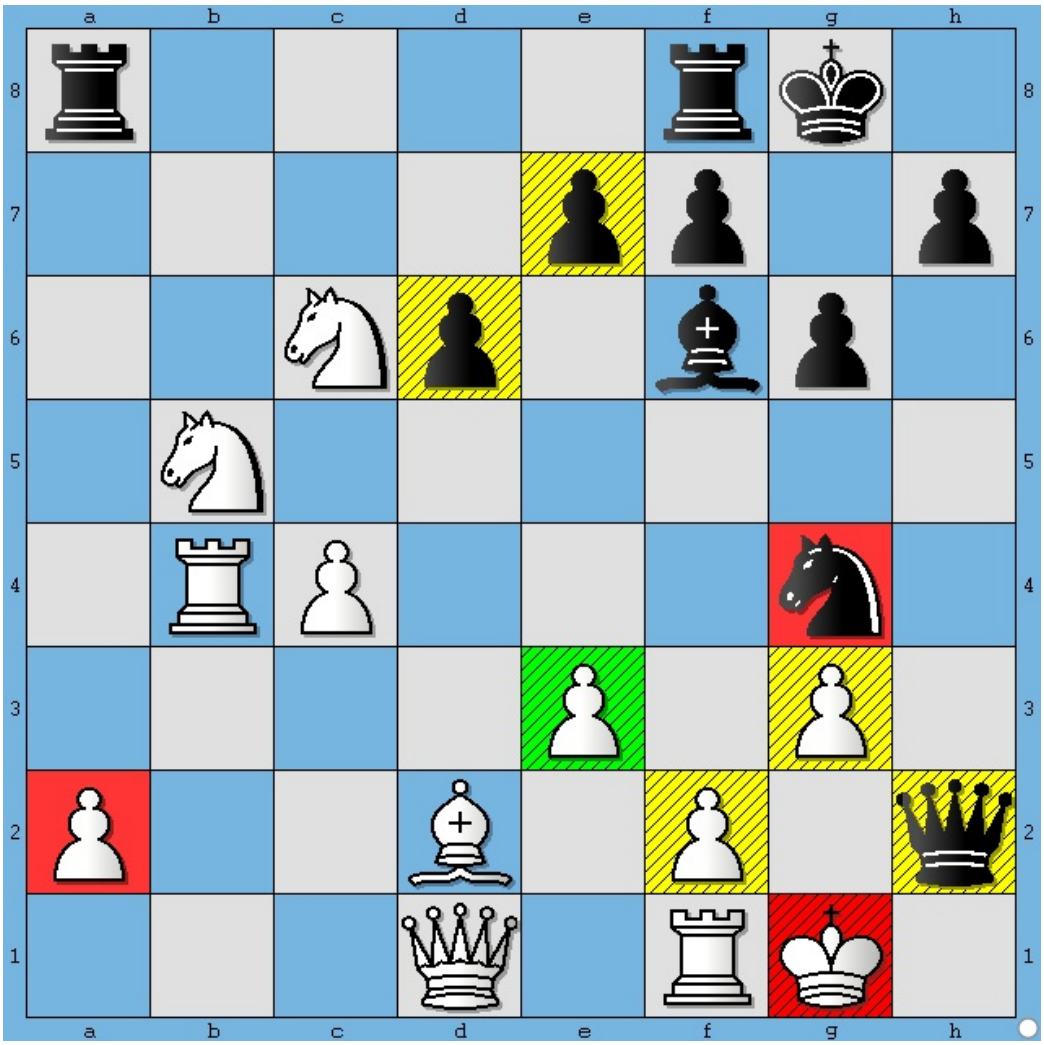
Chess Engine

Group d505e12



AALBORG UNIVERSITET

Det Teknisk - Naturvidenskabelige Fakultet
Datalogi
Selma Lagerlöfs Vej 300
Telefon 99 40 99 40
Fax 99 40 97 98

Title: Chess Engine

Theme:
Intelligent or Massively Parallel Systems

Project period:
P5, the fall semester 2012

Project group:
d505e12

Authors:
Eric Vignola Ruder
Jackie Engberg Christensen
Kimmo Vestergaard Andersen
Nicolai Lund Hasager Kirk
Sune Sylvest Nilausen

Councilor:
Jorge Pablo Cordero Hernande

Total pages: 117

Pages: 99

Appendix pages: 15

Finished 20th of December 2012

Synopsis:

This report is the result of this semesters group project, which is about creating Intelligent or Massively Parallel Systems. We created a chess engine which have aspects from both Intelligent and Massively Parallel Systems. We made a chess engine that can play chess at a reasonable intelligent level, but also made it possible to distribute the system among Parallel Systems. This report goes into the research we have done, and how we have used this research to build our chess engine.

Prefaces

This report was created by five computer science students at Aalborg University. This was written as a part of our P5 project in the fall of 2012.

Jorge Pablo Cordero Hernande has been our supervisor throughout the project. A close collaboration between him and the group has been crucial for the outcome of our project. His extensive knowledge in chess, chess engines and general MI techniques were invaluable, a big thank you is in place!

This semester we had the following courses: Computability and Complexity, Distributed Systems and Networks, Machine Inteligence as well as Software Engineering. The knowledge gained throughout these courses was continuously applied to our project.

Normally, projects focus in one of the two optional courses, which in this case would be Distributed Systems and Networks and Machine Inteligence. In our case, we had students from both courses, so it was decided that our project would be a hybrid: It would mainly focus on the MI aspect, which would be to create a chess engine, as well as incorporating a DSN aspect in it which would assist the main machine in deciding a final move. The aim is to provide the student with ability to analyze and construct parallel and distributed systems as well as the ability to analyze and evaluate the use of intelligent systems to solve problems.

This project is about the development of a distributed chess engine and the choices made by the group while creating this chess engine. The entire source code and an executable application can be found on the attached CD.

Eric Vignola Ruder
Jackie Engberg Christensen
Kimmo Vestergaard Andersen
Nicolai Lund Hasager Kirk
Sune Sylvest Nilausen

Contents

Chapter 1

Introduction

Chess is one of the most popular games in history, it has simple rules yet it is extremely competitive and has hard calculations. But how does modern number crunching machines deal with chess? In this report we deal with this and look into how chess engines work and what their history is. We wil alsol research artificial intelligence theory, techniques and methods, such as advesarial search, minimax, zerosumgames and more, and use it to construct our own chess engine. When making the engine we have to consider state representation, common chess interfaces, movegeneration and more to make our engine as lightweight and fast as possible. Furthermore we will develop a distributed system, by using massively parallel distribution techniques and methods, which allows us to distribute the hard calculation to several computers and run it as a massively parallel system. Lastly we will test our chess engine and parallel distributed system in chess tournaments against other chess engines as well as see how well our chess engine tackles certain chess position problems.

Chapter 2

Problem Analysis

Chess has existed for 1500 years in various forms. The origin of chess is believed to be from India around the 6th century. From here it spread through Persia, and was then taken up by the Muslim world, and from here spread to Southern Europe. Around the 15th century in Europe, chess evolved to its current form. [1]

Chess is a two player strategy game, with each player having an army of 16 pieces. The goal of the game is to defeat the opponent's army by capturing the most important piece; the king. Chess as a game is quite complex; it can quickly be learned but it can take years to master. Players have to be able to see what response the opponent will make according to your own moves. The better the player is, the better he is at looking further ahead in the chain of moves.

Even before computers were invented, great minds had been thinking about how to automate this process and make an automated, perfect chess player. But the theories results in various problems.

In order to determine the best move, an algorithm needs to be established that gives the chess engine some form of reasoning: Why would it choose one move over the other? We have to ensure his reasoning is correct as well so that he does not make foolish moves. Even when you have this problem solved, the algorithm has to have a stop. If we imagine it runs the entire chess game, it would have to evaluate up to an estimated 2^{155} positions. [2]

This would be a little less than $10^{46.7}$ which is still a lot of evaluations. (These are possible positions, so it would not have to evaluate each one, but some of them might be evaluated several times) By this logic, the algorithm would not be usable in any competition which has a time limit.

Even though this would eventually find the best move, a lot of the moves would be illogical, like moving the knight back and forth between 2 positions. A human would quickly see if a line of moves would prove fruitless, but a

computer could not. So you have to give it some sort of intelligence to be able to stop the illogical lines. But what and how would this intelligence be made into an algorithm? How do you define intelligence.

2.1 History

Chess is one of the oldest board games that is still played to this day. No one is sure where the first occurrence of chess was, partly because there isn't much evidence of it but also because there are lots of variants of chess. The oldest known game that resembles chess is *chaturanga* which was played almost 1400 years ago in India. But this does not mean that it was the first game of chess. Pieces similar to chess pieces have been found in Italy that date back to the second century A.D. Though these variants exist, we will focus on the game known as chess and the rules used in official tournaments. This form of chess comes from *chaturanga*. It moved and evolved through Persia, Arabia, Spain and Sicily and came to Western Europe. Though this game wasn't called chess, but instead *Shatranj*, it is the ancestor to our days chess. This game got widely known throughout Europe. Popular amongst both religious orders, kings and soldiers. At some point in the fifteenth century, the game was greatly changed. Some pieces were added, others were changed. At this point the rule of **Promotion**, which will be explained later, was introduced to make the game more exciting. These changes were a huge improvement and the interest for chess spread throughout the rest of Europe. This game replaced the old game completely and more people began playing. Some masters of the game even began to write books on how to play chess(or chesse as it was called at this point). At the start of the 1800s the game began to be popular with ordinary folks, were before it was only for the aristocracy.

2.2 Chess & Computers

As soon as computers became programmable, man tried to make the computer think for itself. They wanted to make the computer intelligent, or at least appear intelligent. Chess is a game of intelligence and tactics and so the concept of AI was used. Over the years a lot of chess brains have been developed, and they have been challenged in various ways, against other AI's or against humans. The first Chess AI to beat the world champion in a human vs. computer chess game was Deep Blue. This AI was developed by IBM around the 1990'. In 1996 it played the reigning chess champion Garry

Kasparow, where it won one game, but lost the match 4-2. A year later in 1997 it tried again, and this time won 3.5-2.5. Deep Blue worked as a brute force machine. Iit could evaluate about 200 million positions per second. It would search to an average depth between six and eight, but could go to a maximum of twenty in some situations.

Deep blue was a super computer and as such is not something ordinary people have access to. Of course since then, processors have been further developed and improved upon, but most computers only have around 2 and therefore cannot handle so many parallel calculations, but maybe this can be solved in another fashion. If the calculations could be split up, then they could be distributed, and then it should be possible to run an large AI on a network of several computers.

2.3 Rules of Chess

There are several different rules in the game of chess. Each kind of piece can only move in a certain way. Some pieces have special moves, and then there are the win/lose conditions. In chess you use a 8x8 board in which coordinates are indicated by letter (a-h) also known as files, and numbers (1-8) known as ranks. In normal Chess you're only able to play 2 at a time, however there have been made some modifications allowing more to play at once. Many other games have inherited some key rules from chess, either in form of the board, or moving pieces.

2.3.1 Initial setting

At the start of the game, each player has 16 pieces; 8 pawns, 2 rooks, knights, and bishops, 1 queen and 1 king. They are all placed on the board in a certain pattern as seen here:

(a) Initial setting of a chess board

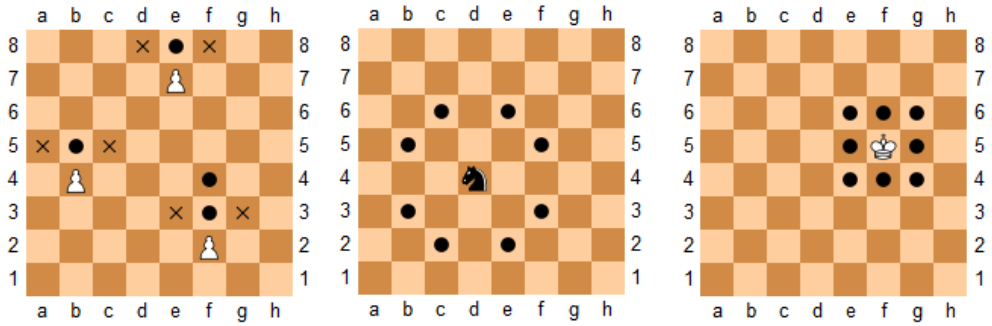
2.3.2 Pieces

The pieces in chess can be divided into 2 groups. There are those that can only move a short distance which are pawns, knights and the kings. There are also rooks, bishops and queens, that can move as far as they want. To get an idea about the rules for each piece we'll talk a little about each piece.

1. Pawns
   This piece can only move one space ahead, and unlike other pieces, can only capture a piece diagonally. The only exception is their first move, where they can optionally move two spaces ahead. Pawns can also make a move called en passant capture, where if one of the opponents move moved 2 squares in his turn, you are allowed to capture it if yu have a pawn located on the same rank as your opponents. This is done by moving diagonally behind the piece, but capturing it anyways.

2. Knight
   This piece can move in any 'L' shape, meaning 2 up or down and one left or right, or 1 up or down and 2 left or right. This piece can also, unlike any other piece, jump over another piece that may be in this 'L' shape, but mustn't land on a space containing a friendly piece.

3. King
   This is the most important piece in the game. If you "lose" this, you lose the game. If the king is threatened by an opponent's piece the king will be in check, and must immediately either block the path for the piece threatening the king, or move the king to a space where it isn't threatened by any of the opponent's pieces. If the king can't move to any space without being in check, the king will be in check mate, or mate, and the player controlling the king must lay the king down as a sign of defeat. The king can only move one space, but in any direction and once in a game it may make a Castling special move.



(b) Pawn (c) Knight (d) King

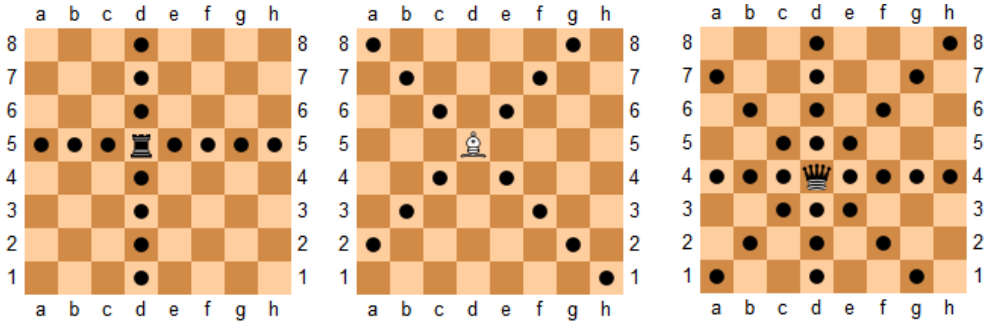
Figure 2.1: Movement of Pawn, Knight, and King

The rest of the pieces can move as far as they want, as long as they don't hit another piece, at which point they can either stop short of it, or capture the piece.

1. Rook
   This piece can only move in a straight line up and down, left and right, but never diagonally.

2. Bishop
   This piece can only move diagonally in four directions, up left, up right, down left, and down right.

3. Queen
   This is regarded as the most powerful piece in the game. It combines the rook and the bishop, and can therefore move in any straight or diagonally line, as far as it wants.



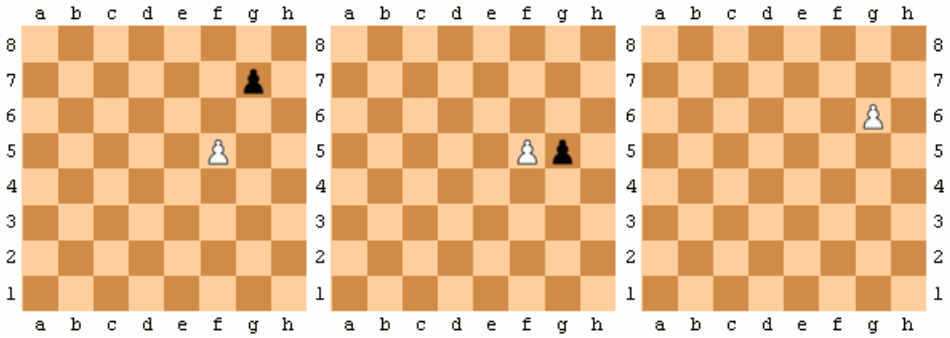
(a) Rook (b) Bishop (c) Queen

Figure 2.2: Movement of Rook, Bishop, and Queen

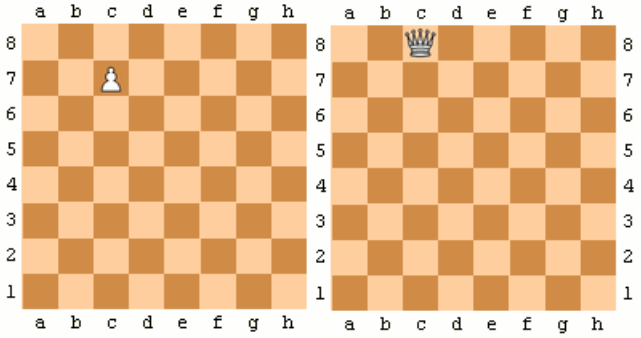
2.3.3 Special moves

There are three special moves:

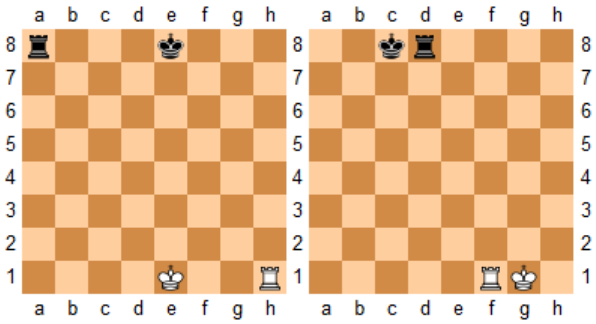
1. En passant
   En passant is used if an enemy pawn is moving 2 spaces ahead, as a first move, and you have a pawn next to the new position. Then you can intercept it by moving diagonal in behind it.



2. Promotion
   Promotion can be used if you manage to get a pawn to the other side of the board unharmed. When the pawn hits the last rank, it can be transformed to another piece of your choice, except a king and pawn.



3. Castling
   The king can move 2 spaces towards the rook, and the rook will move to the space the king passed over. This is only possible as a first move for both king and rook, and there can't be another piece in between them. Additional, the king may not be in check when making this move, after this move, be threatened in any squares the king passes over.



The figures above shows the 2 possible castling moves, kingside- and queen side castling. White has done a kingside castling, and black has done a queenside castling.

2.3.4 Winning

A game is won when a player manages to put the opponent's king in checkmate, when an opponent uses too long time to determine the next move when playing on time, the player having the most points at the end of a timed game, or if a player resigns.

2.3.5 Draw

A game will end in a draw if any of the following conditions occur:

1. A player offers a draw and the other player accepts.
2. If the player to move is not in check but does not have any legal moves, which is also know as a stalemate.
3. The fifty move rule - There has been no capture in the last 50 moves for either player.
4. Threefold repetition - The same board position has occurred three times with the same player to move
5. If there is no possible way of having a checkmate for either side, due to insufficient pieces, E.G.
    - king against king;
    - king against king and bishop;
    - king against king and knight;
    - king and bishop against king and bishop, with both bishops on diagonals of the same color.

2.4 Artificial Intelligence

Artificial intelligence, or AI, is the field that studies the synthesis and analysis of computational agents that act intelligently. Let us examine each part of this definition.

An agent is something that acts in an environment - it does something. Agents include worms, dogs, thermostats, airplanes, robots, humans, companies, and countries. We are interested in what an agent does; that is, how it acts. We judge an agent by its actions.

An agent acts intelligently when:

- what it does is appropriate for its circumstances and its goals,
- it is flexible to changing environments and changing goals,
- it learns from experience, and
- it makes appropriate choices given its perceptual and computational limitations. An agent typically cannot observe the state of the world directly; it has only a finite memory and it does not have unlimited time to act.

A computational agent is an agent whose decisions about its actions can be explained in terms of computation. That is, the decision can be broken down into primitive operations that can be implemented in a physical device. This computation can take many forms. In humans this computation is carried out in "wetware"; in computers it is carried out in "hardware." Although there are some agents that are arguably not computational, such as the wind and rain eroding a landscape, it is an open question whether all intelligent agents are computational. [3]

A perfect AI is often met in movies and book where computers for instance, act like humans. However, in the real world is it hard to make a perfect AI. One of the main reasons is that an AI rarely is able to learn mistakes, both its own and from others and thereby gain the necessary knowledge to perfect itself.

An example of AI usage is games, almost all games have some sort of AI, some more advance than others. A simple AI could be the AI from the classic video game Pacman, where the ghosts move towards Pacman in an attempt to kill him. The AI here is one part randomized and one part path finding. When playing you often notice that at least 2 ghost are chasing you at all time and the 2 other patrol points of interests. An more advanced AI can

be found most newer video games. However, due to the power of the pc's in our century many AIs are actually much dumber than they could be. This is because a pc can keep track of your every move, but this would result in an unfair advantages for the computer. Therefore there often is a limit on how much a computer player, or AI, can see in a game. Take for instance a first person shooter game, if a AI player could react to more than sounds and players in its point of view, this would be an unfair advantage since this is the only feedback, or inout, it recieves from the game. However there is one key factor here, a human player can learn hiding spots and remember where a computer player is hiding, where an AI rarely learn this. Again we see how important learning may be to make a realistic AI.

2.5 Task Environment Analysis

Our chess AI is an agent. An agent perceives and acts in an environment [4, p.34]. To get an overview of this environment, we will provide a specification of the task environment analysis facing our chess AI agent, which is common tool for putting design in context. Normally a task environment analysis would consist of a PEAS (Performance, Environment, Actuators, Sensors) description [4, p.40] however we will only consider actuators and sensors very shortly since there is not much involved in a chess AI agent.

First of all it is obviously in our interest that our chess engine will behave as well as possible, where good behavior is being rational. To evaluate this behavior and measure its success, we will need a form of objective performance measure. Our agent will use this measure to maximize its performance. Our performance measure could embody qualities like maximizing destroyed material value and minimizing lost material value. However in chess using FIDE official rules a game can only be won, lost or draw regardless of the material left. Another could be ensuring a high win percentage of overall games played. It would also be preferable, if dealing with time limits, to prefer low number of moves per win, and minimizing time used per turn. However there is already a generally used and clear performance measure in chess which is called the Elo rating system, and is covered in the next section. The final analysis and the design and implementation of the performance measure will come later.

Secondly we have to consider the environment of our chess agent. The chess agent must deal with chess pieces and their features, such as the abilities, position and value. The agent must also understand the chess board as well

as other chess rules. It may also have to deal with different time constraints which are used in some chess games. Thirdly the actuators is moving of pieces, which includes en passant and castling, as well as including making legal moves. Fourthly the sensors are simply looking at the board.

In the task environment for our agent, we can identify some properties. First of all we know that our agent will have access to the complete state of the environment, thus the task environment is **fully observable**. However this is true since our agent remembers information about the game history (which not all agents do), since rules about castling, en passant capture and draws by repetition are not apparent from the current board state. Another property is that the environment is **deterministic**. This is because the next state of the environment is completely determined by the current state and the action executed by our agent, thus we need not worry about uncertainty, except for the actions of the opponent agent. Another property we can identify is that the environment is **sequential**. It is sequential since the decisions made by our agent will affect all future decisions, e.g. openings and the exchange of pieces will have long term consequences which our agent will need to think ahead about. This is however only valid within a single game. Our environment is also **semidynamic**, unless time is not a factor in which case it would be static. Basically the environment does not change while our agent is figuring out what to do, however since there may be time limits, then our agent will want to figure out a solution while spending as little time as possible. Another property that the environment has, is being **discrete**. Chess has a discrete set of percepts and actions. Lastly our environment is a **competitive multiagent** environment since chess is a one versus one game, thus our agent will need to consider that the opponent will want to minimize our agents performance.

2.6 Elo Rating

Elo rating is a rating system for calculating the relative skill levels of players in two-player games such as chess, go, or similar. This system is named after its creator Arpad Elo. The Elo system was invented to improve the chess rating system, but today it's also used in many other games like go, checkers, etc. It is also used as a rating system for multi-player competition in a number of video games such as real time strategy games, massive multi-player online games and shooters. It has also been adapted in different team

sports such as basketball, baseball, football etc.

Each player has a numerical rating, where a higher number indicates a better player. A players rating is based on the scores win or lose(draw is half a win and half a loss), and the ratings on the opponent rated players. The exact amount is calculated by first getting the expected score of a player, denoted E, which is comprised of the chances of getting a win, lose or draw, and which scores is 1, 0 and 0.5 respectively. So if E is 0.50 then there is 50% chance of winning, 25% chance of losing and 0% chance of drawing, or it could be 25% chance of winning, 25% chance of losing and 50% chance of drawing. This is calculated by the formula: $E_A = \frac{1}{1+10^{(R_B - R_A)/400}}$, where R_A and R_B are the ratings of the two players. The formula shows that for each 400 rating points of advantage for A, it is estimated that A is ten times more likely to win. Now, the expected score is used to give a player points by considering whether his actual score S was lower than expected and update the rating. This is done by this formula $R'_A = R_A + K(S_A - E_A)$, where K is the maximum value the rating can change and for new players it is 30 for FIDE [5]. Lets run an example to clarify: if a 1400 rated player is against a 1500 rated player, the 1400 rated player has a 0.36 expected chance of winning and the 1500 rated player a 0.64 expected chance of winning (together they add to 1). If the 1400 rated player wins his rating is $1400 + 30(1 - 0.36) = 1419$, if he loses it is $1400 + 30(1 - 0.36) = 1389$, if the 1500 rated player wins then his rating is $1500 + 30(1 - 0.64) = 1511$ and if he loses $1500 + 30(0 - 0.64) = 1481$. The update of rating is done after any rating period, e.g. single game or tournament, in which case you use the combined score and the combined estimated score for all matches. In chess an Elo rating of 2000 or more is considered to be an expert's rating and 1000 is considered to be a bright beginner's rating.

2.7 Existing Engines

All chess engines that we have looked at have all been in a win32 binary or executable format. Some engines contain various files that they can call upon, such as application extensions(DLL) or binaries(Bin). In addition to the binary files, most engines also have something called opening book which is basically a a guide for the beginning of the game, since there are so many possible moves and so many different positions that can be achieved. Opening books merely guide the engine by sugggesting moves depending on the opponents move for the entire beginning of the game. The engines we have tested are Crafty, Frizt, and Rybka. Each of them have some similarities, but yet very different.

2.7.1 Crafty 18.12

Crafty is a free, open-source chess engine. One interesting thing about crafty is that you can customize the settings of the engine. Crafty has participated in multiple Computer Chess Championships and ended in second place multiple times and in the 2010 World computer Rapid Chess Championships. Crafty finished behind the first place winner Rybka by only half a point. [6]

2.7.2 Fritz 5.32

In 1995, Fritz 3 won a World Computer Chess Championship, but in the later releases it has not won but never the less it still belongs on a list of top 10 ranked engines. The newer Fritz engines have an Elo rating of more than 3000, which is considered to be really high. [7]

2.7.3 Rybka 2.3.2

As mentioned earlier Rybka has won a championship, and claims to have an Elo rating of more than 3100. Rybka's search is considered one of its main advantages against other chess engines, and the way it implemented bitboards is said to be one of the key factors to its high Elo rating. [8]

2.8 Existing Protocols

A protocol is a agreement on how data between two sources should be handled. In all chess engines when you want to send data between the interface and engine you will need to send and receive information to and from the interface. The 3 most common used chess protocols are: UCI (Universal Chess Interface) WB (WinBoard) ENG (Chessbase). They all seems to use a somewhat equivalent system. We won't go into depths about how each one works since they all work somewhat equivalently.

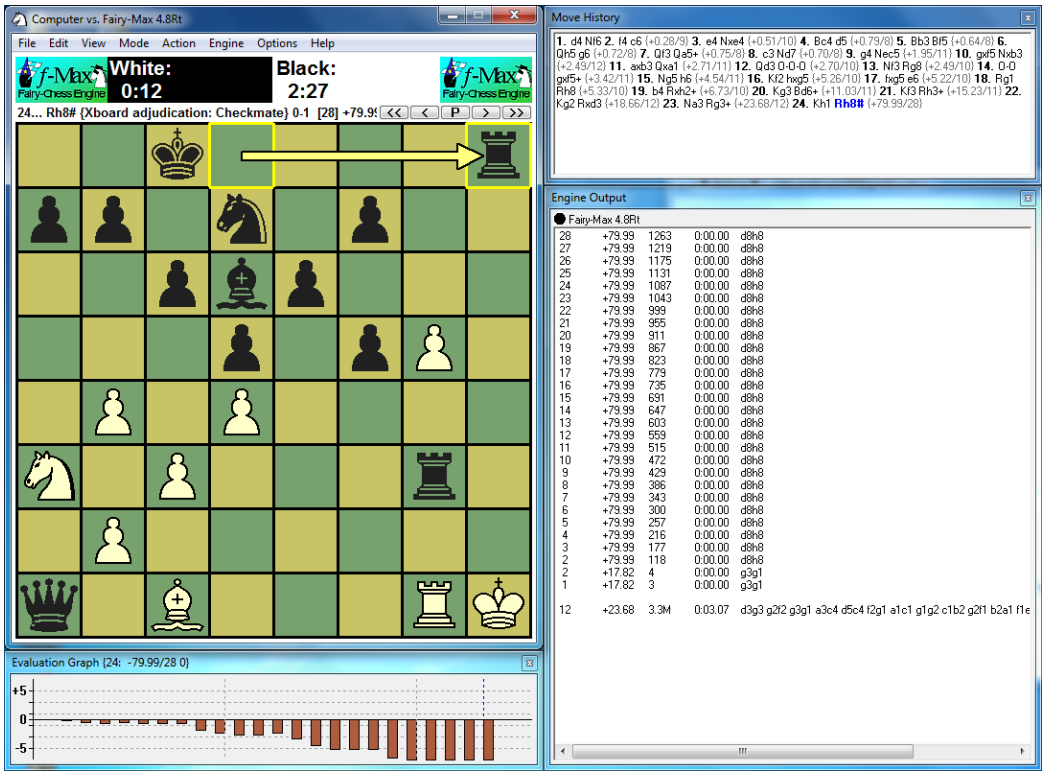
2.9 Existing Chess interfaces

Before we began to build our engine we were looking at difference chess interfaces supporting the UCI protocol. A chess interface is primarily just a way to display the chess board in a user-friendly way and keep track of the game. We looked at 3 interfaces, all having some nice features. In the end we decided that we would utilize arena to run our engine. However we will

go little into details about why we did not choose some of the interfaces we looked at, and why we went with arena in the end.

2.9.1 Winboard/Xboard

Winboard is a widely known free chess interface, the GUI is very simple but yet very clean and shows some interesting game statics such as evaluation, move history, engine output and more is possible. It does have what is needed for a chess interface to function well. However we had alot of trouble getting winboard to work and took us quite some time to figure out how everything worked with it, and therefore we didn't pick this one. [9]



(a) Winboard and its gui

2.9.2 Fritz

Fritz is a commercial chess interface and engine used by many professional chess players. The Fritz interface is in many ways very powerfull and has many small but nice features added. Just like Winboard it has evaluation graphs and engine output, but additionaly it has the ability to teach you and

come up with ideas on what to move by using another engine(s) as a kibitzer for your game. Fritz also allows you to see where you're attacked, defended, and undefended and tell you what you should be aware of. The interface can also adjust the difficulty of the engine/computer to match your needs better. Also a fun detail about this engine is that the engine can give up or ask for a draw if the engine feels this might be the best move and it can also accept or deny a player's offer of a draw match. Overall Fritz is a very nice interface that has some very nice features, especially for teaching purposes, but the GUI may take some time to learn and get used to. We didn't choose Fritz because it wasn't free, and also the version seen in the picture below that we tried didn't support UCI very well from what we understood. [10]

(b) Fritz and its gui

2.9.3 Arena

Arena was the interface we chose. We decided to use this as our main interface for testing, because it was easy to use and we didn't use much time figuring out how to connect it with our engine. The GUI s very basic, and has some of the same functions, such as evaluation graphs, engine output etc. Even

though it is simple, it has everything we need. It is also free-ware so a copy of the program is readily available. [11]

(c) Arena and its gui

Chapter 3

Problem Statement

As mentioned before, there is quite a history regarding computer engines in the chess field. There are also quite a few predicaments regarding the current enignes available. Garry Kasparov, a world famous grandmaster chess player, once said that playing against a computer is like running a marathon against someone on a motorbike. Other than being "unfair" to play against something who can calculate and remember millions and millions of moves ahead, it also leads to boring chess play. What is meant by this is the fact that computers tend to do boring moves that only pay of a lot later in the game, like moving a castle to protect the king in 15+ moves.

3.1 Scope

There are several different things we need to take into account if we wish to create an chess engine. Since we also have a deadline, it is important to state what we want to achieve and what priority they have. After we have done that, we will formulate a problem statement that will guide us throughout the rest of the project.

3.1.1 Chess Interface and Protocol

A big question is whether or not we want to create a chess interface or use an already well established free chess interface. Also, do we want to create our own chess protocol or do we want to use some well established chess protocol so that it is easier for us to test our chess engine against other players and eventually other engines. Although this question would usually be answered in the analysis, we choose to use an already established interface since we do not wish to spend time creating something that has no true relevance to our

project. We also wish to use an already established interface so that we are able to test our chess engine against other players as well as computers later on. The analysis part of this problem will be dedicated to figuring out which interface and protocol best suites our needs.

3.1.2 Distributed System

An idea to further increase performance of the AI was to take advantage of the vast amount of personal computers available in this day and age. The idea was to distribute the calculation load among several different computers instead of letting only one computer calculate all moves. Calculations might be speed up significantly if there are several computers working on the same problem, which is finding the next best move possible. But how do we add a computer to a network? How do we divide the problem among the computers? One thing to note is that this project is mainly a machine intelligence project, so that is where most of our focus will be. There are though some distributed system students in the group, so we would also like to incorporate some aspect(s) of whatever they learn throughout this semester. This is all a matter of time though, so it all depends if we have enough of it.

3.2 Problem Statement

By deciding the main elements we wish to focus on, the following problem statement will be formulated and be used as a guideline of our goal through the rest of the project. This will help us remember what our main goals are and where we should focus most of our effort on. This will also help us decide what features we should keep or cut if we encounter time constraint troubles.

- How can we create an AI that is capable of playing chess on a level where the moves done is smart enough to beat a human player?
- Can we eventually create a distributed system capable of distributing work load among several computers?

Chapter 4

UCI Protocol

Our first goal in our project was to get the chess program talking to the chess engine. It should be able to receive commands, parse them, update whatever needs to be updated and answer accordingly. As mentioned in the analysis, the engine will only support the UCI protocol. That means we must implement the version of that protocol and get the engine to respond accordingly.

The way all chess programs communicate with the engines, according to the UCI protocl, was through the engines standard input and output. The chess program would write to the engines standard output, and expect answers through its standard output window. Getting the commands in and being able to send messages back would be a simple task. A class called *Protocol* was created in order to parse the commands and call methods from an interface called *ProtocolListener*, depending on which command was parsed. A class that implemented ProtocolListener would be passed through initialization. The interface was created so that other classes could then implement it and react to the input differently. This was done so that at some other time, we might implement the xBoard protocol, so that it might someday be used by people who do not use programs that support the UCI protocol. This isn't something that will be implemented in this project though, since most programs do support UCI.

Another class, called *UCIListener*, which implemented the the ProtocolListener interface. From this point onwards, all that needed to be done was to look through the UCI specifications [12] and implement all commands that are sent to the engine and what replies they expect. The UCIListener class would then figure out what to do with the command by calling the respective methods within the program. For example each time the command position is sent to the program, it would figure out whether or not a move had oc-

curred and if so ask the board to update it accordingly by passing a move to it. The position command was sent before every move, even if not prior move has occurred. It is possible to have the engine play for both players, it is just a question of asking the interface to let the engine play the turn. The position command is then sent to the engine, along with all moves up to this point. The engine then tells it to go and expects a reply in the form of bestmove %algebraic notation%.

Whenever the go command was issued, the listener would then query the engine for which move it should take, and whenever the engine was done thinking, it would call back by calling on the Move command on the listener, which would then issue the move back to the chess program.

At any time, the chess program can ask for a newgame, in which the listener reacts accordingly by calling on a newgame method on the board, which resets all piece positions and puts the white player to play next. The chess engine should also be able to receive a FEN string . A FEN string is a string that represents the current board state. It encodes all of the states of each square, as well as active color, castling availability, en passant target square, halfmove clock as well as the full number of moves [13]. This is a fen string for the starting position:

Figure 4.1: Fen string over starting position:
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1

The string is divided by spaces into 6 different parts, each containing information about different aspects of the board.

- The first part is board information. This part is also divided by seven slashes. In between each slash is information about each rank. The first set of letters and numbers is rank 8, and the last is rank 1. Whenever it meets a letter, it adds the corresponding piece to the corresponding position. Capital letters are white pieces and black otherwise. Whenever it meets a number, it skips over the corresponding number of squares. Numbers equate to the number of empty squares in a sequence. As can be seen from the example, ranks 3,4,5 and 6 are all comprised of 8 empty squares in a row.
- The second part corresponds to who is up to play. W is for white, while B is for black.
- The third part corresponds to castling rights. If this part contains a K, that means that white can king castle, and Q means white can queen castle. It also applies for black, but instead the letters are in lower case.
- The next part is an nn passant target square. This corresponds to whether there is a square which can be captured through en passant.
- The fifth part is half move count
- The final is full move count.

If the engine receives such a request, it asks the board to setup the game according to the FEN string and possibly any moves thereafter.

Chapter 5

State Representation

When making computer board games is it important to take into account how you want to represent the board states, especially then it is a game on time, since the way representing the board can change how fast a computer can calculate a move to take.

In this chapter we will discuses ways of representing game boards, as well as protocols to communicate with a chess interface, and benefits and disadvantages of different ways of representing the board.

5.1 Chess Engine Protocol

In this section we discuss the two major Chess Engine protocols; The UCI and the xBoard Protocol. These protocols were created in order to let chess engines communicate with the UI so that they could receive and relay moves. That enabled the construction of chess engines that could receive a move, calculate its counter move and relay that to the interface in a standardized way. The question is then, which one should we use? What advantages and disadvantages are there in using each protocol? We will start by discussing the xBoard protocol.

The xBoard protocol was never developed to intentionally support any chess engine other than GNUChess. Therefore it was created as an ad-hoc solution when other people wanted to utilize it as well. It had a major following up until the late 90's, when UCI launched and gained support of one of the biggest chess programs out at the moment, shredder.

UCI was launched in the year 2000 as replacement for xBoard to address some features that the previous protocol was lacking. These include: [12]

- All engine options can be modified within the graphical user interface so that there is no need to deal with ini files.

- Much better capabilities to display search information of the engine.
- It is more robust, the GUI always knows what the engine is doing.
- Support for endgame tablebases
- Flexible time controls.
- The engine can identify itself
- UCI supports copy protection mechanisms for commercial engines.

In the beginning only a few interfaces and engines supported UCI, but this changed in 2002, where Chessbase, the company that markets Fritz, began to support it [14]. As of 2007 there are well over a 100 engines that support UCI, but some of these kept their support of xBoard. Since the project will be coded in C# and for windows, the major drawback to UCI is not really a drawback at all to us. We do not have plans to implement book learning either, the engine will be using pre established opening books instead. Since the major advantage is the accessibility of UCI and its widespread use, we will be using the UCI protocol instead of xBoard.

5.2 Interface

It is now time to decide what user interface we will use to develop our program. This is not actually that important since as long as the interface implements the UCI protocol, any GUI will serve. And there are quite a few engines. The fact that interfaces can really easily be changed if some issue arises with them makes this a non choice really. We just choose whichever interface works and because of that we have chosen to start out with the chess program Arena [15]. This program was chosen because of how simple it is to connect a chess engine with it. The path to the engine is simply given to it, magic happens, the chess engine connects and the player is now able to play against the engine. If this chess interface turns out to not fit all of our needs, we can quickly find another chess interface to replace that, as long as it implements the UCI protocol. Even Winboard, the chess program created by the same people who created the xBoard protocol, implements UCI.

5.3 Bit Board Representation

One of the first problems encountered when creating a chess engine is how to represent the board internally in such a way that makes accessing each square

fast and painless. This is especially important when generating moves. As an example, take a rook that is located on a file that is not occupied by any other piece. Only taking the file into account, this piece alone has 7 possible moves (note that staying on the same spot is not a move). If its rank is also unoccupied, another 7 moves are possible. What if there are other pieces that are blocking the rook? Can we capture that piece? Is it one of our own? Is it even a legal move? We cannot leave the king in check for example. Now imagine the queen for a second, being able to move just as the rook and the bishop. It is essential that we find a fast way to evaluate all possible movements so that the engine is then capable of choosing the best one among them.

There are two major ways of going around this problem. The first one is possibly the easiest one to grasp since it is the one closest to how the chess board is viewed in reality: the *Offset board representation*. Using this representation, the board is represented as an array of elements, with each element mapped to a square on the board. The other way would be to use bitboards to save information about all pieces and where they are located. Both methods will be discussed, going over each of their strengths and weaknesses.

5.3.1 Two-dimensional array representation

As mentioned before, a two dimensional array is the most natural way to represent the board. A two dimensional array of dimensions 8×8 maps perfectly to our board. Let A1 be the first element of the array, board[0,0], going to H1, board[0,7] and H8 being board[7,7]. This is a perfectly reasonable representation since it facilitates implementation as well as discussing the engine itself. There is no magic here, everything is straight forward and there are no real complications. Well, at least on the surface, there are no complications. This representation can be seen below, on Figure 5.1.

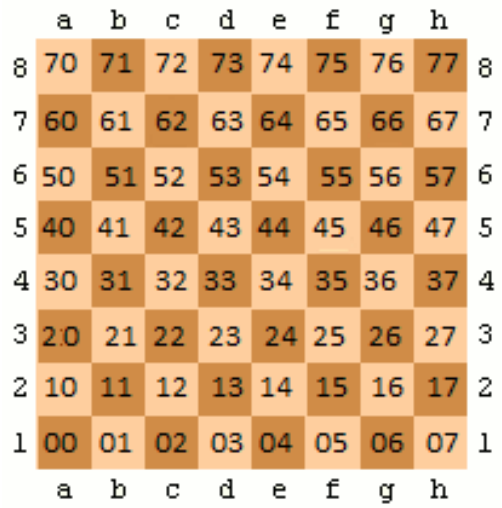


Figure 5.1: A 2D array, where the first digit is the rank and second is file

Remember how multi-dimensional arrays are stored in memory: They are an array of arrays. That means that any time you are trying to access the piece on located on rank R and file F, the multiplication $R \times 8 + F$ is required to access it. Most modern compilers though realize this problem and replace the $R \times 8$ with a bitwise shift ($R \times 8 == R << 3$) [16] . This equivalence should be evident to most competent computer programmers since $8 = 2^3$ and every time we shift a bit we increase it by a power of 2. Even though most compilers can make this switch, it still takes a couple of instructions. This is a problem in move generation when we have to do this for all pieces on the board and all their possible moves to check whether or not they are legal and their consequences (piece capture, putting the king in check).
In checking whether or not moves are legal, we need to make sure that pieces do not slide off the board. That means that all moves need to have the check that the file and rank do not go above 8 or that they don not fall below 0. This would usually be done when the move is generated, so one should check whether the move would "overflow" the board and if so don not add that to the possible move set. That means that we would have to make a check for both rank and file on every possible move for every piece on the table. That is a lot of overhead. There must be a better way. Thankfully, there is.

5.3.2 One-dimensional array representation

This is a very good suggestion to improving upon the two-dimensional array representation. We now map the squares to a single dimension array,

A1 being board[0], H1 being board[7], and finally H8 being board[63]. Now, whenever we want to traverse the board, we simply add a constant to the index. Want to travel left or right? Add or subtract the number of squares you want to move. Up and down? Add or subtract by eight. Diagonal moves are done by adding or subtracting 9 or 7 from the index.

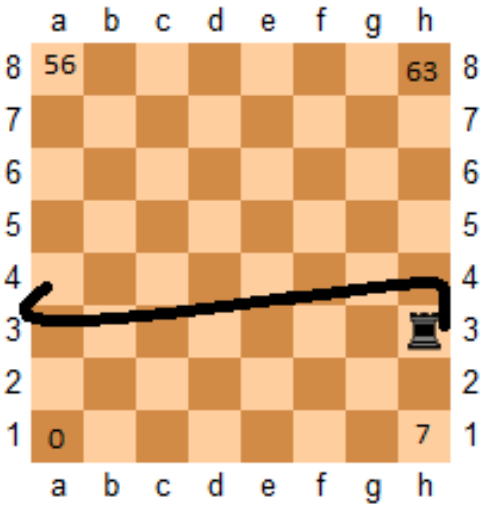


Figure 5.2: A rook on square 23, trying to glide to the right

Another thing we can take advantage of is that it is one less conditional check per piece, since we only need to know if the piece goes above 63 or below 0. Or at least it seems so. As can be seen by figure 5.2 below, what happens when a rook for example is located on square 23 and wants to move to the right? Well, that is easy, just add x squares to 23 and we would have the correct answer. But as can be seen, 23 is actually the H-file and therefore we cannot move to the right with the rook. What actually happens is that the rook wraps around the board and is now located a rank above the one it was previously located at (as long as we don not add more than 8 to the index). This is not a valid move. There are no pieces in chess that have some kind of portal gun. That means that all conditional checks regarding files and ranks still need to be there, to ensure that we do not wrap around the board. Let us see if we can find some kind of board representation that eliminates the need for testing whether a pieces jumps of the edge of the board.

5.3.3 One-dimensional array representation with border squares

Something we could do is to pack the board in an even larger array. We would map A1 to board[20], H1 would be board[27], A8 board[90] and finally H8 would be board [97]. The array would be 120 elements big, thus giving us an extra 2 ranks on each side of the board as well as an extra file on each side of the board as well. The array can visually be represented as shown on Figure 5.3 below

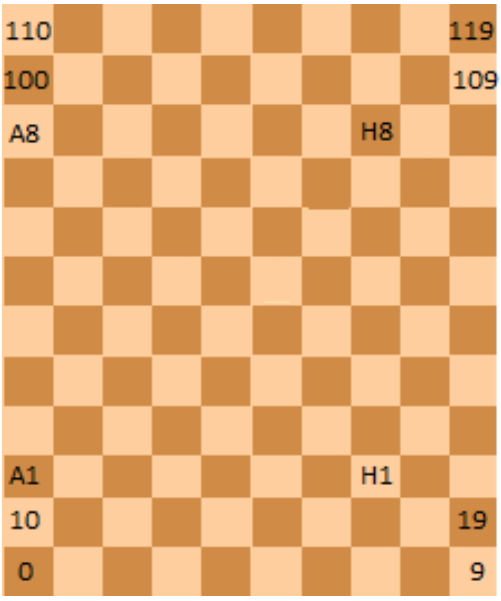


Figure 5.3: Array representation with border squares

The reason why one rank is not sufficient is because of the knights. Remember that they can jump two files or ranks at a time. The reason that there is only one file is because of the problem we had last time, and that is the fact that the ranks wrap around each other. That means that if a knight was to jump to the left two squares while being on file a, he would wrap around and land on file "i", and still be considered illegal. If we were only using sliding pieces, this would not be a problem (think of checkers), but alas it is not so. We still need to check whether or not the move is an illegal one. While it is true, instead of having to use two conditional checks, we can suffice with one. When we decide how to represent chess pieces, all we need to do is to pick a number that is not used to represent them (or the empty square) and fill the border with that number. That way, we can

check whether the landed square contains that number. Let us say that we use -1 to do so; the only check needed would be if(square[landed] == -1). Although this solution sounds elegant since we have removed the major drawbacks from the previous two representations, another question arises: Can we remove more? Remember that in most machines, memory references are quite expensive [17]. It would be quite advantageous if we could remove the check to see whether or not the landed square contained the illegal move code -1. Fortunately, that is what the *0x88 representation* allows us to do.

5.3.4 The 0x88 Board

This representation is based on older computers, where they often had little or no cache memory, which made memory access operations quite expensive [18]. That was one of the problems (it is a problem, we are quite greedy) we had with the previous board representation, the *One-dimensional array representation with border squares*. Every time a move needs to be checked to see whether or not it was legal, a memory reference to the element in the array is needed to see if it contains the illegal move code, in this case -1. Not only is it accessing a memory location, it is also returning a value that isn't used for anything else other then checking whether that move is legal or not. The extra memory reference can be removed and it can be done quite simply by altering our board layout even further. A new array board is used to represent the chess board, this time with 128 elements. A1 gets mapped to board[0], H1 to board[7], A8 to board[112] and H8 to board[119]. This can be visualized as followed:

112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

At first glance, this doesn't provide us any improvement from the array board with borders. But let us for a moment consider the same table, but instead of using a base 10 system to access the elements, let us use a 16 base system, the hexadecimal. Let us see how each element is then labeled.

70	71	72	73	74	75	76	77	78	79	7a	7b	7c	7d	7e	7f
60	61	62	63	64	65	66	67	68	69	6a	6b	6c	6d	6e	6f
50	51	52	53	54	55	56	57	58	59	5a	5b	5c	5d	5e	5f
40	41	42	43	44	45	46	47	48	49	4a	4b	4c	4d	4e	4f
30	31	32	33	34	35	36	37	38	39	3a	3b	3c	3d	3e	3f
20	21	22	23	24	25	26	27	28	29	2a	2b	2c	2d	2e	2f
10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

Now, one of the first things we notice is the fact that for each element, the first digit is equivalent to the rank and the second digit is the file. This is a huge convenience, which makes reading and writing code a lot simpler. But is it truly only a convenience, or is there something more to that? Pay attention that for all elements that are legal, neither the first or second digit never go above 7. That means that all legal moves have the format 0xxx0xxx in binary, where x is arbitrary. As soon as a move is illegal, it becomes either 1xxx0xxx, 0xxx1xxx or 1xxx1xxx. This is where the major advantage of the 0x88 board comes into play. Take the hexadecimal constant 88 and transform that into binary. It looks like this: 10001000. That means that we can perform the bit operation AND on the index and if we get a non-zero value, we know that it is an illegal move. We no longer need to access the array to know whether or not the move was legal or not, we just use the indexer and save ourselves the access time. Note that wrapping is not a problem here, since as soon the piece steps out of bound, we know the move is illegal, because as soon as wrapping occurs, the piece is thrown into the illegal part of the board.

Let us consider move generation for a moment. In whatever form of array representation we choose, we still need to loop through all valid moves targets for that piece to see whether or not it is a valid move. That for example a knight that is located in E4. His valid moves are D6, F6, D2, F2, C3, G3, C6 and finally G6. We need to check all those squares to see if they are occupied, and if they are, can we capture them or are they one of our own. We then need to discard all moves that are not legal, meaning those that land on our own pieces. That is quite a large amount of operations needed for each piece. Something that would be a huge performance boost would be the ability to get all valid target squares in a single operation. But surely, something as great as this is not possible? Let us now leave the realm of array board representation and step into *Bitboard representation*, where such great feats are easily accomplished.

5.3.5 Bitboards

With bitboards, chess boards are represented with a few 64-bit words. The reason for this is because there are 64 squares in a chess board, and 64 bits in a 64-bit word (surprise!). We now map A1 to the first bit, H1 to the eighth bit, and H8 to the 64th bit. For each bit, if it is set to 0, then no piece is in that square, where a 1 represents occupancy. But how do we know what piece is located in each square, if all that is set is whether there is something in that square or not? We do that by instead of only using one bitboard, we use 12. We use on for each piece type of each color.

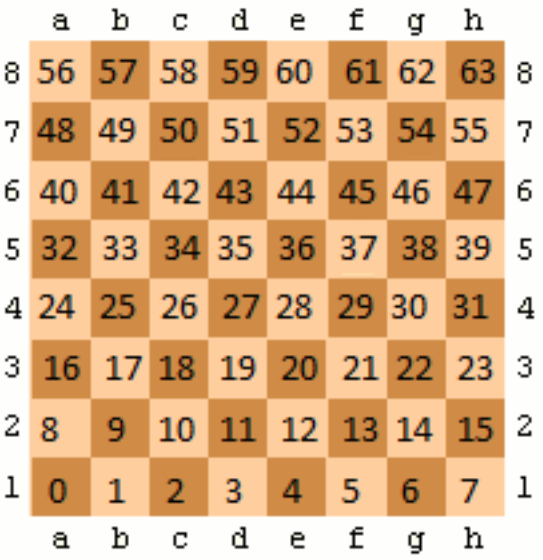


Figure 5.4: A bitboard, where each square is mapped to the corresponding bit index

For example, a bitboard for all white knights in starting position would look like this:

```
whiteKnights = 01000010 00000000 00000000 00000000
   00000000 00000000 00000000 00000000
```

Following this example, if we want to know where the white knights are, we simple check what bits are set in the white knight bitboard. In this example, it would be the second and the seventh bit. This also means that we can get all white pieces or black pieces by simply OR'ing all the white or black bitboards with each other. How can we use this to our advantage? Let us presume for a moment that we pre-calculate all attack tables for all pieces

on all squares. That means that for all knights, we have 64 bitboards, one for each square in the board. Each of those bitboards indicate which squares the knight is able to move to from the given square. That means that we can take this bitboard and, AND it with the negative of the white pieces bitboard (the collection of all white bitboards), resulting in all possible movements for the white knight. This makes move generation extremely fast since we no longer need to loop through all possible movement squares to see whether or not they were occupied and then checking, whether we can capture them or not.
But bitboards come at quite a high price: read-ability and write-ability. Bitboards are incredibly hard to read and write. Please look at Figure 5.5 and imagine the same scenario could be applied for queens and rooks. These are all sliding pieces, which mean that they have unlimited movement in any direction, as long as it is not blocked by another piece. In normal array representation, this would be trivial since you would simply loop in a given direction, and as soon as a non-empty square was found, stop there since the sliding piece can no longer move forward. This is not as simple using bitboards, since they do not tell whether or not a piece is blocked and in what direction. Another problem is that whenever a piece is captured, we need to loop through all of the opponents bitboards to figure out which piece was captured, since there is no way to know which piece was located on the given square.

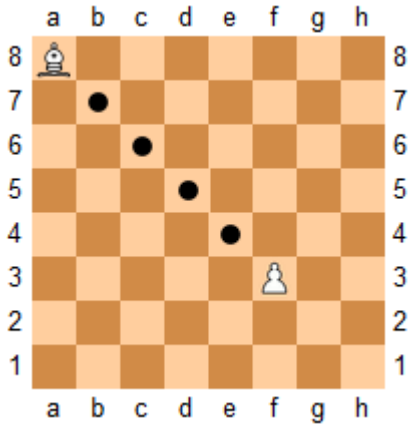


Figure 5.5: while(GetPiece(rank–,file–) != Occupied) { add(move); }
This would not work with bitboards, since we cannot easily ask which piece is occupying the given square!

After going through all this, let us now show a short summary of what

each method of state representation has to offer.
Array Representation:

- No need to access memory to check whether or not a move is valid
- When generating moves, each piece is processed one at a time
- Easy to understand and work with

Bitboards:

- No need to access memory to check whether or not a move is valid
- When generating moves, all pieces of the same type are processed at the same time
- Harder to code and understand

The simple fact that all pieces of the same type are loaded and processed at the same time is a huge benefit when we need to generate moves. Even though understanding bitboards and writing them are a lot more complex than using arrays, we feel that the possible speed gain by far outweighs the complexity of reading and writing bitboards. Therefore, we will be using bitboards in our program.

5.4 Transposition Tables

Transposition tables are used for storing chess positions during the search to avoid searching the same positions that may occur by other move sequences. For instance This position occur:
d2d4, d7d5, c2c4...
As well as: c2c4, d7d5, d2d4...
It makes no sense calculating the same positions again since we already know what the best move is from here.

Let us say we searched 8 half moves from the start position and found a line: d2d4, d7d5, c2c4, c7c6, e2e3, e7e6, g1f3, b8c6... and then evaluated it to get the highest score. When we get to this position after c2c4, d7d5, d2d4... we already know that the best move afterwards would be c7c6 and thereby saved 5 half moves of searching. [19] This way the engine could save a lot of time calculating since many moves would be skipped. We discussed implement this in our project but due to the time limit we had and the scale of work it would be to implement this correctly to be of use, we decided to lay our focus more on other parts of the project.

5.5 Zobrist Hashing

Zobrist Hashing is a way to make unique hashkeys based upon the pieces or moves done on the chess board. Zobrist Hashing is one way to generate this very unique hashkey to represent a hash table. We use Zobrist hashkeys to detect threefold repetition and whether our engine would gain an advantage by attempting to incoke a draw through the threefold rule. But why would we need a new way to represent a board when we already have both fenstrings and board objects to use for this purpose? If we consider a board object it contains a large amount of data that would be useless for checking whether or not that same exact board that has occured before and thereby the large amount of data saved would just be a waste of memory. Fenstrings on the other hand are a somewhat low memory way to store a board position with all information needed, but to generate a fenstring for every move would take up a lot of CPU time, up to 35 % CPU time. [20]

So how does Zobrist Hashing work? The first thing we do is to generate an array of 1024 random numbers, (16 * 64, where 16 is pieces and 64 is squares) and every time we make a move we look up in the array at the given index, XOR it onto the hashkey and then push the key into a stack. This way we get an almost unique key for every move.

Let us demonstrate with an example:
Let us say we have a chess game running and the current Zobrist key is 0. Two moves are then made, which we then look up in the zobrist table and get the values 1100 and 1010. We can then set up the following calculation:
0 XOR 1100 XOR 1010 = 0110
If we now make the same two moves we will have a repetition since:
0110 XOR 1100 XOR 1010 = 0
We would then have the board state "0" 2 times, so if this state would happen again the game would end in a draw.

Below is an example on how we manipulate the hashkey. We first need to remove whatever piece standing on a square by XOR'ing the index in the array with the zobristkey and then move the piece into another square by XOR'ing the index of the piece, to square and so on each move.

```
1 public static void MakeZobristMove(uint _piece, uint _tosq, uint
      _fromsq, Board b)
2 {
3     lock (b.AllStates)
4     {
5         b.zobristKey ^= PIECES[_piece, _fromsq];
6         b.zobristKey ^= PIECES[_piece, _tosq];
7         b.AllStates.Push(b.zobristKey);
8     }
9 }
```

Code example 5.1: Example on how we change the zobrist key

5.6 Board Implementation

As mentioned before, we will be using the Bitboard representation to represent our chess board internally. This is done to greatly improve move generation, since it has to be done so many times. We will now discuss how the board is implemented in this section.

A class called *Board* was created in order to keep all relevant information about boards in one place and to simplify the creation of move trees, which we will discuss later. By encapsulating all information about the boards in one place, it makes it incredibly easy to pass this object around. Let us examine how board information is saved on the board structure.

First, all board objects contain 12 bitboards, one for each piece of each color. The bitboards are saved as ulongs(64 bits), so that each square is mapped to a bit. A method called GetFirstBit() was created using De Bruijn Multiplication [21]. This method returned the index of the first bit and with this it was now possible to know the position of all pieces on all squares. This is because, like mentioned before when discussing bitboards, the index number of the bits that are set to correspond with the square number in which the piece occupies. All that needed to be done was to get each bitboard, scan using GetFirstBit, remove the bit and rescan to get the next. The removal of the bit would be achieved by using an XOR operation with the bitboard and another ulong with the only bit set being the one where the piece in question currently resides.

Besides keeping track of all the bitboards, it is also important to keep track of other information. Besides bitboards, the board object keeps track of who

is playing, number of moves, En-Passant possible squares, castling availability for both colors, as well as whether white or black is in check or checkmate.

Besides having all information about the board, there are also quite a few methods to help around with board management. One of them is called *ParseFen(string s)*. This method takes a fen string(an encoding of the current state of the table), and sets up the table according to the fen string. This method is called every time a new game is started, since it simply sets up the table according to a fen string over the standard position. It is also called whenever the interface wants to set up a specific board state and play from there. That means that our engine is now capable of taking chess puzzles and play them out. This will be crucial when it comes to analyzing our engines performance.

Another vital method is the *Update(Move mv)* method. Its function is to update the current board with a move. It has some helper methods that capture, move and promote pieces accordingly with whatever information was contained in the move it received.

Unfortunately, it was not possible to completely remove an array representation of the board. This is because of how information is saved on bitboards. If we were to completly avoid using an array representation, we would not know which pieces were located at square x without going through all bitboards and all bit indexes until we found a bitboard which had the bit index x set. Let us try to explain this with an example.

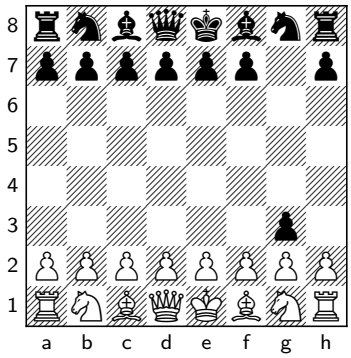


Figure 5.6: An imaginary position, white to play

As can be seen on Figure 5.6, white is up to play and can capture the black pawn located in square g3. Remember now that we do not save in-

formation about what pieces are located in square x. That means that we now have to loop through all bits in all bitboards in the board object until we find a bitboard that has the bit in index x set. In this case, it would be to find a bitboard with the twenty second bit set, which would be the black pawns bitboard. In a worst case scenario, we would be scanning the black pawns bitboard last and would therefore be looping through every single square 12 times (once for each bitboard), and then finally finding the black pawn bitboard, which has the twenty second bit set. This was something we wanted to avoid, so in order to do so we added a one dimensional array to keep track of what pieces where in which squares so that we don't have to loop through all bitboards every time we want information about a square. This isn't much of a concern since the one dimensional array does not check for move legality before updating its information. This is because our move generator only generates legal moves so that the only true drawback in using this array is in memory access times for each element in the array.

The board object now contains all information about a board at any given time. A board object is in fact an encoding for a board position: it holds information about where all pieces are located, castling rights, en passant capture squares, who is up to play, move count, move list etc and whether white or black is in check or checkmate. Each board object is a different state configuration of how the board looks like, or could look like after a move.

Chapter 6

Move Generation

We now need to generate all possible moves for a given board, in order for our chess program to be able to pick one of them and make that move. We will now discuss how moves are generated. We will start out by discussing non sliding pieces (Kings, knights and pawns) since they are simple to understand and will give a nice overview of how things are managed. We would like to mention that move generation was strongly influenced by another existing chess engine [22]. This is because the author was exceptionally good in explaining his ideas and it was very similar to how we wanted to implement our move generation.

6.1 Pre-generated attack tables

The first thing that was done was to create tables that contains all possible moves for all pieces on all squares. This was done in order to save operations when trying to find the best move. Since this would have to be done every time a move was to be generated and that all possible moves for all pieces are predetermined depending on where they stand. Even sliding pieces attacks are predetermined, but their attack moves are also based on the ray state. When we mention ray, we mean whatever direction the sliding piece wants to move. Let us look at an example.

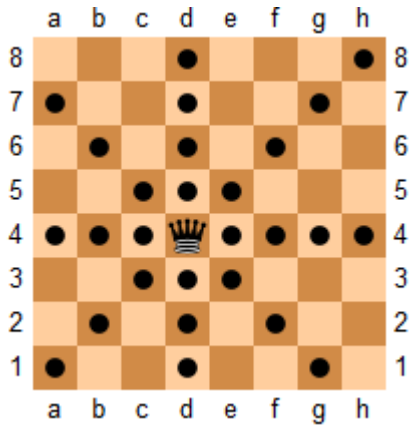


Figure 6.1: Rays for a Queen in square d4

As can be seen in Figure 6.1, the queen has 4 rays. One of the rays would be file d, which are all squares in the file d. Another one would be the rank, which would include all squares in rank 4. The last two would be the diagonal rays, which would be all squares in the ray going from G1 to A7 and the squares from A1 to H8

If it wants to move through the files(left/right), it depends on the occupancy state of the current rank. We will discuss how this is done when we discuss sliding pieces. Let us start with the non-sliding pieces first.

Let us take the knight as an example. Depending on where the knight is located at, it has a set of predefined locations it can move to. To find the set of locations a piece can reach, bounds are set up for where that piece can move. For example if the knight is in a rank that is larger than 1, then we know it can move two squares up and one to the right, as long as its file is smaller than the h file (g file, seventh). This is done for all attack patterns for the knight. Every time it finds a legal move, one that obeys the bounds, it is added to the final attack table for the square the knight is located at. If we look at the move possibilities for a knight in square A1, it is B3 and C2.

The same idea is extended with kings and pawns. An example for a kings attack table on square F5 is shown below on Figure 6.2. All black dots are positions in which the king can move to from its current square. Another example can be seen in the previous Figure 6.1. This example is only true though if the board is empty except for the queen. Pawns are a bit different though, since they are semi-sliding (it can be blocked by other pieces). As an example, think of a pawn located in position A2 and an arbitrary piece located in A3. Even though moving two squares ahead is normally a legal

move, it cannot do so because it is being blocked by the other piece located in A3. It also has different attack moves. This will be managed when we are generating valid moves for the table and not in the attack table itself.

These tables are what we call for attack tables and this is what is used to determine where pieces are allowed to move from depending on where they are located. The index for the attack table in question would be the square number the piece is located in. The attack table in question for that piece would be a bitboard where all squares it can move to have their corresponding bits set to 1 and all others are set to 0.

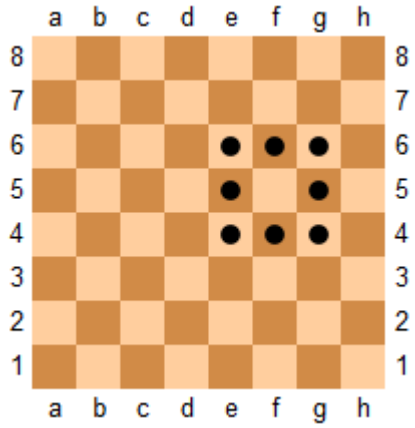


Figure 6.2: Attack table for king on square F5

6.2 Move generation for non-sliding pieces

Let us continue with the knight as an example. We constructed a class called *MoveGenerator* that receives a board object as input and then makes a list of moves accessible through a list called MoveList. Since the board has information about who it is to play, it calls the respective method generating moves for that color and putting those moves in the list for access. That means that the engine can now play for both colors.

Once inside the move generator, it is now time to generate moves for the knight. We now have the bitboard for the knights located on the board. We take that board and use the GetFirstBit() method to get the first bit from the board. It then uses that number as index to find the attack tables for that white knight. We then take the attack possibilities(remember that they

are saved as ulongs, where all possible moves are set with a one, everything else is set to zero), and AND that with the negative of WhitePieces, since all moves are valid except the ones that land on our own. We then loop through that array, extracting one bit at a time and adding that to the moves list. After each move has been added to the list, we remove that bit from the move table and start over. The same idea is applied to the pieces, once we are done generating all moves for one piece, we remove that piece from the bitboard and reloop to get to the next piece and generate moves for that. Here is some pseudo code for move generations:

```
1  whiteknights = board.WHITEKNIGHTS;
2  while(whiteknights != 0){
3      from = GetFirstBit(whiteknights);
4      knightmoves = attacktableknight[from] & ~board.
           getwhitepieces;
5      while(knightmoves != 0){
6          to = GetFirstBit(knightmoves);
7          moves.addmove(new Move(to, from));
8          knightmoves ^= to;
9      }
10     whiteknights ^= from;
```

Code example 6.1: Move generation psuedo code

Let us go through this code example so that we completely understand what is going on. On the first line, we get the position of all white knights on the board. On the second line, the while loop makes sure we still have white knights on the table. If there are no knights, no bits are set and therefore the value of the variable white knights is zero. Line 3 is where we get each piece on the bitboard so that we are able to figure out where that piece is capable of moving to. Line 4 is where we get the attack tables we generated and AND them with all squares that are not occupied by our own pieces. In this example it would be all empty squares and all squares occupied by black. This is because we cannot move to a square that is occupied by our own piece. Knightmoves are all possible locations the knight is allowed to move to. Line 5 is another while loop that checks if there are any more squares the piece can move to. If no bits are set, then there are no more locations to land on, and therefore no more moves to add. The next line gets the first bit set on all possible knight moves, which would be one of the squares the knight can move to. We then add the move to the list of possible moves for the current position. The next two lines remove either the move from knightmoves so that we don't generate the same move twice or remove the

piece from whiteknights so that we don't generate the same moves for the same piece twice.

This gives us a move list for all white knights. This process is repeated for all pieces. Pawns are handled a bit differently because of their special move and attack patterns. They actually have 3 move tables; attack, move and double move. Attack tables are done by ANDing the attack tables with board.getblackpieces. Normal moves are done with the negative of occupied squares, since they can't capture on a normal move, and finally double moves gets ANDed as well with occupied, and if another piece is found in its double move, it is not a legal move.

6.3 Move generation for sliding pieces

As mentioned before, move generation for sliding pieces is not as straight forward as move generation for non-sliding pieces. This is because their moves depend on how their rank, file or diagonal looks like at the moment. That means that a bishops' moves depend on were other pieces are located on their diagonal. Let us now refer to files, ranks and diagonals as rays, and consider them the same thing. We do this because they are actually the same thing for sliding pieces. The longest diagonal has 8 fields, the same as ranks and files. We simply push a diagonal or a file down to a rank. That means that we can create a two dimensional array with sliding attacks that covers all possible state occupancy for a ray for a piece on a given square. What is meant with state occupancy is what squares are occupied in a ray. So if for each square a piece can occupy in a ray, there are 7 other squares in the ray that can be either occupied or empty. That means that there are 2^7 possible occupancy states for each position in a ray. That means that for all rays, there are a limited number of moves a piece can make, depending on where the piece is located on the ray. This also means that they are predetermined and therefore we can create pre-generated attack tables for sliding pieces as well. These tables take form of a two dimensional array, where the first index is the square the piece is located at and the second is the ray occupancy state. Let us first discuss how we turn a file or a diagonal into a ray. Ranks are trivial since we simply take the first bit in the rank and get the next 8 elements in the bit sequence. To extract files and diagonals, we need something called magic bitboards.

6.4 Magic Bitboards

There are two different kind of magic bitboards we will use in order to make it possible to pre-generate attack moves for sliding pieces. The first one that will be discussed is the mask magic bitboard. What these masks do is to cover up all unnecessary information, only giving us what we need. Let us use a bishop that is located on A1 as an example. Also, remember that sliding pieces that can move diagonally can move in 4 directions, which means that there are 2 sets of diagonals in a board. These will now be referred as A1H8 diagonal (the diagonal that goes from A1 to H8) and the A8H1 diagonal (the one that goes from A8 to H1). Note that these only refer to the direction: there are actually 15 diagonals for each set of diagonals (A1H8, A8H1), as can be seen on Figure 6.3, where each black line is a diagonal. This means that there are 30 total diagonals on the board. That means that we need 30 magic boards to extract all pieces for each diagonal. These magic bitboards are created by doing the following: for each diagonal, we create a bitboard where only the bits in that diagonal are set. We found these magic numbers instead since they are trivial to create [22]. After these numbers are found, we AND them with a bitboard and now we have all pieces in a diagonal. Also note that we need to associate each square with its correct mask, therefore we create another array to which points to its correct ray mask to use.

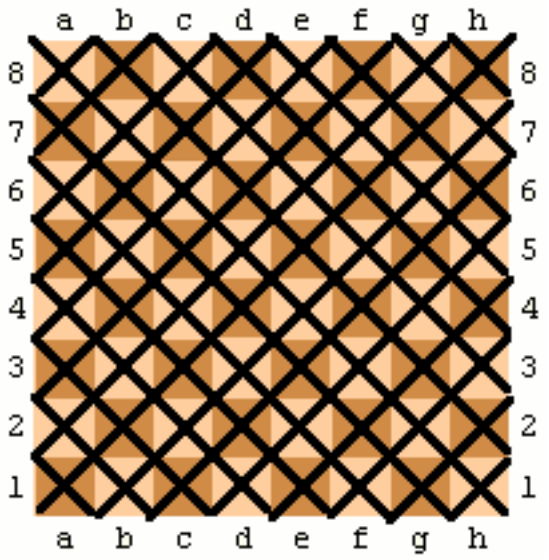


Figure 6.3: All diagonals in a chess board

The same idea applies to files, but they are even simpler to create since there are only 8 files. As mentioned before, no magic bitboards are needed for ranks since it is trivial to get the first bit of the rank sequence and count 8 elements ahead.
Now that we have ray occupancy for diagonals, ranks and files, we need to know how to create a single byte that describes the rays occupancy state. These are achieved by multiplying the result of the aforementioned technique (a bitboard where only the bits which occupy the same ray are set) with a magic number, which pushes all pieces of a ray to a single rank. This can be seen on Figure 6.4, where the left picture is a picture of the rays of a rook, masked by the corresponding file ray mask so that only the pieces on the same file as the rook are set. The picture to the right represents the same ray, after it has been multiplied by the magic number, pushing all the bits into the first rank, so that we can cut of everything above the eighth bit and have a ray representation. We now have a byte sequence that represents ray occupancy, as well as where the current piece is located. These means that we can now access the predefined attack table for sliding pieces. But how is this table generated?

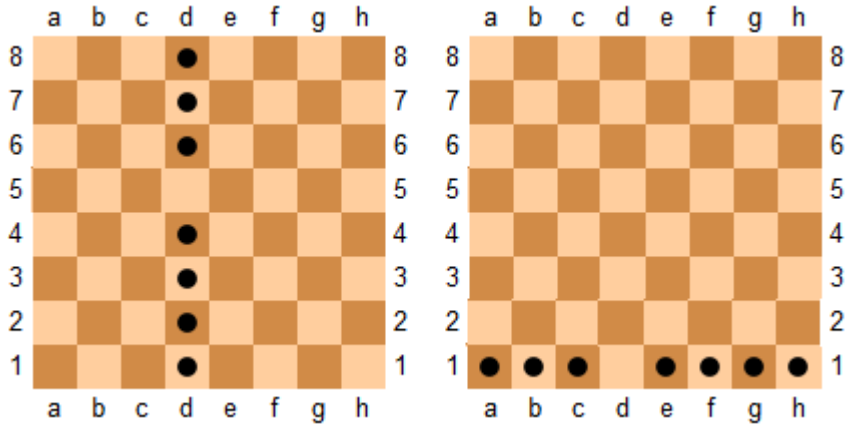


Figure 6.4: The picture to the left represents the file ray for a rook, which has been masked by the corresponding file mask.
The picture to the right represents the same ray multiplied by the corresponding magic number.

6.5 Attack table for rays

This is done by creating 2 loops, which iterate over where the current piece is located at as well as ray occupancy states. That means that the loop goes

through all possible combinations for piece location and state occupancy. This is done by simple adding and removing bits until all combinations have been tried. This enables us to try to move the piece to the left and the right for all occupancy states. This is first done for the left direction. If the bit is not set to the left of the piece position, a move is added, and it moves to the left again to check for the next bit. If a bit is found though, it is not a legal move and therefore no move is added. It also means that we cannot move the piece any further left, so we exit the loop and apply the same idea for the right side. After both loops are done executing, the array *SlidingPieceAttacks* now contains information for all sliding attack moves. We then create 4 different arrays, rank, file and both diagonal attacks for each square and ray state occupancy. We then loop through each of these arrays and use the sliding attack table to fill these arrays up. Let us tie a concrete example to make sure these ideas are understood.

6.6 Sliding Piece Example

The rook will be used as an example here. We start out by placing a rook in A1. To get all possible moves for this piece, we need to apply a mask to get its file occupancy state. The rank is trivial, we simply get the rank number, multiply by 8 and subtract one, and now we have the bit index for the rank in question. We shift 8 times and we have the ray occupancy state for that rank. For the file though, we get the file mask associated with the square A1 and apply it to the bitboard. We now multiply it by the magic number to push it down to a single byte sequence. We now have the ray occupancy state for the file. We now use a method called Rook attacks which asks both the rank attacks array as well as file attacks and OR's the two results to get a bitboard with all possible target squares, exactly like non sliding pieces. We can now use the above code snippet 6 to generate all moves for the rook. The bishop follows the same idea, except it asks the both diagonal tables (A1H8, A8H1) for possible target squares. Queens in turn OR's the result of Rook attacks and bishop attacks to get all its possible moves.

6.7 Is the square attacked?

A critical functionality in our chess engine is the ability to check whether or not a square is under attack. This would be needed in order to check whether or not the king is currently in check and evaluate accordingly, to check for move legality when generating moves as well as checking whether

or not the king can castle (will the king come under check in any of the squares it passes).

To get this information the engine would need to know whether an enemy piece is in a position that can attack that square and being sure nothing is in the way. The obvious way to get this information would be iterate through all enemy pieces and check whether it can attack the square. This would be a rather long calculation and since it would have to check whether or not the king is in check for every move generated, this method would take far too long.

Since the logic of the game of chess states that if an enemy piece can take one of your pieces of the same type, you are also able to take his. This gives the possibility to reverse the function. Instead of checking every enemy piece we take the position and for each type of piece, we simulate that piece in our color on the square. The logic then dictates that if we can capture one of a similar type, the square is attacked by that piece. This logic also applies to pawns, since they move in opposite directions. 6.5

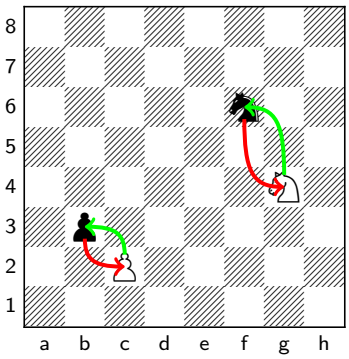


Figure 6.5

The function does not only give us the possibility to know if a square is attacked, but by reversing the function, meaning calling it with the opponents color, we can know whether or not a square is protected. By checking to see if we, from the square, can capture one of our own pieces, we will know that the same piece would be able to capture whatever lands on the square.

```
1 IsSquareAttack(square)
2   foreach (Type of piece)
3   {
4     if ((Attacktables(Type, square) & enemypieces(type)) != 0)
5       return true;
6   }
7   return false;
```

The way we check is to call IsSquareAttack with the square we want to know about. In the function we loop over all types of pieces and see if the

attack tables for that square can capture one of the enemy pieces of the same type. If it can, then it will return true since this is enough information. Else it will continue in the loop, and if no pieces can be captured, it will return false, implying that the square is safe from all enemy pieces.

6.8 Complexity

We have calculated the time complexity of our move generation. This can be seen in our appendix. It has been broken up into several parts in order to better fit the pages. Each part represents the generation of moves for each type of piece. The final results of our time complexity analysis are the following :

- Generating moves for knights: number of knight pieces * number of possible moves
- Generating moves for kings: number of king pieces * number of possible moves
- Generating moves for queens: number of queen pieces * number of possible moves
- Generating moves for bishops: number of bishop pieces * number of possible moves
- Generating moves for rooks: number of rook pieces * number of possible moves
- Generating moves for pawns: number of pawn pieces * number of possible moves

That means that the complexity of our move generator would be the sum of all previously mentioned routines. As can be seen, the complexity is equal to the sum of all possible moves for all possible pieces for a given possition. This can be expressed as:

$$\sum_{i=0}^{k} NumOfMoves(p_i)$$

where k is the number of pieces for a given color, p_i is piece number i, and $NumOfMoves()$ equals the number of moves for a given piece.
This formula also equals the branching factor for a given position. The branching factor is generally refered to as b, which would mean that our move generator has the complexity $O(b)$

Chapter 7

Distributed system

Now that we have found out have to implement and design our engine, it is time for us to consider our second problem, how do we distribute our system so we can have multiple computers calculating to find the best move.

In this chapter we will be describing our path towards a distributed system, including the decisions made and the methods used.

7.1 Connection method

In order to create a distributed system it is necessary to use a common connection platform. When looking at such there are several different options, one of the most basic being sockets. Note that we are considering built-in options in the .NET 4.0 paradigm. The usage of network communication to and from our chess engine would however require sending and receiving an entire object - a node from our tree. If this were to be done using sockets it would require us to serialize the object into a byte array, send it to the receiver and then deserialize it into an object again on the receiving computer. There is however another way which implements automatic serialization and deserialization of object called WCF (Windows Communication Foundation). In WCF contracts are used to tell the client and the server what types will be passed and returned when calling a given method. These are called operation contracts since they define a callable operation on the server. These contracts sometimes needs to be backed up by data contracts. Data contracts are used when the returning type or one of the parameters of an operation is a custom class. Data contract is a tag set on the custom class that will be used in some operation between the client and the server. Once this tag, called "DataContract", is set it tells WCF that this custom class can be used and serialized for communication purposes. However when having big

classes it is not always necessary to pass every property of that class over the network. Some of these properties might only be needed locally and is as a result not necessary to include in the passing object. WCF determines which properties should be used for this through the "DataMember" tag on every property that should be included in the sending object. See 7.1 for an example of a basic custom class called "Hello". This shows that the value of the property "id" will not be included in the object once it is transfered to its recipient. This could be used in a case where the "id" of each object is handled locally while the "message" contains the message for the server.

```
1 [DataContract]
2 class Hello {
3   int id;
4
5   [DataMember]
6   string message;
7 }
```

Code example 7.1: A basic custom class

Because of the ease of use we will be using WCF as our communication base. With this being said there are still a few decisions to be made which will be covered in the next section.

7.2 Communication

We have three different roles in our distributed system: master, slave and manager. A description of the different roles:

Master

The master is responsible for handling all connections to the slaves associated, both handling the distribution of nodes created by the master and updating the scores of each nodes in the final node tree.

Slave

Each slave is responsible for listening on a channel for connections and stating out to the network that it is an available resource ready to be used. Slaves will be using the same algorithm for searching for the best move as our chess engine would when playing without the distributed part.

Manager

A manager was implemented to make it easier for the master to search for available slaves outside the LAN (Local Area Network) range, by implementing a list of active slaves. By having the manager placed at a public available location, in the terms of network access, it is possible for the master to connect to available slaves all over the world.

Knowing what roles are needed we can now develop a client-server environment. Taking the responsibility of the three roles into consideration the master will need to run a client, while the slave and manager will be hosting a server. This is because both the slave and the manager will be receiving operation calls, while the master will be the one calling them.

The hosting service, which will be listening on an address for input, will be called ListenerService while the client/connecting service will be called EngineService. These service classes mostly contains technical information such as timeout, security aspects etc. This however will not be covered in this report as it is not relevant to the reader. Only significant or otherwise important parts of these services will be described in detail.

ListenerService

The interesting part about this service is that it actually also contains an EngineService . This is used only by slave roles to connect to a manager to add itself to the list of active helpers. Likewise there is a similar method to remove itself from the list of active helpers once the program is shut down both intentionally and unexpectedly.

EngineService

A core part of this service is the GetNewScore method which calls the assigned slave with a node and returning once it receives a result. This part of the code is illustrated in pseudocode in 7.2.

```
1 while (Result not received)
2   if (Stop requested || Time is up)
3     Invoke Stop on connected slave
4   else if (ForceStop requested)
5     Invoke ForceStop on connected slave
6     return
7 return result
```

Code example 7.2: GetNewScore - Waiting for response from slave

The difference between Stop and ForceStop in the above code is described later in the interface contracts.

The communication between all computers, regardless of their role, is given by an interface which shows what operations are available to other computers. Since all three roles and the code for these are contained within the same project file, all these operations exist in the same interface. However if the different roles each had their own executable, it would have made more sense to split up the interface for each role. The operations currently being used are showed in 7.3.

```
1  [ServiceContract(CallbackContract = typeof(ICallback))]
2  public interface INodeDistributor {
3    [OperationContract]
4    Computer RequestBenchmark(string remoteAddress);
5
6    [OperationContract]
7    void ConfirmConnection(string remoteAddress);
8
9    [OperationContract(IsOneWay = true)]
10   void GetNewScore(RootNode lf);
11
12   [OperationContract]
13   void Stop();
14
15   [OperationContract]
16   void ForceStop();
17
18   [OperationContract]
19   void AddAsActiveSlave(string remoteAddress);
20
21   [OperationContract]
22   void RemoveAsActiveSlave(string remoteAddress);
23
24   [OperationContract]
25   List<string> RetrieveActiveSlaves();
26 }
27
28 public interface ICallback {
29   [OperationContract(IsOneWay = true)]
30   void ReturnResult(Node result);
31 }
```

Code example 7.3: The interface contracts used

The master is the role calling most of these operations including RequestBenchmark, ConfirmConnection, GetNewScore, Stop, ForceStop and RetrieveActiveSlaves. It also implements ReturnResult from the ICallback interface. This is to handle the result from the GetNewScore call.

The slave implements the operations the master is calling with the excep-

tion of RetrieveActiveSlaves which is implemented by the manager. Implementing these operations means that the slave is responsible for creating the logic that perform the decided action of the operation and return the values of the type given by the interface.

The manager implements AddAsActiveSlave, RemoveAsActiveSlave and RetrieveActiveSlaves. It is not calling any operations and serve only as a point of reference between the addresses of the slaves and the master.

RequestBenchmark

receives a string, representing the address from where the operation was called, in this context the address of the master. The slave then performs several actions in order to fill up a "Computer" class containing all necessary information about the given slave such as address, benchmark test result, amount of cores etc. This information gives the master a way to determine how powerful the slave is compared to other slaves and is necessary to create a proper workload division between connected slaves.

ConfirmConnection

is a simple method to determine whether or not the master can connect to the slave. It also takes a string as argument to represent the address of the master and is used for output on the slave console.

GetNewScore

can be seen as the core operation between the slave and master. The master passes the node it want to update the score for, and the slave then uses our algorithms to find the best move from the given node. When the algorithm is done either as a result of completion or as a result of a stop call from the master, it returns the resulting score, contained in a node, back to the master. However, as you can see in the interface, the operation is set to return void rather than a node. This is due to the fact that we need to slaves to run asynchronously for each operation. If this run synchronously we would not be able to invoke a stop command to the slave since it would wait until the currently working operation is complete. The result is instead returned through a callback to the master using the ReturnResult operation in the ICallback interface. The logic behind ReturnResult is as mentioned implemented on the master.

Stop

is used to invoke a stop on the selected slave, notifying the slave to stop any further computation and return the result it currently has acquired. This is used in terms of time restraints when the master decides that no further

time should be spent on computing the current move.

ForceStop
implements a slight variation of Stop in the sense that the current result is irrelevant to the master and forces the slave to immediately stop computation and simply return nothing. This is used if the master learns that the node the slave is working on is a part of a cut branch in the tree, making it useless to the selection of the best move.

AddAsActiveSlave & RemoveAsActiveSlave
is implemented by the manager and is called by a slave when it respectively starts and stops. The slave simply informs the manager of its address thereby telling the manager that it is available/unavailable to be used by a master. On the managers side it simply adds and removes the received address to a list of slaves.

RetrieveActiveSlaves
is also implemented by the manager but is instead called by the master to retrieve the list of currently active slaves. It should be noted that this list currently both included addresses of local and remote addresses even though the local addresses might not be available to the master if it is not in the same network as the slaves with local addresses. However, the master does confirm the connection to each slave before adding them to the final list.

ReturnResult
from the ICallback interface is implemented in the master and is responsible for handling the result and insert it as the result from the corresponding slave it received it from. This is done using a specific identification variable for each slave and for each node, which are both checked to verify the correctness of the result.

Managing all our slaves is done through a class called DSHandler, which includes a list of slaves along with methods such as AddSlave and RemoveSlave. It is also in this class we determine which role the program will be having(master, slave or manager). This is simply done through command line arguments when the program is executed.

An interesting feature of the DSHandler is a method called SearchForSlaves. This method is split into two areas, one being an automatic search for hosted slaves on the LAN which is achieved by using a feature from WCF which searches for hosted WCF services on the LAN. When this search is done, the second area of the DSHandler kicks in by contacting a manager and receiving a list of active slaves. This list is then handled, confirmed and

cross-referenced with the list of slaves found on the LAN to remove any duplicates. Once done, this leaves us with a complete list of all available slaves stored in the DSHandler for later use to the master.

7.3 Distribution

Now that we have the methods and tools needed to communicate between several computers we need a way to properly distribute our nodes for computation.

7.3.1 APHID Algorithm

APHID is an algorithm that was created in an attempt to implement parallelism in an alpha beta game search in order to achieve better results [23]. There are several benifits to APHID over several other avaiable algorithms - APHID use itterative deepening to reach a certain fixed depth which is below its total target depth. At this point, it evaluates those current nodes and generate bounds for alpha and beta, since the moves at the moment are estimates at best since we are looking to evaluate the leaves at a deeper depth to obtain a more accurate score from the tree. This removes synchronisation points from the alpha beta routine, since we can now make an estimate as to whether or not these nodes belong to a cut branch. If they do, they no longer need to be evaluated further. This means that it is possible to list all possible moves that lie within the bounds and continue our search in each of them in parallel. This is the main idea behind APHID.

The master is in charge of generating the upper part of the tree. If our target would be a depth of d, we would generate $d - d'$ on our master, and have the slaves search from d' to d. Every time the algorithm reaches a leaf node, it assigns that node to one of its slave computers. The slave computers then in charge of determining an exact score for the node they are assigned. The master then continuously propagates the tree, updating its bounds. If an accurate score is available from one of the slaves, it updates its current values accordingly. Every time the master has done a complete sweep of the tree, it must check whether or not all nodes have been searched by the slaves. If they have, we now have an accurate score and can return the best move.

Communication between slave and master is done through a table called the APHID table. Each slave has its own table with the master. Every time a node has been visited, the node gets added to one of the slaves APHID

table in a round robin fashion. Information on those nodes include moves needed to reach that current position from the root, current depth and target depth. That part is only writable by the master, when he adds nodes to the slaves list of nodes. The slaves on the other hand can access the other part of the table where it can write information about the current node including whether or not it has been visited, its core as well as the depth in which the score is based on.

The slave has very few routines: it asks for a node to calculate an accurate score for, as well as updating its search bounds with the master. It then writes this new information to its portion of the APHID table. It must also be told to abort computation on its currently working node in case it has fallen out of bounds of the current search. Once a slave has finished calculating all of its nodes, it picks the one with the best score, and continues to search that node through iterative deepening in an attempt to achieve an even more accurate score for that node.

The master, once it sees that all nodes have been searched to d, the target depth, propagates the tree one last time, and returns with the score for the best possible move.

Our solution is a variation of APHID, working by first retrieving the list of available slaves using our previously mentioned DSHandler.

Two different classes were created in order to handle distribution of nodes between the master and the slaves. The class responsible for time and node management is called "Master". The second class is called "Slave" which is responsible for calling the appropriate methods in order to compute the list of nodes it has been assigned. There will only be one instance of the master class running at any given time. This class in turn has a list of "Slave" classes, one for each slave computer available. Though the name "Slave" might imply that the class is being run on the slave computers, it is not - both the "Master" and the "Slave" class are used only on the computer with the master role.

Whenever a new instance of the master class is created, its constructor is called(see 7.4), adding available slaves followed by a computation of an initial tree based on the depth received. This initial tree continuously calls AddNodes in the master(described later in 7.6) thereby adding nodes to the slaves at the same time as the initial tree is still being computed. When the initial tree has finished its computation the code runs into a while loop checking whether all slaves are done or if the time has elapsed. If none of

those conditions are true it keeps updating the initial tree as well as balancing the slaves.

```
1  static List<Slave> slaveList;
2
3  Master(Board, int masterDepth, int slaveDepth, int time) {
4    foreach (Computer in DSHandler.ListOfHelpers)
5      slaveList.Add(new Slave(Computer, this, slaveId++));
6    // The initial tree of depth "masterDepth"
7    FindBestMove(boardToSearch);
8
9    while (Slaves not done && Time not elapsed)
10     UpdateTree();
11     BalanceSlaves();
12   // The final update
13   UpdateTree();
14 }
```

Code example 7.4: Pseudocode for our Master class (constructor)

The balancing of slaves is done by running through all slaves twice in two loops as can be seen in 7.5. The method first determines an average amount of total nodes based on the the amount of nodes from each slave divided by the amount of slaves. A relocation of nodes is then performed between all slaves with the goal to have all slaves have the same amount of node as the previously determined average. When done the slaves are considered balanced.

After a node is done computing on the slaves, its value is ready to be updated to the initial tree in case no cuts occurred. This is done using our UpdateTree method(see 7.5) which updates our initial tree with all the new available scores. Finally it runs through all slaves to check if their currently working node is a part of a cut node.

```
1  // Balancing the slaves my relocating nodes between all slaves
       until all slaves has the same average amount of nodes to
       compute
2  void BalanceSlaves() {
3    int missingNodes = 0;
4    foreach (Slave in slaveList)
5      missingNodes += Slave.Nodes.Count;
6
7    int average = missingNodes / slaveList.Count;
8    foreach (Slave s1 in slaveList)
9      if (Slave.Nodes.Count > average)
10       foreach (Slave s2 in slaveList)
11         while ((s2.Nodes.Count < average) &&
```

```
12                  (s1.Nodes.Count > average))
13              s2.Nodes.Enqueue(s1.Nodes.Dequeue());
14  }
15
16  // Updates the tree in the movefinder with the newly found
        values and checks if any nodes have been cut
17  void UpdateTree() {
18    Update tree in movefinder
19    foreach (Slave in slaveList)
20      Slave.CheckCut();
21  }
```

Code example 7.5: Pseudocode for our Master class (Slave balancing and tree update)

Nodes is, as mentioned, continuously added to the slaves using AddNode(see 7.6) while the initial tree computes. This method keeps track of a counter used to pass nodes to the next slave in line every time it is called. It is however necessary to run each slave in a new thread so that parallel computation and handling is possible.

```
1  int count = 0
2  // Add nodes while the initial tree is still being computed
3  void AddNode(Node) {
4    Node.DepthToSearch = DepthTotal - DepthPrime;
5    slaveList[count].AddNode(Node);
6    // Checks if the slave has been started
7    if (slaveList[count].FirstRun)
8      slaveList[count].FirstRun = false;
9      new Thread => slaveList[count].StartWorking();
10   count++;
11   if (count >= slaveList.Count) count = 0;
12 }
```

Code example 7.6: Pseudocode for our Master class (Adding Nodes)

The "Slave" class has several different properties to determine identification, completion, states and so on, but the main method is the method called StartWorking(see 7.7). This method is responsible for making sure that the class instance keeps sending nodes to the associated slaves until the list of nodes has been depleted. It contains two loops because of the fact that if the slaves completes computation of all their nodes before the master has completed computation of the initial tree the slaves would go inactive. Once we are sure that the initial tree is done, the list of nodes associated with the given slave can be depleted knowing that it is the final run.

```
1 while (Initial tree not complete)
2   // Nodes denotes the nodes awaiting computation in this slave
3   while (Nodes.Count > 0)
4     // Send the node to the associated slave using the
          EngineService class
5     DispatchNode();
```

Code example 7.7: Pseudocode for our Slave class (The StartWorking loop)

Chapter 8

Adversarial Search

Adversarial search is used in games where a player attempts to maximize their chances to win over another player.

The adversarial search is a basically a tree where each node represents a possible move. The adversarial search can in most cases be compared with a depth first search with a limited depth. Some sort of adversarial search is used in many turn based games such as tictactoe, go, and of course chess, to determine the best possible outcome from a move.

This chapter will set focus on adversarial search and how we use it in our chess engine to determine what move it should take and how it should act depending on the its opponent's moves.

8.1 Game Trees

When we are playing chess, we often discuss the possible moves someone can make, and what positions those moves will lead to. If we for example have a single pawn on the board in its starting position, then the pawn has two valid moves; a single move as well as a double move. That means that at the current board position, there are two possible positions that can be achieved, depending on what move is taken. We could go a step further and discuss what possible positions those two positions might lead us to. For simplicities sake let us just assume, for this example, that pawns can always move one or two moves regardless of whether or not it is their first move. Figure 8.1 shows a visualization of such a tree:

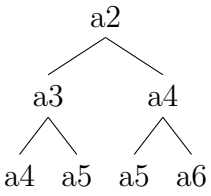


Figure 8.1: A game tree for a table with a single pawn on square a2, that can always move one or two squares forward

From the above Figure 8.1, the root of that tree would be the a2 node. This is where all branches of the tree comes from, which also means that all future position originate from this one position. Leaf nodes are located at the end of each branch. In this example, the leaf nodes would be a4, a5, a5 and finally a6. These are the positions that can be achieved, depending on what branch is taken. All other nodes are intermediate steps from the root to the leaves, which would be all moves that lead to that position. That means that every time we go down a branch, we go one level deeper in our tree. We do this every time someone has made a move. In the above example, the tree is a 2 node deep tree.

As mentioned, this is what is called a game tree. It is a tree composed out of all possible moves from that position, and all the moves that those moves generate, and so forth. The idea behind game trees is to be able to give the engine foresight so that it can see the positions after making certain moves and giving it the ability to analyze these positions in order to determine which ones it would like to achieve. This also gives the engine a way to search for a path towards these positions since it now knows what moves lead to that position. In theory it would be possible to create a complete game tree over the game of chess because of the 3 move repetition rule, which leads to a draw as well as the 50 move rule, which also leads to a draw. This effectively removes the possibility of an infinite loop. It is important to note that this is all theoretical: there are 318,979,564,000 different ways of playing the first four moves! It is estimated that there are more 40 move chess games then there are galaxies in the universe (100+ billion) [24]. So, while theoretically possible, it is not feasible with current technology. That means that we have to limit our tree to a certain depth in order to be able to achieve anything, or else our computer would be stuck generating a game tree for generations to come.

The way a game tree is created in our engine is fairly simple. A class called *MoveFinder* was created that was in charge of finding the best move to play. Note that the tree is created at the same time we search for a move.

We will discuss how we are searching in the next section. Another class was also created called the *Node* class. Each node class contains a Board class, which represents the current state of the board. Each node also contains a list of children, which represents a list of possible positions that can occur from this current position. We generate these children by creating a new node class, passing a board as well as a move. The node then makes sure to update its copy of the board with the move given to it. This newly created node is then attached to the list of children of its parent node. By doing this x number of times, we can create a game tree that looks x moves ahead for all pieces.

Now that we have discussed how game trees are created, we will discuss how we search for moves to make, as well as deciding which of the moves we will make. We start out by determining a depth (number of half-moves forward) we want to look at. We do this so that the computer doesn't spend generations trying to achieve the final leaves (positions) of a chess game. From the starting position, we generate all possible positions caused by every possible move. We then do this same procedure for all its children, and we repeat this until we reach the leaves (end positions) of the tree. Once this is done, we evaluate that position and assign that score to the node. This is done by using our utility function which evaluates the board for whoever is searching and assigns a score accordingly. This means that the score of each node is either its own utility value, or its children's, depending on how deep we are searching and how deep in the tree we currently are. The reason why only leaves are evaluated is because they are just more specific versions of their parents. If for example a parent node is missing a queen, its child node will reflect that, as well as reflecting any other moves that are made. If it was possible to predict which move someone would take, it means that we could use a node's child(the one achieved by the move we predict our opponent to make) as its own score, since that position will always lead to the next move, making the current node irrelevant since we have a child node which is more "specific", because it has information of a move we know our opponent will make.

Imagine for a minute that we are playing a game of chess. We are infinitely intelligent, so we always know which move is the best possible move to make at any given time. We are playing against someone that has no understanding of chess. Not only that, he is incredibly unlucky and happens to choose the best move for you every single time and we know this and abuse this fact as well. If this was the case, finding the path towards our desired position would be simple. We are now at the leaves of our game tree: these are positions that is achieved after we make a move. If we know that he picks the best

move for us every time, we simply evaluate all leaves and return the best possible position. Once we return this value, we are now in a position that is achieved after we make our own move. Since we know what move our opponent will make in response to all of our moves, it is a simple matter of finding the best possible end-result and follow that path. An example of such a game tree can be seen on figure 8.2

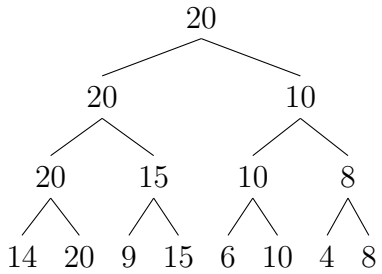


Figure 8.2: An example game tree following the best possible moves at all time

Let us go through this tree as an exercise to make sure the reader fully understands the concept of why we are able to cherry pick a position and lead ourselves to that position. Also note that in this example, only two moves are possible for each position. This isn't the case in real chess. There is an average of 30-40 possible moves for any given position [24]. We start out on the leaves by finding the position that gives us the best possible score. Note that the score is relative to us, the currently playing player. The higher the score, the better the position is for us. In this case, it would be the second leaf, the one with the score of 20. Higher scores are better since they indicate a better position for us according to our utility function. This position is achieved after one of our moves, which comes from the leaf's parent node. In this position, we can either go to a leaf that has the score 14, or the aforementioned leaf with the score of 20. Since we are infinitely intelligent, we will always pick the best move possible, which would be the one with the score of 20. This means that the parent's score is irrelevant since we will make that move, and therefore that position will always lead to the next one with the score of 20. If we go up further, it is now the opponents turn to play and he can either choose a move that will lead to the score of 20, or one that leads to the score of 15. Since he is infinitely stupid, he will of course choose the best move for us and choose the node with the score of 20. The same logic applies as before: since we know for certain which move he will make, the score of the current node is irrelevant since we know that from this position, the next one will always occur and therefore we can use that

node's child score as its own. We now go up the tree one last time and find ourselves at the current position. It is our turn to move, and we can now choose to follow the path that leads us to a position with a score of 20, or the score of 10. We will of course choose the one that leads to the score of 20. It is important to note that the logic we followed throughout this branch is applied to all branches. We evaluate all leaves and always choose the best possible move since both players are playing towards the same goal, which would be you winning.

Luckily, this isn't how chess is played. If every game revolved around making sure one player won in the most glorious way possible, chess would devolve into a game where winners were decided by their ability to argue for playing black and performing a Fool's Mate [25] on their opponent. This would cease being interesting after the first game.

But if the game isn't played this way, how can we predict which moves the opponent will play? What use is it to search for beneficial positions if we have no control over what moves our opponent will play? Well, instead of assuming our opponent is infinitely stupid, we can assume the exact opposite. If we assume that the opponent is infinitely intelligent, just like us, we now have some form of qualified guess of which move the opponent will choose. Also, in case the opponent doesn't choose the best possible move, it means that we now have a small advantage over him since we in turn did take the best possible move. This is the core idea behind the Mini-Max search algorithm that is used in our program.

8.1.1 Zero sum game

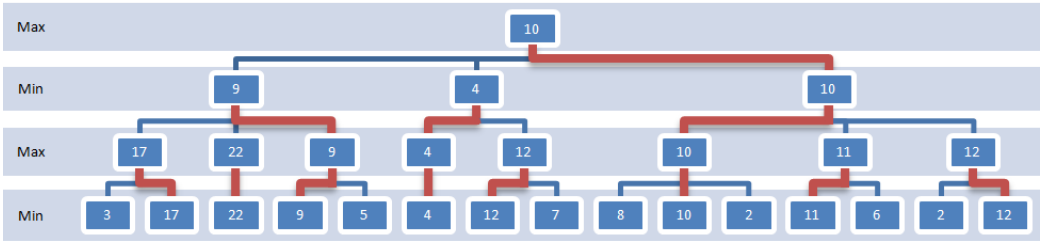
In game theory, a zero sum game is a situation where one participant's gain of utility is the other participant's loss of utility, for instance if one participant gains +1 utility, the other participant gets -1 utility so in other words the utility values are always equal and opposite. If the total of both participants utility is summed, it will equal to zero, thus a zero sum game. A zero sum game is strictly competitive since if one player wins, the other player loses. Chess is by this definition, a zero sum game, since two players act alternatively against each other to kill off the others king. One players advantage could be being ahead one pawn which gives a good chance of winning. A secure advantage is equivalent to about three pawns and should give almost certain victory. From the opposite player's view, being one pawn behind is very bad and being three pawns behind is almost certain defeat. [4] In a later section we will discuss the minimax theorem which takes advantage of the

zero sum property.

8.1.2 Recursive Mini-Max

The minimax theorem states that a two-player zero-sum game has optimal mixed strategies for each player, where the expected payoff V for player 1 is the negative equivalent to the expected cost for player 2, which would be $-V$. Vice versa the expected payoff for player 2 is same as the cost for player 1. This theorem was established and proved by John Von Neumann in 1928 [26]. Minimax is used in decision theory, game theory, statistics and even philosophy. A simple form of minimax is used when dealing with games such as tic-tac-toe. In a tic-tac-toe game, say a player 1 can make a move which leads to player 2 winning and another move leading to a draw. Player 1 will want to minimize player 2's chances of winning, and thus increase his own chances of winning since it is zero sum, and in this case choose to lead to a draw.

The main idea behind the Mini-Max algorithm is that every time our turn is up, we want to choose the best possible move out of a series of possible moves, and that the opponent will always pick the move that is most detrimental for us, the worst move. We still apply the same logic as before: we always know which move we will take, and we presume to know which move the opponent will make, which would be the worst move for us. When we know which moves will be taken, we can base the parent's node score on the child's score. This means that whenever it is our turn, we follow the path with the *Max* possible score. Whenever it is our opponents turn, we follow the path with the *Minimum* possible score. This is where the name comes from: the minimum of the maximum. Let us look at an example of a mini-max tree below on figure 8.3a



(a) An example of a minimax tree.

As can be seen, this is a depth 3 tree, which means that the leaves are positions that occur after we make a move. Since this is the deepest level of

the tree, we evaluate all the leaves with their respective scores. Remember, we are scoring the leaves according to who is playing! Good moves have really high scores for the player searching while not so good moves have lower scores. We now go up a level, and are going to search for the best possible move for us. Since we are the one making the move, we want to pick the *Max* possible score to use. As can be seen, all nodes in this level inherit the score of their child node with the highest score of them all. We now move up one layer in the tree. It is now the opponents turn to play, and he wants to minimize our advantage. We assume he is infinitely intelligent, and assume he will pick the worst possible move for us. That means he is searching for the *Minimum* node of all its children. Once it is found, the parents inherit that score as well. Going up again to the last layer, we now want to get the best possible move and therefore end up picking 10 as the best possible achievable score. We then pick up that move to lead us forward.

Let us now look at a concrete example in 8.3.

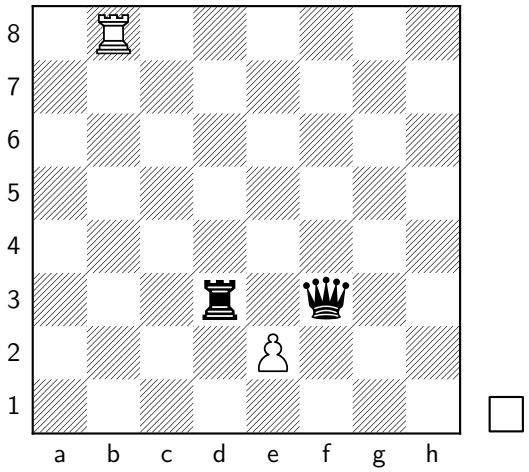


Figure 8.3: White to move, can capture queen or rook

Even though the example is unrealistic since there are no kings, it serves perfectly to demonstrate how mini max works. It is now white's turn to play, and he has two moves. Regardless which move white makes, black can only make 2 counter moves. Those moves depend on our choice of initial move. Let us construct this tree to better visualize how it works.

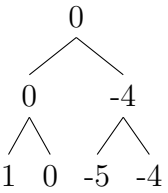


Figure 8.4: Trimmed game tree for 8.3

The above game tree is a trimmed one. We assume for the sake of simplicity that the white rook cannot move and that the only moves both players can make are capture moves. That means that they can either capture something or move somewhere else. In this case, the only evaluation feature that matters is material value, so that is the only thing that will be taken into consideration. The current evaluation score is now currently one rook and a pawn for white (6 pawns score) against a rook and a queen (15 pawns score). That means that the total score is -9 pawns. The left branch of the tree equals us capturing the queen. Our opponent can then make 2 moves, either capture the pawn or escape. If escapes, the score would then be a rook and a pawn against a rook, meaning that the score would then be a +1 pawn. If black captures the pawn, then it is one rook against one rook, meaning that the score would be 0. This can be seen in the tree as the children of the left move, the one where we would capture the queen.
The right branch is white capturing the rook. Here, our opponent has two moves again. Capture the pawn or escape. If he escapes, the pawn will still be alive, meaning that the score would be 1 rook and a pawn against a queen. The score would then be -5 pawns. If he captures the pawn, the score would then be -4, since it is a queen vs. a rook. Just like the other branch, these positions are the children of the right branch.

Minimax would work as following. The engine would find itself in the current position, and asked to find the best move. It would generate all moves for that position, and start out by making the first move and visiting that child. Once there, it would apply the same procedure; generate all children and visit the first. We are now located at the leaves of our tree, that means that it is time to evaluate and return the score. Now that we have done that, we repeat the same procedure for the next move and return the score. That means that we now have the score for both moves that black can play if we play our first move, e2f3, which would capture the queen. The opponent wants to maximize his score, so he chooses to capture the pawn, which is the move that ends with the 0 score. That means that this particular node gets the score 0 as well, since that is the best move that the opponent can make.

We now examine the other branch. We apply the same procedure as before and we are now analyzing for the best move black can make; either capturing the pawn or escaping. Again, the opponent wants to pick the best move possible, so he chooses the best move he can make, which would be capturing the pawn, and thereby determining that nodes score to -4, which would be the move in which we capture the pawn.
We have now analyzed both possible outcomes of our moves. We then go through all our moves until we find the one with the highest score, which in this case would be capturing the queen, this giving the current node a score of 0, since if the engine made that move it would ultimately end up with the rook capturing the pawn we used to capture his queen.

Let us look at some pseudo code for the minimax algorithm:

```
1  RootMiniMax(Board board, int depth){
2      score = int.minValue;
3      GenerateMoves(board);
4      for all (moves) {
5          result = MinMax(board(move), depth);
6          if(result > score){
7              score = result;
8          }
9      }
10     return score;
11 }
12
13 MinMax(board, depth){
14     if(depth = 0) { return Evaluation(board); }
15     score = int.maxValue;
16     GenerateMoves(board);
17     for all (moves){
18         result = MaxMin(board(move), depth -1);
19         if(result < score){
20             score = result;
21         }
22     }
23     return result;
24 }
25
26 MaxMin(board, depth){
27     if(depth = 0) { return Evaluation(board); }
28     score = int.minValue;
29     GenerateMoves(board);
30     for all(moves) {
31         result = MinMax(board(move), depth - 1);
32         if(result > score){
33             score = result;
```

```
34            }
35        }
36        return result;
37    }
```

Code example 8.1: Mini Max psuedo code

From the above code snippet, we can see that our minimax code is mutually recursive since MinMax calls MaxMin, which in turn calls MinMax until the leaves of the tree are reached. This is also the main idea behind minimax: we want to maximize our own utility, while the opponent wants to minimize it. Therefore, whenever it is our turn to move, we choose the path with the maximum utility, and when it is our opponent, we choose the path with minimum utility.
We start out at the root, always trying to maximize our utility. Once we have generated all the moves for the current table, shown in line 3, we then go through each of them and ask them for their child node with the lowest score, since that is the path our opponent will choose in order to minimize our utility. This can be seen in line 5 and 31 of Code example 8.1. We then compare that node's score with our currently highest score, as can be seen in lines 6 and 32. If it is found, we then update our currently highest score with the one that was found. The same idea applies to the MinMax method in line 13, except that this time around, it is the lowest score that is we are searching for because, as mentioned before, our opponent wants to minimize our utility. This can be seen in line 19. We also ask for the score of the child with the highest utility instead of the lowest, since we are trying to maximize our utility. This can be seen in line 18. Whenever a leaf node is reached, the current node is evaluated and its score returned, which is done in line 14 and 27 where the engine checks to see if the current depth is zero.

The avid reader will have noticed that the initial values of scores are always set to the highest and the lowest possible values for an int. This is because otherwise the engine might risk not making a move at all. Imagine that it is now white to play, and he is in a horrendous position. No matter what move is chosen, the utility function will never be above 0. This means that if we used the default value for an int, which would be zero, and no move yields a higher utility value than zero, then no move will ever be set and the engine would then in turn return a 0 move. This was called internally for the A1A1 syndrome, because if no move was chosen, it would return a default move which was also set to zero. This means that it's from square and to square would also be zero. A1 to A1. The same idea applies when searching for the lowest possible value. If nothing is below zero, no move is chosen.

Therefore, the initial score value is set to be the maximum value for an int.

8.1.3 Negamax

Negamax is a search variation derived from minimax. This search algorithm relies heavily on the fact that $max(a, b) = -min(-a, -b)$. This means that it relies on the zero sum property of the game, in the maner that if a position has a heuristic value of x for white, then it is $-x$ for black. One of the main advantages of nega max is the fact that all code is implemented in a single function, in comparison to minimax which has 2 functions, one to get the max child and one to get the min child. Let us look at some pseudo code for nega max:

```
1
2  int negaMax( int depth, int color ) {
3      if ( depth == 0 ) return color * evaluate();
4      int max = -oo;
5      for all (moves)  {
6          score = -negaMax( depth - 1, -color );
7          if( score > max )
8              max = score;
9      }
10     return max;
11 }
```

Code example 8.2: Negamax psuedo code

As seen in code snippet 8.2, there is only one function instead of two to get the score of a node, regardless of who we are searching for. As mentioned before, that is because the max of two numbers is equal to the negated minimum between the same two numbers negated. Nega max is closely related to minmax, as both have fairly similar structures. In line 1, if we have reached our target depth, it is time to evaluate the current position and return that score to the callee. In line 3, we again set the best move as negative infinity, in order to ensure that a move is choosen. We then iterate through all possible moves for each position. This is where things start to differ from the standard minimax. We now assign a score to the negated score of its child in line 5. We then check to see whether or not that is the best move possible for that given position. If it is, we then update the max with that score and

continue iterating over all the possible moves.

The reason why negamax was not used in our engine was due to the fact that we had some major complications with our evaluation function, since there was some major confusion as to what score the evaluation had to return: should it return the score based on who is currently searching or should it evaluate the board based on who's turn it is to play, according the board. We continuously attempted to get negamax working, but in the end we felt that we were wasting too much time and decided that we would not use negamax, since it took us a very short time to implement a working minimax.

The correct way to use negamax though is to evaluate the board depending on whose turn it is to play on the board it is examining, and not for who is searching. This can be seen in the methods signature and how it is called. If you look closely at line 5, you see that we reverse the current color every time negamax is called in order to get the score of its child.

As mentioned before, there are a lot of moves that can be made. From the start position there are a total of 20 moves. From the second level, an opponent can respond with 20 different moves. That means that there are now 20×20 moves possible. It is now the white player's turn again. At this level things start to get a bit complicated since we have possibly released the queen or the bishop from the back rank. If the pawn at square E4 was removed, another 27 moves are possible for that position. Let us say that an average of 25 moves are possible for all prior moves, and the same thought applies for black. We now have $20 \times 20 \times 25 \times 25$ different positions for the first move. That's 250 000 positions! That's quite a few different positions we have to evaluate, and that's only for a depth of 4 for the first few moves. Imagine when all sliding pieces are unblocked. The time complexity of the minimax is $O(b^d)$ where b is the branching factor, which is the amount of legal moves in our case, and d is the depth we are running the algorithm at. [4, p.165] We can unfortunately not eliminate this exponent, however we can use a technique that effectively cuts it in half.
If we know which moves will be taken, it should be possible to trim the tree significantly. As an example, we reach a position where the we can take the black queen with our pawn. We also have the possibility of taking the queen with one of our knights. We can see that the queen is protected, so that once it is captured, our own piece will be captured as well. We first see that by taking out the queen with our pawn leads to us losing our pawn. This might give us the score of 10. If we take it with the knight, it gives us a score of 8. The engine has now found a move that is worse then the previous move, which was to capture the queen with the pawn, regardless of which move the

black player plays or uses to capture our piece. We are no longer interested in evaluating all his moves and can therefore prune the rest of the branches for that tree.

This technique is called alpha-beta pruning and we will discuss this in detail now.

8.1.4 Alpha-Beta Pruning

Alpha beta pruning is a technique used in order to trim branches, and cut search times so that we can traverse our tree faster, than if no branches were pruned. The general idea behind alpha beta, is to ignore branches that will yield lower, or higher results, than branches we have already visited. Alpha beta is implemented on top of our mini max algorithm. This can clearly be seen, when looking at some pseudo code for alpha beta(see 8.3), when only a single line is added as well, as the parameters for the methods.

```
1
2  AlphaMin(board, alpha, beta, depth){
3      if(depth = 0) { return Evaluation(board); }
4      GenerateMoves(board);
5      for all (moves){
6          result = AlphaMax(board(move), alpha, beta, depth -1);
7          if(result >= alpha) { return alpha; }
8          if(result < beta){
9              beta = result;
10         }
11     }
12     return beta;
13 }
14
15 AlphaMax(board, alpha, beta, depth){
16     if(depth = 0) { return Evaluation(board); }
17     GenerateMoves(board);
18     for all(moves) {
19         result = AlphaMin(board(move), alpha, beta, depth - 1);
20         if(result <= beta) { return beta; }
21         if(result > alpha){
22             alpha = result;
23         }
24     }
25    return alpha;
26 }
```

Code example 8.3: Alpha-Beta psuedo code

What are the alpha beta variables that now get passed around together with the board and depth? Alpha beta are variables that indicate the highest and the lowest score the engine has found until now. Alpha indicates the highest while beta indicates the lowest score we have found yet. This is used when we visit a node's children in order to cut branches that either give us lower or higher score than branches we have already visited. This is done in line 8 and 20. Remember that these methods are mutually recursive, that means that they call upon each other. AlphaMin is searching for the lowest score and at the same time setting the beta variable, which is the lowest score we have found yet. This can be seen in line 9. After that variable has been set, it then goes and calls AlphaMax for the next child in its child list, with the updated beta variable. We are now searching for the child with the highest score. Remember that our calling method, AlphaMin, has already found a suitable move to take with score x. This has been passed down to us through the beta variable. We now visit the first child and the returning score is $x + 1$ this score is higher than the previous move we found from our calling method , which is x. This means that whatever happens, we now have a move that has a higher score than what we previously found. This is where the alpha cut off happens, which can be seen in line 7. This is because our calling method, AlphaMin, is searching for the lowest possible score. We have just dived into a branch in which regardless of what happens, there is already a move which has a higher score than what we can achieve by following the other path. This is why we stop searching down the branch at this point: the score returned from our child node will always be higher than what we have already found, therefore we no longer need to search the other children since we already have a path that leads to better results(lower score), than this subtree we are currently searching in. The same idea applies to beta cut off, where we find a branch that has a lower score than a branch we have previously visited. Since another branch with a better score has been found, there is no need to keep searching since we know that if we do take that branch, the resulting move will be worse than the other branch we have found (it could be a lot worse, but we have found a move that the opponent will prefer to make over our own, leading us to a worse position than our previous branch).

Let us try to tie a concrete example.

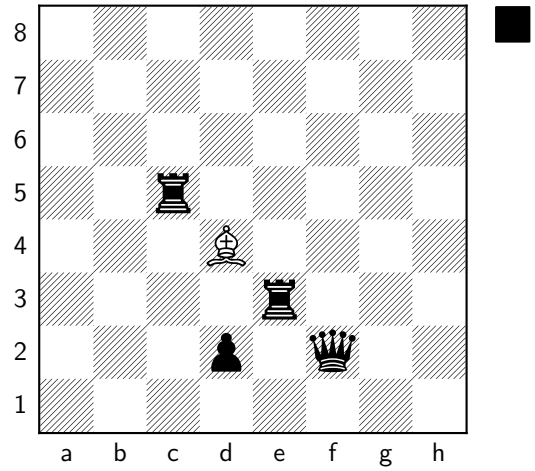
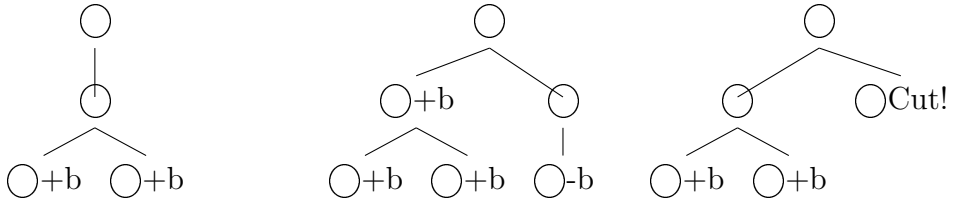


Figure 8.5: Alpha Beta example

In the example 8.5, it is white to move. White can either choose to capture the rook on square c5 or the rook on e3. Let us first try to explain this example in words before showing the game tree. We first look at what black can do if we capture the rook on c5. There are no responses from black that lets him capture our bishop. We now look at the responses from black if we capture the rook on e3. As soon as we capture the rook on e3, we see that he can capture our bishop with his queen. We immediately stop searching in that branch since white has found a better move previously, which was to capture a rook on c5 without losing anything. All other responses from black, including capturing the bishop with the queen do not matter to us, since we know that at least, he will capture our bishop with the pawn. That is, at worse, the worst response black can make. So we know that at least, he will capture our bishop. Therefore, that branch is now pruned since we are no longer interested in that move. Let us try to show a game tree to cement our explanation.



(a) d4c5 branch (b) d4e3 branch (c) Alpha cut-off

This is not a complete tree since it does not show all moves. Only in-

teresting moves are shown here, as well as the value of the leaf nodes only reflect whether or not we keep or lose the bishop. This have been done to simplify the example. The engine starts out by visiting all nodes for the d4c5 branch as seen in figure 8.6a. This gives the parent node a score of +b. Now that we are done visiting that branch, we continue on to the d4e3 branch, as can be seen in figure 8.6b. We go down the first response by black which would be d2e3, capturing the bishop. The evaluation function reflects that by returning score - b. Once we return from that position to the parent node, we can now see that that move, d4e3, is worse than our previous moves. We aren't interested in searching for that branch anymore, since we are able to refute that position by simply making another move, and therefore that branch gets cuts, which can be seen in figure 8.6c.

This also works the other way around. The opponent can also refute moves that would be to beneficial to us. This is called a Beta cut-off, in comparison to the previously mentioned Alpha cut-off.

In layman terms, Alpha-Beta pruning is about refuting positions that would be beneficial to the opponent by ensuring that moves that lead to the undesired position are pruned.

8.1.5 Quiescence Search

One of the problems in artificial intelligence is the horizon effect. This appears because in some games the number of possible game states are immense, and since there often is a time limit for the search, the portion of the states that the computer can search is limited. So a computer which only searches a limited part of the tree, has the possibility of finding a state that looks promising, while yet could prove to lead to an unfavorable state later in the game. But since it cannot search that far into the tree, it would not see the problem. With human players the problem becomes near null, since with intuition, a human have the possibility to look at a state, probably noticing what would be illogical to search, and as a result cut the branching factor to a minimum. But since computers does not have this intuition, we will have to simulate something similar. One of the ways to do this is called Quiescence Search. The idea is to cut the branching factor as much as possible, by only focusing on very interesting moves. This is however dangerous to do from the beginning, since there is a high possibility that we will cut branches, simply because the top node is uninteresting. Instead the Quiescence Search is called at the end of the normal search. Instead of just evaluating this leaf state in the tree, the Quiescence Search searches a bit further. This poses to

questions, how deep should the Quiescence Search go, and what defines an interesting move. These settings are different from one implementation to another. Some just searches one ply further down, while others keeps searching until there are no more interesting moves. In our implementation we will only search one ply down, since the speed of the calculation is quite important, and to keep searching will take too long. The other question is more interesting. What is an interesting move. It depends on both what knowledge you have, but also what it is you are trying to solve with the search. In our implementation we want to get a better idea on whether we are making proper captures, meaning captures where we gain something, and where we will not lose it the next round. This means that we want to primarily look at captures.

Through some of the position tests we ran, we discovered that if we did not search deep enough, we sometimes did not find moves that resulted in a promotion. Since adding a ply to the search depth would add to much time to the search, we changed the move generation for the quiescence search, to not only generate attack moves, but also generate promotions. This way, if we are one ply from finding the promotion, the quiescence search will find it, without adding too much time to the search.

Since we use minimax, the easiest way to use Quiescence Search together with the normal search is to make two functions, one for beta cutoffs and one for alpha cutoffs.

```
1  public int Quiescence_Search_Min(Board board, int alpha, int
       beta, Node parent, int currentDepth) {
2              MoveGenerator moveGen = new MoveGenerator(board,
                   true);
3              int score = Evaluation.Evaluate(board, SearchingFor)
                   ;
4              List<Move> moves = moveGen.MoveList;
5              if (moves.Count != 0 && currentDepth < TargetDepth +
                    2) {
6
7                  if (board.ToPlay != Color.Black) {
8                      if (board.BlackCheck) {
9                          score = AlphaBetaMax(board, alpha, beta,
                                currentDepth, parent, false, 4);
10                     }
11                 }
12                 else if (board.ToPlay != Color.White) {
13                     if (board.WhiteCheck) {
```

```
14                          score = AlphaBetaMax(board, alpha, beta,
                                currentDepth, parent, false, 4);
15                      }
16                  }
17
18                  if (score <= alpha) {
19                      return score;
20                  }
21
22                  if (score < beta)
23                      beta = score;
24
25                  foreach (Move move in moves) {
26                      if (move.GetCapture() != 0 || move.
                            GetPromotion() != 0) {
27                          board.MakeMove(move);
28                          score = Quiescence_Search_Max(board,
                                alpha, beta, parent, currentDepth +
                                1);
29                          board.UnMakeMove(move);
30                          if (score <= alpha) {
31                              return alpha;
32                          }
33                          if (score < beta) {
34                              beta = score;
35                          }
36                      }
37                  }
38                  return beta;
39              }
40              return score;
41          }
```

Code example 8.4: Quiescence Search Minimum

In the first couple of lines we instantiate the different variables we will need. Movegenerater with a boolean as second parameter, generates only capture moves and promotions, this is our definition of interesting moves. It does this by exchanging the normal target (excluding own pieces) with the opponents pieces. The differences is that the empty squares now is disregarded as targets. Then we need a base score, to see whether or not the new score is better. This is also used to make alpha cutoffs. In line 5 we check to see if there are any interesting moves and if we are done with the quiescence search. Since there is the possibility for an enemy move to put us in check or even mate, we now check for this, and since the escape from the check is not always a capture move we have to use the normal AlphaBeta search to get out of the situation. In line 18-20 we make the alpha cutoff,

and line 22-23 updates the beta value. Now we again check each move, to see if it is a capture or a promotion. The move is made and the search continued Quiescence_Search_Max. Quiescence_Search_Max is the same as Quiescence_Search_Min except where Min makes alpha cutoffs, Max makes beta cutoffs. When this returns we unmake the move to get the old board again. We check the score, make the cut or update the value, and continue to the next move. When we are done with all the capture- or promotion-moves, we return the best score we have found.

8.2 Evaluation Function

An evaluation function is more or less an internalization of a performance measure, where the performance measure is more general, and gives a score to a sequence of states. Our evaluation function on the other hand will deal with a single board state and will be called by the minimax algorithm and then return an evaluation value, which specifies the utility of a state, to the algorithm.

If we had infinite time (and hopefully also a supercomputer) our agent could just search out all possible configurations and find the best path to a winning terminal state. However since we do not, we will apply a heuristic evaluation function to states in a search, which will give an estimate of a position's utility. There is a number of properties that should be considered for a position, and what things should be encouraged or discouraged by our AI, and we have considered what those should be for our evaluation function. First of all we want to consider material value, which among other things is important for winning piece exchanges. Another to consider is center control, since controlling the center offers more mobility than being pushed into a corner. Mobility is also something we will consider. We also want to consider king safety to ensure the king is sheltered and in a safe position. Another thing to consider is good pawn structure, for instance it is generally considered bad to have multiple pawns on a file. Lastly we also want to consider passed pawns, since pawns close to getting promoted are worth more than those in start position. Our final evaluation score will represent a sum of these factors.

8.2.1 Material value

First of all the material value is very important to the value of a position. We will use a common assignment of point values using centipawns, which is commonly used in computer chess AI, where a pawn is worth 100 centipawns,

a queen 900, rook 500, knight 300, bishop 300 as seen in figure 8.6. The value of a king is not defined since it cannot be captured. The reason for using centipawns is to be able to evaluate strategic values of a position which can be worth less than a single pawn. One of these strategic values is the value of knights, which will be higher than a bishop in early to mid-game, since knights can jump over the many pieces in early game, whereas a bishop is easily blocked. The material value of a bishop is also higher when a player has a pair of them. This is because bishops use only one tile color and if a bishop is lost, an opponent can avoid the remaining bishop by preferring the opposite colored tiles. We apply material value evaluations to our bitboard by simply counting the amount of pieces in a given board configuration and apply the material values to them. [27] The summation function is $MV = w_1p_1 + w_2p_2 + \cdots + w_nf_n = \sum_{i=1}^{n} w_ip_i$ where n is the amount of pieces of a type, w_i is the value of the piece type as denoted in the table 8.6 and p_i is a piece.

Material	Values
Rook	500 cp
Knight	300 cp
Bishop	300 cp
Queen	900 cp
Pawn	100 cp

Figure 8.6

8.2.2 Center Control

It gets trickier when it comes to dealing in the strategic factors like center control, king safety, pawn structure etc. But we can make a table over the most strategically important positions for a piece to be in, with a value assigned to all locations on a board. This is called a piece-square table and is a fast scheme and good for quick evaluation. Then we can apply this table to the bitboards of pieces and the pieces will get extra points the better position they are in. For center control we can make tables which give extra points to pieces who helps center control. But first let us consider what control of the center is exactly. Center control is about placing pieces so that they attack or occupy

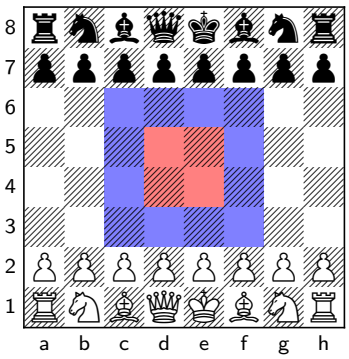


Figure 8.7

the central 4 squares of the board. This is important
because from the center, pieces can access most of the board, in other words they have more mobility when in the center. We have made our piece-square tables by giving a position 5 points for each of the 4 center squares it can attack which is the red area in figure 8.7, and twice the amount if they occupy it. Similarly a position also gets 1 point for each of the 12 squares marked by blue in figure 8.7 and twice the amount if they occupy it. This results in a point distribution as seen in listing 8.5 for the positions of knights.

```
1 private static readonly int[] Knight_CC = new int[64]
2  {0, 1,  1,  2,  2,  1, 1, 0,
3   1, 2,  6,  7,  7,  6, 2, 1,
4   1, 6, 14,  9,  9, 14, 6, 1,
5   2, 7,  9, 14, 14,  9, 7, 2,
6   2, 7,  9, 14, 14,  9, 7, 2,
7   1, 6, 14,  9,  9, 14, 6, 1,
8   1, 2,  6,  7,  7,  6, 2, 1,
9   0, 1,  1,  2,  2,  1, 1, 0};
```

Code example 8.5: Point distribution for Knights

This is used on knights, bishops, rooks, queens and pawns, however pawns is from whites perspective and reversed for black. We are however not evaluating the kings center control value since the king should not be used for center control. The total center control value is calculated as a sum of weighted factors like so: $CC = w_1c_1 + w_2c_2 + \cdots + w_nc_n = \sum_{i=1}^{n} w_ic_i$ where n is the amount of pieces, c_i is the calculated center control value for a single piece, remember we find center control value for one piece at a time, and w_i is a weight used for tuning.

8.2.3 Mobility

Although mobility overlaps a bit with center control as centered pieces obviously have more move choices (Knight for instance has 8 move choices in the middle and 2 in a corner), it is also a very important feature. Mobility is a measure of how many choices a player has in a given board configuration. The idea is that the more move choices you have, the stronger the players position is. In our evaluation we will consider the sum of possible legal

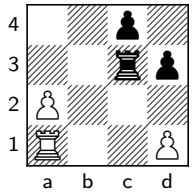


Figure 8.8

moves. So for instance in figure 8.8 the white rook
has 2 possible moves and black rook has 4. This is done by calling the move generators count of possible moves (including attack moves but not defend moves) for each piece type. Similarly to center control we also calculate the total sum of mobility as a sum of weighted linear factors like so: $Mobility = w_1m_1 + w_2m_2 + \cdots + w_nm_n = \sum_{i=1}^{n} w_im_i$ where n is the amount of piece types, m_i is the calculated mobility for a piece type, remember we find mobility for all pieces of a type one at a time, and w_i is a weight used for tuning.

8.2.4 King Safety

Another very important feature is king safety. King safety is about evaluating how safe the king is by considering the kings pawn shield(after castling) and the relative danger of nearby attacking pieces also known as king. We can check how well the king is protected by a pawn shield by looking at whether there is pawns in the red/blue marked areas in figure 8.9 and if the king is in the field marked by the dot. It is preferable if the pawns are on the same rank and take up either the blue or red fields so they form a line. If there are 3 pawns in the first(red) rank in front of the king then it is a lot better than if there is 2 pawns in front of the king and one in the next(blue) rank. This will be checked if the king is in either of the spots in the white corner shown in figure 8.10 and identically for the opposite white corner as well as for the two black corners but with the direction reversed obviously. The value is calculated by counting the amount of pawns in the red field and blue field, if there is 3 pawns in the red field then the pawn shield is good, if there is less than 3 pawns in the red field but 3 pawns in blue and red combined, then the pawn shield is open which is bad. Even worse however is if there are less than 3 pawns in both fields combined then the pawn shield is very open which is very bad.

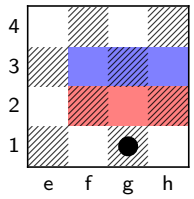


Figure 8.9

8.2.5 Pawn Structure

This is another important feature in our evaluation function. The important aspects of pawn structure is to discourage getting 2 or more pawns per file as well as getting 2 or more pawn islands. To discourage getting 2 or more pawns per file we simply check how many pawns there are on a file. To check how many pawn islands there are, we simply check if there are files without

pawns. To check how many pawns on a file we simply run through a loop checking each square on a file for a colors pawn and after running through an entire file we give points accordingly.

8.2.6 Passed Pawns

The last important feature in our evaluation function is passed pawns. We have implemented passed pawn in a simple form by just giving a bonus that increases as the pawn advances. The purpose of this feature is to encourage promotion, otherwise it would only be the material value feature that encourages promotion of pawns and it has the issue that it needs up to 12 in depth to see it. Another reason for this feature is to protect pawns that are close to being promoted, as well as seeing the danger of opponent pawns moving forward. We implement this feature simply by making a piece square table with increasing values in higher ranks, and these values are then given to the pawns occupying these squares.

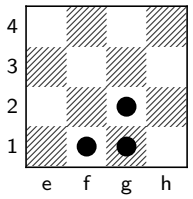


Figure 8.10

8.2.7 Combining the factors

Now since we have several features and since we can potentially add more features to make our evaluation better, we would like to have the total evaluation sum be calculated as a sum of weighted factors. This is also because it gives a better overview as well as the possibility to tune each factors and give them their appropriate influence on the total sum. Each color is evaluated using a summation function on the form of $Eval = w_1 f_1 + w_2 f_2 + \cdots + w_n f_n = \sum_{i=1}^{n} w_i f_i$ where each f_i is a feature, e.g. Mobility, and each w_i is a weight we apply to the feature. It will also have to accommodate the search algorithm which could be Minimax or Negamax. For Minimax we need to return a score that represents the color that is playing and how they are doing relative to the opponent. Thus it will be the playing color's total value subtracted by the opponent colors total value.

Chapter 9

Testing

There are several aspects of our chess engine that we need to test. We will need to tests our evaluation function's features and tune the weights on them to try to get the most optimal evaluation. Another thing we will test is how good our chess engine is at calculating the correct move for different board configurations. We also need to tests distributed parallelism in our chess AI and see how much it helps, and how much of a benefit each computer gives etc, where only 1 move is known to be the best. Lastly we need to find our chess engine's Elo rating through playing a tournament versus other engines.

9.1 Opponent Chess Engines

For our testing we needed to find some opponent chess engines that we can test against. We chose to start with using the weakest chess engines we could find and in case they were to easy we could easily go back and find some better ones to match our chess engine if needed. All the engines we could find was from a French site [28] who ran a UCI engine league which purpose is to compare engines especially when developing. We have listed the engines and their Elo rating in the table 9.1 . However we do not know exactly how they have tested the engines, although they do describe that they have started in one single specific opening for each tournament. Still we can't be sure if the Elo ratings of the engines represents exactly how well they play, for instance it could be that some engines lose to engines with far lower Elo rating. Also remember that it is not fair to compare chess engines Elo

Chess engines	
Name	Elo Rating
Ace	991
qutechess	1344
TREX	1440
Nanook	1443
eden	1506
Suff	1543
Predateur	1586

Figure 9.1

rating to human players Elo rating, since they play completely differently and against their own kind of rating to human players Elo rating, since they play completely differently and against their own kind.

9.2 Evaluation Feature Testing

As we discussed in section 8.2 we have weights which can be used to document and tune the relevance and impact of each feature, and individual pieces impact on a feature, on the total evaluation. To figure these factors out, we will have our chess engine run a lot of games against opponent chess engines at a fixed search depth, the matter of time and search speed is then not considered thus making the performance rely more on evaluation function, which we are testing. For this reason we also chose to remove the influence of opening books by making all chess engines in the test use the same opening book from Arena. We would also have liked to remove endgame tables completely from affecting the performance of any chess engine, however it is not possible with many engines so we will prefer using engines, which does not use them anyway, and engines, which we are able to beat anyway. These preferences led us to choose to use Predateur 0.1.5 and Trex. The reason for using only 2 is that we rather want a lot of games versus a few than less games against many engines, which we think will make the test result less random.

We will start with enabling one feature at a time starting with material value, and then run a session of games, and see how many wins, draws and losses we get out of 100 games vs each chess engine, thus making this a quantitative analysis rather than qualitative. Then we will enable another feature and see how much our engine's results improves. Once we have done this for all features, we will experiment with some different weights internally in some features if we think they could be better, for instance in material value we could say a queen is worth 10 pawns instead of 9, would improve the score, and likewise for the other features. This can be iterated a few times to make sure everything has the best values. We will do this at a depth of both 2 and 4, we would have liked to have deeper depths as well but it would take a lot more time.

Game results				
MV only	2 Halfmoves		4 Halfmoves	
Predateur	0-62-38	19%	1-88-11	7%
Trex	3-49-11	27%	6-55-39	26%
MV and CC	2 Halfmoves		4 Halfmoves	
Predateur	8-63-29	23%	1-82-17	10%
Trex	23-46-37	42%	9-48-43	31%
MV, CC and Mo	2 Halfmoves		4 Halfmoves	
Predateur	15-56-29	30%	5-66-29	20%
Trex	21-41-38	40%	11-29-60	41%
MV, CC, KS and PS	2 Halfmoves		4 Halfmoves	
Predateur	13-53-34	30%	5-68-27	19%
Trex	35-29-36	53%	12-37-51	38%
MV, CC KS, PS and PP	2 Halfmoves		4 Halfmoves	
Predateur	14-52-34	31%	5-76-19	15%
Trex	30-26-44	52%	19-35-46	42%

Figure 9.2

We have compiled the results from the testing, in the table 9.2. We can see that there is a clear improvement between running material value(MV) only and running it, with center control (CC). While supervising the test we noted that with MV only, especially at 2 depth, the engine often chose a move and then reversed the move several times. This was also even more evident in preliminary test where we put our engine against itself with only MV enabled, no opening book and low depths, which resulted in some very boring matches. Enabling Mobility (Mo) also seemed to improve the evaluation score overall except for 2 depth vs trex where it did slightly worse. It is however evident that improvement gets less clear the more features we have enabled. When enabling King safety (KS) and Pawn structure (PS) we do not get any improvement at all, and it might even have worsened the evaluation function. We can however not be certain so we will retest KS and PS and give them more influence on the total evaluation. Similarly we seem to have the same issue when enabling Passed pawn (PP), and we will retest this again where PP has more influence on the total evaluation score.

Now looking internally in material value we have noticed that there actually is a big difference in what material values that different chess engines use. Currently we use the basic rule of thumb and easy to remember 1,3,3,5,9 values for pawn, bishop, knight, rook and queen respectively. However we have chosen to test if Larry Kaufman's(Grand Master and chess engine re-

searcher) [29] material values would improve our evaluation function. He concluded that the best values for human players is 1, 3.5, 3.5, 5.25, 10 for pawn, bishop, knight, rook and queen respectively, and half a pawn value for a bishop pair.

Game results 2. Iteration				
New MV values	2 Halfmoves		4 Halfmoves	
Predateur	13-55-32	29%	5-70-25	18%
Trex	30-27-43	52%	32-18-50	43%
Higher PS/KS weight	2 Halfmoves		4 Halfmoves	
Predateur	12-56-32	28%	11-72-17	20%
Trex	29-36-35	47%	17-29-54	44%
Higher PP weight	2 Halfmoves		4 Halfmoves	
Predateur	11-59-30	26%	6-61-33	23%
Trex	34-18-48	58%	31-29-40	51%

Figure 9.3

We ran a second iteration of tests where we first tried the new material values, then increased the weight on PS and KS so they had a bigger influence and tested it, and similarly we did the same with PP. We found out that the new MV values, as evident in the table 9.3, did not make a huge difference and we cannot conclude it is better from this test. We do however think it most likely that Larry Kaufman's MV values should be better than the rule of thumb ones, so we decided to keep them. As for the higher PS/KS weight result we are uncertain whether it has made a positive effect if any, however it seems that it proved to be stronger against predateur on 4 halfmoves depth giving over twice the regular amount of wins. Lastly, the higher PP weight seemed to give overall positive results despite 2 depth predateur being a bit lower.

Overall for this test we would like to conclude that it is hard to judge what effect our feature changes have made. We think we would need to run a larger set of test for each feature change, e.g. 300 instead of 100, but an issue we would have had with larger tests would be time constraints, since 100 games at 4 depth already takes almost 2 hours. In retrospect we could have focused more on the depth 2 games since they are on average 4-5 times faster to run, so we could have gotten 5 times as many games to judge from. It would also be better if we had time or speed to search deeper, but currently our engine would take at least 12 hours for 100 games if we searched in depth 6. We also think that our tests could have been better if we had played the same 50

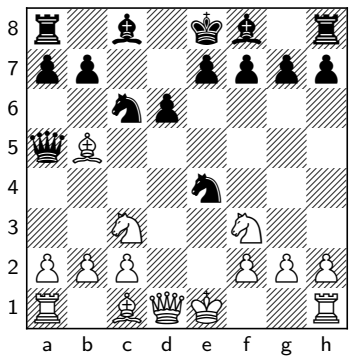
openings twice(once as white and once as black), but we realized this feature was available to us late in the testing. However after some 2400 chess games, it is quite clear that the last configuration where we put the PP weight up, is obviously a lot better than having only MV enabled.

9.2.1 Evaluation of certain positions

We also wanted to see how the engine evaluates certain positions and how that changes by adding/removing features. In figure (a) you can see the values for our engines evaluation of the position (b). Remember that this is after the weights have applied, so for instance we have a weight of 10 on material value so a pawn is 1000 points. You can see that Black is ahead 1 pawn which makes up the most of his lead in score.

Evaluation Score		
Feature	White	Black
Material	41000	42000
Mobility	337	270
Center Control	172	204
King Safety	0	0
Passed Pawn	60	80
Pawn Structure	-40	-20
Total	41529	42534
Current player(B)		1005

(a)

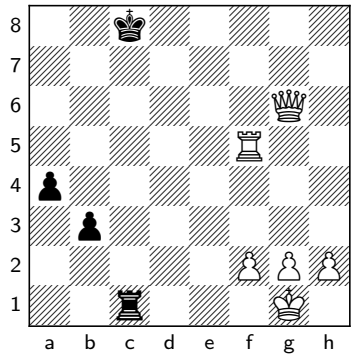


(b)

In figure (c), which displays the values for figure (d), you can see that although white has a lot more score, mostly from material value, it is still set to -200,000 because it is checkmated. White also has a king safety bonus since it is behind a pawn shield, although in this case it doesn't help at all. Black also has a relatively big passed pawn bonus since its pawns are not far from being able to be promoted, which makes them worth more.

Evaluation Score		
Feature	White	Black
Material	18250	7250
Mobility	11	145
Center Control	92	36
King Safety	20	10
Passed Pawn	30	220
Pawn Structure	-100	-100
Total	18303	7561
Current player(W)		-200000

(c)



(d)

9.3 Determining our Elo Rating

As we discussed in section 2.6 we can find our Elo rating by playing a number of games versus Elo rated opponents. We intend to put our computer against other chess engines in a round robin tournament and see how many wins, losses and draws we get and calculate our Elo rating from these results. We will then use the number of wins, draws and losses, the elo rating, and tournament placing as a performance measure. We have also considered putting our engine against human players, however we think it is better for us to put it against other engines since they are more reliable and gives us more options. These options will help us test the real strength of our engine, since we can disable opening books and jump to middle game, where we can really test our engine. We will also need to either disable endgame tables for the opponent chess engine, since when we get into endgame then the tables will tell the engine the outcome of the game if both play optimally, so basically the game would be over since there is a finite number of possibilities left. Another thing to take into account is the openingbook. To have a strong beginning is important in chess, but since this is not implemented in our engine, this could give the opponent an unfair advantage. To counter this, arena have an opening book that you can force the engines to use, even if they have their own. With this enabled we ran a tournament with our engine and 6 other engines, on a single computer, with a depth 6 and a time limit of 15 minutes, to get a benchmark of how it would improve to use the distributed system. The other engines were Ace, Eden, Nanook, Predateur, Qutechess and Trex. 9.1

Results from 1. tournament			
Opponent	Won	Lost	Draw
Ace	2	0	0
Eden	0	2	0
Nanook	0	0	2
Predateur	0	2	0
Qutechess	0	1	1
Trex	1	1	0

To calculate the initial rating we use the FIDE Initial Rating Calculator [30]. Since our opponents have an average rating of 1385 and we scored 4.5 out of 12, we get an initial rating of 1298. While the tournament was playing, we could see the other engines input to arena, and we could see that they searched quite a bit deeper than us. Where we searched in a depth of 6, some of the engines got to over a depth of 20. This is most likely the deciding factor of who won, and since our search was too slow to search deeper within the time limit, they won most of the time.

The second tournament was meant to show whether or not it helped if we used several computers, and with the added computing power, searched deeper in the tree. With the added depth we got the result:

Results from 2. tournament			
Opponent	Won	Lost	Draw
Ace	2	0	0
Eden	1	1	0
Nanook	1	1	0
Predateur	0	2	0
Qutechess	0	1	1
Trex	0	2	0

The test was done at a depth of 8 and we had 8 slave computers. The master computer made the initial tree with a depth of 4, and then the notes were distributed and searched again in a depth of 4. If we compare the 2 results, we see that almost all the games resulted in the same, and overall we got the same score, which gives us the same Elo rating as before. The only difference was in the games versus Eden and Trex. Versus Eden with an elo rating of 1506 we first lost both games, but then in the second tournament we only lost one, but won the other. Versus Trex it was the other way. First we were tied, with one loss and one win, but in the second tournament we

lost both games.
The conclusion we can draw from this, is that the problem with searching deeper is that if it takes to long, the timer will run out and the tree cut. Then if a result was not found by the search, the time used to search was wasted, and the tree is not getting updated. So instead of giving the result of a search at depth 8, we will only get the result of the initial tree, at a depth of 4.
To see if we could improve on the search without adding computers or change the algorithms, we tried to change how we searched. Where in the earlier test we searched in a depth of 8, with an initial tree of depth 4 and slaves searching each note in a depth of 4, we would try to make an initial tree of depth 6 and let the slaves search in a depth of 2. Our reasoning was composed of 2 things. First, as we knew there would be a lot more nodes, but since we started sending nodes to the slaves as soon as we found the first, this would not be as big a problem. Another difference was that even though we say that in the earlier test, the slaves searched in a depth of 4, when we add quiescence search 8.1.5 the search get an additional ply do search, even though it is not the complete ply, but only the capture moves.

Results from 3. tournament			
Opponent	Won	Lost	Draw
Ace	2	0	0
Eden	0	2	0
Nanook	1	0	1
Predateur	0	2	0
Qutechess	0	2	0
Trex	0	2	0

In our 3. tournament the score went a bit downhill despite high hopes. We had seemingly had some issues with the results that our slaves report back. We do not really have any way to confirm it, but suddenly our engine were making choices we had never seen before and which should not be part of its logic. It is hard to troubleshoot this as it takes a lot of time dealing with this many computers. We however concluded that to avoid this issue we need to make sure that every slave is installed correctly and identically. We calculated our initial rating Elo to be 1227 for this tournament.

Chapter 10

Discussion

We have done research and constructed a chess engine with a search algorithm and distributed the workload of that algorithm, so that the chess engine can play at a decent level. We do now have a finished project, however there is still a lot that could be done but due to the time limitations this will be noted as "future work". Some of this work will be described in this chapter and we will discuss why we did not implement it.

10.1 Transposition Tables

As mentioned in 5.4, transposition tables can speed up the search by a great factor, since transposition tables are very memory- and computing-friendly. We did not really know about this way of representing the board states, before we were about half way through the project period and due to the implementation time of a good transposition table, we did not use transposition tables for more than checking for threefold repetitions.

10.2 Opening Book

Even though we have implemented an opening book for our engine, we quickly realized that it was a really weak opening book. We did not have a tree like we had when we generated the moves in the move generator. Later on we then try to load the book into a tree similar to the tree we generated in the move generator, but this was not a big success since we ended out having huge memory leaks. Once we realized that the chess interface Arena could provide engines with their own opening book, we decided to lay our focus on more critical parts of the engine such as a evaluation function, and removed

the opening book since this was not a part of any of the main goals in this project.

10.3 End Game Tables

At this time our engine is very weak in the end game. This is mainly due to the fact that we do not have end game tables. We do however change the piece square tables to have a minor advantage in endgame, but an end game table could really improve the overall end game a lot, rather than just using the engine in the end game.

10.4 Human like Behaviors

In the very beginning of our project we talked about not having a chess engine that always found the best move, but rather having an engine that made interesting and human like moves. We wanted to give the chess engine a human aspect to it, not just "search" and find the best possible solution, but instead to take some risks. This way of playing against our engine would cause a rather dynamical gameplay and rarely end out in "boring" moves. However, after discussing this topic we decided not to get in over our heads, since making an human like chess engine would require a large amount of time, that simply was not available to us, especially given the fact that this was our first attempt at a chess engine in general.

10.5 Human opponents

We would also have liked to have played tournaments or at least elo rated matches against human players so we could get an idea of how well our engine performs against to humans. This however was also not done, due to lack of time.

10.6 Improved Evaluation Function

The evaluation function is very important to a chess engine. However a big part of writing a good evaluation function is about understanding chess, its theories and features. The purpose of this report however is not for us to get good at playing chess, but building an engine that is. There is however still a lot that could be improved in the evaluation function in order to improve

performance of our chess engine. It would also be beneficial if the evaluation function was able to mutate its weights dynamicaly according to the current phase in the game. For example if we are currently in the early to mid game, having the king safely tucked away in a board corner surrounded by pieces is a very good thing. The same position however is not good in the late game where fewer pieces are available. You want the king in an open spot with opportunities to escape.

10.7 Time distribution

A big issue whenever we played chess games with time constraints, was the way we distributed our engines calculation time. An optimal time distribution would be to spend less time and to search smaller depths when in early and endgame, and prioritize most of the time for middle game and search deepest there. This could be done by analysing the opponents move and allocating time depending on how strong the move was. This could be accomplished by some form of bayesian network.

10.8 Multi-core Support

A way to speed up the search for the best move even more, would be to run multiple calculations at the same time. Since most newer computers have at least 2 processors this could speed the calculation by 100%. We tried to look into this at one point but when implementing multi-core support this resulted in a lot of technical problems with distributing the system and gave us many race conditions that we were not able to solve easily.

10.9 Improved Search Algorithm

When looking back in our project, we would really like to improve upon our current search algorithm by implementing Iterative Deepening. This would greatly improve the engine by giving it a much stronger move ordering and more balanced move scores. At the moment, once the time runs out, the target depth is merely reduced by two which greatly speeds up the rest of the search, and enabling us to complete the search of the tree with a depth of $d - 2$. This is not an optimal solution though since we there might be moves just above the chess engine's horizon on the part of the tree that had its depth pruned. While it is true that the horizon effect is always present, we feel that by using iterative deepening, we could implement a much better

time schedueler for our search. We could for example tell the engine that once it has x time left, it should turn on quiescent search and continue searching until the time has been elapsed. Once it has, stop searching, find the best move and return that score.

10.10 Improved Distributed System

Once we implement iterative deepening search on our chess engine, we will have a really strong move ordering in our tree. This will greatly increase the power of our distributed search algorithm. This will ensure that the first nodes we visit will be the most interesting ones and the first nodes that get searched by our slaves. At the moment, since there is no move ordering, it is quite likely that the nodes we initially search will be outperformed and cut off by other moves we discover later in our tree. That effect is reduced when using iterative deepening since it forces the most interesting moves to be searched first. This would be by far the largest improvement in our distributed system.

10.11 Slave Workload Management

We would have liked to have better workload management for our slaves. We would like to have had some good heuristics on how powerful a computer is, to be able to allocate important nodes to the powerful slaves and in likewise avoid giving important nodes to unstable or slow slaves. This could probably also be achieved with some form of bayesian network that recieves a computer's hardware information and gives it a priority or increases/decreases its list of nodes to work on.

10.12 More Tests

We would like to have run several more tests on our engine. The sample size was too small to conclusively determine our engine's elo rating. We would also have liked to run more tests in order to be completely sure how our evaluation function weights effect our engine's search. While we have done some testing, we feel that spending more time in testing would have been beneficial. We would also have liked to perform search speed tests with limited depth. For example how long does it take our engine compared to others to find a solution to a chess problem in a limited depth scenario.

10.13 Security and Stability

Finally we would have liked to make our slave service more stable and secure. At the moment, there is no control as to whether or not the engine is doing what we are asking it to do. If anyone got ahold of our service interface, they would be able to create a program that posed as a slave but in reality returned bogus results to the master. We would like to avoid that as well. We would also like to avoid situations where slaves are not all updated to the latest version and might be running with different evaluation weights than our master. By implementing some kind of authentication in our engine, we would be able to ensure that it is not only up to date, but that the service is what it says it is.

Chapter 11

Conclusion

Our goal in this project, as stated in the problem statement (3.2), was to create a chess engine that could beat a human player, and that would have an ELO rating around 1300-1400. In addition to this we wanted to use a distributed system to distribute the workload among several computers.

We have created a board representation(5.3), that makes us able to generate the possible moves for all the different pieces on a chess board. It also gives us the possibility to evaluate the board (8.2) and see who has the advantage at that point. With this we can generate a tree of nodes that can be searched, in order to find the best move.

We also got a distributed system consisting of a master and a set of slaves working(7), where we create an initial tree, and distribute its leaf nodes to the slaves. They then calculate the nodes and returns the result. The master then updates the initial tree with the new scores in order to finally find the best move from that tree. We see this as a satisfying result although it could be improved upon.(10.8)

Lastly we will look at the end result and evaluate the entire engine. To do this we use ELO rating(2.6) to show how well our engine plays compared to other engines. We have tested our engine in 3 tournaments against 6 other engines.(9.3) This has given us an idea of our ELO rating. As you can see in the chapter we found 3 different ELO ratings. This was because we could not accuratly determine our rating. This is partly because we had some problems with our engine in the tournaments, but also because our rating is based on how the engine is used: local computation or distributed computation. Depending on how many slaves you have and how powerful they are, the engine would be able to search deeper in shorter time, and therefore get a higher rating.

Bibliography

[1] Wikipedia history of chess. http://en.wikipedia.org/wiki/History_of_chess.

[2] John's chess playground. http://homepages.cwi.nl/ tromp/chess/chess.html.

[3] David L. Poole and Alan K. Mackworth. *Artificial Intelligence, Foundations of computational agents*. Cambridge University Press, 2010, 2011.

[4] Stuart J. Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach*. Pearson Education, Inc., third edition, 2010, 2003, 1995.

[5] FIDE. Fide ratings change calculator. http://ratings.fide.com/calculator_rtd.phtml.

[6] Crafty chess engine. http://www.craftychess.com/.

[7] Fritz chess engine. http://www.chessbase.com/.

[8] Rybka chess engine. http://www.rybkachess.com/.

[9] Winboard and xboard chess interface. http://www.gnu.org/software/xboard/.

[10] Fritz chess engine. http://www.chessbase.com/.

[11] Arena chess interface. http://www.playwitharena.com/.

[12] Stefan-Meyer Kahlen. The uci protocol. http://wbec-ridderkerk.nl/html/UCIProtocol.html.

[13] Wikipedia. Forsyth-edwards notation. http://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation.

[14] chess guide.fateback.com. Universal chess interface. http://chess-guide.fateback.com/computing/universal_chess_interface.html.

[15] Arena. Arena website. http://www.playwitharena.com/.

[16] Peter Kankowski. What your compiler can do for you. http://www.strchr.com/what_your_compiler_can_do_for_you.

[17] David R. O'Hallaron Randal E. Bryant. *Computer Systems, A Programmer's Perspective*. Pearson, 2011.

[18] Robert Hyatt. Chess program board representations. http://www.cis.uab.edu/hyatt/boardrep.html.

[19] Transposition tables. http://mediocrechess.varten.org/guides/transpositiontables.html.

[20] Zobrist keys. http://mediocrechess.varten.org/guides/zobristkeys.html.

[21] Chess Programming Wiki. Bit scan. http://chessprogramming.wikispaces.com/BitScan.

[22] Stef Luijten. How to create a chess engine in 99 steps. http://www.sluijten.com/winglet/, 2011.

[23] Mark G. Brockington and Jonathan Schaeffer. The aphid ab search algorithm, 2008.

[24] Chess Poster. Did you know..? http://www.chess-poster.com/english/notes_and_facts/did_you_know.htm, 2000.

[25] Wikipedia. Fools mate. http://en.wikipedia.org/wiki/Fool's_Mate.

[26] Mike Goodrich. Mixed strategies and minimax. http://students.cs.byu.edu/ cs670ta/Lectures/Minimax2.html.

[27] Chess programming wiki space piece-square tables. http://chessprogramming.wikispaces.com/Piece-Square+Tables.

[28] Le Fou numérique. Uci engines ligue (uel). http://lefouduroi.pagesperso-orange.fr/tournois/uci/uel.html.

[29] Larry Kaufman. The evaluation of material imbalances. http://home.comcast.net/ danheisman/Articles/evaluation_of_material_imbalance.html.

[30] FIDE. Fide initial rating calculatorr. http://ratings.fide.com/calculator_rp.phtml.

Chapter 12

Appendix

Move Generator Complexity

```
1
2   ulong target = 0;                                                          1
3   if (CaptureMode) target = board.GetBlackPieces;                            1
4   else target = ~board.GetWhitePieces;                                       1
5
6   Move move = new Move();                                                    1
7   move.SetPiece((byte)Piece.WHITEKNIGHT);                                    1
8   ulong tPiece = board.WHITE_KNIGHT;                                         1
9
10  while (tPiece != 0) {                                                      10 (knights)
11    uint from = Constants.GetFirstBit(tPiece);                               1
12    move.SetFromSq(from);                                                    1
13    ulong tMove = target & PieceMoves.KnightAttacks[from];                   1
14    while (tMove != 0) {                                                     8 (moves)
15      uint to = Constants.GetFirstBit(tMove);                                1
16      move.SetToSq(to);                                                      1
17      move.SetCapture(board.squares[to]);                                    1
18      if (!Check(move, Color.White)) {                                       1
19        whiteKnightMoves.Add(move);                                          1
20      }
21      tMove ^= PieceMoves.LonePiece(to);                                     1
22    }
23    tPiece ^= PieceMoves.LonePiece(from);                                    1
24  }
25                                                                             10 * 8
26                                                                             knights * moves
```

```
1   Move move = new Move();                                                    1
2   move.SetPiece((byte)Piece.WHITEPAWN);                                      1
3   ulong tPiece = board.WHITE_PAWNS;                                          1
4   while (tPiece != 0) {                                                      8 (pawns)
```

```
5       uint from = Constants.GetFirstBit(tPiece);                                  1
6       move.SetFromSq(from);                                                       1
7       ulong tMove = 0;                                                            1
8       if (!CaptureMode) {                                                         1
9         tMove = PieceMoves.WhitePawnMoves[from] & ~board.GetOccupied;             1
10        if ((Constants.GetRank(Convert.ToInt32(from)) == 2) && (tMove != 0)) {    1
11          tMove |= (PieceMoves.WhitePawnDoubleMoves[from] & ~board.GetOccupied);  1
12        }
13      }
14      tMove |= PieceMoves.WhitePawnAttacks[from] & board.GetBlackPieces;          1
15      while (tMove != 0) {                                                        3 (moves)
16        uint to = Constants.GetFirstBit(tMove);                                   1
17        move.SetToSq(to);                                                         1
18        move.SetCapture(board.squares[to]);                                       1
19
20        if (!Check(move, Color.White)) {                                          1
21          if (Constants.GetRank(Convert.ToInt32(to)) == 8) {                      1
22            move.SetPromotion((byte)Piece.WHITEQUEEN); whitePawnMoves.Add(move);  1
23            move.SetPromotion((byte)Piece.WHITEKNIGHT); whitePawnMoves.Add(move); 1
24            move.SetPromotion((byte)Piece.WHITEROOK); whitePawnMoves.Add(move);   1
25            move.SetPromotion((byte)Piece.WHITEBISHOP); whitePawnMoves.Add(move); 1
26          }
27          else {
28            whitePawnMoves.Add(move);                                             1
29          }
30        }
31        tMove ^= PieceMoves.LonePiece(to);                                        1
32      }
33      if (board.EpSquare != 0) {                                                  1
34        if ((PieceMoves.WhitePawnAttacks[from]
35          & PieceMoves.LonePiece(board.EpSquare)) != 0) {                         1
36          move.SetPromotion((byte)Piece.WHITEPAWN);                               1
```

```
37            move.SetCapture((byte)Piece.BLACKPAWN);                    1
38            move.SetToSq(Convert.ToUInt32(board.EpSquare));            1
39
40            if (!Check(move, Color.White)) {                           1
41              whitePawnMoves.Add(move);                                1
42            }
43          }
44        }
45        tPiece ^= PieceMoves.LonePiece(from);                          1
46        move.SetPromotion((byte)Piece.EMPTY);                          1
47      }                                                                8 * 3
48                                                                       pawns * moves
```

```
1   Move move = new Move();                                              1
2     ulong target = ~board.GetWhitePieces;                              1
3     ulong tPiece = board.WHITE_KING;                                   1
4     move.SetPiece((byte)Piece.WHITEKING);                              1
5     while (tPiece != 0) {                                              1 (kings)
6       uint from = Constants.GetFirstBit(tPiece);                       1
7       move.SetFromSq(from);                                            1
8       ulong tMove = PieceMoves.KingAttacks[from] & target;             1
9       while (tMove != 0) {                                             8 (moves)
10        uint to = Constants.GetFirstBit(tMove);                        1
11        move.SetToSq(to);                                              1
12        move.SetCapture(board.squares[to]);                            1
13
14        if (!Check(move, Color.White)) {                               1
15          whiteKingMoves.Add(move);                                    1
16        }
17        tMove ^= PieceMoves.LonePiece(to);                             1
18      }
19      tPiece ^= PieceMoves.LonePiece(from);                            1
```

```
20    }                                                                    1 * 8
21
22    if (board.whiteKingCastle) {                                         1
23      if ((board.GetOccupied & whiteKSMask) == 0) {                      1
24        move.SetToSq(6);                                                 1
25        move.SetCapture(0);                                              1
26        move.SetPromotion((byte)Piece.WHITEKING);                        1
27
28        if (!board.IsSquareUnderAttack(4, Color.White) &&
29        !board.IsSquareUnderAttack(5, Color.White)) {                    1
30          if (!Check(move, Color.Black)) {                               1
31            whiteKingMoves.Add(move);                                    1
32          }
33        }
34      }
35    }
36    if (board.whiteQueenCastle) {                                        1
37      if ((board.GetOccupied & whiteQSMask) == 0) {                      1
38        move.SetToSq(2);                                                 1
39        move.SetCapture(0);                                              1
40        move.SetPromotion((byte)Piece.WHITEKING);                        1
41        if (!board.IsSquareUnderAttack(4, Color.White) &&
42         !board.IsSquareUnderAttack(3, Color.White)) {                   1
43          if (!Check(move, Color.Black)) {                               1
44            whiteKingMoves.Add(move);                                    1
45          }
46        }
47      }
48    }                                                                    kings * moves
```

```
1     Move move = new Move();                                              1
2     ulong target = 0;                                                    1
```

```
3   if (CaptureMode) target = board.GetBlackPieces;                                  1
4   else target = ~board.GetWhitePieces;                                             1
5
6   ulong tPiece = board.WHITE_BISHOP;                                               1
7   move.SetPiece((byte)Piece.WHITEBISHOP);                                          1
8   while (tPiece != 0) {                                                            10 (bishops)
9     uint from = Constants.GetFirstBit(tPiece);                                     1
10    move.SetFromSq(from);                                                          1
11    ulong tMove = PieceMoves.BishopMoves(Convert.ToInt32(from), target, board);    1
12    while (tMove != 0) {                                                           13 (moves)
13      uint to = Constants.GetFirstBit(tMove);                                      1
14      move.SetToSq(to);                                                            1
15      move.SetCapture(board.squares[to]);                                          1
16
17      if (!Check(move, Color.White)) {                                             1
18        whiteBishopMoves.Add(move);                                                1
19      }
20
21      tMove ^= PieceMoves.LonePiece(to);                                           1
22    }
23    tPiece ^= PieceMoves.LonePiece(from);                                          1
24  }                                                                                10 * 13
25                                                                                   bishops * moves
```

```
1
2   ulong target = 0;                                                                1
3   if (CaptureMode) target = board.GetBlackPieces;                                  1
4   else target = ~board.GetWhitePieces;                                             1
5
6   Move move = new Move();                                                          1
7   move.SetPiece((byte)Piece.WHITEQUEEN);                                           1
8   ulong tPiece = board.WHITE_QUEEN;                                                1
```

```
9
10    while (tPiece != 0) {                                                    9 (queens)
11      uint from = Constants.GetFirstBit(tPiece);                             1
12      move.SetFromSq(from);                                                  1
13      ulong tMove = PieceMoves.QueenMoves(Convert.ToInt32(from), target, board);  1
14      while (tMove != 0) {                                                   27 (moves)
15        uint to = Constants.GetFirstBit(tMove);                              1
16        move.SetToSq(to);                                                    1
17        move.SetCapture(board.squares[to]);                                  1
18
19        if (!Check(move, Color.White)) {                                     1
20          whiteQueenMoves.Add(move);                                         1
21        }
22
23        tMove ^= PieceMoves.LonePiece(to);                                   1
24      }
25      tPiece ^= PieceMoves.LonePiece(from);                                  1
26    }                                                                        9 * 27
27                                                                             queens * moves
```

```
1  ulong target = 0;                                                          1
2    if (CaptureMode) target = board.GetBlackPieces;                          1
3    else target = ~board.GetWhitePieces;                                     1
4    Move move = new Move();                                                  1
5    move.SetPiece((byte)Piece.WHITEROOK);                                    1
6    ulong tPiece = board.WHITE_ROOK;                                         1
7
8    while (tPiece != 0) {                                                    10 (rooks)
9      uint from = Constants.GetFirstBit(tPiece);                             1
10     move.SetFromSq(from);                                                  1
11     ulong tMove = PieceMoves.RookMoves(Convert.ToInt32(from), target, board);  1
12     while (tMove != 0) {                                                   14 (moves)
```

```
13          uint to = Constants.GetFirstBit(tMove);                    1
14          move.SetToSq(to);                                          1
15          move.SetCapture(board.squares[to]);                        1
16
17          if (!Check(move, Color.White)) {                           1
18            whiteRookMoves.Add(move);                                1
19          }
20          tMove ^= PieceMoves.LonePiece(to);                         1
21        }
22        tPiece ^= PieceMoves.LonePiece(from);                        1
23      }                                                              10 * 14
24                                                                     rooks * moves
```

Tournament Tech Info and Results

[Settings]
TournamentName=Test
Type=0
CurrSwissRound=0
Rounds=2
CurrentPair=21
CurrentRound=2
Hardware=Intel(R) Core(TM) i5 CPU M 460 @ 2.53GHz 2527 MHz with 3,9 GB Memory
OS=Windows 7 Ultimate Professional Service Pack 1 (Build 7601) 64 bit
ArenaVersion=Arena 3.0
LastGameDate=2012.12.19
LastGameTime=17:34:06
[PairingResults]
ChessEngine —— Nanook=1½
Eden-0013-jet-ja —— Trex=½1
ACE —— Qutechess=00
Eden-0013-jet-ja —— Predateur 0.1.5=11
ChessEngine —— Qutechess=00
ACE —— Trex=00
Nanook —— Qutechess=11
ACE —— Predateur 0.1.5=00
ChessEngine —— Trex=00
ACE —— Eden-0013-jet-ja=00
Nanook —— Trex=1½
ChessEngine —— Predateur 0.1.5=00
Qutechess —— Trex=1½
ChessEngine —— Eden-0013-jet-ja=00
Nanook —— Predateur 0.1.5=0½
ACE —— ChessEngine=00
Predateur 0.1.5 —— Qutechess=1½
Eden-0013-jet-ja —— Nanook=00
Predateur 0.1.5 —— Trex=1½
ACE —— Nanook=½0
Eden-0013-jet-ja —— Qutechess=11
[Participants]
Engine1=ChessEngine
Engine2=Trex
Engine3=ACE

Engine4=Qutechess
Engine5=Eden-0013-jet-ja
Engine6=Nanook
Engine7=Predateur 0.1.5

Test positions

We got some position test, to find if our engine would find the right move, from a certain game state. We tested these on different computers, but only a single computer at the time, meaning we didn't use the distribution of nodes, but this shouldn't matter since it is dependent on the evaluation and search, not the speed. We avoided the use of time limitation. This doesn't mean we ran all the test without it, but we will get to that. We got 8 tests, and to get the right move, we used Rybka 2.2 64-bit to run the test. We knew who should win, and on almost all of them, Rybka found a series of moves that lead to the right conclusion. In 2 of them 8,9 Rybka never finished, but we decided that for the moment, six test were enough and we would get back to the last two when the rest were done.

Zero

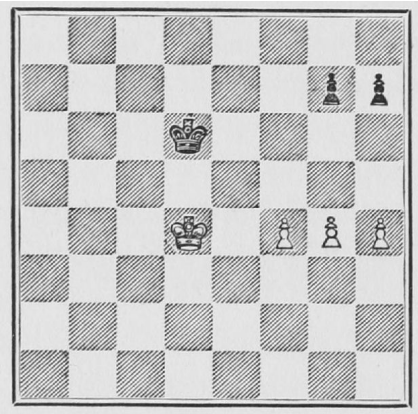


Figure 12.1: Position test ZERO

First we played Rybka vs Rybka to get the right move for both sides.
In the first test figure (12.1) This gave the result:

$g4g5 - g7g6 - d4e4 - d6e6 - h4h5 - e6d6 - h5hg6 - h7g6 - f4f5 - d6r7 -$ *Black resigns*

When we run the game with our engine in a depth of 6 we get:

$f4f5 - h7h6 - g4g5 - h6g5 - h4g5 - d6e7 - d4d3 - e7f7 - g5g6 - f7f6 - d3e4 - f6e7 - e4e3 - e7f6 - e3e4 - f6f7 - e4e3-$

We see that the last couple of moves are the same, and since nothing was caught the board results in the same game state, and can conclude that it will be stuck here for the foreseeable future. This can be explained by looking at the result from Rybka. If Black had not resigned, white would have move his pawn up and gotten a promotion. But from the point of the loop with our engine the depth of six that we search, we wouldn't see the promotion and therefore only think about protecting ones pieces.

if we increase the depth to 8, we get another result.

$f4f5 - h7h6 - g4g5 - h6g5 - h4g5 - d6e7 - d4d3 - e7f7 - g5g6 - f7f6 - e4f4 - f6e7 - f4e5 - e7e8 - e5e6 - e8d8 - f5f6-$

The first moves are the same as with a depth 6, but at this point both black and white finds that white will move a pawn up and get a queen for it. Though the route is different from Rybka, the result is the same. Since these searches was made without Quiescence Search (Subsection 8.1.5) and we have made it so the quiescence search only searches one more depth down, but only catches and promotion, we can see that if we used quiescence search we would have found the promotion at depth 6.

First

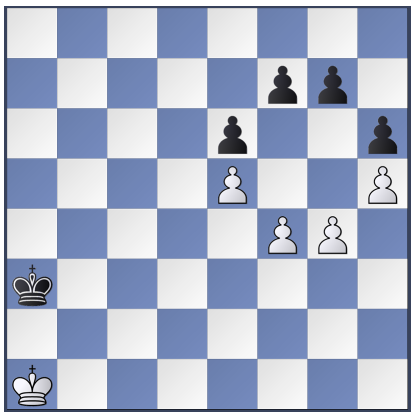


Figure 12.2: Position test First

In the second test (figure 12.2) Rybka finds the moves:

$f4f5 - a3b3 - f5f6 - g7f6 - e5f6 - e6e5 - g4g5 - h6g5 - h5h6 - b3c4 -$ *Black resigns*

Looking at the game state after the last move we see that white would be able to get a promotion, and this is why black resigned.

When we run the test with a depth 6, but with quiescence search we get the moves:

$f4f5 - a3b3 - f5f6 - g7f6 - e5f6 - e6e5 - a1b1 - e5e4 - b1c1 - b3c3 - c1d1 - c3d3 - d1c1 - d3e2 - g4g5 - h6g5-$

At this point we can see in the tree, that both players find that white will get a queen as promotion So even though it takes longer to find the result, it is still the same result as rybka found.

Second

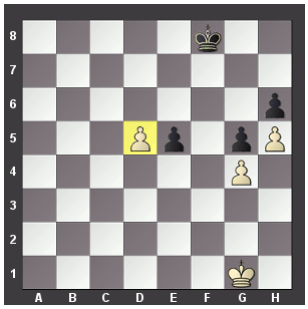


Figure 12.3: Position test Second

In the third test (12.3) Rybka finds the moves:

$f8e7 - g1f2 - e7d6 - f2e2 - d6d5 - e2d3 - e5e4 - d3c3 - d5e5 - c3c2 - white\ resigns$

We see at the end state that black can keep protecting the pawn and move it towards a promotion without danger from the white king.

With our engine, the result is a bit different:

$f8e7 - g1f2 - e7d6 - f2e1 - d6c5 - e1d1 - c5d5 - d1c1 - d5c4 - c1d1 - e4f3 - b1c1 - f3g4 - c1b1 - g4h5$

At this point black king have captured all white pawns, and so rendered white defenseless. The reason black does not move pawns forward to get a promotion is that the search tree does not contain this. It is too far down and the gain for passed pawns does not trumpf the value for capturering a pawn at this point.

Third

Figure 12.4: Position test Third

In the fourth test (12.4) Rybka finds the moves:

$b3b4 - c5d5 - c3d3 - d5e5 - d3c4 - e5e4 - b4b5 - e4e5 - b5b6 - e5d6 -$ *black resigns*

Again we see that before black resigns, white is close to getting a promotion.

When we run the test on our engine, again with a depth of 6 we find:

$b3b4 - c5b5 - c3b3 - b5a6 - b3c4 - a6b6 - b4b5 - b6a7 - c4c5 - a7b7 - b5b6 - b7b8 - c5c6 - b8c8 - b6b7 - cc8b8 - c6b6 - h6h5 - g4h5 - g5g4$

At this point white has blocked all black pawns and have a free run towards a promotion. If black king tries to block one, then the other can get a promotion.

Fifth

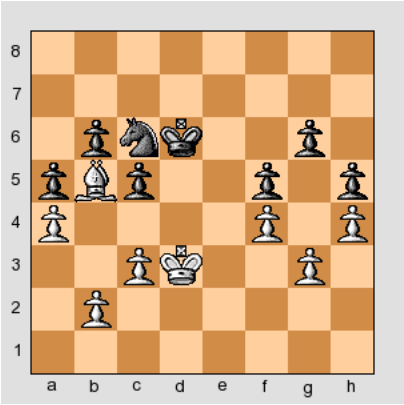


Figure 12.5: Position test Fifth

In the fifth test (12.5) Rybka finds the moves:

$b5c6 - d6c6 - d3c4 - c6d6 - c4b5 - d6c7 - b5a6 - c5c4 - a6b5 - c7b7 - b5c4 - b7c6 - b2b4 - a5b4 - black\ resigns$

In the last game state we see that white can capture the rest of the black pawns and still have pawns left. Then he will be able to move that pawn up while under protection of the king, and at last get a promotion.

When testet on our own engine we find the moves:

$d3d2 - c6e7 - b5e2 - d6c6 - e2f3 - c6d6 - b2b3 - e7d5 - f3d5 - d6d5 - c3c4 - d5c6 - d2c1 - c6d6 - c1b1 - d6c6 - loop$

At this point we go into a loop. The engine finds that if they move one of the possible pawns, it will get captured by the enemy pawn. Because the engine does not see the possible promotion, they will not sacrifice the pawn, and instead just moves the kings around.

Sixth

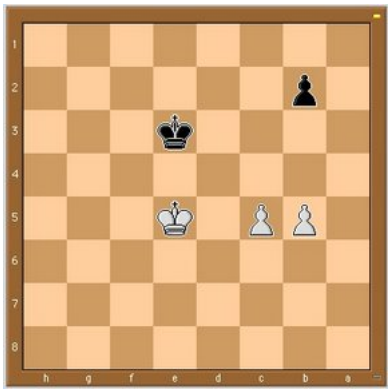


Figure 12.6: Position test Sixth

In the sixth test (12.6) Rybka finds the moves:

$f4f5 - d6e7 - d4e5 - e7f7 - g4g5 - g7g6 - f5f6 - f7f8 - f6f7 - f8f7 - black\ resigns$

Though black resigns, it is not because white wins, but because that they run into a draw. White loses one of the pawns, and then the 2 left are stuck.We can see that instead of the last move $f6f7$ the king should have moved forward to protect the pawn. Then white would have the possibility to get a promotion and win the game. This also show that this version of rybka would rather lose than go into draw.

$f4f5 - d6d7 - d4c5 - d7e7 - g4g5 - e7e8 - c5d5 - e8e7 - d5d4 - e7f7 - g5g6 - f7f6 - d4e4 - f6g5 - e4e5 - g5h5 - e5f4 - h5h6 - f4e4 - h6g5 - loop$

At this point we go into a loop. The search tree generated is not big enough to see the promotion we could get, and therefore it will just keep its pieces protected.

If we instead use a depth of 8, we find the moves:

$f4f5 - d6d7 - d4d5 - d7e7 - g4g5 - e7f7 - d5d6 - f7g8 - d8e7 - g8h7 - f5f6 - h7g6 - f6f7 - g6g5 - f7f8$

Here the engine saw that it could get a promotion, and keeps protecting the piece and moving it forward.