ,   10  ,   10  ,   10  ,   10  ,   0   ,   -10 ,
    0   ,   10  ,   15  ,   15  ,   15  ,   15  ,   10  ,   0   ,
    0   ,   10  ,   15  ,   20  ,   20  ,   15  ,   10  ,   0   ,
    0   ,   10  ,   15  ,   20  ,   20  ,   15  ,   10  ,   0   ,
    0   ,   10  ,   15  ,   15  ,   15  ,   15  ,   10  ,   0   ,
    -10,    0   ,   10  ,   10  ,   10  ,   10  ,   0   ,   -10 ,
    -50 ,   -10 ,   0   ,   0   ,   0   ,   0   ,   -10 ,   -50
};
```

```
77
78  const int KING_SQ_VAL[64] = {
79      0   ,   10  ,   10  ,   -10 ,   -10 ,   10  ,   20  ,   5   ,
80      0   ,   0   ,   5   ,   0   ,   0   ,   0   ,   5   ,   0   ,
81      -10 ,   -10 ,   -10 ,   -10 ,   -10 ,   -10 ,   -10 ,   -10 ,
82      -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70 ,
83      -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70 ,
84      -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70 ,
85      -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70 ,
86      -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70 ,   -70
87  };
```

Code 4.1: Table-base lookup. For white a1 is top left, the mirror array is used when looking up values for the black pieces. The other arrays are values for each piece standing on a specific square. For example, when making the move e2e4 the engine will gain 30 points as PAWN_SQ_VAL[e2] equals -10, while PAWN_SQ_VAL[e4] equals 20

4.5.3 Piece Specific Score-Tweaks

To further tweak the score of the piece. The pieces are weighted by using piece-specific metrics.

Pawns The Pawns are the most advanced. An isolated Pawn, which means that there are no Pawns on either of the ranks directly next to the Pawn is given a score of -20 because it cannot be supported by another Pawn. A passed Pawn, meaning there is no opposing Pawn in the three ranks in front of the Pawn, gets a score bonus. The further down the board a passed Pawn is the higher score it gets.

```
1  int WHITE_PAWN_PASSED[8] = {0, 5, 10, 20, 40, 80, 160, 0};
```

Code 4.2: A passed Pawn gets its base and table score added with the PAWN_PASSED score. The further down the board it is, the higher the score added. A passed Pawn on the 7th rank gets an added score of 160.

As an example; a passed Pawn on the 7th ranks gets a base score of 100 + a table base lookup of 20 + a passed Pawn score of 160, a total of 280 points. The passed Pawn score is doubled in the endgame, meaning a passed Pawn on the 7th rank can be worth more than a minor piece.

Bishop and Knights The Bishop and the Knight gets its tweak in the endgame. Knights are given the following score,

score = ((number_of_pawns-10)*6)

Bishops are given the same but opposite score,

((10-number_of_pawns)*6)

Basically it means that, in the endgame, a position with many Pawns gives the Knight a high score, while a position with few Pawns gives the Bishop a high score.

Rooks Rooks get a plus score of 100 in an open rank, meaning there are no Pawns in this rank. It gets a plus score of 50 in a semi-open rank, meaning there are only opposite colored Pawns in that specific rank.

Queens Queens are not tweaked in any way, they will always have a base score of 1000.

Kings The King is the only piece with two tables, one for the early game and one for the endgame. It favors to castle and stand on the first rank in the early game and favors to walk towards the center and get what is called an active King in the endgame when there are fewer threats.

4.5.4 Tapered Evaluation

The engine has a tapered evaluation function to get a gradual transition from the midgame to the endgame. This means that it evaluates both a early game and an endgame score for every position, which are then weighted by the value of all big pieces (Knights, Bishops, Rooks and Queens) on the board.

```
1 phase = (Queens*10) + (Rooks*5) + (minors*3);
2 phase = MIN(64, phase);
3
4 score = (score_mg/(phase+1)) + (score_eg/((64-phase)+1));
```

Code 4.3: The tapered evaluation functions weighs midgame-score versus endgame score. Phase is an integer calculated this way: number Queens on the board multiplied by ten + number of Rooks multiplied by 5 + number of Knights and Bishops multiplied by three. The integer is capped at 64. The lower the number, the more weight is put on the endgame score.

This is derived from the tapered evaluation function in crafty. With fewer big pieces more weight is put on the endgame score.

4.5.5 Material Draw

The engine has a function which tries to detect material draws. This is to avoid trading a Pawn for another piece to gain material advantage if this will cause a draw in the position, or vice versa. It is based on simple rules which can tell if the position is most likely a material draw or not. As an example it will detect that a King versus two Knights and a King is a material draw.

4.6 Hash Table

The hash table will consist of the following items:

- Move
- Evaluation
- Flag
- Age
- Depth
- Hash key
- Checksum

Move The move that, from the search, was valued the best move to take.

Evaluation Evaluation is the score the position, from the search, was evaluated as.

Flag The flag tells if the evaluation saved is either an exact score, a beta cutoff or an alpha value. The exact score means the branch was fully evaluated, the alpha or beta score means the score was outside the search window and is not an exact score.

Age The age tells what ply the board was when the score was updated. The age can be used to remove old entries, meaning entries with a low ply.

Depth The depth tells how deep the search went from the position. If a thread finds an entry in the hash table, but the depth is shallower than what is to be searched, the score can not be used.

Hash key The hash key is a 64 bit hash of the position. The hash is made with a hashing technique called *Zobrist hashing*. Zobrist hashing works by XORing different random 64 bit integers to create an (almost) unique hash of the current state of the position. At startup multiple random 64-bit integers are created to represent each of the following states the position can have.

- 64 integers for each of the 12 different types of pieces (separating between white and black), one corresponding to every square of the chess board.
- 1 integer for side to move
- 4 integers corresponding to castling rights

- 8 integers corresponding to a file with a valid en passant capture

One of the reasons to use Zobrist hashing is that the XOR-operation is own inverse, meaning the operation is undone simply by XORing the same value once more. This means that the hash of the position can be continually updated. When a piece is moved, the engine XORs (to remove) the value corresponding to the square the piece moves from, and then XORs (to add) the value of the square the piece moves to.

Checksum The checksum is what makes the hash table thread safe. If two threads write to the table at the same time the checksum will not compute and the entry is discarded. A more detailed explanation follows.

4.6.1 Lockless Hash Table

To make the hash table thread safe it will be implemented as a lockless hash table. The lockless hash table works by having a checksum of all the values in the table. The checksum is calculated locally, before inserting the values into the table. It then inserts all the values, including the checksum, into the table. If multiple threads writes to the table at the same the entry is broken. If a thread later finds the entry in the hash table, the values in the table will not match with the checksum and the entry is discarded.

The reason it was decided to make the hash table thread safe this way, was to avoid synchronization overhead when accessing the hash table. The hash table is big and entries are sparse, threads will very rarely write to the same hash entry at the same time. Making the hash table thread safe by using locks was therefore deemed an unnecessary step.

4.7 Opening Book and Endgame Tablebases

To reduce the scope of the thesis it was decided to not implement either an opening book or endgame tablebase. The engine would play better chess if this was implemented, but it still plays decent without it, thus following the requirements given.

4.8 UCI-Shell

To be able to play with other chess engines, the engine has implemented support for the Universal Chess Interface. This is necessary, because without some kind of communication interface or GUI an engine is pretty much useless. There are not many differences between the various communication standards and the UCI standard was selected for convenience, as it is the one the author knows the best. The engine supports the following commands defined by UCI[30].

- uci
  - Explanation: Asked by the UI, to see if our engine supports UCI
  - Action: prints 'uciok'
- ucinewgame
  - Explanation: Tells our engine to start a new game from the startposition
  - Action: The engine clears all stored information, for example from the hash table, then sets up the chess startposition
- position FEN-string [moves]
  - Explanation: Tells the engine to set up the following position with an option to tell the engine to also make the following moves
  - Action: The engine sets up the position defined in the FEN-string and makes the moves if given any
- stop
  - Explanation: Tells the engine to immediately stop the search
  - Action: The engine tells searching threads to stop searching and return
- isready
  - Explanation: Used by the UI to ask if the engine is ready, for example after the command ucinewgame command
  - Action: prints 'readyok' when ready.
- go [possible arguments as time left etc, or depth to search to]
  - Explanation: Used by the UI to tell the engine to start searching
  - Action: The engine starts searching with the given arguments as settings
- setoption name Hash value [size]
  - Explanation: Used by the UI to tell the engine to use a hash table of a specific size in MB.
  - Action: The engine frees up the its current hashtable and allocates a new one, with the size specified by the UI.
- setoption name Threads value [count]
  - Explanation: Used by the UI to tell the engine to use a specific amount of threads
  - Action: The engine kills all searching threads and creates new ones.

- quit
  - Explanation: Tells the engine to quit immediately
  - Action: The engine releases all resources, kills the threads then quits the program

All these commands are handled by an IO-thread which does not do any of the actual search itself. The IO-thread is not counted toward threads searching in the parallel search.

4.9 Parallel Search Algorithm, Lazy SMP

A significant part of this thesis is to implement and report test results from the parallelization algorithm lazy SMP. Why the lazy SMP algorithm was chosen is explained in chapter 1, but to reiterate the main point; it has given promising results in multiple engines, but there is a lack of research and knowledge of the algorithm.

The lazy SMP implementation in this thesis has the following traits. Every thread start its search from root, and search to various depths. Every thread is able to report its results to the interface, so for any given depth it is a race to find the answer faster than other threads.

4.9.1 What Information is Shared Between Threads

In this implementation of lazy SMP the threads share only two pieces of information, a depth counter and a hash table.

Depth Counter

We only want a single result from each depth of the iteratively deepening search, as the results from the same depth will always be the same. Since multiple threads are set to search to the same depth we have to have a way for the threads to know what depths are already reported to the interface. This is done with the depth counter. When a thread is done searching a specific depth it will check if the result it found is the deepest yet, if so it will try to grab the lock to the depth counter. When given the lock it will increment the depth counter to its own search depth and then report its result to the interface. Once this is done the lock to the depth counter is released, and new threads can report their results if theirs is the deepest result found yet. This ensures that only one thread reports to the interface at a time and only a single result from each depth is reported.

Hash Table

All threads use the hash table to report the result of every position they explore. When the branch of a position is fully explored, the result of the

search is reported to the hash table. The result can then be reused by any thread. The *evaluation* from the hash entry can only be used if the depth of the entry is big enough for the current search (meaning the same, or deeper than the current search's depth). The *move*, however, can still be used to improve move-ordering.

4.9.2 Private Heuristics

For the lazy SMP algorithm to work well it is important that threads search down slightly different paths of the search tree, so that when other threads eventually search down the same path they find the information already in the hash table. The simplest way this is done is by setting the threads to search to different depths. This is not efficient, especially when having a large amount of threads. The engine implemented search down to a depth of 10-12 fairly quickly. After that the size of the game tree gets too big for a thread to return a result within reasonable time. As a result of this the lazy SMP algorithm commonly sets multiple threads to search the same depth.

4.9.3 Thread to Depth Ratio

As explained, the threads search to various depths. This is to nudge threads to different branches of the tree and make them benefit from the shared hash table. The weighting of the thread to depth ratio is one important aspect of reducing the search overhead. Multiple suggestions have been made as to what is the best ratio, and in *Stockfishs* implementation of lazy SMP this was discussed in detail[38][31]. The thread to depth ratio needs more testing before an optimal value is found, and will most likely be implementation specific. This is because different engines behaves differently when searching, for example some engines value depth highly, whereas others value evaluation.

The engine in this thesis will uses the following thread to depth ratio: Half of the threads should search at current depth + 1, a quarter of the threads should search at current depth + 2, 1/8 of the threads should search at current depth + 3, and so on. For example, with eight threads the thread to depth ratio would be like this:
current depth + 1:
4 threads searching
current depth + 2:
2 threads searching
current depth + 3:
1 thread searching
current depth + 4:
1 thread searching

This is implemented using a modulo N counter, a counter which has a maximum of N states and then resets to zero, where N is the number of threads. The counter, which in the implementation is called a search_id,

is synced across all the threads. For a thread to get the depth it uses the following formula.

current_depth + 1 + count_trailing_zeroes_from_binary(search_id)

Since the binary number 1 has no trailing zeroes the thread that gets search_id 1 searches at current depth + 1 + 0. Whereas a thread with search id 4 searches at current depth + 1 + 2 (since 4 in binary has 2 trailing zeroes).

4.10 Debugging Functions to Assert Correctness of the Program

To assert the correctness of the engine some debugging functions have been implemented. There are two major test-functions as well as an assert-function - a function that throughout the program asserts that values of variables are correct.

4.10.1 Assert Function

The assert function is used throughout the program to assert that the values of variables and parameters are as expected. The assert function is only present when the engine is compiled with a debug flag.

```
1  static void add_move(const S_BOARD *b, S_MOVELIST *list, int from, int to,
       int piece,
2                       int capture, int promoted, int ep, int castling)
3  {
4      assert(valid_sq(from));
5      assert(valid_sq(to));
6      assert(valid_piece(piece));
7      assert(valid_piece_or_empty(capture));
8      assert(valid_piece_or_empty(promoted));
9      assert(valid_bool(ep));
10     assert(valid_bool(castling));
11
12     ...
13 }
```

Code 4.4: An example of how the assert function is used. All the parameters of the add-move function are checked for correctness, checking that the to and from squares are valid squares, that the pieces are valid, as well as that the booleans are valid. The assert function is only called when the debug flag is set during compile time.

4.10.2 Perft Function

A performance test, commonly known as perft, was used to test and control that the engine correctly generates and makes moves. The way perft works is that the engine gets a position, a depth, and a target node count. The

engine then calculates all legal moves, ignoring repetition rules, down to that depth and check the number of leaf nodes in *its* move-tree compared to the target node count. For example; from the starting position white has 20 legal moves, and black has 20 legal responses. A perft test from the start position with depth 1 will have a target leaf node count of 20. A perft test from the start position with depth 2 will have a target leaf node count of 400. Note that only the number of possible end-positions are counted (leaf nodes), and not total number of positions encountered. Hence why the target node count is 400 and not 420 (perft depth 1 + perft depth 2). Since only the end-positions are counted, a branch that terminates before reaching the specified depth, i.e. by mate or stalemate, is not counted. A test suite containing various test positions and their leafnode called perftsuite.epd from *rocechess.ch* was used in this testing[37]. Following is an example of how this function was used to find a bug in the move generation. The position in the example failed at depth 2 of the perft test.

```
Perft from this position:
  +---+---+---+---+---+---+---+---+
8 | n |   | n |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
7 | P | P | P | k |   |   |   |   |
  +---+---+---+---+---+---+---+---+
6 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
5 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
4 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
2 |   |   |   |   | K | p | p | p |
  +---+---+---+---+---+---+---+---+
1 |   |   |   |   |   | N |   | N |
  +---+---+---+---+---+---+---+---+
    a   b   c   d   e   f   g   h
side:W, castle_perm:0, ep:0, fifty_move:0


depth ||    target |     count |
-------------------------------------
    1 ||        24 |        24 |
    2 ||       496 |       493 |
```

Figure 4.2: Perft function example, at depth 2, one can see the count of leaf nodes is off the target count

Already at depth 2 at this position the engine is missing 3 legal moves, as one can see from the last line in the figure. The target number of moves was 496, but the engine only found 493. There was a bug in the move-generator, but it can be almost anywhere and it can be very hard to find if given only this information. To narrow down the search for the bug, the engine was compared against another engine using another function named perft-divide.

4.10.3 Perft-Divide

Perft-divide is a subfunction of perft. Perft-divide outputs all legal moves at at depth 0, and the number of leaf nodes of the sub-branch of each position. The thesis perft-divide function was compared against roce chess engine's perft-divide.

```
Roces perft-divide    Our engines perft-divide
e2f2 24               Move: b7a8N, count: 21
e2d2 24               Move: b7a8B, count: 21
e2e3 24               Move: b7a8R, count: 22
e2d1 24               Move: b7a8Q, count: 21
e2f3 24               Move: b7b8N, count: 5
e2d3 24               Move: b7b8B, count: 22  <==
b7b8 23   <==         Move: b7b8R, count: 24
b7b8 24               Move: b7b8Q, count: 22  <==
b7b8 23   <==         Move: b7c8N, count: 19
b7b8 5                Move: b7c8B, count: 6
b7c8 4    <==         Move: b7c8R, count: 19
b7c8 19               Move: b7c8Q, count: 3   <==
b7c8 6                Move: f1d2,  count: 24
b7c8 19               Move: f1h2,  count: 24
b7a8 21               Move: f1e3,  count: 24
b7a8 22               Move: f1g3,  count: 24
b7a8 21               Move: h1f2,  count: 24
b7a8 21               Move: h1g3,  count: 24
f1g3 24               Move: e2d1,  count: 24
f1h2 24               Move: e2d2,  count: 24
f1d2 24               Move: e2f2,  count: 24
f1e3 24               Move: e2d3,  count: 24
h1f2 24               Move: e2e3,  count: 24
h1g3 24               Move: e2f3,  count: 24
Moves: 24             move count: 24
Nodes: 496            total node count=493
```

Figure 4.3: Perft-divide. The left side is roces perft-divide output, the right side is the thesis's engine output. The arrows shows where the count differs.

The output of the perft-divide function, Roces output is on the left, the engine implemented in this thesis is on the right. The arrows are used to highlight where the count is off. Its only false where a Bishop or a Queen is moving, so the search is already narrowed to a single pieces move-generation (the Queen moves are just Bishop and Rookmoves added together). To specify it even more we make another move, b7c8Q (bxc8Q).

```
d7e7 1                Move: Kd7d6,  count: 1
d7d6 1                Move: Kd7e7,  count: 1
d7c6 1    <==         Move: Kd7c8, count: 1
d7c8 1                move count: 3
Moves: 4
```

Figure 4.4: Perft-divide 2. Again, left is the roce engines output, right is this thesis's engine output. The arrow shows the missing move.

The engine is missing the move d7c6 (Kc6). The bug was related to the King in check-function. The engine falsely thought the King was in check by the Bishop/Queen after moving to c6, while it was in fact not.

4.10.4 Evaluation Flip

Another debug function is the evaluation flip function. This functions asserts that the evaluation is the same when playing black and white pieces. The engine should play equally good given either color.

The evaluation flip function takes a position, evaluates it, and saves the score. It then changes the side to move, and mirrors the board by replacing all white pieces with black pieces, and flipping the ranks and files of the board. It evaluates the board again (now as playing the opposite color) and checks that this score is equal to the original evaluation score.

4.11 Summary

To be able to achieve the goals of the thesis the following requirements were given to the engine; *a complete understanding of all the rules, play decent chess, communicate with a UI, take advantage of multicore CPUs*.

To fulfill the requirements a set of specifications where given. The specifications were loosely based instructions on what the engine should implement. A part of the specifications dealt with programming language and frameworks. Here it was specified that the engine is to be developed in the C programming language. It was also specified that the multithreaded part is to be implemented using POSIX threads.

Then the design and implementation choices of the engine where given. The engine is to be implemented using the bitboard board representation with magic biitboard move generation. The move structure is to be a 32 bit integer which contains the from square, the to square, the piece moved, the captured piece (if any), the promoted piece (if any), an en passant bit (if a capture was done en passant) and a castling bit (if the move was a castling move).

The evaluation function is calculated using a base material score, added with a tablebase lookup as well as a piece-specific tweak. The engine is to implement tapered evaluation, by always evaluating an early- and midgame score and weigh this score based on the number of Queens, Rooks, Knights and Bishops left on the board. The evaluation also is to have a material draw function which tries to detect material draws.

The hash table will be implemented as a lockless hash table to make it thread safe. This is done by having an entry named *checksum*, which is calculated locally by each thread inserting in to the hash table. Later, when a threads gets this entry it can assert its correctness by calculating and

comparing the checksum.

The implementation of the UCI shell was given. Following this was an explanation of the lazy SMP algorithm. The 2 pieces of information shared between the threads are a depth counter, stating which depths are already searched, and the lockless hash table. This means that the each thread has its own private heuristics which can help nudge threads down slightly different paths of the game tree.

The engine has multiple debugging functions. One of the debugging functions is the perft function. Perft is used to assert the correctness of the move generation. The perft function is accompanied by the perft-divide function, a function to localize bugs in the move generation.

Chapter 5

Testing and Results

This chapter presents the result of the testing done on the chess engine. It will first give an explanation of what test metrics are used, then, in short describe the test set, hardware and engine settings used to test the engine.

The test results are presented. The first test gives the reader an idea of the ELO of the engine. The second test gives results that indicate the scaling capabilities of the engine.

5.1 Test Metrics

There are many ways to test a chess engine, and various tests gives different information about the engine.

One of the requirements of the engine was that it should play decent chess. To give the reader an idea of how strong the engine is, a test to estimate the chess engines ELO is given.

The goal of the thesis was to improve the chess engine on multicore CPUs using the lazy SMP algorithm. The main metric chosen to represent this improvement is time to depth. Time to depth gives information as to how long it took the engine to complete a search to a certain depth. This gives a direct indication of speedup of the engine.

With the lazy SMP algorithm threads gets information via the hash table from other threads searching at a higher depth. This means that the results from an engine using multiple threads are at least as good, and maybe better, at any given depth. So the time to depth may not be the perfect metric when compared to playing strength.

5.2 Test Sets

5.2.1 ELO Test

One of the requirements of the engine was that it should play well. A good metric for this is ELO. To measure ELO precisely the engine needs to a lot

of games versus similarily skilled opponents, for example by adding it to the *Computer Chess Rating List*, a site which test engines up against one another. Unfortunately there was not enough time to do it this way. Instead of this it was chosen to run a test to approximate the ELO of the engine.

The ELO test used here is taken from a german computer chess site, and is called BT2630[2]. The way it works is that the chess engine is fed 30 poisitions with a 'move to find'. For each position the engine is given 15 minutes to search for the move. The time in seconds it takes the engine to find the correct move is recorded. If it does not find the correct move a maximum of 900 seconds (15 minutes) is recorded. One then finds the average of the times and subtract the average from 2630 to find the chess engines rating.

$$rating = 2630 - average\ time\ to\ solve\ in\ seconds$$

The ceiling rating of the test is 2630. This is not an exact test, and is only here to give the reader an idea of the strength of the engine.

The engine did this test by reading one position's FEN-string at a time from a file and then set itself to search for 15 minutes. The output of the search were then gone over manually to calculate the rating.

5.2.2 Lazy SMP Test

This test is done on a set of a thousand positions taken from a test set used by Stockfish[31]. For each position a search was done down to a depth of 9 ply. In this test the actual result of the search was not considered. As the engine searches with the same evaluation function the result of the alpha beta function should be the same, with the exception that when the search is done multithreaded threads can get trickles of information from deeper searches of other threads via the hash table.

5.3 Engine Settings

The engine only has 2 settings. Thread count, which indicates number of threads searching and hash table size, which indicates the memory usage of the hash table.

Settings, ELO test:

Thread(s)	1
Hash table size	64 MB

Settings, lazy SMP test:

Thread(s)	1, 2 and 4
Hash table size	64 MB

Table 5.2

5.4 Computer Hardware

Following is the specifications of the hardware the tests were run on. A thing to note here is the 'threads per core'-entry. Normally, new Intel CPUs have hyperthreading. This was turned off after preliminary tests gave bad results. The CPU used has only four cores, and therefore a maximum of four threads were used while testing the engine.

CPU specifications:

Architecture:	x86_64
CPU op-mode(s):	64-bit
CPU(s):	4
On-line CPU(s) list:	0-3
Thread(s) per core:	1
Core(s) per socket:	4
Socket(s):	1
Model name:	Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	8192K

Table 5.3

RAM specifications (two of this memory chip, total 16 GB):

Data Width:	64 bits
Size:	8192 MB
Type:	DDR4
Type Detail:	Synchronous
Speed:	2133 MHz
Manufacturer:	SK Hynix

Table 5.4

5.5 Results, ELO Test

Of the 30 problems given, the engine solved 21 of the problems within the allotted time of 15 minutes. The result of the test gave the engine an estimated ELO of 2238, the positions, moves and times can be found in appendix B.

5.6 Results, Lazy SMP Test

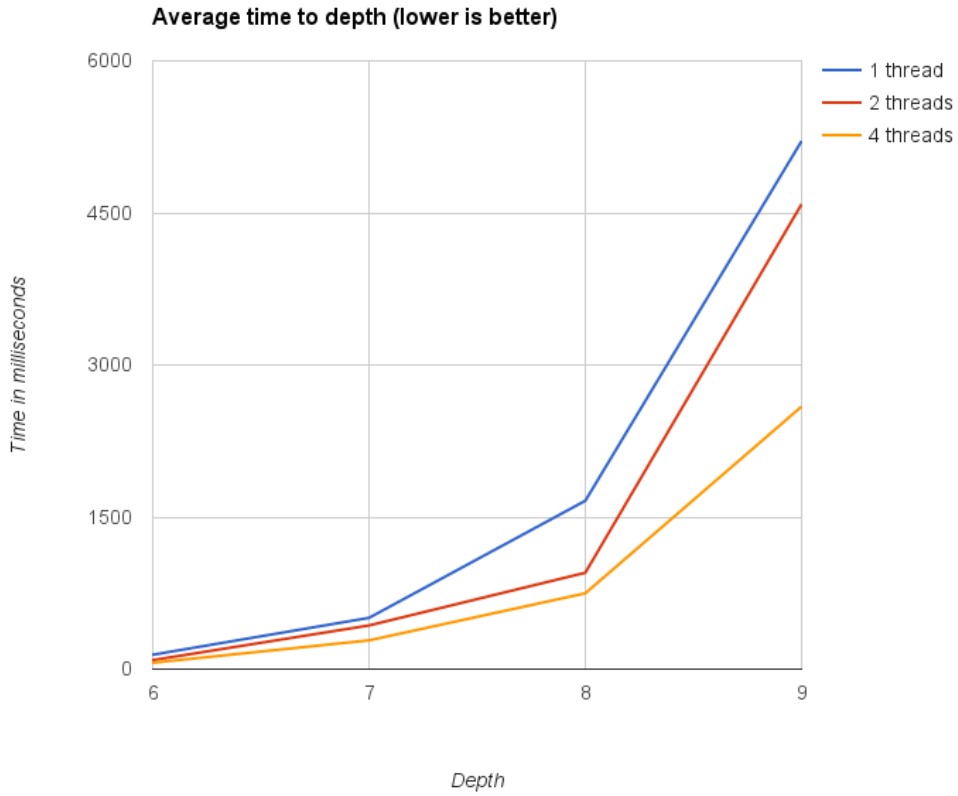


Figure 5.1: Average time to depth, at depth 6-9. Running with 4 threads is about twice as fast as 1 thread.

Average time to depth 9, over the 1000 positions:

Threads	Time (ms)	Speedup (compared to 1 thread)
1	5209	1
2	4586	1.14
4	2590	2.01

Table 5.5

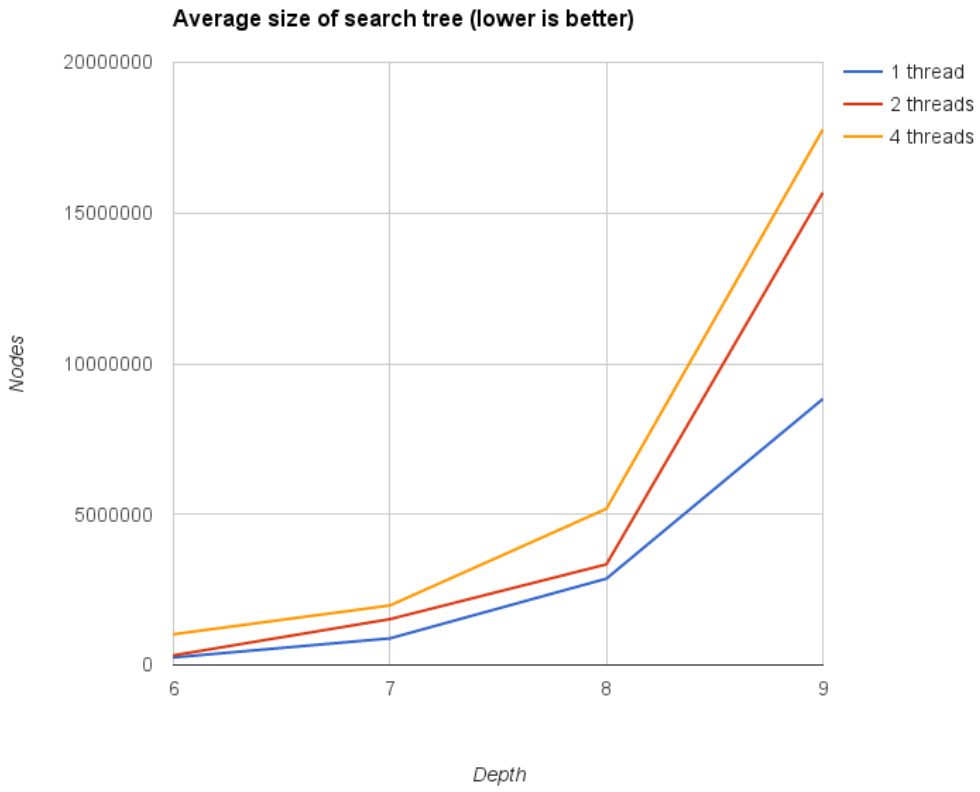


Figure 5.2: The number of nodes searched to reach depth. Also called game tree size.

Average size of search tree, over the 1000 positions:

Threads	Nodes searched (x10 000)	Search overhead (compared to 1 thread)
1	882	0
2	1567	0.78
4	1777	1.01

Table 5.7

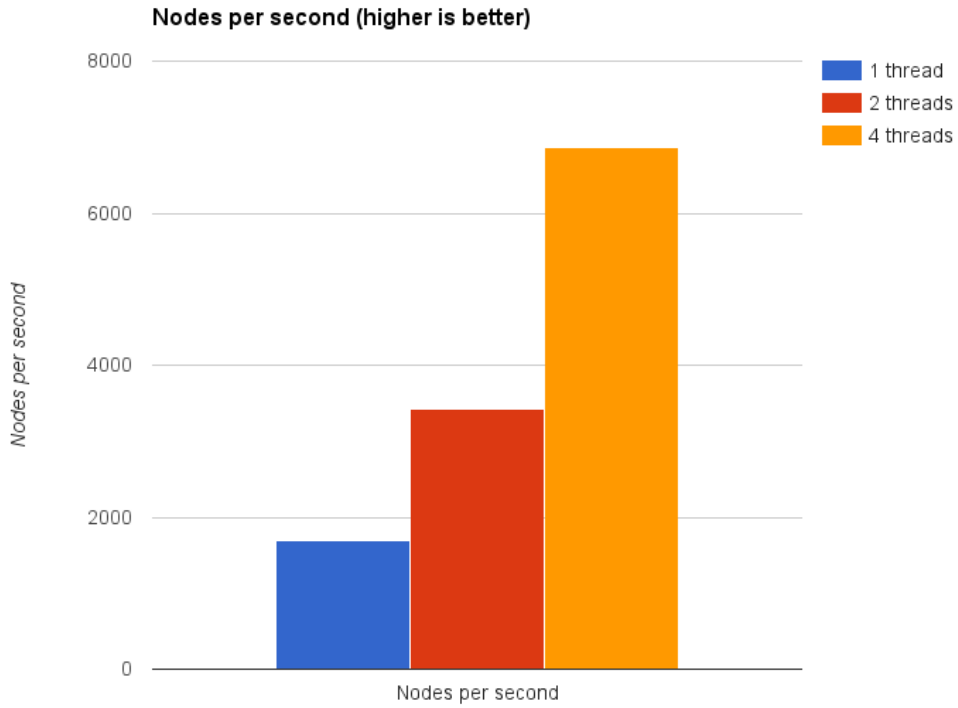


Figure 5.3: The number of nodes per second. Because of the nature of the lazy SMP algorithm, this number should scale 1 to 1 compared to number of threads

Nodes per second:

Threads	Nodes per second	Scaling (compared to 1 thread)
1	1695	1
2	3418	2.02
4	6861	4.05

Table 5.8

5.7 Summary

There are two tests performed in this thesis. One is a smaller ELO test to give the reader an idea of how strong the engine is. Another one is a bigger test which tests the scaling of the multithreaded engine. The metric chosen to represent the scaling is time to depth, which shows how fast the engine searches down to a selected depth.

During preliminary testing it was found that hyperthreading throttled the scaling of the program. All results shown in this the chapter are with hyperthreading turned off.

The ELO test gave the engine an estimated ELO of 2238. The lazy SMP results showed that the engine doubled in search speed using 4 threads compared to 1.

Chapter 6

Discussions

In this chapter the results shown in the previous chapter are discussed. The chapter will first discuss the ELO test, explaining that the test should be taken with a grain of salt. It then discusses the lazy SMP test, discussing how search overhead is the main bottleneck of the lazy SMP algorithm.

6.1 The ELO Test

The ELO test was only included to give the reader an idea of the engines playing strength. It does not do much beyond giving an indication of the engines true ELO and should not be given too much of an emphasis.

The chess engine in this thesis does not have an opening or endgame table. Since the ELO test only contains midgame positions the true ELO of the engine may be lower than indicated.

6.2 The Lazy SMP Test

The main result taken from the lazy SMP test is the time to depth. This shows the actual speedup gained from the threads searching to depth 9. With 2 threads there was a speedup of 1.14, and 4 threads doubled the speed of using 1 thread, with a speedup of 2.01.

The other 2 graphs give some interesting insights to how this was achieved. The first thing to notice in the results is how the nodes per second scales 1:1 with number of threads as shown in figure 5.3 Lazy SMP with a lockless hash table should have no synchronization or search overhead, and this is clearly shown to be the case in the nodes per second result. In fact, the result reports a slightly higher than one to one scaling, however, this is most likely because threads are not suspended from the search when the nodes are counted, so the node count reported is a bit higher than the actual count.

The search tree size graph shown in figure 5.2 shows the search overhead to be 0.78 using 2 threads and 1.01 using 4 threads. This means that when

searching with 2 threads the amount of nodes searched to reach depth 9 were 78% bigger than with 1 thread, and with 4 threads the tree size was 101% bigger than with one thread.

While the nodes per second scales 1:1 with number of threads searching, the bottleneck of the lazy SMP algorithm is the search overhead. Since the algorithm with 4 threads search over twice as many nodes, the speedup of the engine is only doubled using 4 threads versus 1 thread.

6.3 Summary

The ELO test should not be given too much of an emphasis, especially considering how the engine has not implemented an opening or endgame table.

The lazy SMP algorithm had a speedup of 2, using 4 threads. The main bottleneck is the search overhead caused by the lazy SMP algorithm.

Chapter 7

Conclusion

This thesis has shown how the lazy SMP algorithm can be used to speed up a chess engine using threads on a multicore CPU.

The results show that the main caveat of the lazy SMP algorithm is the search overhead. The minimal communication between threads allows the nodes per second to scale 1:1 with number of threads, however, it also causes threads to do redundant work which negates a lot of the speedup.

7.1 Summary

The background research for this thesis found that there were little information concerning the new and promising parallel algorithm lazy SMP. The aim of the thesis was to build a complete chess engine, parallelize it using the lazy SMP algorithm and, with this, shed some light on how and why the algorithm works.

The thesis explained how modern chess engines work and how the nature of the search chess engines use makes them hard to parallelize. Multiple algorithms that try to solve this problem were presented, each with their own drawbacks.

The lazy SMP algorithm is a newly discovered algorithm that with its simplicity quickly made its way into many of today's strongest chess engines. It works differently from the other algorithms in that threads keep communication and synchronization between themselves to the minimum.

The chess engine in this thesis implemented its own lazy SMP algorithm with success, cutting the time to depth 9 in half, showing that even simple chess engines can benefit from this algorithm.

7.2 Future Work

The lazy SMP algorithm works because of some random factors. An example of this can be whether or not a thread gets to write its result to

the hash table before another thread starts searching the position. Another example is the randomness of the private heuristics.

As explained in section *4.9.3 Thread to Depth Ratio*, a lot of focus has been given to improve thread to depth ratio of the lazy SMP algorithm. I do not think this is very important. After the initial tweak it will probably not give any significant speedup. Focus should instead be put on reducing the random factor of the algorithm. Lazy SMP could heavily benefit from achieving close to the perfect scenarios more often. This should not be done at the expense of a higher communication and synchronization overhead. Communication between threads should still be kept minimal, as is the spirit of the lazy SMP algorithm. Following are two suggestions I think can help reduce the random factor of the lazy SMP algorithm.

The threads searching should at times go back and search through its move history to check the hash table for possible new hits. This will make the threads stop redundant search as fast as possible.

The lazy SMP algorithm could heavily benefit from guessing if it is in a CUT node or an ALL node before deciding which child it should explore. If the node explored is a CUT node the best move heuristics should be used, but if the thread is in an ALL node it should choose a random child to explore. Note that I say random, as I do not think more communication between the threads is the solution and selecting a random child should be enough to ensure threads are most likely not searching down the same path. As to how the thread should guess if a node is an ALL or a CUT node, this can maybe be done in the same fashion the DTS algorithm selects splitpoints, although the details of this is not known to me.

Bibliography

[1] Georgy Adelson-Velsky, Vladimir Arlazarov, and Mikhail Donskoy. "Some Methods of Controlling the Tree Search in Chess Programs." In: *Artificial Intelligence* 6.4 (1975), pp. 361–371.

[2] Bednorz-Tönissen. *BT 2639 Test*. (accessed 2016-10-29). n. d. URL: http://www.schachcomputer.at/bt2630.htm.

[3] Alexander Bernstein and Michael de V. Roberts. "Computer v. Chess-Player." In: *Scientific American* (June 1958), pp. 96–105.

[4] B Boskovic et al. "The representation of chess game." In: *27th International Conference on Information Technology Interfaces, 2005*. IEEE. 2005, pp. 359–364.

[5] Mikhail Moiseevich Botvinnik. *Computers in chess: solving inexact search problems*. Springer Science & Business Media, 2013.

[6] Michael Brudno. *Competitions, Controversies, and Computer Chess*. (accesed 2016-07-11). 2000. URL: http://www.cs.toronto.edu/~brudno/essays/cchess.pdf.

[7] chessdom. *TCEC Season 8 - Superfinal*. (accessed 2016-07-11). 2015. URL: http://tcec.chessdom.com/archive.php?se=8&sf&ga=46.

[8] D. J. Edwards and T. P. Harts. "The alpha beta heuristic." In: *MIT memo* (Oct. 1963).

[9] R. Feldmann, P. Mysliwietz, and B. Monien. *A Fully Distributed Chess Program*. 1991.

[10] Trevor I Fenner and Mark Levene. "Move Generation with Perfect Hash Functions." In: *ICGA Journal* 31.1 (2008), pp. 3–12.

[11] Johannes Fürnkranz and Miroslav Kubat. *Machines that learn to play games*. Vol. 8. Nova Publishers, 2001.

[12] Onno Garms. *Speedup by bitboards*. (accessed 2016-11-17). 2007. URL: http://www.open-aurec.com/wbforum/viewtopic.php?t=6651#p31061.

[13] Richard D. Greenblatt, Donald E. Eastlake III, and Stephen D. Crocker. "The Greenblatt Chess Program." In: *Fall Joint Computer Conference* (1967), pp. 801–810.

[14] Daniel Homan. *Lazy SMP, part 2*. (accessed 2016-07-20). 2013. URL: ihttp://talkchess.com/forum/viewtopic.php?t=46858.

[15] Robert Hyatt. *Crafty sources*. (accessed 2016-07-12). 2016. URL: http://www.craftychess.com/crafty-25.0.1.zip.

[16] Robert Hyatt. *mvv/lva vs SEE capture ordering test results*. (accessed 2016-07-18). 1995. URL: https://groups.google.com/forum/#!topic/rec.games.chess.computer/H6XjY2L13eQ.

[17] Robert Hyatt. *The DTS high-performance parallel tree search algorithm*. (accessed 2016-07-19). 1994. URL: https://www.cis.uab.edu/hyatt/search.html.

[18] Hiroyuki Iida, Nobuo Takeshita, and Jin Yoshimura. "A Metric for Entertainment of Boardgames: Its Implication for Evolution of Chess Variants." In: *Entertainment Computing: Technologies and Application*. Ed. by Ryohei Nakatsu and Junichi Hoshino. Boston, MA: Springer US, 2003, pp. 65–72. ISBN: 978-0-387-35660-0. DOI: 10.1007/978-0-387-35660-0_8. URL: http://dx.doi.org/10.1007/978-0-387-35660-0_8.

[19] Intel. *Deep Blue FAQ*. (accessed 2016-07-11). 2003. URL: https://www.research.ibm.com/deepblue/meet/html/d.3.3a.html.

[20] Intel. *Intel's First Microprocessor*. (accesed 2016-07-11). n.d. URL: http://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html.

[21] Intel. *Moore's Law and Intel Innovation*. (accesed 2016-07-11). n.d. URL: http://www.intel.com/content/www/us/en/history/museum-gordon-moore-law.html.

[22] Jez9999. *AB pruning*. (accesed 2016-07-11). 2007. URL: https://en.wikipedia.org/wiki/File:AB_pruning.svg.

[23] H. Kaindl, R. Shams, and H. Horacek. "Minimax search algorithms with and without aspiration windows." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 13.12 (Dec. 1991), pp. 1225–1235. ISSN: 0162-8828. DOI: 10.1109/34.106996.

[24] Donald E Knuth and Ronald W Moore. "An analysis of alpha-beta pruning." In: *Artificial intelligence* 6.4 (1976), pp. 293–326.

[25] D. Levy and M. Newborn. *All about Chess and Computers*. Computer chess series. Springer, 1982. ISBN: 9783540119326.

[26] David Levy, ed. *Computer Chess Compendium*. New York, NY, USA: Springer-Verlag New York, Inc., 1988. ISBN: 0-387-91331-9.

[27] David Levy and Monty Newborn. *How Computers Play Chess*. New York, NY, USA: Computer Science Press, Inc., 1991. ISBN: 0-7167-8239-1.

[28] Ronald de Man. *Syzygy Tablebases*. (accessed 2016-07-13). 2013. URL: http://www.talkchess.com/forum/viewtopic.php?topic_view=threads&p=513255&t=47681.

[29] T Anthony Marsland and Murray Campbell. "Parallel search of strongly ordered game trees." In: *ACM Computing Surveys (CSUR)* 14.4 (1982), pp. 533–551.

[30] Stefan Meyer-Kahlen. *UCI Protocol*. (accessed 2016-07-14). n.d. URL: http://www.shredderchess.com/chess-info/features/uci-universal-chess-interface.html.

[31] Mikael. *Lazy SMP*. (accessed 2016-10-10). 2015. URL: https://groups.google.com/forum/#!topic/fishcooking/GVdyWSWEpQY.

[32] Alan Newell, Cliff Shaw, and Herbert Simon. "Chess Playing Programs and the Problem of Complexity." In: *IBM Journal of Research and Development* Vol. 2 (Oct. 1958), pp. 320–335.

[33] Alan Newell and Herbert A. Simon. "Computer Science as Empical Inquiry: Symbols and Search." In: *Communications of the ACM* 3 (Mar. 1976), pp. 113–126.

[34] benchmark task performance. *Java programs versus C gcc*. (accessed 2016-23-10). n.d. URL: https://benchmarksgame.alioth.debian.org/u64q/java.html.

[35] *Post subject: Re: Explanation for non-expert?* (accessed 2016-07-20). 2015. URL: http://talkchess.com/forum/viewtopic.php?t=55368&start=17.

[36] Joseph Racknitz. *The Turk 3*. (accesed 2016-07-11). 1789. URL: https://commons.wikimedia.org/wiki/File%3aLambdaPlaques.jpg.

[37] rocechess.ch. *perftsuite.epd*. (accessed 2016-10-10). n.d. URL: http://www.rocechess.ch/perft.html.

[38] roha...@gmail.com. *Helper Thread Depths in the Lazy SMP algorithm*. (accessed 2016-10-10). 2015. URL: https://groups.google.com/forum/#!topic/fishcooking/Sq8HJ7Xq0Ww.

[39] Tord Romstad. *RE: Speedup by bitboards*. (accessed 2016-10-17). 2007. URL: http://www.open-aurec.com/wbforum/viewtopic.php?t=6651#p31065.

[40] Jonathan Schaeffer. "Distributed game-tree searching." In: *Journal of parallel and distributed computing* 6.1 (1989), pp. 90–114.

[41] Claude E. Shannon. "Programming a computer for playing chess." In: *Philosophical Magazine*. 7th ser. 41.314 (Mar. 1950).

[42] Kathe Sprcklen and Dan Spracklen. *First Steps in Computer Chess Programming*. (accessed 2016-07-13). Oct. 1978. URL: http://archive.computerhistory.org/projects/chess/related_materials/text/4-4.First_Steps.Byte_Magazine/First_Steps_in_Computer_Chess_Programing.Spracklen-Dan_Kathe.Byte_Magazine.Oct-1978.062303035.sm.pdf.

[43] Stockfish. *Stockfish 7*. (accessed 2016-07-20). 2016. URL: http://blog.stockfishchess.org/post/136483912072/stockfish-7.

[44] Herb Sutter. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software." In: *Dr. Dobb's Journal* 30.3 (Mar. 2005).

[45] Alan Turing. "Chess." In: *Faster Than Thought* (1953), pp. 286–295.

Appendices

Appendix A

Running the Chess Engine

The engine's source code can be found on my github page.

https://github.com/emilfo/master

A.1 Linux

The program is compiled with the GNU C Compiler (gcc). To compile; open a terminal, navigate to the master/src folder and type make. The executable is named KholinCE.

To play against the engine you need a chess GUI. I recommend using the small and lightweight *Pychess*. To install on Ubuntu open a terminal and type sudo apt-get install Pychess. If you are on another distro instructions can be found on their site.

A.2 Windows

For windows, a pre-compiled version of the program can be found in the github repository, under the folder *executables*.

You need a chess GUI to play against the engine. I recommend Pychess. A windows installer file for Pychess can be found on their site.

A.3 Pychess

Pychess is a simple chess GUI. Instructions to install it can be found on their download site.

http://www.pychess.org/download

To run the engine, open Pychess, click Edit->Engines. A new window, *manage engines* pops up. Push the *'New'* button, and navigate to the

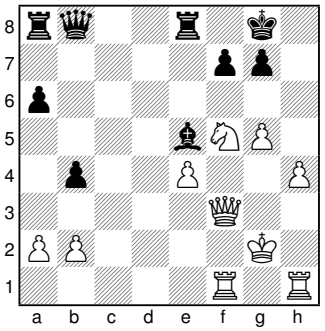
compiled version of my engine and add it. Close the pop up window.

In the main window select KholinCE as your opponent. Choose color and click *Start Game*.

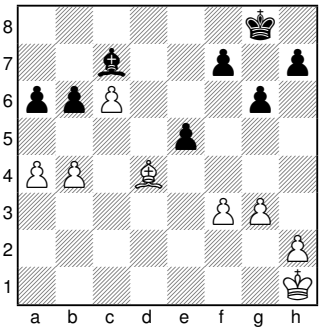
Appendix B

Test Positions, ELO Test

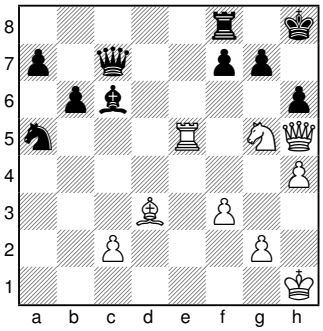
Following are the test problems with solutions for the ELO test. It also includes the move the engine suggested, and the time it used to solve the problem if the solution was found within the allotted 15 minutes.



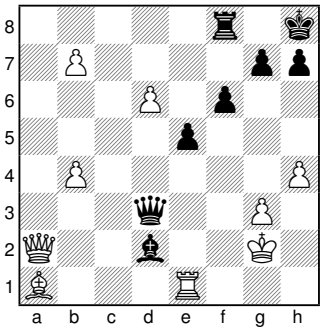
Best move: Nxg7
Engine move (time):
Nxg7 (44 seconds)



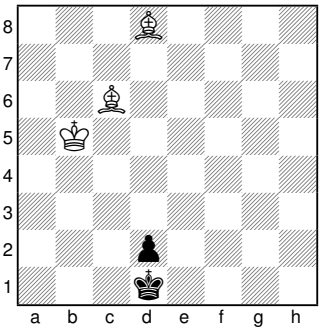
Best move: Bxb6
Engine move (time):
Bxb6 (11 seconds)



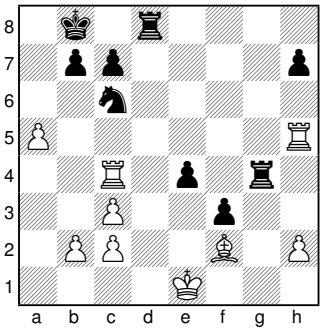
Best move: Re6
Engine move (time):
Re6 (229 seconds)



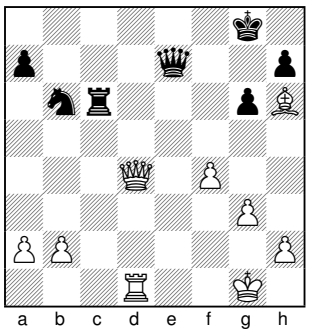
Best move: Qf7
Engine move (time):
Qa7 (-)



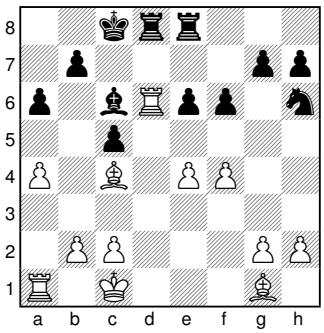
Best move: Ka6
Engine move (time):
Ka6 (1 second)



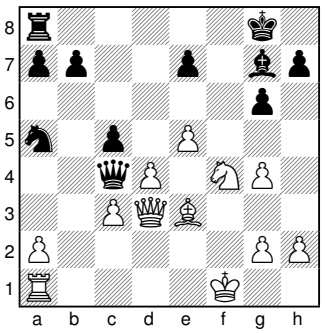
Best move: e3
Engine move (time):
e3 (11 seconds)



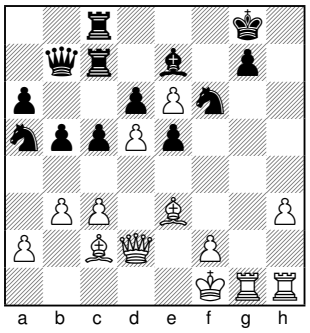
Best move: Rd6
Engine move (time):
Rd6 (>1 second)



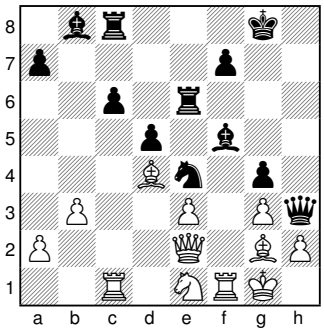
Best move: Rxc6
Engine move (time):
Rxc6 (1 second)



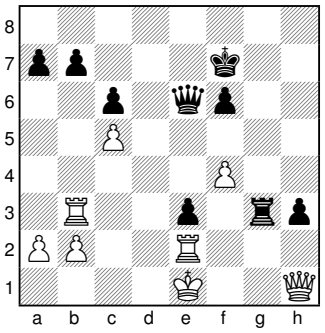
Best move: g5
Engine move (time):
g5 (1 second)



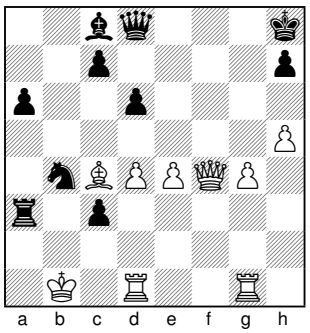
Best move: Rxg7
Engine move (time):
Rxg7 (44 seconds)



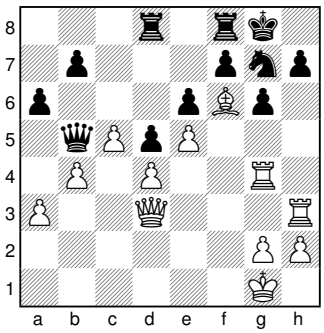
Best move: Qxh2
Engine move (time):
Qxh2 (1 second)



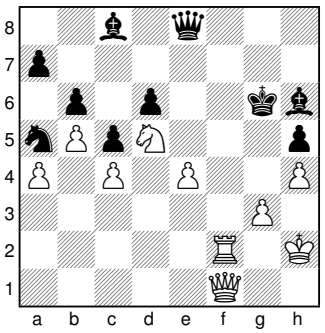
Best move: Qe4
Engine move (time):
Qe4 (1 second)



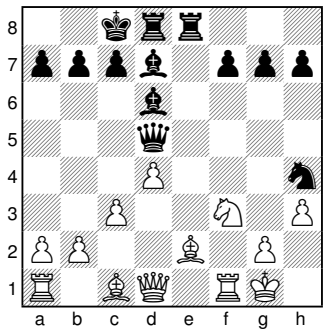
Best move: Be6
Engine move (time):
c2 (-)



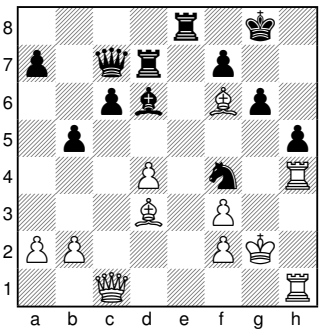
Best move: Rxh7
Engine move (time):
Rxh7 (367 seconds)



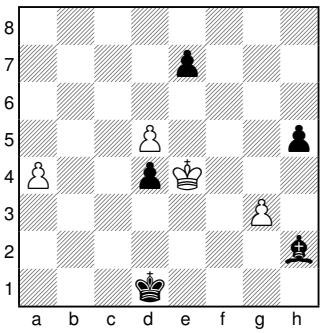
Best move: e5
Engine move (time):
Rf6 (-)



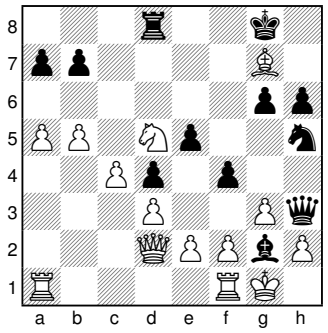
Best move: Nxg2
Engine move (time):
Nxf5 (-)



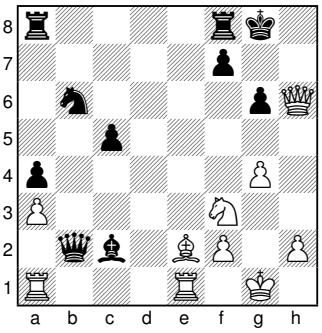
Best move: Qxf4
Engine move (time):
Qxf4 (4 seconds)



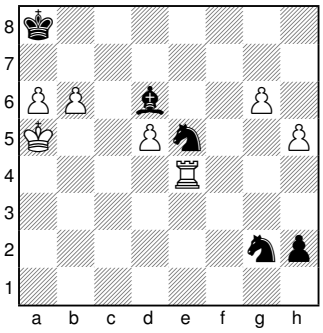
Best move: d6
Engine move (time):
d6 (1 second)



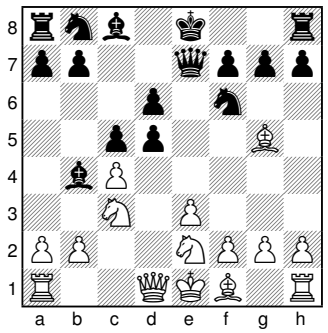
Best move: f3
Engine move (time):
f3 (157 seconds)



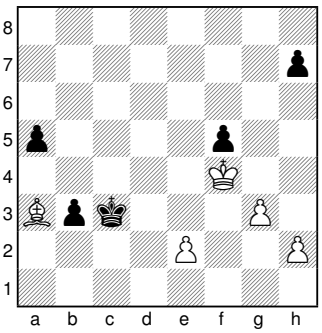
Best move: Ra2
Engine move (time):
Ra2 (729 seconds)



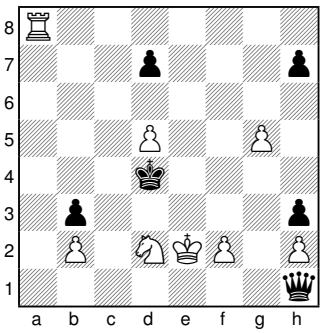
Best move: Re1
Engine move (time):
g7 (-)



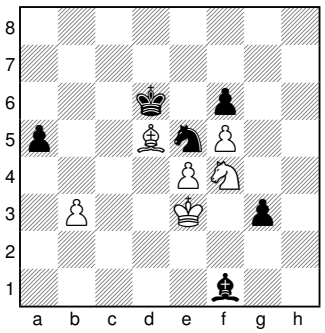
Best move: a3
Engine move (time):
Bxf6 (-)



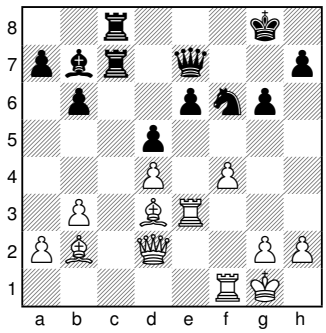
Best move: g4
Engine move (time):
g4 (22 seconds)



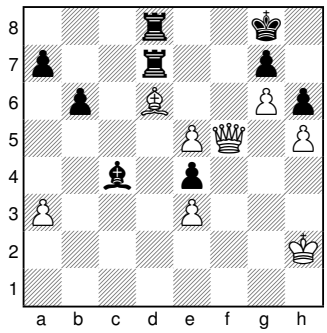
Best move: g6
Engine move (time):
g6 (38 seconds)



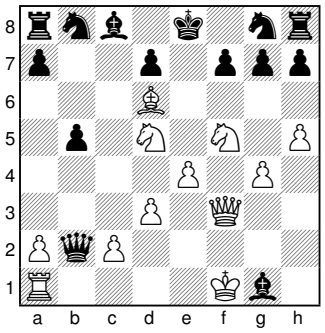
Best move: Nd3
Engine move (time):
Nd3 (142 seconds)



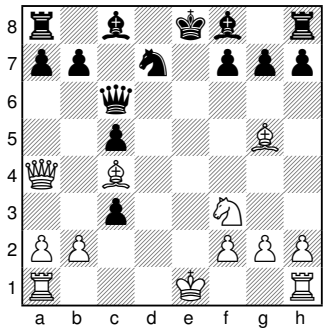
Best move: f5
Engine move (time):
Rfe1 (-)



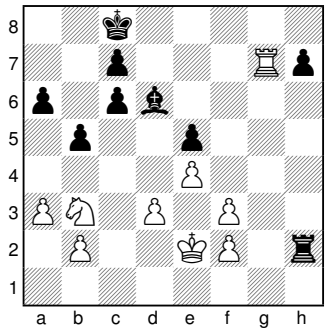
Best move: e6
Engine move (time):
e6 (8 seconds)



Best move: e5
Engine move (time):
Nc7 (-)



Best move: O-O-O
Engine move (time):
Bf7 (-)



Best move: f4
Engine move (time):
f4 (80 seconds)