A Chess Engine

Chess Mantis

A Chess Engine

Paul Dailly Dominik Gotojuch Neil Henning Keir Lawson
Alec Macdonald Tamerlan Tajaddinov

University of Glasgow
Department of Computing Science
Sir Alwyn Williams Building
Lilybank Gardens
Glasgow G12 8QQ

March 18, 2008

Abstract

Though many computer chess engines are available, the number of engines using object orientated approaches to the problem is minimal. This report documents an implementation of an object oriented chess engine. Traditionally, in order to gain the advantage of speed, the C language is used for implementation, however, being an older language, it lacks many modern language features. The chess engine documented within this report uses the modern Java language, providing features such as reflection and generics that are used extensively, allowing for complex but understandable code. Also of interest are the various depth first search algorithms used to produce a fast game, and the numerous functions for evaluating different characteristics of the board. These two fundamental components, the evaluator and the analyser, combine to produce a fast and relatively skillful chess engine. We discuss both the design and implementation of the engine, along with details of other approaches that could be taken, and in what manner the engine could be expanded. We conclude by examining the engine empirically, and from this evaluation, reflecting on the advantages and disadvantages of our chosen approach.

Education Use Consent

We hereby give our permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name:		Signature:	
Name:		Signature:	
Name:		Signature:	
Name:		Signature:	
Name:		Signature:	
Name:		Signature:	

Contents

1 Introduction **7**

1.1 Motivation 7

1.2 Aims 7

1.3 Background 8

1.4 Preliminaries 8

1.5 Outline 9

2 Background **10**

2.1 History of Computer Chess 10

2.2 Traditional Engine Design 12

2.3 Present Day 16

2.3.1 Engine Ratings 16

2.3.2 Current Chess Engines 16

2.3.3 Man Versus Machine 17

3 Design **19**

3.1 Overview 19

3.1.1 Chess Engine Communication Protocol 19

3.1.2 Major Classes 19

3.1.3 Threading 20

3.1.4 Rejected Designs 20

3.2 Board 21

3.2.1 Board representation 21
3.2.2 Move representation 24
3.2.3 Piece representation 24
3.3 Analyser 25
3.3.1 Time Control 25
3.3.2 Alpha Beta Pruning 25
3.4 Evaluator 26
3.4.1 Material Evaluation 27
3.4.2 Passed Pawns 27
3.4.3 Crowded Files 28
3.4.4 Pawn Quantity 28
3.4.5 Crippled Pawns 29
3.4.6 Isolated Pawns 30
3.4.7 King Pawns 30
3.4.8 Piece Freedom 31
3.4.9 Center Control 33
3.4.10 King Attack 33
3.4.11 Center King 34
3.4.12 Has Castled 35
3.4.13 End Game Multiplier 36

4 Implementation **37**

4.1 Overview 37
4.1.1 Language Choice 37
4.1.2 Language Style 37
4.1.3 Game Class 37
4.1.4 Communication between objects 37
4.1.5 Threads 38

4.1.6 Chess Engine Communication Protocol versus Bespoke UI 38
4.2 Board 38
4.2.1 Board representation 39
4.2.2 Move representation 40
4.2.3 Piece representation 42
4.2.4 Move legality 48
4.2.5 Checks, checkmates and stalemates 51
4.2.6 Castling 52
4.2.7 Promotions 54
4.2.8 En Passant 56
4.2.9 Move committing and reversing 58
4.2.10 Legal move generation 59
4.2.11 Board hashing 60
4.3 Analyser 63
4.4 Evaluator 65
4.4.1 Material Evaluation 67
4.4.2 Passed Pawns 68
4.4.3 Crowded Files 69
4.4.4 Pawn Quantity 70
4.4.5 Crippled Pawns 71
4.4.6 Isolated Pawns 71
4.4.7 King Pawns 72
4.4.8 Piece Freedom 73
4.4.9 Center Control 74
4.4.10 King Attack 75
4.4.11 Center King 76
4.4.12 Has Castled 76

4.4.13 End Game Multiplier 77

5 Evaluation **78**

5.1 Testing Rules 78

5.1.1 Outline 78

5.1.2 Piece Movement 78

5.1.3 Captures 84

5.1.4 End of Game Recognition 85

5.2 Player vs Computer 87

5.3 Computer vs Computer 88

5.3.1 GNUChess vs Chess Mantis 89

5.3.2 Crafty vs Chess Mantis 90

5.3.3 Beowulf vs Chess Mantis 92

6 Conclusion **93**

Appendices **95**

A Contributions **95**

A.1 Paul Dailly 95

A.2 Dominik Gotojuch 95

A.3 Neil Henning 96

A.4 Keir Lawson 97

A.5 Alec Macdonald 97

A.6 Tamerlan Tajaddinov 98

B Status Report **99**

B.1 Product 99

B.2 Deficiencies 99

B.2.1 XBoard Commands 99

B.2.2 Features 99

B.2.3 Rules 100

C Class Diagram **101**

D Requirements Specification **103**

D.1 Functional requirements 103

D.2 Non-functional requirements 103

E Coding Style **104**

Chapter 1

Introduction

1.1 Motivation

Many chess playing programs, or ”engines”, are available, some available free over the Internet, others serious commercial ventures. Given the volume of chess engines, the motivation behind this project may not be immediately apparent. However, the world of computer chess is highly competitive, and consequently, a lot of information and research in the area is kept secret so as to maintain an advantage against competitors. Though there are now a small number of tutorials on the Internet with regards to programming chess engines, these generally do not cover the entirety of the process, meaning that significant portions of the problem are completely undocumented publicly.

Clearly the lack of documentation makes the project a challenging one, allowing us both to practice our software development skills, but also to learn more about computer chess, specifically the mathematics that lie behind it. This understanding also allows us to get a better grasp of how other, similar, games such as Checkers or go are implemented on a computer.

Though there are a few open source chess programs, the vast majority of these are written in the C language for maximum efficiency. This not only makes them harder to read and follow, but also means that these engines are more tricky to make truly modular, and cannot take advantages of the modern features of object orientation. To our knowledge there are no open source chess engines that are written in an object oriented language, and consequently, the decision to implement our program in Java differentiates us from the vast majority of open source chess engines. Admittedly this decision was taken for other pragmatic reasons such as the ease and speed of implementation, but it does allow us to design our program in a way that differs significantly from the other open source chess engines.

1.2 Aims

The primary goal of the project was to create a chess engine capable of giving an interesting game to the casual chess player. This means the engine must be able to adequately open and close chess games, and have knowledge of at least basic chess strategies and formations. This meant that our engine had to be fast enough to maintain the interest of the player, whilst still giving itself time to compute a good move to play.

We also set ourselves the goal of using the ”Chess Engine Communication Protocol”, a standard way for chess engines to communicate with user interfaces, such as XBoard, enabling the swapping of interfaces without modifying the underlying chess engine. This meant that we did not have to develop our own user interface, and we could play other chess engines that also implemented the protocol. The protocol is reasonably well documented, but as it uses standard input and output for communicating with the chess engine, it presented some minor challenges.

Our chess engine will be capable of checking its opponent's moves for legality, as well as supporting features such as loading previously stored games and allowing for user defined time constraints on moves.

1.3 Background

As previously stated, we are by no means the first team to create a chess engine. Even before computers were fast enough to begin playing chess properly, academics researched the possibility of one day using computers to play chess. As computers advanced greatly in terms of speed, the discipline of computer chess took off. Only within the last two decades, however, have said engines become powerful enough to present a serious challenge to human masters.

A combination of improvements in the algorithms and methods used within chess engines, and vast improvements to the underlying hardware, specifically in speed and capacity, has meant that in the last few years computer chess engines at the top of the field have matched the top human players in terms of skill. Obviously our engine, being limited both in terms of the software we have time to create, and the hardware we have access to, will not reach this standard.

Many of the techniques that we use today for implementing chess engines were developed decades ago, and have not been improved substantially since. Others however have only been made public recently. Generally the older the technique, the better documented and consequently easier to implement it is. It is clear that the success of our project does not solely rest on our own efforts, without the many chess engines that have come before ours we wouldn't even know where to start, let alone be capable of producing the engine we have.

Game theory plays an especially important role in computer chess, specifically the minimax algorithm and its various descendants, a depth first search algorithm that traverses a game tree to a certain depth in order to find the value of a given move. These algorithms have been central to the creation of chess engines since they were first proposed for the task in 1950.

1.4 Preliminaries

In order to understand this report the reader is required to have a firm grasp of programming concepts, specifically Object Orientation and more advanced techniques such as reflection and inheritance. Though Java is the programming language in which Chess Mantis is implemented, familiarity with it is not essential to understand this report as it only provides a broad overview of the project, without going into excessive technical detail about the implementation. An understanding of algorithmics would also be beneficial to the reader, specifically in the area of depth first search trees, however such knowledge is not essential.

An understanding of the rules of chess is required, and whilst a knowledge of more advanced chess tactics and obscure rules, such as en-passant captures would be advantageous, it is not required.

1.5 Outline

The rest of this dissertation is organized as follows.

Chapter 2 examines the history of computer chess in order to provide a degree of context for the rest of this document, and to inform the reader of the development of the various technologies on which our engine is built.

Chapter 3 gives an overview of the architecture and construction of our program, detailing how the various components interact, and what each component's purpose is. This section will also explain which parts of the engine can potentially be swapped out for alternative implementations, and which areas would be easily expanded on.

Chapter 4 details the implementation of the design outlined in the previous chapter, providing a more in-depth discussion of the key areas of interest, namely the board, the analyser and the evaluator, not only detailing their implementations, but also discussing broader issues such as algorithms, and possible future enhancements.

Chapter 5 describes the testing and evaluation that was carried out on the engine, including playing the engine against both humans and other computer chess engines. This chapter also explains the various test cases used to check that various chess rules were implemented and checked for properly by our engine.

Chapter 6 concludes by giving an overall evaluation against the stated aims of the project, to gage whether or not the project can be deemed a success.

Chapter 2

Background

2.1 History of Computer Chess

The ambition to develop a chess-playing machine dates back to the eighteenth century. At that time an idea of creating an automaton, a self-operating machine, was dominating the minds of engineers, though limited advances in the research area led to just a few hoaxes developed in 18-19th centuries, The Turk, Ajeeb and Mephisto, all operated by concealed human directors. It was not until 1912, when the first true automaton, El Ajedrecista, was developed, however even it was capable of playing only a three-piece ending, involving a King and a Rook versus a King[6].

The research into chess-machinery required a technological advancement, so with the appearance of digital computers in 1950s a new wave of research began. Initially researchers were motivated by studying human cognition, though this branch of research was soon abandoned. Computer chess engines tend to analise a large number of positions several moves ahead, which makes them different to human players, who primarily focus on identifying patterns and relying on experience. Despite this, research into computer chess goes on, however focusing on creating an efficient chess algorithm for solo entertainment or competing in computer chess championships.[5]

The first research paper on the subject of computer chess was published by Claude Shannon in 1950 before anyone had programmed a computer to play chess. It discussed two possible approaches, Type A, which referred to an exhaustive move evaluation, and Type B, which referred to a selective approach, where the engine evaluates only "good moves". At the time, the latter was a more attractive technique, since even the most sophisticated computers could not calculate positions beyond 3 plies (moves made by each of the players) in a reasonable amount of time. World Chess Champion Mikhail Botvinnik was one of the few chess grandmasters to devote time to the research in computer chess. The hardware available to him could not permit him to perform a full-width search to an acceptable number of moves, so his research also focused on the selective approach.

An important milestone in the development of chess engines was the appearance of Chess 4.0, developed by Northwestern University. Chess 4.0 was the first chess engine to abandon the selective search approach and instead focused on the full-width search using Alpha-Beta and negamax, optimised versions of the Minimax algorithm, discussed by Claude Shannon. Chess 4.0 created a paradigm used by the majority of modern chess applications, the popularity of which was simply because it produced better chess engines. Additionally, the full-width search approach made it easier to debug the code, making it more predictable. Section 2.2 will further expand on the subject of evolution of chess algorithms.

The twentieth century saw a number of important chess plays between a human and a machine. For many years after the intense research into computer chess began it remained an open question whether an artificial intelligence (AI) could reasonably compete with the best of human players. In 1968 International Master David Levy made a famous bet, that he would not loose to a machine in ten years. In fact, it took twenty-one years until 1989, when he lost to the computer Deep Thought in an exhibition match. Despite this win, Deep Thought was not yet at a world champion's level as demonstrated by Gary Kasparov winning 2-0 in 1989. It wasn't until 1996, when Gary Kasparov, then World Champion, lost a chess game to Deep Blue in the first game of the match. Despite Kasparov regrouping to win three and draw two of the remaining five games, the first game was the first time a reigning chess world champion has ever lost to a chess engine. During the rematch in 1997 Kasparov lost to Deep Blue 2.5 - 3.5. However Kasparov subsequently alleged, that IBM cheated by using a human player to change the engine behaviour midgame. A documentary made about the confrontation in 2003, titled Game Over: Kasparov and the Machine, suggesting an analogy with the aforementioned hoax the Turk, operated by a concealed human.[8] Nevertheless, this was the first time a computer defeated the World Champion in a match of several games. Deep Blue was dismantled after the match and never played again, however human-versus-machine matches continued to be played. As the computational power of workstation computers continued to increase, more chess engines became able to challenge the world class players.

An interesting match between Viswanathan Anand and Rebel 10, which took place in 1998 resulting in Anand losing 3-5, was remarkable since the games were played at different time controls. In four blitz games, played at five minutes plus five seconds per moves, Anand lost 1-3; in semi-blitz games, fifteen minutes each, Anand lost one and drew the other, however in the tournament time controls, Anand managed to win 1 1/2 - 1/2, suggesting that a human player may be stronger in the slower time controls, where the player rating is determined.

Game tree is an abstraction used to describe the combination of all possible moves on the board down to a certain maximum number of moves. Some chess masters have made claims that they could win against the smartest of chess AIs using 'anti-computer tactics'. The essence of them is trying to achieve a long-term advantage with computer being unable to see it in its game tree. An example of an attempt to employ such tactics is Brains in Bahrain, an eight-game match between Vladimir Kramnik, then reigning world champion, and Deep Fritz 7 in October 2002. Kramnik used the above tactics successfully in the second and third games, securing winning positions. The fifth game was lost due to a blunder. However the sixth game has been described by some as 'spectacular'. Kramnik in early mid-game sacrificed a piece in an attempt to gain a better position and launch an attack, however Fritz managed to create a watertight defense, thus leaving Kramnik in a bad position. Kramnik resigned, however the post game analysis showed that the engine could not force a win, and Kramnik gave up a drawn position. The overall match ended as a draw 4-4 with each of the sides winning two and drawing four of the games.[2]

Unlike Checkers, Chess has not been solved, however attempts are being made by exploring the use of endgame databases. The essence of the endgame databases is creating a winning position on the board and then applying a retrograde analysis, a technique employed by chess problem solvers to determine which moves would need to be played up to a given position. Presently a weakness of even the best chess AIs is that they are unable to see a winning position if it is too deep in the game tree. Endgame databases are used by many of the current chess engines to solve this problem, so when the engine encounters a position that is present in the database, it simply stops analysing and follows the set of moves from the above database. The size of the database is only limited by the amount of computer memory, as it grows exponentially as the number of chess pieces increases. Currently all endgame positions have been solved for less than six pieces, and most of the six piece endgames. As the research goes on, eventually the endgame databases may assist in completely solving the game of chess.

When developing a chess engine, there are a number of implementation issues to consider. Thus the developer needs to make a choice of board representation in the memory, game tree search techniques, when choosing the best move from those available, and leaf evaluation functions, when evaluating a given position on the board. The next section will expand in further detail on the variety of algorithms available currently.

2.2 Traditional Engine Design

As has been mentioned in the above section, algorithms have proved central to computer chess, specifically the minimax algorithm, and its various descendants. Minimax works in tandem with an evaluation function that looks at a specified board state and returns a score indicating how great the advantage or disadvantage is to the current player.

An evaluation function typically uses various heuristics, computer measurable advantages to one player, ranging from the most basic measurements such as weighted piece count to more advanced indicators such as pawn formation. Each of these heuristics will return a score which must then be combined with the scores of other heuristics in a balanced manner in order to produce an overall score for the board.

Evaluation functions for chess engines are not very well documented as this part of the engine is considered critical to its skill. However many evaluations that human chess players take into account when analysing a board can be translated to computer heuristics, and many of these are well documented, though some are subject to debate.

Unlike computer board evaluation, the analyser algorithm, minimax, is very well documented, having been central to computer chess from the beginning of the discipline. The minimax algorithm is an important tool within the area of Game Theory, a field within applied mathematics. Game theory generally, and hence minimax, finds wide use not only within computer science, but also within the fields of economics, biology and philosophy. The ability to represent behaviour, be it of a market, of natural selection, or any other similar process, as a game with simple rules, is immensely useful in enabling the prediction and explanation of behaviour.

The minimax algorithm was first proposed by the prominent computer scientist and mathematician John von Neumann in 1928 [26], as part of his pioneering work on game theory. This algorithm is of use in zero-sum games, that is, games in which one player's gain is equal to the sum of the loss of all other players. In two-player games such as chess or Checkers, this means that when one player makes a move that gains her an advantage, her opponent has a disadvantage of the same magnitude as her advantage.

Minimax also normally requires that the subject game is a game of perfect information, that is, the complete board state is known to all players, in theory no player will ever have any more information about the board state than any other. There are variations of the minimax algorithms for games that lack the perfect information quality, however this is beyond the scope of this document.

The algorithm traverses a game tree, a conceptual model of the manner in which Minimax operates. Note that Minimax does not actually require a tree structure to traverse, this model is just to ease human understanding. The game tree is composed of all the possible future board states, generated by all possible future move sequences, within certain bounds. Typically, the game tree is constrained by having a maximum depth, specified in ply. A ply in a two-player turn-based game is half a turn, in the case of chess, either a black move or a white move, so to evaluate to 6 ply is to evaluate 3 turns into the future.

Minimax is a depth-first search algorithm, meaning that the algorithm first searches one branch of the tree to its leaf node, then another, sequentially, until all leaf nodes of the tree have been visited.

Minimax's function can be best described with the phrase ”minimising the maximum possible loss”, and from this phrase minimax derives its name. Each node in the tree, a board state that results from a future move sequence, has a score assigned to it. At leaf nodes this score is generated by the evaluator function, which analyses the board and assigns the board a numerical score based on how advantageous the board is for the initiator of the search. The value of a non-leaf node is calculated by taking either the maximum or minimum score of said node's children, and so eventually the score propagates up to the root node. The root node's value is the value of the calculated best possible move from the given board state. Obviously, for a computer to utilise the minimax algorithm, it must be modified slightly so that it can return the move that is deemed the best possible, rather than just the score of that move.

Evaluating the speed of the minimax function is of great importance, as minimax, along with the evaluator, forms the core of modern computer chess. The best way to do this in the abstract, without actual timings of a specific implementation, is to calculate the number of nodes that will be evaluated. If b is the average branching factor, that is, that average number of children of a given node, and d is the depth to which we are to search, then b^d nodes will have to be evaluated. In chess, there can be as many as 100 possible moves in the mid game. The average branching factor of chess is around 35. So a search with a depth of 6 ply, 3 turns, will evaluate around 35^6, or roughly 1,800,000,000 nodes. Although not all of these nodes are evaluated using the evaluation function, clearly the number of nodes necessary for evaluation with naive minimax grows exponentially. Though the time taken to perform a single evaluation is minuscule, when performed over billions of board states, deep searches are made very time consuming.

Numerous enhancements to naive minimax are available, such as alpha-beta pruning, principal variation search and the killer move heuristic. Chess Mantis implements a subset of these improvements, and these are detailed further within chapters 3 and 4.

As well as algorithmic enhancements, a form of caching is also commonly implemented on the game tree, in order to increase search speed. The ability to cache important results also means that if a search is terminated before completion, it has not been completely wasted. This is implemented in the form of transposition tables within computer chess, an idea first pioneered by Richard Greenblatt at MIT. In 1967 the chess program ”Mac Hack”, running on an DEC PDP-6, became the first program to utilise this technique.

The basic principle of transposition table caching is that a single board state can be reached by two or more move sequences. For example, from an initial board, moving the leftmost pawn one

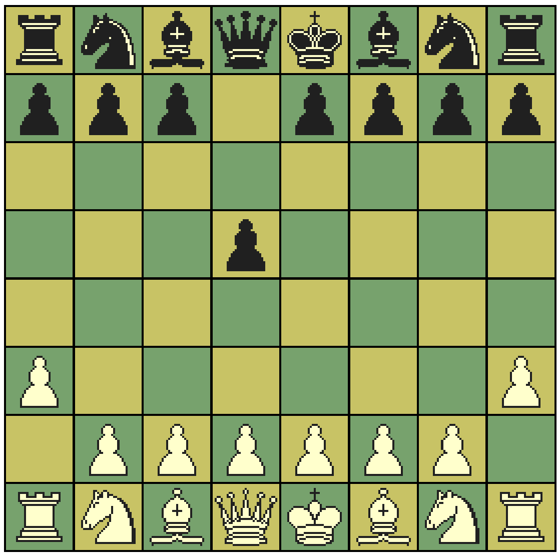


Figure 2.1: Potential transposition

space forward (a3) and then the rightmost pawn one space forward (h3) might well produce the same resulting board as if the move order were reversed (see fig. 2.1), if the opponent handles this the same way. These identical positions with different histories are known as transpositions. The deeper the search, the further into the future it looks, the more transpositions will occur, and so for fast chess engines such a table is critical.

In order to check if a certain board position is the same as another, a means of comparing two chess boards is required. As a board is normally a complex construction within the chess engine, a comparison of all its fields would be so time-consuming as to negate the time savings afforded by a look-up table, and so a more efficient, but preferably equally precise method is required.

The most prominent and well-documented method of comparing boards is called Zobrist hashing, a scheme first published by Albert L. Zobrist in 1970 [29]. This method does not find use solely in chess, but can equally be applied to other similar games such as Checkers. Zobrist hashing allows chess boards to be given a near unique hash depending on the position of the various pieces.

For each square on the board, and for every piece that might occupy that square, a random number is generated at the beginning of the game. It is extremely important that these numbers are highly random, as any duplicate values will result in ambiguity between boards. To generate the key for a given board, the random values associated with each piece present on the board are XORed together, to create what should be a unique value. Board positions that are only different by one move produce completely different keys.

The advantage of Zobrist hashing over other methods is speed. Because of the nature of the XOR operation, in order to undo a move one simply has to XOR the move in again, i.e. the move is XORed twice to return to the original state. As making a move is typically two XOR operations, the time required to generate the key is negligible, as such binary operations are generally very fast.

Zobrist hashing has other uses within computer chess, besides being excellent for transposition table implementation. Zobrist can be used for other constructs such as opening books or pawn structure tables.

Another feature that has been subject to historical development is precomputed databases, usually in the forms of opening books and endgame databases. Opening books have been used by human players long before the conception of computer chess, and so creating a database of good openings is not particularly technically challenging, its simply repetitive work.

Endgame databases, are of more interest. Created by a process of retrograde analysis, that is, forming checkmates or stalemates and then tracing the possible moves back in order to produce a game-ending move sequence. These move sequences are then stored in an indexable format so that if a situation documented by the database occurs, then the chess engine can stop thinking and simply follow its provided move sequence. Because of the massive amount of possible moves, and the extreme complexity of some check mates, these databases are always generated by computers, and so far have only been created for six or fewer pieces on the board.

Besides the obvious advantage gained through the speed of looking up a predefined move sequence rather than using a minimax algorithm, and the guarantee of best play (i.e. using a database you will always play the best possible move), endgame databases are also of huge importance in combating the horizon effect.

For any checkmates that are further ahead into the future than the chess engine can see, for example if check mate is seven ply away but the engine only calculates to six, a computer engine will find it very difficult, if not impossible to close a game. This is known as the horizon effect, as the engine cannot see beyond the horizon of however many ply the minimax algorithm is set to search to. Clearly, if the checkmate is stored in the endgame database then this horizon problem is entirely avoided. However it is worth noting that currently databases have only solved chess for 6 piece and fewer checkmates, so a checkmate move sequence for a board state with more than six pieces still requires manual analysis, and are hence is still subject to the horizon effect. The horizon effect is also severely problematic during the rest of the game, as a board state may appear upon evaluation to be very good for the current player, however the next turn may result in a major capture. This is sometimes partially countered by the quiescent search addition to minimax, an extension that partially evaluates the children nodes of the evaluated node, checking only for captures, however this improvement in itself does not completely compensate for the horizon effect.

Besides the various algorithms used for searching, the way the board is represented within the engine is also of great importance. Operations on the board are called billions of times during a standard search, and so it is of huge importance that the board be as efficient as possible.

A naive implementation would simply be an 8 by 8 array of pieces, represented by references or characters, however this makes checking whether or not a piece moves off the board with a proposed move potentially difficult. A slightly improved version of the 8x8 board is commonly known as 0x88.

0x88 boards are comprised of 128 squares, double that of a standard chess board. Alongside the standard 8x8 chessboard there is a second dummy chessboard to it's right, this second board means that checking whether or not a piece lies out with the bounds of the board after a move is much easier.

A significant improvement over the 0x88 representation method comes in the form of a bit board. Bit boards were first invented by Arthur Samuel in the 1950s [22], it was initially designed for his

Checkers engine. Bit boards have not been substantially improved upon since their creation, and are the current standard in competitive chess engines.

The Bit board works by maintaining several boards each represented by a 64bit integer, one bit for each square. Each board represents a distinct piece of information, such as the position of all the white pieces on the board, or the positions of all pawns on the board. A logical AND can then be applied to these two boards to produce a board representing the white pawns on the board. This makes for an extremely efficient implementation because of the efficiency of operations like AND and OR, and most checks only needing a limited amount of information about the board. This is especially true on a 64bit platform, as the Integers are naturally 64 bits in length.

2.3 Present Day

In the early stages of chess engine development, the focus of interest was, almost exclusively, engines which were designed to run on supercomputers using dedicated hardware, created specifically for the individual engine in some cases. However, in the present day of computer chess, that focus has shifted. Now, the main area of interest is researching the development of chess programs that can run on a standard computers and matches between world class players and computers.

2.3.1 Engine Ratings

There has always been a fierce sense of competition within the computer chess community, with a great need felt by all, to prove their engine to be better than their rivals. This desire to be rated most highly, has, if anything, only become more evident in present-day computer chess development. A large number of international tournaments are dedicated to computer versus computer chess playing, in order to determine which engine deserves to come out on top. However, many critics refute engine ratings based on tournament performance alone. This is mainly due to the fact that engines don't play enough games during tournaments to give a true representation of relative strength. Also, with many tournaments allowing engines to play on varying specifications of hardware, a proper engine against engine comparison, at the software level alone, cannot take place[28]. As a result, a wide variety of testing suites have been developed. These are used to produce chess engine rating lists, which aim to provide a statistical measurement of relative engine strength. The tests themselves usually ask the engine to find the one best move in a given position. This position may be geared towards positional, tactical or endgame play, depending on the suite being used. Ratings are given to engines based on the Elo Ratings System [25], the official rating system of the United States Chess Federation, for both human and computer chess players alike.

2.3.2 Current Chess Engines

Today, computer chess programs have become popular with both the academic and the commercial world. Engines are usually classed under one of two headings: Commercial or Freely Available. Of those which come under the heading of Freely Available, some are also available as open-source. It is usually the commercial programs which enjoy most success in competition, with the top four engines at the World Computer Chess Championships in 2006, being commercial. The engine which finished fifth that year, Zappa, was, at that point, freely available. However, success of such engines is somewhat of a self-fulfilling prophesy, with Zappa becoming a commercial engine as a result of its strong performances.

Currently, the top-rated engine is Rybka, positioned top of all notable chess engine rating lists as of November 2007, with an Elo rating of between 2900 and 3120[24]. The engine has enjoyed numerous success, most recently winning the 2007 World Computer Chess Championships. Rybka has also twice won the International Chess Championships in 2005/06 as well as finishing first in the Dutch Open Computer Championships with a perfect score of 9 out of 9.

Rybka, whilst not being an open-source engine, has had some of its implementation details revealed by its creators. It uses a bit-board representation (see 2.2) and group of related Boolean variables describing the board. This allows for queries on the board to be performed in a single logical operation. Rybka also uses an aggressive form of alpha-beta pruning (see 3.3.2). The details of the evaluation process are unknown, however, it has been said to be based on material imbalance studies which were conducted by Larry Kaufmann [16].

Probably the most successful commercial engine at present is the Fritz software. The Fritz program comes in many forms including Fritz X3D, complete with 3D graphical interface, as well as Deep Fritz, which runs on multi-core processors. It has enjoyed success in each of its forms, with its most notable triumph being when it defeated grand master Vladimir Krammick 4-2 in 2006. The internal details of the Fritz engines are unknown, due to the concealed nature of commercial chess engines,. However, it is known that Fritz can evaluate 8 million positions every second, and look ahead up to 18 ply (see 2.2) during the mid game.

Open-source chess engines are difficult to come across in general, with strong engines, in particular, being somewhat of a rarity. The Fruit engine, however, started out as being a freely available open-source engine. Although now it has become a closed-source commercial product, the previous versions are said to have been very influential in the development of many other major engines in recent years. The most notable engine to have been challenged with such a suggestion, is the currently top ranked, Rybka [27]. The versions of the software which were open source, have remained so. The most unique feature of the Fruit engine was its 16x16 board representation. It also implemented the classic Negascout [19], a search algorithm which has the potential to be faster than AlphaBeta at coming to the same conclusions. This particular engine made use also of Iterative Deepening, another searching technique, used to find the best possible next move, via the use of a breadth-first search [21] . The main difference between Fruit and other open-source engines is its strength of play, which earned it an Elo rating of 2842, while still in open-source form.

2.3.3 Man Versus Machine

As mentioned above, there has been much interest in not only comparing the various available chess engines against each other, but also against human players. In particular, as the strength of chess engines has grown, they have posed more and more of a challenge to more advanced players. With the standard of play top chess engines are able to reach at present, many Grand Master chess players are suffering defeat at the hands of a machine. Currently, the top-rated chess player in the F.I.D.E rankings is Vladimir Kramnik, who has an Elo rating of 2799. Compare this with the aforementioned Rybka engine, which has been ranked top of the chess engine rankings with an Elo rating of between 2900 and 3120. Whilst chess has yet to be solved, unlike checkers for example, it is most likely the case that chess playing software will continue to advance to a point, where it can play such near perfect chess, that for even the top players in the world, there will still be no realistic prospect of the human emerging victorious.

Chapter 3

Design

3.1 Overview

3.1.1 Chess Engine Communication Protocol

Conformity to the Chess Engine Communication Protocol [18] allowed integration with many standard user interfaces and many other engines which also conform. The protocol is asynchronous, communicating as soon as ready across standard in/out with a predefined set of commands and formats to communicate data. This standardisation allows any engine that conforms to the protocol to work with any other that also conforms. The protocol is constructed in such a way that it envelopes the internet chess protocol, meaning that any conforming engine can also be used over internet chess.

Ensuring Chess Mantis conforms to the CECP allows us to concentrate fully on the chess engine, and not lose time to developing a user interface. Instead, any of the conforming interfaces can be swapped in and out.

3.1.2 Major Classes

The engine exists as six major components: XBoardIO, Game, Board, Analyser, Evaluator and TimeControl. The structure of the engine is shown on the class diagram, included in Appendix C.

XBoardIO is the Input/Output system; the interface between the engine and the GUI. This is the part that must conform to the Chess Engine Communication Protocol, see 3.1.1, fielding all commands the GUI sends and pass them on, then sending correct responses back out. Based on the command received, XBoardIO either instantly responds, or packages the message it receives up and queues it up for processing.

The Game class addresses this queue, taking each message now packaged into one of six types of Event, and processes it. Depending on the type of the event, Game will do different things, but ultimately can be regarded as the hub of the engine. It is the class that is called upon instantiation of the game and is responsible for starting up the rest of the elements.

The board is the engine's internal representation of the chessboard and is used to store the state of the game at any given time, as well as by the evaluator to assess potential board states. It's the largest class codewise, as it contains all the methods to test and manipulate the board.

Next, the analyser and the evaluator work in tandem to request a list of all possible moves from the board, assess the current position then make each of the moves in turn. These board positions are then assessed and so forth until the TimeControl class deems there is no more time available, or a maximum depth is reached, at which point based on the assessments the evaluator picks the best path and so chooses the move to start on that path. This chosen move is handed to the Game class which then passes it to XBoardIO to pass to the GUI.

As already mentioned, another component in the engine is TimeControl, which determines whether or not the engine will have time to assess another ply or not. The TimeControl works on its own thread, noting the time at the start of the move, the time available to make a move, then after each ply, the time taken to assess that ply. A crude heuristic is then used to calculate whether there is time still available to investigate another ply (see 3.3.1).

3.1.3 Threading

The engine utilises multi-threading to get the most out of each alloted time space to choose a move. The main thread of execution is assigned to the Game class, as this is the main hub of the engine. Game will yield to the board, the analyser and the evaluator as necessary. To make the engine constantly responsive to interactions from the GUI, upon startup, the game class starts a new thread for our input/output system (XBoardIO). To keep correct timing, the game class also starts another thread for TimeControl and sets that off on its own.

3.1.4 Rejected Designs

Architecture

The initial design was to revolve around the internal board, with the Board class driving the engine, invoking the evaluator and analyser as and when it required, inviting them to operate on it directly. This was rejected due to the major classes being too tightly coupled and also the Board class being too heavily relied upon.

Board

Many of the designs that came before the current one were mostly variants of how the board was internally represented. One of the major discussions was whether to have one board that steps on and rolls back, or a seperate board for every change in board state during evaluation. The driving factor behind the choice to use only one board was down to our chosen representation of the board. Multiple boards would require a lightweight representation of the board such as bit boards (see 2.2), whereas the chosen design of the board was a single array of objects so would be too memory intensive to multiply.

Piece

Another laboured design decision, somewhat led from the board discussion, was how to represent a piece. Piece representation is mandated in bit boards from the board design, whereas the various array choices gives more flexibility. Each piece on the board can be represented in the array as a piece object and with Object Oriented design, deriving an actual piece can be done in several different ways. A white piece can inherit from piece, then a specific piece from there, meaning a three-teir heirarchy for piece. A second option considered was each piece object to be the same, but to carry a piece type and a colour, so a white king would just be a piece, with attributes white and king.The design actually chosen was somewhere in the middle, with each specific piece implementing a Piece interface and carrying in its attributes its colour.

3.2 Board

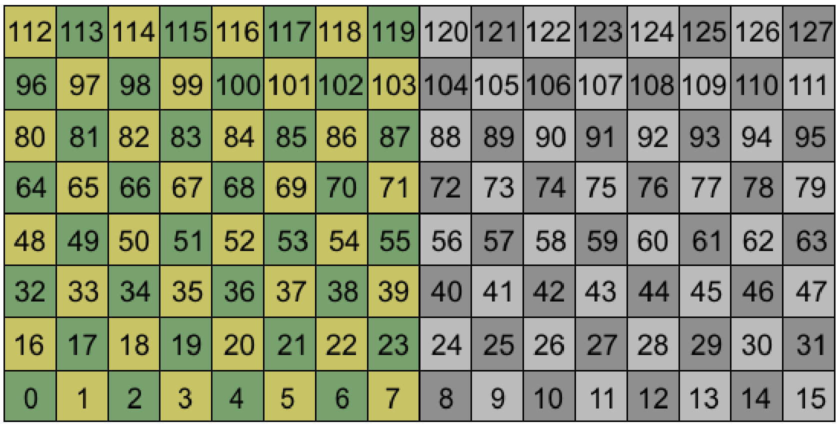
3.2.1 Board representation

As there are number of ways to implement the board representation and its contents in a chess engine [14], the design was aimed at achieving a flexible structure that could be easily modified, if certain changes were to be made to the used board representation or if a completely new representation was to be introduced and integrated with the engine. Chess Mantis provides a set of board functionality rules, which have to be followed in any chosen implementation of the board representation, allowing the engine to take full advantage of a custom made board.

The most commonly used board representations in chess programs include simple arrays, 0x88 representation and bit boards. The first concept of board representation to be analysed for Chess Mantis was a simple two dimensional array of 8 by 8 cells. It is natural for humans, as it seems to be a very high level abstraction, which directly maps a physical chessboard to a digital representation. Even though it is easy to imagine and build, it is deemed very inefficient and at times difficult to control, especially because of the possibility of pieces going off the board. Another idea, which is a variation of the simple array solution, is a single array of 64 fields representing the chessboard squares with an additional layer of 64 fields, which serve as a safety margin for solving the problem of pieces running over the edge of the board. This solution evolved into the representation widely known as 0x88, taking advantage of a single array with 128 fields, which finally was picked as the choice for engine's board representation. In the process, probably the most efficient board representation technique known as bit boards was also considered [17]. Its high efficiency results from using a set of 64-bit sequences of 0s and 1s. Each of these represents a certain situation on the board, such as the positions of all the white pieces. By combining these sets with simple logical operations the information required from the board can be extracted quickly. However, due to its complexity and the fact that the piece representation was designed to be object oriented and residing directly on the board representation, bit boards were decided to be an enhancement, which might potentially be included in the chess engine in the future.

The 0x88 representation is a good choice, because of the fact that it inherits the easiness of manipulating the board state from more simple array representations. The movement rules of each piece type can be clearly defined in terms of the subtraction of the value of the target field and the initial field. These deltas remain the same for specific direction of movement on every situation on the board. This implies that single horizontal moves will always change the position by -/+ 1, single vertical moves by -/+ 16 and single diagonal moves by -/+ 15 or -/+ 17.

The 0x88 representation, which is currently used in the application, can be pictured as two 8 by 8, regular chessboards connected together on the edge [15], as shown on Figure 3.1.



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

Figure 3.1: 0x88 board representation with square numbering

The board to the left in the above figure is the one used to store the state of the game in terms of the position of chess pieces and can be directly mapped to a physical chessboard or a graphical representation of chessboard. The second board is a dummy clone, which is used to detect off-board moves during the move search by the engine and provides an additional protection against illegal moves by pieces moving on the diagonals (queen, bishop, pawn), columns (queen, rook, king, pawn), ranks (queen, rook, king) and knight jumps. Off-board moves happen, when an attempt is made by the engine to move a given piece, according to its movement rules, from a legal board square to a non existent square, outside the boundaries of the board. An example of such a situation is moving a knight from any of the squares in column H to the right. In terms of piece movement rules, such a jump is assessed to be legal, in practice however, it causes the knight to escape over the edge of the board, as shown on Figure 3.2.

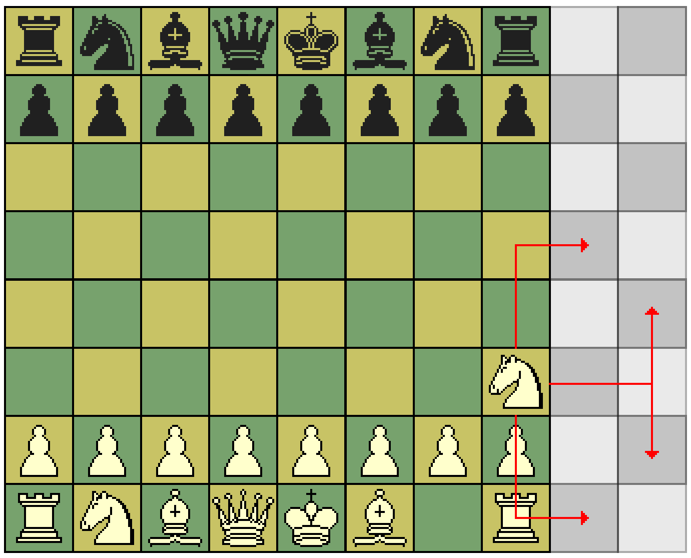


Figure 3.2: Potential knight escape moves off the board

Although a human will instantly understand this kind of move is impossible, as there are no board squares left to go to, the application working on numerical values will only confirm that such a move is legal for the knight. With 0x88, detecting and discarding such off-board moves is extremely efficient, as the test requires only one binary AND operation to verify whether a target square is outside the boundaries of the actual board. That is because the binary values of all squares on the dummy board have their 4th or 8th bit set to 1, whereas these bits are always 0 for the squares on the real board [20].

The manipulation of the board representation is conducted by a set of methods located within the board class. Most of these are responsible for ensuring the board legality and consistency. These test if a given piece is moving according to its movement rules, that its path is not blocked by other pieces and if a capture, promotion, en passant or castling move is not performed. The board also operates the detection of checks, checkmates and stalemates. The board representation also contains methods providing means of performing a given move or reversing it, taking into account all properties of the board and the manipulated pieces. It was also necessary to introduce different types of constructors, which allow to set up the pieces on the board in a standard way or loading a custom situation on the board. Other routines are also encapsulated within the board class, many of which play an important part in delivering relevant data to board manipulation procedures. This includes the legal move generation method, which finds out all legal moves for all the pieces of a give side and a given board situation.

For the purposes of certain chess optimisation solutions, such as transposition tables, the board was also equipped with a hashing function, based on the idea by Albert Zobrist [29]. The function produces a unique, 64 bit integer, which represents the situation on the board by logically combining values related to the properties of the moves made by the pieces on the board.

3.2.2 Move representation

A piece move forms one of the key building blocks in a chess engine. The importance of choosing an optimal design arises from the necessity for the engine to process vast numbers of moves when evaluating a game. A poor choice of representation may result in an impaired overall performance of the engine if unnecessary computations are imposed on the client code, when decoding the move. While the choice of possible move representations is vast, a number of 'better' options had to be considered when making a design decision, thus the following options have been evaluated:

- A string in the form “a2a4”. This is the most human readable notation, that gives just enough information to identify the piece moving as well as the destination. In the above example, a piece (not essentially a pawn unlike the standard chess notation) moves from square “a2” to square “a4”. The problem with this design is that the amount of information that can be contained in a move is rather small and conversion is computationally expensive.
- Two integers: first corresponding to the from square, second to the targer square. Although more computationally efficient, this also contains just a limited amount of information.
- A Move object, which may contain extra information, e.g. whether it's a capture or a promotion, the piece captured, and also the piece promoted to, where appropriate. While less time and space efficient, an object approach contributes to the object orientation of the design.

Based on the above findings, a decision was made to implement moves as objects, storing the original square and the target square as numbers between 0 and 63. A Move object also stores the information on whether this is a promotion and the figure promoted to, and whether this is a capture and the piece captured.

3.2.3 Piece representation

Since the project adopted an object oriented approach towards the design and implementation of the chess engine, the pieces were represented as separate objects residing on the squares of the chessboard. Although most chess engines take advantage of the performance benefits resulting from minimised or primitive representation of the pieces, Mantis's rich object representation allows straightforward and intuitive board manipulation, as well as flexibility in extending the existing piece representation. That was especially useful in regard to features that were introduced late in the development process, and will definitely prove to be helpful for adding new features even after the program is released.

Each type of piece has its own class, encapsulating information about its properties. That includes individual data, such as the value of the piece (value used in the evaluation), hashing value (used for generating the board hash key), piece colour, and methods for controlling the rules of movement of the given piece. Since all pieces share most characteristics, the piece classes are the subclasses of an abstract piece class, which defines common methods and fields for all pieces. This allows maintaining a uniform interaction with all the piece objects, when the board is manipulated, during move making and reversing, as well as setting the board or performing tests on the contents of the board. The generic piece class is finally guarded by the implementation of a Piece interface, which declares general methods used by the piece classes.

3.3 Analyser

The analyser component of the engine follows the standard approach of using a depth first search algorithm based upon minimax to deliver its result. A broad overview of how minimax works is given in section 2.2. Our analyser also contains several optimisations, the most significant of which is Alpha Beta pruning.

3.3.1 Time Control

As computer chess games are normally timed, i.e. a certain number of moves must be completed within a number of minutes by each side, the analyser needs to ensure that it does not take too long in calculating the next computer move, as running out of time means losing the game.

Clearly the engine must track the time and moves remaining, and then calculate how much time should be allocated to the next move. This simplest method of doing this is to allocate the next move the time remaining divided by the moves remaining, however this solution is far from perfect. If an endgame database is being utilised, then moves near the end of the game will take a very small time, of the order of milliseconds, and so clearly it makes sense in this scenario to allocate more time to earlier moves. Further, opening moves can be extremely important in setting up a game, so it is desirable to allocate more time to these than mid-game moves. For our engine we opted for the simplest option, as this approach meant that time control would be simple to debug, and that resources could be spent on other parts of our program.

If a move that is being calculated runs out of time, then we need to be able to cancel the analyser. The precise details of how our engine does this are presented in chapter 4, however, cancellation presents a problem. When a search is cancelled, it leaves us with a move that is potentially terrible (if after being cancelled it is capable of returning a move at all), as the full analysis of the game tree has not been completed. In order to avoid this problem, and to scale easily to any timing constraints, a technique called iterative deepening is used. Iterative deepening is simply the process of sequentially starting analysers at ever increasing depth, until the time for the move in question is over. The move returned by the last analyser to complete is then the one made by the engine. For example, an analyser might be run on depth of two ply, then three, and then cancel an analysis of four ply as the time has expired; the engine would then make the move returned by the three ply analysis.

This would, at first glance, appear to waste a lot of time on analysis that is simply thrown away, however, because of the exponential increase in the time taken to analyse one ply deeper, iterative deepening generally does not produce a significant decrease in the speed of an analysis. Further, when combined with techniques such as transposition tables, iterative deepening can actually be faster than traditional fixed depth.

3.3.2 Alpha Beta Pruning

Alpha Beta pruning is the most basic and commonly deployed enhancement over naive minimax. Since its debut in 1958, it has become omnipresent within computer chess. The essence of the alpha beta algorithm is that some subtrees within a game tree are inherently redundant, and will not be

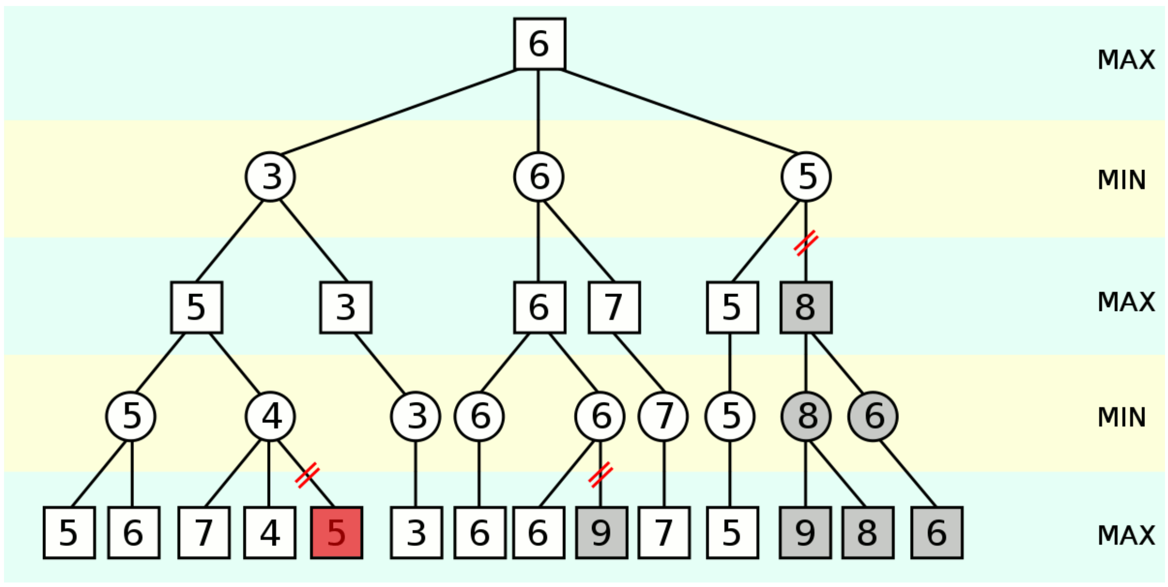


Figure 3.3: An example of alpha-beta pruning [1]

considered irrespective of their outcome. For example in figure 3.3, the subtree highlighted in red need not be analysed, as no matter what number is in place of this 5, it cannot possibly change the outcome of the minimax search. For example, if this subtree was instead analysed and produced the score 3, this would indeed change the result of the super tree, as minimisation would select it as the smallest value, however, at the next level up, a maximising node, it would be discarded anyway.

The elimination of redundant subtrees is achieved algorithmically by using two variables, alpha and beta. Alpha represents the worst possible score that the engine can be forced to take by its opponent, while Beta represents the opposite, the worst score the opponent can be forced to take by the engine. So, for example, if one subtree of a node has a score that is greater than or equal to beta, we can stop processing the node, as it is redundant.

Alpha Beta pruning generally causes a sizeable speed increase, roughly 300% at the game opening, and in spite of this will always produce the same result as the naive minimax algorithm. No other enhancement to the minimax algorithm yields such an increase in efficiency.

3.4 Evaluator

The purpose of the Evaluator is to take any given board and calculate how good the board is using a given set of heuristical evaluations. These heurstics are a set of evaluations that calculate specific things, for example the Material evaluation (see 3.4.1) takes the total value of pieces present on the board. These heuristics are gathered from the few open source chess engines that are willing to divulge the methods used to evaluate a chess board to see whether the position is a good one or not, and also from basic knowledge of chess on behalf of the group. The Evaluator extends the Board, thus meaning that it has access to all the non-private functions that Board has, and thus can use these in evaluating the chess board more efficiently and effectively. The remainder of this section covers the various heuristics that the Evaluator performs on the board, and why we perform this heuristical analysis on the board.

3.4.1 Material Evaluation

Material evaluation assigns a value to each piece on the board, and adds the value of this piece to a running total of pieces on the board[16]. Piece values have been assigned in terms of the value of a pawn. Knights are worth three pawns, Bishops are worth three and a quarter pawns, Rooks are worth five pawns, and the Queen is worth nine pawns. The King does not have a value, as the King is not a piece that can be taken by any other piece. The values that is typically assigned to a pawn is one hundred, which is just an arbitrary value, but the more important thing is that each piece is scaled in relation to the value of a pawn. In our evaluation we take the total value of the black pieces on the board, and minus the total value of the white pieces, thus giving us a positive difference if black is doing well, and a negative difference if white is doing well. This means that the evaluator always evaluates the board from the perspective of black. The evaluation will be calculated by using the protected access to the board array held within the Board class (see appendix C), and running through this array finding what pieces are left on the board, and adding or subtracting the appropriate value for each piece.

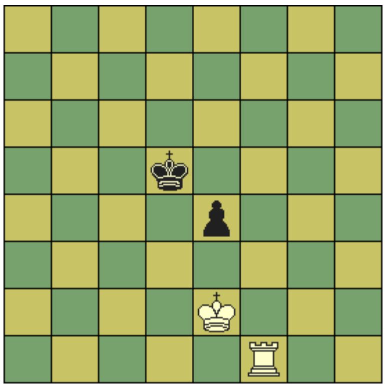


Figure 3.4: Simple Board for Material Example

The above board is a simple example of a board that allows a simple calculation of material value. Assuming a pawn has a value of one hundred, black would have a score of one hundred only this turn, as the king does not have a value (as it cannot be taken off the board). White has a score of five hundred as the rook is worth five pawns.

3.4.2 Passed Pawns

A passed pawn is one which has advanced so far that it can no longer be attacked or rammed by enemy pawns.[17]. This is desirable as it means that there is less chance of the pawn in question being captured. That alone is advantageous due to the fact that a passed pawn can then attempt to further progress to the final rank on the board, where it is promoted to another piece. A bonus equal to a whole pawn value is given to a passed pawn, in order to encourage the chess engine to actively seek to promote its pawns.

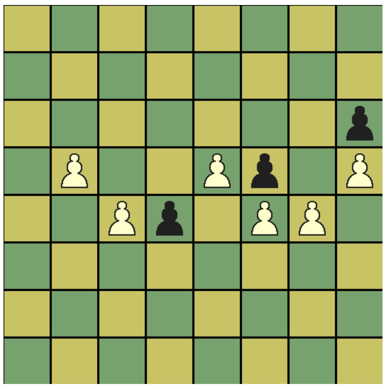


Figure 3.5: White's pawns on b5, c4, and e5 are passed pawns. Black's pawn on d4 is passed.

3.4.3 Crowded Files

Pawns are considered to be crowding each other if there are two or more pawns, of the same colour, in the same file [17]. This is usually viewed as a disadvantage to the player whose pawns are crowded, due to the fact that crowded pawns restrict each others movement on the board. Applying a penalty to a board which has pawns which are crowding each other, will encourage the chess engine from making pawn moves which result in crowding. Obviously, the greater the number of pawns in the same file, the greater the effects on the restriction of movement of those pawns. Therefore, a lesser penalty is applied for those files containing two pawns, whilst a slightly greater penalty is issued when more than two pawns are in the same file.

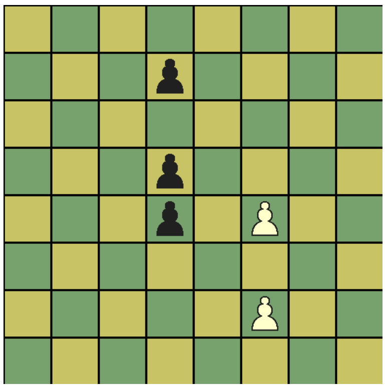


Figure 3.6: White has doubled pawns on file f, Black has tripled pawns on file d

3.4.4 Pawn Quantity

Whilst it is generally viewed as beneficial to have a lot of pieces still on the board for as much of the game as possible, there are also situations where an abundance of material, in particular pawn pieces, can be less desirable[17]. The main reason for viewing too many pawns as less advantageous is that they can clutter the board, particularly in the opening phase of the game when they restrict the movement of other pieces. However, there is a fine line between not wanting too many pawns cluttering up the board and having to few pawns, such that other, more valuable pieces become vulnerable to attack, due to being exposed. Therefore, a slight penalty is given when all eight pawns are present for a particular side, with an even slighter penalty given for having seven pawns in play. Any number of pawns less than this however is viewed as being insignificant to this particular heuristic and will be subsequently ignored.

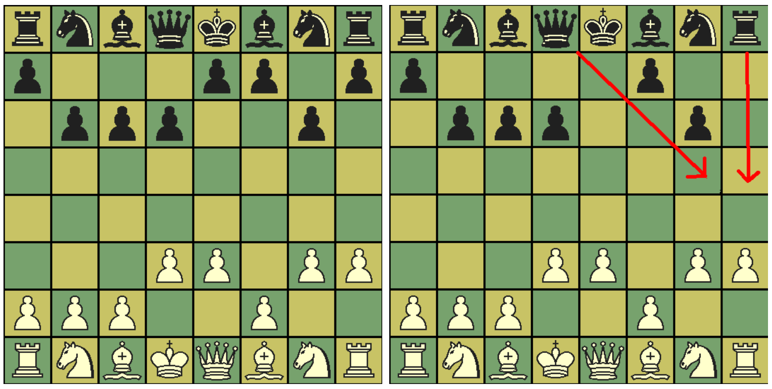


Figure 3.7: Before: Pieces sitting behind black pawns are restricted in movement. After: Black Queen can now advance diagonally, black rook can advance up file h.

3.4.5 Crippled Pawns

A crippled pawn is one which is located on either of the two outer files of the board. The notion of such a pawn being 'crippled' is due to the fact it is only able to control one file, as opposed to being able to control two files like other pawns[17] The ability to only control one file is due to the fact it can only capture opposition pieces on the inside file of that particular pawn, whereas other pawns can capture on both their inside and outside files. Therefore those pawns which are classed as crippled will be given a value which is half that of a normal pawns value.

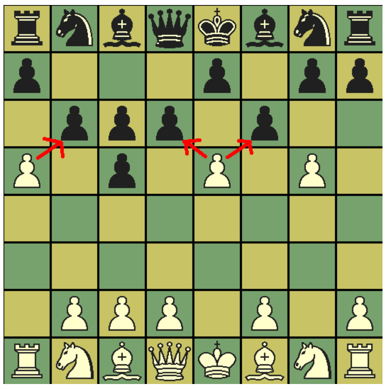


Figure 3.8: Here white pawn on a5 can only capture on inside file, whereas white pawn on e5 can capture on both its left and right hand files.

3.4.6 Isolated Pawns

An isolated pawn is a pawn that has no friendly pawns in an adjacent file. [12] Isolated pawns are weak because they have no protection from other friendly pawns, and are therefore vulnerable to attack. Pawns are awarded a bonus if they have protection from pawns that reside in either of their neighbouring files. This bonus will be set to be ten points, which should be a sufficiently high value to encourage the chess engine to keep its pawns protected no matter their position on the board. The bonus is also sufficiently low that the pawns will make a better move if such a move exists, an example would be when pawn isolation actually aids an attack on the enemy king.

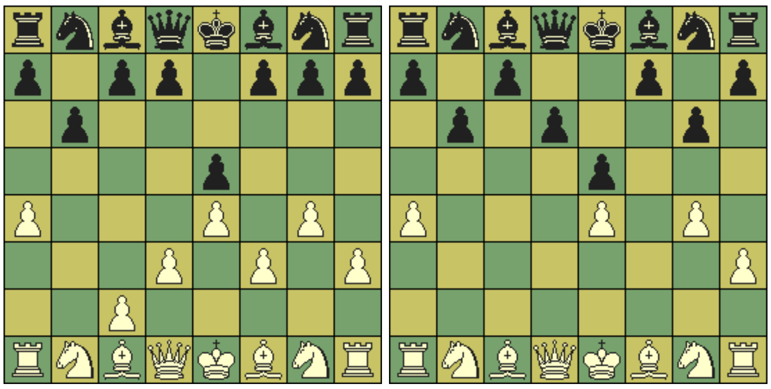


Figure 3.9: Left: Pawn in a4 is isolated Right: White has 3 pawn islands

In the figure above, the left board shows that white has an isolated pawn, the pawn in position a4. This pawn is now vulnerable to attack and therefore does not receive a bonus. The right board shows the other case that this heuristic covers, pawn islands. White has three distinct pawn islands in this board, the pawn in a4, pawn in e4 and the pawns in g4 and h3. This means that each of these pawn islands has no protection from any other pawns, and are therefore vulnerable to attack from other pieces.

3.4.7 King Pawns

The King is a vulnerable piece on the board, it being the object of the enemy's attacks, and needs to have a strong defense to protect it.[23] The King gains a good defense from having a line of pawns protecting it from enemy attacks. These pawns help stop enemy pieces directly checking the King, and create a wall that gives the King a safe position to reside, especially before the endgame situations arise. The Pawns are given a bonus of ten points for protecting their King, enough of a bonus to encourage them that protecting the King is a good move, but not too high to distort blatantly better moves from taking place.

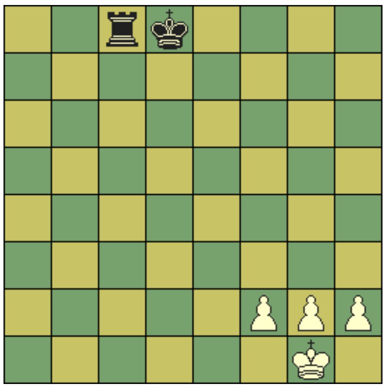


Figure 3.10: White King defended by pawns, black is not

As is seen in the above figure, the white king has defense from three pawns, guarding it from attack, whereas the black king has no such defense. Although this is a completely trivial example, it illustrates this heuristic in its entirety. In this example white would receive a bonus of 30 for the pawns protecting the king.

3.4.8 Piece Freedom

An important part of position evaluation is evaluation of piece freedom. It arises from one of the key principles in chess, piece development. Most pieces at their initial positions can make only a limited number of moves, thus reducing their value in terms of attacking ability. Additionally such pieces may become more vulnerable to attacks and can't mobilise easily to protect their position, other pieces or even their own king from the opponents attacks. It is extremely important that light pieces, i.e. bishops and knights, are developed early in the game, thus creating a positional advantage on the board. Heavy pieces, i.e. rooks and queens, should also be capable of moving freely, as these possess a critical checkmating power.

Development of figures typically results in either an increase in the number of moves the piece is capable of making or a decrease in the number of moves available to the opponent.

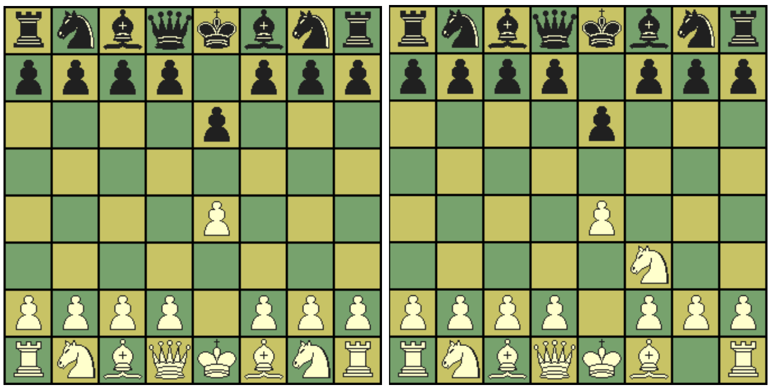


Figure 3.11: Developing knight in order to increase its freedom

Figure 3.11 shows how white knight makes a move from g1 to f3, thus making itself able to make further moves to d4,e5,g5,h4 or even back to g1. This move is better, than, for example, Nh3, as then the knight has only three squares it can move to: f4, g5 and b1. This worked well in combination with the center attack discussed in further detail in subsection 3.4.9.

Figure 3.12 shows white bishop moving from c1 to g5, thus half-pinning black knight on f6, limiting its ability to move effectively. It's worth mentioning that in this example the black side can still move the knight and even capture the pawn on e4, however this move would result in a capture of the queen on the following move Bxd8, thus creating a material in balance in favour of the white side. It is important, that the piece freedom values are chosen carefully as an overestimate may result in excessive piece sacrifices in favour of a more open position, while an underestimate will result in the engine playing excessively conservatively.

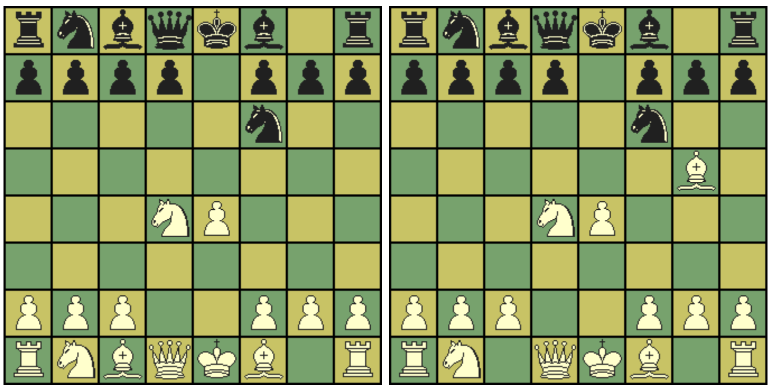


Figure 3.12: Developing a bishop, half-pinning the opponent's knight

If the piece freedom and material evaluation are balanced well, then the engine should be capable of attempting to prevent opponents moves, resulting in them gaining an advantage in piece freedom, as long as the analyser can forsee it in its game tree. Consider the following example in figure 3.13:

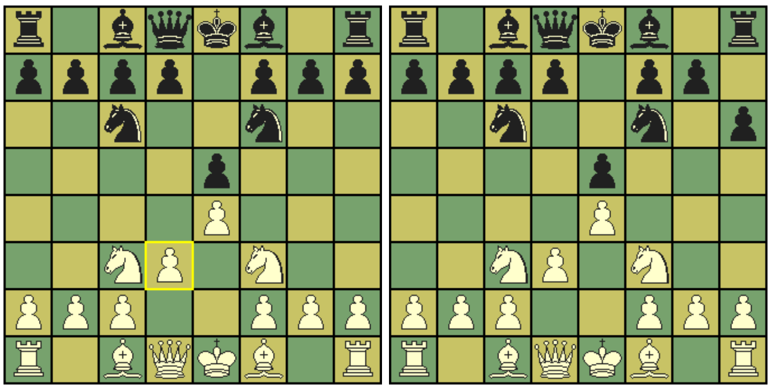


Figure 3.13: Preventing white bishop from half-pinning the black knight

Here black plays h6, thus preventing white playing Bg5. (White can still play Bg5, which is a legal move, however, provided the analyser is capable of calculating the tree to sufficient depth, the engine should catch, the following move hg, creating a material inbalance in favour of the black side, increased freedom of rook and a worsened black pawn formation.) This example shows how the piece freedom heuristic relies heavily on other heuristics in order to produce accurate results.

3.4.9 Center Control

While the ability of pieces to move freely on the board is an important factor in the chess strategy, such moves should be directed at achieving some sort of advantage over the opponent. While the main goal in any chess game is check mating the opponent's king, achieving a material advantage is commonly perceived as a good way of winning a game. While an immediate material advantage is not usually possible, this may be achieved through positioning pieces and targeting them at strategic points on the chess board. One of the key strategic parts of the board is the board center.

At their initial positions, kings are located in the middle of the board, thus a prompt and early movement of pieces towards the center of the board can enable an attack to be launched on the opponent's king. Even after the castling, pieces positioned in the center of the board can move easily to attack the opponent's king irrespective of whether the castling was performed in the short or the long side.

Typically, experienced human players find strong pawns and knights in the center to be a highly graded position, while bishops can simply point at the center and heavy figures like rooks and queens provide back-line support. Although openings exist where one side makes moves, that are not immediately targeting the center, e.g. 1.g3 or 1.b3 and their reflections from the black side's perspective, these openings tend to produce complex closed positions eventually targeting the center, however extending the length of the opening. Since the engine is designed to operate without an opening book, but to rely on its heuristical analysis, such openings are not preferred, as the complexity of the required strategy is likely to exceed the depth of its game tree.

Thus at the stage of designing the evaluator's behaviour, it's been decided to reward pawn and figure advancements towards the center, in order to acquire a good position as well as maintain an open game.

As with any other heuristic, a careful balancing is essential in order to ensure, the engine does not sacrifice its pieces unnecessarily in order to gain a seemingly strategically advantageous position in the center of the board.

3.4.10 King Attack

The King is the most vulnerable piece on the board, as he is the piece whose checkmate will lose the game for either player. Opening up attacking lines towards the enemy King is an important tactic for any chess player, and the engine is no different. By giving a small bonus to a piece for having the ability and positional advantage required to take a piece that surrounds the King, and thus opening up an weakness in the enemy Kings defense, the AI can (where appropriate) sacrifice pieces to gain a tactical advantage on the enemy King. The bonus has to be sufficiently small so that the AI will not recklessly throw away good pieces for the small tactical advantage gained from such a move, but also sufficiently large that when appropriate, the AI will sacrifice a piece to perhaps gain a checkmate, or remove a valuable enemy piece. The bonus will be set to a fiftieth of a pawns value, a very small bonus but an important one nonetheless. The heuristic checks if any given piece has *a view* on the enemy King, or more precisely can it make a move that would potentially open up an attack on the enemy King. This is used because it means that the pieces will receive a bonus for having a good position on the board, without actually having had to make a sacrifice when there was no good reason to do so.

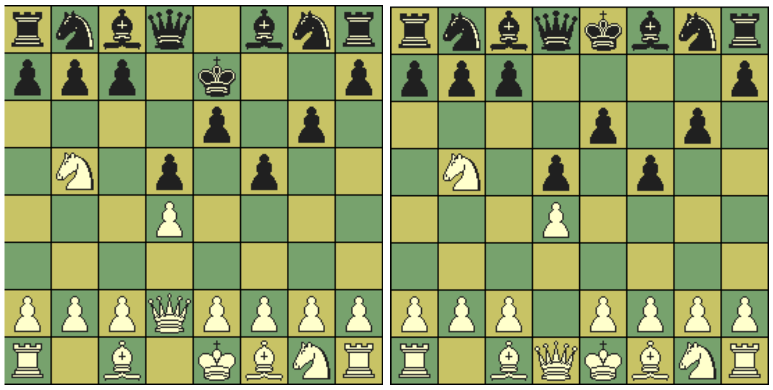


Figure 3.14: Left: Attack on Inner Aura of King Right: Attack on Outer Aura

The King is assumed to have two rings around him that are important positions in opening up an attack on the King, the first and second aura. In the example on the left it shows the King having an attack on the first aura around the king, thus granting white the bonus. The example on the right, it shows the white knight this time attacking the second aura that surrounds the king, and thus receives a smaller bonus of a hundredth of the value of a pawn, as although this position is a good position, it is not as important as the attack on the first aura around the king. Both these examples show how opening up an attack on the king can lead to a greater chance of their being checkmates, and thus the engine winning the game.

3.4.11 Center King

An important Heuristic to consider when trying to protect the King, is the position of the King on the board. During the opening and middle game it is important that the King is protected and remains away from the exposed center of the board. This is due to the high concentration of pieces that can move to the middle of the board and the fact that there are still many pieces left on the enemy side. Conversely the King should head to the center of the board at the end game, as king safety is no longer a major issue in the endgame, but a centralised king is more powerful than one at the edge of the board during this later phase of the game. This means that the AI will have to use the mechanism for detecting what part of the game we are currently in (start, middle, or end game) and act accordingly about this (see 3.4.13). This multiplier will be used to change the bonus from giving the King residing outwith the center in the start game a good score, to giving a bonus for the king residing in the center in the end game. This bonus is equal to ten points, but this is multiplied by the end game multiplier from 3.4.13.

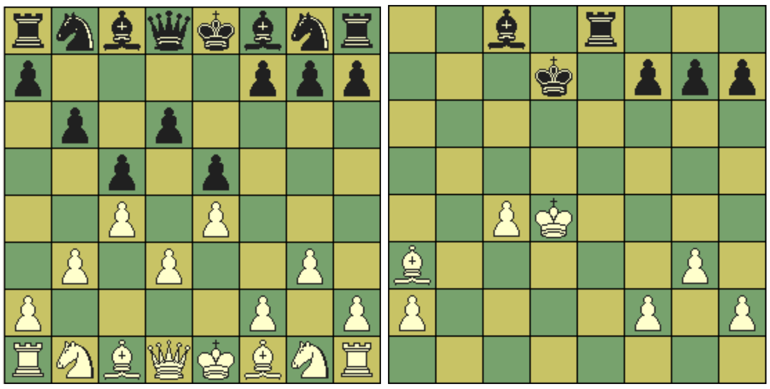


Figure 3.15: Examples of King Center heuristic

In the example above, the left example shows the early game, where the white King is safely away from the center of the board. In this early stage of the game it is important to keep the King from the center, so the white King would be given a bonus for this behaviour. The example on the right shows the white King positioning himself in the center of the board, and since the game that is in progress is in the end game stage (near to it is completion) this is a good position for the king to attain. The white king would receive the appropriate bonus for gaining this good position, so late in the game.

3.4.12 Has Castled

The has castled heuristic is an important test for the mobility of the King[23]. Castling can provide a much better defensive position for the King in certain situations, and the ability to castle means that the King will always have this option available to him. This heuristic will encourage the King to castle, as it will give him a bonus of ten points if he has castled on the board. This means that the King is then encouraged to castle, but the bonus is not to great to discourage other potentially better moves.

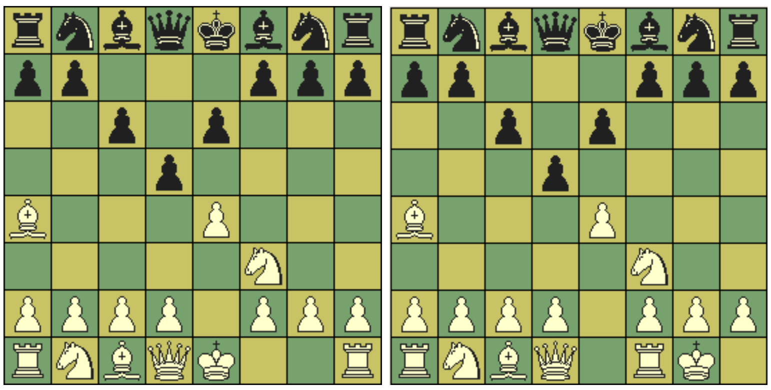


Figure 3.16: Examples of castling

The example on the left shows the white king having the ability to castle, but not having actually castled. Thus it will not have the bonus applied to it. The example on the right shows the white king having castled, and thus for the remainder of the game white will gain the bonus of ten points.

3.4.13 End Game Multiplier

The end-game multiplier is used to determine what state the game is currently in. It is important for the Evaluator to know what state the game is in, for it to be able to change its behaviour to best suit the state of the game. For example, it is worth more to have a passed pawn in the end game than it is at any other stage. A passed pawn before the end game has a much greater chance of being taken, and thus makes it a vulnerability. But during the end game when there are significantly less pieces on the board it becomes a great bonus, as this pawn has a much greater chance of promotion if it is a passed pawn, and a much lower chance of being taken as there are less pieces remaining on the board that can take it. The end game multiplier should be a floating point value that ranges between 0.0 and 1.0 respectively. The value will be calculated by taking the total value of material left on the board, and dividing it by total value of material on the board at the start of the game. By dividing the current value of the board by eighty pawns we will get a value between 0.0 and 1.0, which is what we wanted.

Chapter 4

Implementation

4.1 Overview

4.1.1 Language Choice

Although Java is over twice as slow as C and other compiled imperative languages [11], it was chosen for this project as it offered a well structured, modular and object oriented paradigm by which a good, clear design could be formed. As the primary reason for the project was to serve as an educational exercise, this was thought a preferable trade-off for speed. Most chess engines, whose remit is to be as fast and efficient as possible, have chosen C.

4.1.2 Language Style

After choosing Java as the language to write in, see 4.1.1, the group agreed on one coding style and wrote up formal documentation declaring it (see appendix E) so that it would be easy for one group member to understand another's code.

4.1.3 Game Class

The final implementation made use of a hub class, from which everything else derives, and which deals with the distribution of data. The Game class is this hub, and is also the driver of the engine.

After initialising the other parts of the system, it sits in an infinite loop for the life of the engine, just handling the messages from XBoardIO and passing them as needed.

4.1.4 Communication between objects

The communication between objects was possibly the most laboured implementation decision, with a great amount of time poured into discussion over the merits and drawbacks or different designs.

Discussion revolved around which objects should have knowledge of which, and which attributes of each object should be free to be manipulated directly by other objects. Finally, a compromise approach was taken, balancing the object orientation of the program with tight coupling of certain components (the board, piece and move representation, and weaker links to these from the analyser and the evaluator), which is typical for chess applications, in order to gain efficiency.

4.1.5 Threads

Originally the engine was written as a single-threaded unit, but due to the wealth of information in need of processing in a asynchronous and timely fashion, two extra threads were added. Having chosen to write the engine to conform to the Chess Engine Communication Protocol (see 4.1.6) as well as constantly being able to respond to interrupts required a second thread to handle Input and Output (see 3.1.2). As time controls (see 3.3.1) were introduced another thread was introduced.

4.1.6 Chess Engine Communication Protocol versus Bespoke UI

Early on in the project, discussion ensued regarding the scope of the project. It was agreed that as much time as possible should be spent on the engine, and a standard user interface would be 'plugged in'. Our own bespoke interface would allow flexibility in the design, but would take time away from the engine itself and thus weaken it.

Although implied by the choice of a pre-written UI, it was seperately decided that the engine should conform to the Chess Engine Communications Protocol so that our engine could play other engines for testing.

4.2 Board

The board representation was designed in a flexible fashion with a Java interface on the top, which contains the set of methods that are necessary for implementing a working, digital chessboard. These methods can be implemented with any board data structure the developer chooses to use, as long as they conform with the declarations in the interface. This approach allows the developer to swap the used board representation for a new one, even if it is completely different from the original solution.

Although it is very easy to change the board representation, its implementation must work closely with other components of the engine, as they are tightly coupled with the board structure, namely the move representation (Moveable) and the chess piece representation (Piece). This practice goes against the rules of object oriented programming, however it was a necessary solution, adapted in virtually all chess engines. It increases the efficiency of the board operations and hence the performance of the analyser, which uses the board operations hundreds of thousands of times, when searching the game tree.

The methods that ought to be implemented for every used board representation are:

- boolean makeMove(Moveable m) - Performing the given piece move on the board structure
- boolean reverseMove(Moveable m) - Reversing the given piece move on the board structure
- boolean isLegalMove(Moveable m) - Test if the performed move is correct, according to chess rules
- boolean isStalemate() - Test if the current situation on the board is a stalemate
- boolean isCheckmate() - Test if the current situation on the board is a checkmate
- boolean isInCheck(boolean colour) - Test if the king of the given side is in check. The notation adapted by the board uses true to denote white side and false to denote black side
- boolean isWhiteToPlay() - Test which side is playing at the moment, returning true if it is white and false if it is black
- void generateHashCodes() - Generates random codes for hashing values, used for Zobrist hash key generation and creating the initial hash key of the board
- void generateZobristKey(Moveable m) - Generates a new hash key for the board, on the basis of the performed move
- List<Moveable> generateLegalMoves() - Generates a list of legal moves that can be made in the current board situation

Each board representation also contains custom private methods, which assist the public methods in performing their tasks. The most important of these private methods, the public methods, the currently used board data structure, and the specifics of the move and piece representations implementations will be discussed in detail in the following sections.

4.2.1 Board representation

Although the design of the engine was based on an object oriented approach, the implementation of the board representation had to embrace more than that. By its nature, the board is tightly coupled with the representation of the chess pieces and the moves, in practice making all of these a part of a single segment of the engine. The result of that is also the length of the board representation class, which makes it the longest class in the application. However, these aspects of the program, which can also be found in other chess engines, do not compromise the tidy and elegant, object oriented design of the application, which makes it very flexible and easy to manipulate.

The 0x88 board was implemented according to the popular solution of declaring an array of 128 elements, each representing a single square on the real or dummy board. The introduction of this data structure implied taking advantage of its properties in terms of performing efficient binary operations and simple additions for piece movement. Such operations were used in different methods, implemented as the part of the board, as well as the pieces' representation. The array houses objects of type Piece, which make it easy to manipulate or access the piece information, calling piece methods in the same way for all types of pieces.

Apart from the board data structure, certain data is recorded in the private fields of the board class. That includes isWhiteToPlay field, which is used for keeping track of the playing side. If the field holds true, the white side is playing, and if false, the black side is. The board also tracks the positions of the black and white kings, which are used in tests for king checks, checkmates, stalemates and castling. Many factors, such as already mentioned checkmates and stalemate, rely on the data about the currently available set of legal moves, additionally a dynamic collection of all legal moves for the current playing side, for the current situation on the board, is also recorded. It was implemented as a Java LinkedList.

The methods operating within the board class can be divided into three groups: constructors, accessors and modifiers. There are three constructors for the board, which initialize the board data structure in distinct ways. The parameterless constructor creates a standard game with a typical positioning of black and white pieces on the opposite sides of the chess board. The other two constructors are responsible for setting the board to a custom state, on the basis of a provided string, representing the state of each of the 64 board squares, or a 64 element array of Piece objects. The string constructor is used for loading custom board situations, for instance allowing the user to load a previously saved game or playing a non standard game (for example chess riddles). It was also extensively used, along with the piece array constructor, to test the functionality of the engine.

The accessors in the board are different methods dedicated to extracting information from the elements of the board representation or testing the state of these. The collection of operations focused on testing the legality of performed moves are a part of this group. That set of methods is discussed in more detail in the section 4.2.4. There are also smaller, private accessors, whose purpose is to aid the generally used methods in performing their tasks.

The most important modifiers are the methods responsible for performing a given move and reversing the move. These methods directly manipulate the contents of the board data structure and their usage determines how the analyser and evaluator see the board and then, how the user does. Due to their direct interaction with the array representing the actual board state, most of the bugs encountered during the development process were caused by these methods. That is because an incorrect manipulation of the board array corrupts its contents, making them inconsistent. That results in engine crashes, as inconsistencies are propagated to the thousands of operations performed by the analyser, during a search. One such critical bug, nicknamed ”The Knight Bug” was a result of only partially tracking of board changes related to castling, which were exposed by an awkward behaviour of knights. The modifier methods are described in depth in the section 4.2.9.

4.2.2 Move representation

As discussed in section 3.3.2 in order to decouple the code as much as possible (having already tightly coupled certain segments of the engine), a decision was made to implement move objects, so they would contain a 64 square representation of the board, which does not directly correlate with the 0x88 internal board representation, discussed in section 3.3.1. This decision was conscious as the Move class should make no assumptions about what is the internal representation of the board. Additionally, the Move object is also shared by the Game class and even referred to by the evaluator, so a complete decoupling was essential here.

The most significant drawback in this case is a necessity to convert between the 64 and 128 square representation. This can be solved by implementing two functions as outlined in Algorithms 1 and 2:

```
Algorithm 1 Map a 64 square board position on to a 128 square representation
Require: a is a number between 0..63 corresponding to a square number on a regular board
Ensure: b is a square number in a 128 square representation between 0..128
  $b \leftarrow \lfloor a/8 \rfloor * 16 + (a\, mod\, 8)$
```

```
Algorithm 2 Map a legal square in 0x88 representation on to a 64 square board
Require: a is a number between 0..127 corresponding to a legal square number on a board in 0x88
  representation
Ensure: b is a number between 0..63 corresponding to the related position on a real board
  $b \leftarrow \lfloor a/16 \rfloor * 8 + (a\, mod\, 8)$
```

As mentioned above, the Move object is not supposed to make any assumptions about the board representation, so neither of these two functions were implemented as methods of the Move object, instead they were implemented within the BoardArray class, responsible for manipulations with the board, however optimisation had to be taken into account at the early stage of making implementation decisions. Unfortunately all of the operations required for the above algorithms are computationally expensive in terms of the required processor cycles. Due to Move objects being a center piece of the entire engine a special care had to be taken to find balance between a good object oriented and decoupled design and a reasonable level of efficiency. Luckily, these form a special case, since all of the involved constants are powers of two. Due to this fact, all of the above costly operations can be reduced to just a few inexpensive binary operations. To remind the reader, $i >> 1$ means to do a bit shift to the right by 1 bit, which also implies a division by 2, and so on for each successful bit shift. Conversely $i << 1$ implies multiplication by 2. Similarly, to find *a%b* if *b* is some power of 2, it's sufficient to compute *a&(b-1)*, where '&' means 'binary and'. The proof of mathematical correctness is beyond the scope of this document, however this is guaranteed to work for any unsigned integer and has been tested for all integers in the concerned range.

In the view of the above optimisations, the algorithm, converting from a 64 to a 128 square representation [1] can be optimised to *(*$a >> 3$*)* $<< 4 + a\&7$. Similarly, its converses [2] implementation optimises to *(*$a >> 4$*)* $<< 3 + a\&15$. Each of these formulae now have a cost of just 4 binary operations, which is negligible.

In order to improve the maintainability of the application a decision was made to pass around objects of type Moveable, an interface implemented by class Move, so to allow for a choice to change the representation of Move objects at a later stage with the minimum number of changes to the code, limited only to them parts of it, where the Move object is instantiated.

Initially the Move class was equipped with just a few basic routines, including a constructor, accessors for the square numbers corresponding to the initial and final positions on the board and a number of accessors and mutators for private states, i.e. promotion and capture. As the number of implemented rules expanded, in order to implement support for complex rules, like castling and *en passant* a few more private states and corresponding accessors and mutators had to be added, including rook position before and after the castling and the position to which the capture needs to be returned, should the move need to be reversed. As suggested in the previous paragraph, when choosing methods for the Move object, a special care had to be taken to ensure, that every move could be accurately reversed in an efficient manner. The reason for support for reversals will be discussed in full detail in relevant parts of this report.

4.2.3 Piece representation

Similarly to the board representation, the piece is originally represented by an interface providing a set of method declarations, which can be reused for different implementations of the piece classes. By creating the requirement of implementation of specific methods, we achieved the enforcment of necessary degree of consistency, which is the factor that allows to easily switch to a completely different piece representation, if desired. It is possible, because of the advantages of the object oriented design and modular apporach, which we have followed. The methods of the standard Piece interface are:

- boolean isLegalMove(Moveable) - Tests if the move indicated by the given delta is legal, according to the movement rules of the piece, which calls the method. True is returned if the move is legal, false if it is not
- boolean hasMoved () - Tests if piece has already moved, by checking if the move counter is greater than 0
- void setMoved () - Increment the recorded number of moves the calling piece made
- boolean isWhite() - Tests if the calling piece is of white colour, by returning true if that is the case and false if it is black
- int getValue() - Extracts the value of the piece, which is used in evaluation
- void unsetMoved() - Decrements the recorded number of moves the calling piece made
- List <Integer> getDeltas() - Extract the possible movement patterns on the board in terms of deltas
- void setColour(boolean) - Sets the colour of the piece object on the basis of the provided boolean (true stands for white, false stands for black). It is used for board setup and promotions, as well, as restoring pieces on the board when capture moves are reversed
- int getID() - Extract the value representing the type of the piece, used in Zobrist hashing

Although not directly part of the board classes, the piece representation is tightly coupled with the board, by relying on the board data structure. As explained before, such tight coupling increases the efficiency of the chess engine, which is crucial for achieveing a better gameplay. That makes the piece representation an integral part of the board focused block of the program.

The piece interface is implemented by a single abstract class GenericPiece, which encapsulates the common properties of all of the chess pieces found on a chessboard. These are:

- protected int hashID - Stores the value representing the type of piece, used in Zobrist hashing
- protected short value - Stores the value of the piece, which is used in the evaluation
- protected boolean colour - Stores the colour of the side the piece belongs to, with true being the white side and false being the black side
- protected int moves - Stores the number of moves the piece made so far

Since these fields are protected, they are inherited by every subclass that extends the abstract GenericPiece class. This also includes an implementation of the basic methods, which work in the same way for every piece. Most of them such as, isWhite() or getID() consist of a single field return statement, others increment or decrement the appropriate counters. The unsetMoved() methods includes an additional guard test, which does not allow the count of moves to drop below 0, as it cannot happen. The isLegalMove(int delta) and getDeltas() methods are not implemented as a part of the generic class, as their bodies are specific for each piece type.

Every piece type is represented by a seperate subclass, extending the GenericPiece abstract class. This approach equips all the pieces with the basic piece functionality, as described above, and only leaves the implementation of piece specific methods up to the piece related subclass. There are six pieces that can be found on the chessboard: king, queen, rook, bishop, knight and pawn. Apart from the implementation of isLegalMove(int delta) and getDeltas(), the classes for these pieces contain a standard constructor, which accepts the boolean value of colour as its only argument. The constructors initialise the standard fields for every piece, namely the integer moves (always set to 0 when the piece is created), the boolean colour (assigned the value of the given argument), the integer hashID (piece type hash value) and the short integer value (piece evaluation value). These two fields are individual for each piece:

Piece type	value	hashID
King	1000	1
Queen	900	2
Rook	500	3
Bishop	325	4
Knight	300	5
Pawn	100	6

Each class also contains a parameterless constructor, setting only the value and the number of moves made for the piece. These are used for special tasks within the evaluator and the analyser. The information on implementation of the movement rules for all the pieces is shown in Algorithms 3 - 8. The algorithms are accompanied by figures 4.1 - 4.6, which picture the movement capabilities of each piece, with the deltas required for the given move presented on the chessboard.

Algorithm 3 King movement rules

```
$adelta \leftarrow Abs(delta)$
if $adelta == 1$ then
    return true {Left up and right down jumps}
end if
if $adelta == 15$ then
    return true {Right up and left down jumps}
end if
if $adelta == 16$ then
    return true {Up left and down right jumps}
end if
if $adelta == 17$ then
    return true {Up right and down left jumps}
end if
if $adelta == 2$ then
    if piece has not moved yet then
        return true {Castling control check, double field castling jump}
    end if
end if
return false {Attempted move is not legal for a king type piece}
```

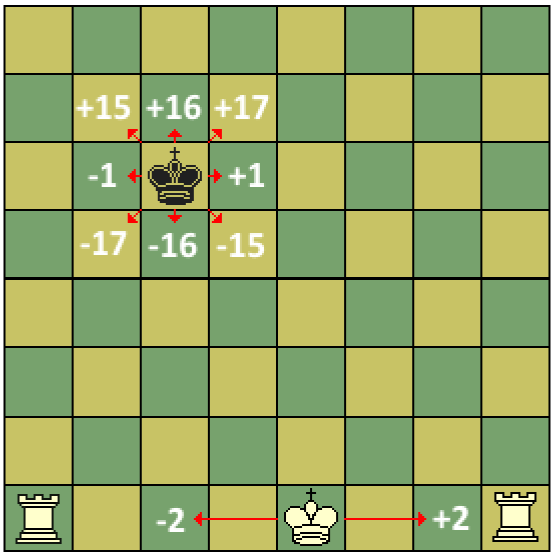


Figure 4.1: Possible moves for a king piece

Algorithm 4 Queen movement rules

```
if Abs($delta$) $< 8$ then
    return true {Horizontal movement, up to 7 squares}
end if
if $delta\ AND\ 15 == 0$ then
    return true {Vertical movement, up to 7 squares}
end if
if $delta\ MOD\ 15 == 0$ then
    return true {Left diagonal movement, up to 7 squares}
end if
if $delta\ MOD\ 17 == 0$ then
    return true {Right diagonal movement, up to 7 squares}
end if
return false {Attempted move is not legal for a queen type piece}
```

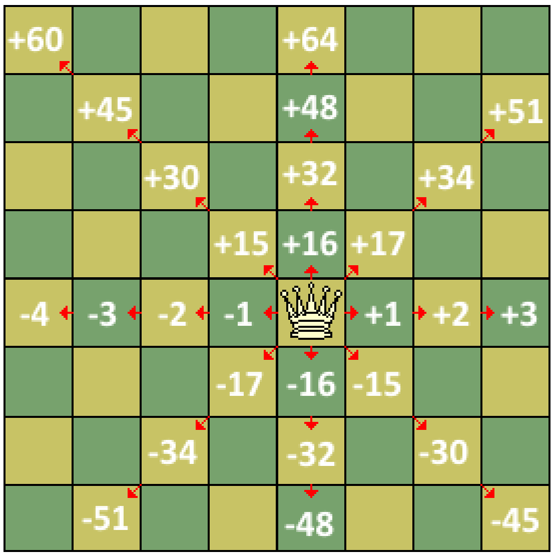


Figure 4.2: Possible moves for a queen piece

Algorithm 5 Rook movement rules

```
if $delta\ AND\ 15 == 0$ then
    return true {Vertical movement, up to 7 squares}
end if
if $Abs(delta) < 8$ then
    return true {Horizontal movement, up to 7 squares}
end if
return false {Attempted move is not legal for a rook type piece}
```

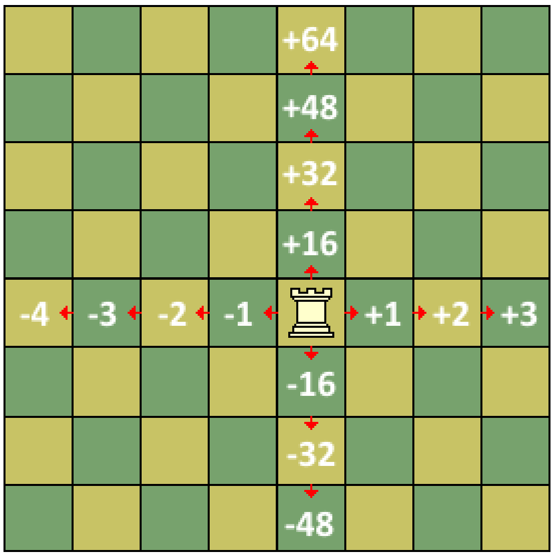


Figure 4.3: Possible moves for a rook piece

Algorithm 6 Bishop movement rules

```
if $delta\ MOD\ 15 == 0$ then
    return true {Left diagonal movement, up to 7 squares}
end if
if $delta\ MOD\ 17 == 0$ then
    return true {Right diagonal movement, up to 7 squares}
end if
return false {Attempted move is not legal for a bishop type piece}
```

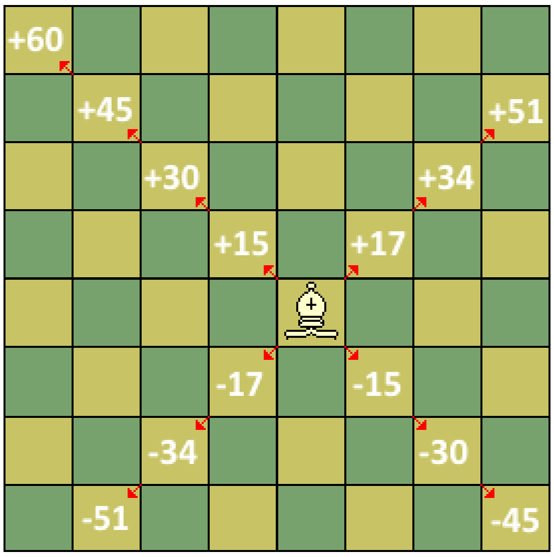


Figure 4.4: Possible moves for a bishop piece

Algorithm 7 Knight movement rules

```
$adelta \leftarrow Abs(delta)$
**if** $adelta == 14$ **then**
    **return true** {**Left up and right down jumps**}
**end if**
**if** $adelta == 18$ **then**
    **return true** {**Right up and left down jumps**}
**end if**
**if** $adelta == 31$ **then**
    **return true** {**Up left and down right jumps**}
**end if**
**if** $adelta == 33$ **then**
    **return true** {**Up right and down left jumps**}
**end if**
**return false** {**Attempted move is not legal for a knight type piece**}
```

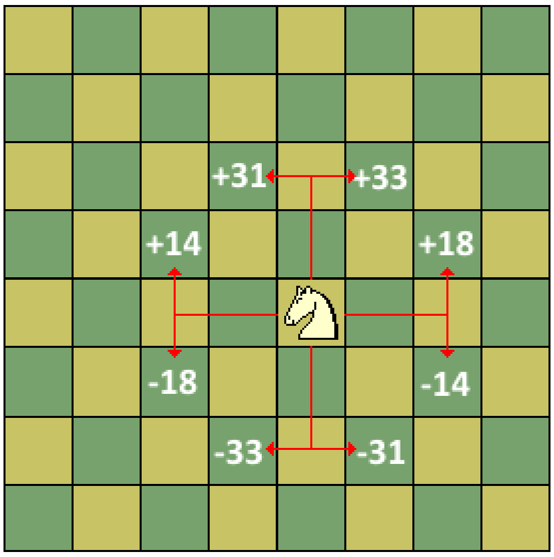


Figure 4.5: Possible moves for a knight piece

Algorithm 8 Pawn movement rules

```
if (piece is white AND delta $< 0$) OR (piece is black AND delta $> 0$) then {Test if piece moved in the right direction}
    return false
end if
$adelta \leftarrow Abs(delta)$
if $adelta$ == 32 AND piece has not moved then {Test if piece not moved, hence allowed a two field jump}
    return true {
end if}
if $adelta$ == 16 then {Test if piece moved, hence allowed a single field jump}
    return true
end if
if ($delta$ == 15 OR $delta$ == 17) then {Test for piece capture}
    return true
end if
return false {Attempted move is not legal for pawn type piece}
```

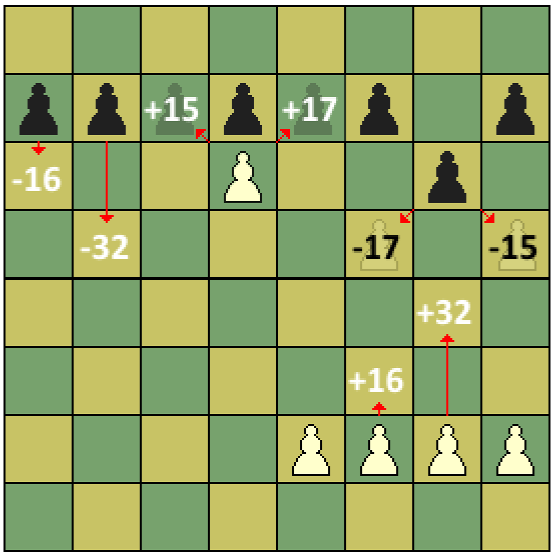


Figure 4.6: Possible moves for a pawn piece

4.2.4 Move legality

Since a game of chess relies on specific movement rules for different pieces, as well as on specific situations on the chessboard, the implementation of the enforcment of these rules is critical for building a working chess engine. Because of the numbers of these rules and their importance, the tests are spread across the piece classes and the board representation methods, which are calling the piece rule test methods and test the consistency of the board.

The piece classes contain methods for testing the legality of a move by comparing it with the legal movement patterns for the given piece. This encapsulation allows to call these methods from the board class in exactly the same way for any type of piece, which is one of many advantages of the object oriented construction of the application. The details of the implementation of the movement rules are described in the previous section (see 4.2.3).

The board class contains a number of methods responsible for testing different rules of chess. These tests are performed before the move is executed on the board, so that the validity of the move is confirmed before any manipulation of the board takes place. These methods include:

- boolean isLegalMove(Moveable)
- boolean isPathBlocked(int, int)
- boolean isInCheck(boolean)
- boolean isCheckmate()
- boolean isStalemate()
- boolean canCastle(Moveable)
- boolean isPlayerTurn()

Most of the tests are located within the boolean isLegalMove(Moveable) method, which returns true if all of these tests are passed correctly. If any of the tests detects a flaw in terms of chess logic it exposes an incorrect move by terminating the method and returning false. The tests conducted within the isLegalMove method include:

- is the move made outside the boundaries of the board (off board)
- is the piece moved to the field where it is currently standing
- is the piece of the wrong colour moved
- is the move an attempt of capturing one's own piece
- is the move legal, according to the movement pattern of the moved piece
- if the piece is a pawn, is it trying to capture a piece by moving forward, instead of moving on diagonal
- if the piece is a pawn, is it trying to capture an empty square by moving on diagonal
- if the piece is not a knight, is the movement path of the piece blocked by other pieces
- does the move put the moving side's king in check

As mentioned before, the movement pattern validity is confirmed by calling a local isLegalMove(int delta) method for the piece involved in the move. There is also another call made to a seperate board class, private method for the purpose of controlling the movement path of the piece (boolean isPathBlocked(int, int)). This method tests each square on the piece's way, as well as the destination square, for presence of other, friendly or foe, pieces. If the route is blocked, the move cannot be made. The implementation is based on the pseudocode of algorithm 10.

Algorithm 9 Blocked movement path test

```
Ensure: $board$ is the board data structure
Ensure: $finalSquare$ is the target board square of the move
Ensure: $originSquare$ is the original position of the moved piece
  $delta \leftarrow targetSquare - originSquare$
  if $delta > 0$ then {Piece moving to higher ranks}
    $direction \leftarrow 1$
  else {Piece moving to lower ranks}
    $direction \leftarrow -1$
  end if
  if ($delta$ AND 15) == 0 then {Vertical path control}
    for $i$ in $originSquare$+($direction$*16) to $targetSquare$ advancing by $direction$*16 do
    {Consecutive squares in the given direction}
      if there is a piece on square $board[i]$ then
        return true
      end if
    end for
  end if
  if (Abs($delta$) < 8 then {Horizontal path control}
    for $i$ in $originSquare$+$direction$ to $targetSquare$ advancing by $direction$ do {Consecutive
    squares in the given direction}
      if there is a piece on square board[i] then
        return true
      end if
    end for
  end if
  if ($delta$ MOD 15) == 0 then {Left diagonal path control}
    for $i$ in $originSquare$+($direction$*15) to $targetSquare$ advancing by $direction$*15 do
    {Consecutive squares in the given direction}
      if there is a piece on square board[i] then
        return true
      end if
    end for
  end if
  if ($delta$ MOD 17) == 0 then {Right diagonal path control}
    for $i$ in originSquare+(direction*17) to $targetSquare$ advancing by $direction$*17 do
    {Consecutive squares in the given direction}
      if there is a piece on square board[i] then
        return true
      end if
    end for
  end if
  return false {Path is not blocked}
```

The tests performed within the boolean isPathBlocked(int) method were reasonably easy to implement and are also quite efficient, due to the properties of the 0x88 board representation. Another method called as a part of boolean isLegalMove(Moveable) is the boolean isInCheck(boolean) method, which investigates the situation of the king for the given side colour. The set of all possible legal moves of the opposing side is tested against the position of the king of the given side (according to board notation, true for white side, false for black side) and if any of them is a potential capture, a check is confirmed.

The boolean isLegalMove(Moveable) itself and the rest of move legality control facilities are called from the boolean makeMove(Moveable) method, which is responsbile for performing the move from the XBoard interface (human player's move) or from the analyser (computer player's move). Although it seems reasonable to hide all the rules' tests in one method and then call the method once, it turned out to be impossible. Such a structure of the application would require the board to be manipulated several times, with moves performed more than once. Hence, castling and promotion checks are executed in the makeMove method.

Although most of the move legality tests are grouped together, the checks for an instance of a stalemate or a checkmate are performed elsewhere, in order to keep track of the game state appropriately. The stalemate and checkmate methods rely on calling the boolean isInCheck(boolean) method and testing the number of possible, legal moves available. If the king is in check and there are no legal moves possible, the game ends with a checkmate. If the king is not in check, but there are no moves to be made, the game ends with a stalemate.

4.2.5 Checks, checkmates and stalemates

Obviously, the desired outcome in any game, is to defeat your opponent. The way this is done in the game of chess is by 'checkmating' you opponent's king. The king is in checkmate when he finds himself in a position, whereby he is able to be captured by an opposing piece should he remain there, but is unable to legally move himself to a position where he would evade capture[4]. When the king is in a position, whereby he can be captured by an opposing player's piece, that king is said to be 'in check'. There is also the ability for a player to draw with their opponent. This is known as stalemate and occurs when a player is not currently in check, but is unable to make a legal move, without putting their king in check, as a result.

Our engine had to be able to recognise the end of a game, therefore it was necessary to give it methods which would allow it to recognise checkmates and stalemates, whilst trying to play for them also. Although both checkmate and stalemate are fairly straightforward algorithms in themselves, they both rely on the notion of whether or not the king is in check. Allowing the chess engine to have the ability to determine whether or not the board was in a state of 'check', required careful consideration, in terms of implementation.

The first problem to consider was the fact that a state of check would need to be considered from various 'views' or potential scenarios. Three potential scenarios were identified. Firstly, the most basic check was to determine whether the board was currently in check at that time. The second notion to consider, was checking to see if moving the King would be legal, on that given board i.e the move would not result in the King moving into check. The third scenario highlighted for consideration, was ensuring moving an arbitrary piece on the board, which wasn't the king, wouldn't result in the King finding himself in check.

As a result of highlighting three separate scenarios which could potentially lead to the board being in a state of check, it was concluded that an overloaded method would be the best way to implement the isInCheck method. However, the main body of each of the three oveloaded methods were the same in principle, performing the same activities for each. Therefore, one of the three overloaded methods contains the main body of code, whilst the other two methods simply call this third method, passing it default parameter values where appropriate. This third method takes, as its parameters, a variable indicationg which side's turn it is to play, followed by both a from and a two position, indicating the move of a piece from one position to another. Therefore the most basic method simply deals with the case when checking to see if the board is currently in check. As it does not involve any pieces moving, this method passes a large negative value as its second and third parameters, in its call to the main check method. Similarly, the second of the three methods, takes a value coinciding with the square that the king is considering moving to. It then passes this value, plus a large negative value, as its second and third parameters, to the main third procedure.

The algorithm itself, whilst being fairly lengthy, is quite straightforward in principle. The function requires a variable denoting the colour of the side whose turn it is to play. It also requires two other variables, associated with squares on the board in the case when considering either a king or other arbitrary move, or simply a default large negative value. The premise of the algorithm is as follows: Pretend it's the other side's turn to play, then, playing as that side, attempt to make a move which would capture the original side's king. If such a move is possible, then the original side's king is in check. Any king or other pieces that were moved on the board are reversed, as is the parameter denoting which side's turn it is to play. The methods which check for stalemate and checkmate are easily implemented on the back of the isInCheck method. Both the checkmate and stalemate methods call the above method to test for check. If this returns true, both then perform a check to determine whether there are any legal moves available to the side in question. If there are legal moves available, then this is stalemate, if not then checkmate occurs.

4.2.6 Castling

Two methods control whether castling is possible: isCastling identifies whether the current move is an attempt at castling, then canCastle determines whether such an attempt is possible.

isCastling

isCastling is a crude attempt at determining whether the current move is an attempt at castling. As it is considered for every move made, it must return false quickly for those moves that aren't concerned with castling.

It firstly checks whether the piece to be moved is a king, then if it is, if the move is greater than one square left or right it deems this move to be a castling attempt. At this point it is assumed that move attempts from the king greater than one square are two squares, as moves greater than this have already been ruled out by a validity check against the piece's deltas.

Algorithm 10 isInCheck() Method

```
Ensure: $sideToPlay \leftarrow !sideToPlay$ pretend it's other side's turn to play.
Ensure: $toSquare$ the second parameter passed to this method
Ensure: $fromSquare$ the third parameter passed to this method
  $kingpos$
  $pieceMoved \leftarrow$ false
  $inCheck$
  if $toSquare > -\infty$ then {Signifies case 2 i.e a King potential king move}
    $kingpos \leftarrow toSquare$
  else if $fromSquare > -\infty$ then {Signifies case 3 i.e a potential arbitrary move is being considered}
    Make the move from the square in $fromSquare$ to the square in $toSquare$
    $pieceMoved \leftarrow true$
    $kingpos \leftarrow sideToPlay$ king's board position
  else {Signifies that a basic check for 'check' is being considered}
    $kingpos \leftarrow sideToPlay$ king's board position
  end if
  for each Piece $p$ on Board $b$ do {Run through each square on board}
    if $p$ is same colour as $sideToPlay$ then {only consider pieces of opposite colour to the colour of the king in question}
      make new move $m$ from position $pos$ of $p$ to $kingpos$
      if $m$ is legal then {check this move for legality}
        $inCheck \leftarrow$ true
        if $piecemoved =$ true then {consider case when an arbitrary piece was moved on board}
          Reverse the move from $toSquare$ to $fromSquare$
        end if
        $sideToPlay \leftarrow !sideToPlay$
        return $isInCheck \leftarrow$ true
      end if
    end if
  end for
  {No legal move can be made by opposition which would allow them to capture the king}
  $sideToPlay \leftarrow !sideToPlay$
  if $piecemoved =$ true then {consider case when an arbitrary piece was moved on board}
    Reverse the move from $fromSquare$ to $toSquare$
  end if
  return $isInCheck \leftarrow$ false
```

canCastle

If the move passes the isCastling criteria, the legality of castling given the current board state is then fully tested. This is done on eight criteria:

The first is whether the moving piece is a king. Sequentially this is superfluos as this test has already been made in isCastling but is in here for modular completeness.

Secondly, this castle must be the king's first move, so a check is made to ensure the piece hasn't moved yet.

The third test is interested in the rook. It comes in two parts; firstly checking whether the rook is where it should be, and then ensures the rook hasn't moved yet.

Once it has been determined all the involved pieces are where they should be and with the right state, the path between them is tested. There should be no pieces between the king and the castling rook.

The next three checks concern themselves with threats from the other side. The king cannot end up in check after a castle move, so a check is made to determine whether the king would be moving into check.

Also, the king cannot pass through any square under threat, so the king's intended path is checked for threats from the other side.

The king cannot make a castling move from a position of check so the current board state is tested to ensure the king attempting to castle is not currently in check.

The final check is to ensure the king and the concerned rook are on the same rank. This rule is a late addition to chess, introduced in 1972 after it was shown that with the current rules it would be possible, after e pawn promotion, to castle vertically [3].

If all these tests are satisfied, castling is allowed and performed.

4.2.7 Promotions

Promoting pawns to other pieces on the board involves adapting a number of classes to handle this.

1) Move class had to be altered to store the piece a pawn would be promoted to, and also to say if a move is a promotion.

2) XBoardIO had to be changed to respond to promotions made by an opponent on the board and also communicate promotions made by the engine back to the interface.

3) BoardArray had to be changed to correctly handle changes to the state of the board.

While (1) involved implementing just 3 extra methods: setPromotionFigure(), getPromotionFigure() and isPromotion(), (3) involved a substantial number of changes to support rules of promotion and is discussed in further detail below. (2) is beyond the scope of this section and is discussed in further detail later in this chapter.

When considering changes required to be made to the board implementation, the following two possibilities need to be considered:

i) Opponent makes a move involving a promotion, this move already contains the new piece, so it simply needs to be placed on the board.

ii) The engine makes a move resulting in a promotion, an appropriate piece needs to be placed on the board and communicated back to the interface.

In order to handle both of the above scenarios, the following algorithm was implemented and merged into the makeMove() method of BoardArray.

Algorithm 11 Algortihm for handling promotions

```
Require: m is a move generated elsewhere
Require: isWhiteToPlay is true if it is white's turn to play
Require: cb is an array of size of 128 of chess pieces
Require: post is the square the piece has moved to
Ensure: m is updated to reflect the promotion if this hasn't already been done
Ensure: cb reflects the board state after the promotion
  if m.isPromotion() then
    Set the colour of the piece stored within the move to match the colour of the side whose turn
    it is to play
    cb[post]=m.getPromotionFigure()
  else if cb[post] is Pawn AND post/16 == (isPlayerTurn?7:0) then
    cb[post]=new Queen(isWhiteToPlay)
    m.setPromotionFigure(cb[post])
  end if
```

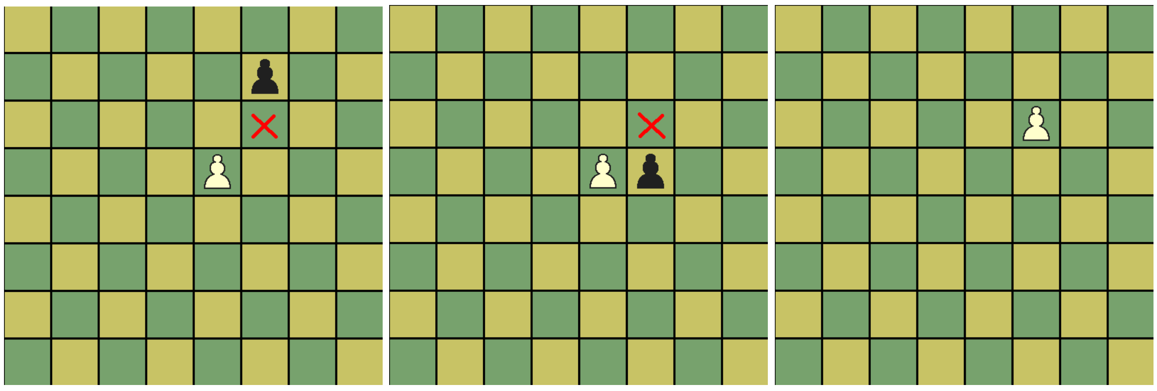
A number of points deserve to be highlighted in the above algorithm. First of all, the algorithm is used as part of makeMove() routine, called both during the move generation within the board, when it is not known yet if the move is going to result in a promotion, and by the main game loop, that sets up the move to contain the necessary promotion information if it is a promotion move.

In the first branch of the *if-statement* [see Algorithm 11] we assume, that the move is passed to the board from the interface, so it already contains all of the information relevant to the promotion, so the board state needs to be changed to reflect this. One of the steps involved in merging the promotion information into the board state is extracting the new piece from the move object. At this stage, the colour of the piece is not set, so this needs to be done explicitly by the algorithm [11].

In the second branch, the move is not already marked as a promotion, however it may in fact result in one. This is generally a result of calling generateLegalMoves routine discussed later in this chapter, that generates all posible moves on the board, which may result in a pawn reaching the final rank on the board (8th for white pawns and 1st for black pawns). This logical branch deals exactly with this scenario and produces a promotion to a queen, since this is the result of the majority of promotions.

4.2.8 En Passant

En passant is a special capture made immediately after a player moves a pawn two squares forward from its starting position, and an opposing pawn could have captured it as if it had only moved one square forward. In this situation, the opposing pawn may, on the immediately subsequent move, capture the pawn as if taking it "as it passes" through the first square; the resulting position would then be the same as if the pawn had only moved one square forward and the opposing pawn had captured normally. En passant must be done on the very next turn, or the right to do so is lost.[9].


(a) The black pawn is in its initial location. It could be captured by the white pawn if it moves to f6

(b) Black moved his pawn forward two squares from f7 to f5, "passing" f6

(c) On the next move, White captures en passant, capturing the pawn as if it had moved to f6

Figure 4.7: Performing an en passant capture.

Signalling the capture

The en passant capture was a very complex rule to implement. The main reason for this was due to the fact that the capture happens over two successive moves, one from each side. Firstly, when a pawn double jumps from it's starting position, this signals that there could potentially be the opportunity for the opposing side to capture that pawn 'en passant'. The capture is only available if the opposition side has a pawn positioned either one square to the left or right of the square the pawn double jumped to. Therefore, it was decided that anytime a pawn should move two squares from its starting position, a check to see whether an en passant capture by the opposing side would be possible. A method was implemented which performed such checks. However, the main problem was deciding upon how the board would 'remember', when the next move was taking place, whether the previous move had led to a potential en passant capture during this move. It was decided that the best way to signal to the engine that an en passant capture could be made on next move was to set a global status flag within the board, whose value would remain over as many moves as desired. This was a problem in itself as the opportunity to capture is only available on the immediate next move, made by the opposing side. As a result, the status flag always had to be set to false after any move on board was made, to ensure the capturing side was not able to perform the capture more than one move after the opposition's pawn had double jumped.

Recording the Move

With the ability of the board to correctly signal the availability of an en passant capture to the engine, the second problem was how the board could communicate what move would be required, in response to the double jump of a pawn, in order to perform the capture. The need to somehow record the move was due to the fact that two opposition pawns could be sat side by side on a rank, such that it would have been possible for one of those pawns to have had its first move be a double jump to its current position. However it may have been the case, for example, that the pawn in question moved to single squares at a time, on two seperate moves. This situation would not qualify as a valid en passant capturing opportunity for opposition side. As a result, the board also encapsulated a move value, which stored the details of the move required to be made by opposition, should they wish to take the opportunity of the en passant capture, during the resulting move.

Performing the Capture

Deciding upon and making the en passant capture for a particular side was fairly straightforward, given the preperation which was done to communicate the fact a capture was possible. The previously implemented method which dealt with the making of moves had to altered however. Within the method, a check is made to see whether the status flag, signalling the potential for a capture, is set. If it is, the current move being made is compared with the move stored globally as the required move for making the capture. Should these two moves happen to be the same then the capture is performed. As the make move method checks the move it is given for legality, the method which checks the legality of moves had to be altered also. This is due to the fact that a pawn is not normally allowed to move diagonally into an empty square. The method was altered so that such a move is allowed to be made by a pawn, provided it is making the en passant capturing move, whilst the capture flag is set.

Reversing the Capture

Obviously, the en passant capture is not like other conventional captures performed in chess, whereby the capturing piece captures on the square it moves into. In an en passant capture, the capturing piece captures on a aquare which is different to the square into which it moves. The standard operation for reversing a capturing move, was for the board to look at the square the capturing piece moved to, when move was performed and subsequently place the captured piece back onto that square. It was decided to overload the method that recorded a capture, so that it would take an extra perameter, corresponding to the square on the board a piece was captured on, should should that happen to be different from the square the capturing piece moves to. A check was then included in the method that dealt with the reversing of a capture move, so that it looks at the value corresponding to the square the piece was captured on. If this value is set to a default of minus one, the capture is dealt with in the conventional manner. Should the value happen to correspond to a square on the board however, the captured piece is placed back on that square.

4.2.9 Move committing and reversing

Move commiting

When commiting a move on the board, the engine needs to ensure that all implemented game rules are complied with and the integrity of the board is preserved. Move commiting is implemented by the makeMove() routine. The high-level algorithm of makeMove subroutine can be summarised in Algorithm 12:

Algorithm 12 makeMove() algorithm

```
Require: m is a move, requiring being commited
Ensure: result is true if move has been commited successfully, false otherwise
  if m is malformed then
    return $result \leftarrow false$
  end if
  if !isLegalMove(m) then
    return false
  end if
  if isCastling(m) then
    if canCastle(m) then
      Castle
    else
      return $result \leftarrow false$
    end if
  else
    if capture then
      Do Capture
    end if
    Move piece
  end if
  if Piece is King then
    if White King then
      Store the new position of the white king
    else
      Store the new position of the black king
    end if
  end if
  if promotion then
    Handle Promotion
  end if
  Tell the piece it's moved
  if Board is in check then
    Board state is illegal if a move causes a side to get in check or stay in check
    Reverse the move
    return False
  end if
  return True
```

As it can be seen from the above algorithm, it relies heavily on a number of subroutines, when deciding if the move is legal. isLegalMove() checks if the piece movement satisfies an appropriate pattern, i.e. direction and whether the path is blocked for most of the pieces, except knights. Castling, promotion and capture subroutines ensure, that the relevant complex rules are followed during the move. Finally, isInCheck() ensures that the integrity of the board is preserved even after the move has been commited. Each of these subroutines is discussed in detail in the relevant sections of this chapter. For a move to be successfully commited, all of the applicable chess rules have to be satisfied, otherwise, the routine aborts immediately, making no change to the board state. If a change has been commited to the board, but the resulting state is illegal, the move is reversed. The process of reversing moves is discussed in the following section.

Move reversing

Given a move, the board needs to be able to reverse its state to the one prior to the move being commited. During the move reversing, the engine needs to ensure the board state is changed accurately, any private board states are changed accordingly and relevant methods of pieces are called to revert their private states.

The high-level algorithm of the reverseMove() method is presented below:

Algorithm 13 Algorithm to reverse moves on the board

```
Require: m is a move, that needs to be reversed
  if m is malformed then
    return false
  end if
  if wasCastling(m) then
    Revert Castling
  end if
  if m.isPromotion() then
    Revert promotion
  else
    Move the piece to its old square
  end if
  if m.isCapture() then
    Put the captured piece back on the board
  end if
  Tell the moved piece, its move has been reversed
  Update pointers to locations of kings
```

4.2.10 Legal move generation

The task of generating all possible legal moves on the board is performed by a subroutine generateLegalMoves(), contained within the BoardArray class. The essense of it is trying to move each piece of the appropriate colour on the board in every way, that can be legal in some cicumstances and checking if that move is legal. If the move is not legal, then the board does not permit the move to be made, so nothing is changed. If the move is legal, then it is recorded and then reversed. At the end of the routine, the list of generated moves is returned back to the caller. This method produces only the moves that are legal, and they have to satisfy all of the implemented rules, including ensuring the king is not left in check. Effectively, the routine depends on:

i) Pieces knowing which directions they can move in,

ii) Correct implementation of makeMove(), which is responsible for implementation of chess rules and ensuring that they are adhered to,

iii) Correct implementation of reverseMove(), which is responsible for reversing a given move, ensuring that the board state is fully reverted, not leaving any trace of changes made by the original call to the makeMove() routine.

The algorithm is summarised below:

Algorithm 14 Generate Legal Moves

```
Require: MAXSQUARES is the number of squares on the board
Require: cb is an array of pieces of size MAXSQUARES
Require: isWhiteToPlay is true if it is white's turn to play, black otherwise
Ensure: legalMoves is a list of all legal moves on the board
  for $pos \leftarrow 0..MAXSQUARES - 1$ do
    if $(cb[pos] \neq null) AND (cb[pos].isWhite() == isWhiteToPlay)$ then
      $deltas \leftarrow cb[pos].getDeltas()$
      for all int delta : deltas do
        $m \leftarrow newMove(pos, pos + delta)$
        if makeMove(m) then
          legalMoves.add(m)
          reverseMove(m)
        end if
      end for
    end if
  end for
```

4.2.11 Board hashing

The concept of hashing the board to a unique integer relies on the random codes generated for different movement and piece states, which may be introduced during the game play. All of these states and their associated random codes are stored and reused for generating new board keys. That includes the set of values for each square of the board, as well as special move states, such as capture, castling and promotion, which have their own, unique codes. All of these values are necessary for generating the Zobrist hash keys for the situation for the board, as their uniqueness guarantees the uniqueness of the board key and functionality of the structures taking advantage of it. These values are stored as private, local fields of the BoardArray class, and so is the integer zobristKey, which holds the current hash code for the board. The keys for each square are recorded in a local, three dimensional array, indexed by piece type, colour and board square number. The type used for these values is a long (64 bit) integer. Although many chess engines, which implement hashing, use 32 bit integers, it is much more sensible to use larger numbers, which reduce the risk of producing identical values during the generation process.

The process of generating the random codes and maitaining the Zobrist hash key for the board is divided between two methods. The generateHashCode() method is responsible for initialising the fields holding the random values of different states for later use in evaluating new hash codes. The values are generated with the application of standard Java random long number generator and the Math.abs() function (absolute value), which ensures the resulting value is unsigned and can be used as a table reference in such constructs, as transposition tables. The same process is repeated for the array representing the board squares. The initialising loop running through all the squares of the board also picks the occupied squares and calculates the initial Zobrist key of the board by performing a logical XOR operation on the zobristKey value and the values associated with the given square and the piece located on it.

Algorithm 15 Hashing the board

```
$zobristKey \leftarrow 0$
$sideKey \leftarrow Abs(Randomlong)$
$castlingKey \leftarrow Abs(Randomlong)$
$enPassantKey \leftarrow Abs(Randomlong)$
$captureKey \leftarrow Abs(Randomlong)$
$promotionKey \leftarrow Abs(Randomlong)$
for every board square $s$ do
  if square does not belong to the dummy part of the board then
    for every piece type $p$ do
      $boardKeysArray[p][blackside][s] \leftarrow Abs(Randomlong)$
      $boardKeysArray[p][whiteside][s] \leftarrow Abs(Randomlong)$
    end for
    if there is a piece on square $s$ then
      $pieceType \leftarrow type\ of\ piece\ on\ s$
      $pieceColour \leftarrow colour\ of\ piece\ on\ s$
      $zobristKey \leftarrow zobristKey\ XOR\ boardKeysArray[pieceType][pieceColour][s]$
    end if
  else
    $s \leftarrow s + 7$ {Advance to next row if entered the dummy board}
  end if
end for
```

The generateZobristKey(Moveable m) method focuses on analysing the given move and logically combines the random keys, associated with its properties, with the existing board Zobrist key, using the XOR operation. Although the generateHashCode() method could have been reused for the purpose of producing the new hash code, iterating through the entire board for every move made, proves to be inefficient. The routine examines the following aspects of the move:

- which side is playing
- if it is a castling move
- if it is a capture or an en passant capture

- if it is a promotion or a regular move

The newly formed Zobrist board key is then stored within the board class, to be used in the tasks that are to follow. This method is used both for making and reversing moves, as repeating a set of XOR operations twice on the board, using the same move object, results in restoring the original value of the key.

Algorithm 16 Generating new hash key

```
if white side's turn then
  $zobristKey \leftarrow zobristKey\ XOR\ sideKey$
  $colour \leftarrow white$
end if
if castling is performed then
  $zobristKey \leftarrow zobristKey\ XOR\ castlingKey$
  $XOR\ boardKeysArray[rook][colour][rookoriginalposition]$
  $XOR\ boardKeysArray[rook][colour][rooktargetposition]$
end if
if capture is performed then
  $captureColour \leftarrow opposite\ of\ colour$
  $zobristKey \leftarrow zobristKey\ XOR\ captureKey$
  if capture is en passant then
    $zobristKey \leftarrow zobristKey\ XOR\ enPassantKey$
    $XOR\ boardKeysArray[pawn][captureColour][enpassantposition]$
  else
    $zobristKey \leftarrow zobristKey$
    $XOR\ boardKeysArray[capturedpiecetype][captureColour][pieceposition]$
  end if
end if
if promotion is performed then
  $zobristKey \leftarrow zobristKey\ XOR\ promotionKey$
  $XOR\ boardKeysArray[pawn][colour][pawnoriginalposition]$
  $XOR\ boardKeysArray[piecepromotedto][colour][pawntargetposition]$
else {normal move performed}
  $zobristKey \leftarrow zobristKey\ XOR\ boardKeysArray[movedpiecetype][colour][originalposition]$
  $XOR\ boardKeysArray[movedpiecetype][colour][targetposition]$
end if
```

The Zobrist board hashing was implemented in Chess Mantis for the purpose of introducing transposition tables and history heuristics and possibly other techniques, which take advantage of the stored board's hash keys. Although a bit more complicated approach was examined at first, featuring seperate random keys for every castling possibility and additional random keys for pawns, we have decided to use a simplified version with single random code for the pawn piece type and castling. The simplified version is still very reliable and works as required.

4.3 Analyser

Chess Mantis has within it a defined analyser interface, that ensures all analysers have certain functions, such as the ability to retrieve a move once the analyser has finished, the ability to test whether or not it has finished and a means of canceling the analysis whilst it is being performed. Two sub-interfaces extend this interface, one for stable analysers, that is, analysers that will always produce the best possible move as per the minimax algorithm, for the given cut off, and the other to be implemented by unstable analysers, whose results aren't necessarily the best move for the given cut-off.

We have implemented two stable analysers: one for naive minimax, the other for alpha beta pruning to gain a speed increase. We have attempted to implement further analyser enhancements, both stable and unstable. However, more complex designs proved to be too difficult to implement within the time constraints. Nevertheless, the design of the analyser component allows us to in future easily expand upon any of the analysers, and allows us to distinguish between stable and unstable analysers should we need to discriminate in future.

Algorithm 17 Maximise

```
if depth is maximum depth or $node$ is terminal then
  return evaluation($node$)
end if
$bestscore$ ← -infinity
while Moves left do
  make $move$
  $movescore$ ← Minimise($move$)
  reverse $move$
  if $movescore > bestscore$ then
    $bestscore$ ← $movescore$
  end if
end while
return $bestscore$
```

Algorithm 18 Minimise

```
if depth is maximum depth or $node$ is terminal then
  return evaluation($node$)
end if
$bestscore$ ← infinity
while Moves left do
  make $move$
  $movescore$ ← Maximise($move$)
  reverse $move$
  if $movescore < bestscore$ then
    $bestscore$ ← $movescore$
  end if
end while
return $bestscore$
```

The simplest analyser, is the naive minimax implementation, upon which all the other current analysers are based. The essence of the algorithm employed is outlined in algorithms 17 and 18. However this algorithm only returns a score for every node, whereas what is needed is a move to be returned at the root. This is achieved by simply adding in a best move variable that is returned instead of the best score at depth 1.

Algorithm 19 Alpha Beta Maximise

```
if depth is maximum depth or $node$ is terminal then
    return evaluation($node$)
end if
$bestscore \leftarrow$ -infinity
while Moves left do
    make $move$
    $movescore \leftarrow$ Minimise($move$,$alpha$,$beta$)
    reverse $move$
    if $movescore > alpha$ then
        $alpha \leftarrow movescore$
    end if
    if $alpha >= beta$ then
        return $alpha$
    end if
end while
return $alpha$
```

Algorithm 20 Alpha Beta Minimise

```
if depth is maximum depth or $node$ is terminal then
    return evaluation($node$)
end if
$bestscore \leftarrow$ infinity
while Moves left do
    make $move$
    $movescore \leftarrow$ Maximise($move$)
    reverse $move$
    if $movescore < beta$ then
        $beta \leftarrow movescore$
    end if
    if $beta <= alpha$ then
        return $beta$
    end if
end while
return $beta$
```

The alpha beta analyser, based on a copy of the naive minimax analyser, is naturally similar in implementation. The engine's and its opponent's best score are stored in the alpha and beta variables respectively. Algorithms 19 and 20 show the changed max and min functions.

The implementation of analyser cancellation is quite simple, and is consistent throughout all the implemented analysers. The cancelling function simply sets a Boolean to true. At each node the analyser checks the Boolean, and if it is true, throws an exception; this exception is then passed up the levels of recursion until the function exits. The initial implementation of cancellation had only checked the Boolean at nodes on depth 1, as this avoided throwing and catching errors, however, this solution did not scale as for larger searches of 6 ply or more, cancellation could take some time, time that could be better spent on analysis of a future move that would not be discarded for time keeping. Further, checking a Boolean once every node is an extremely inexpensive operation, even if repeated billions of times per analysis.

4.4 Evaluator

The evaluator is designed as a generic interface that contains the relevant evaluate function, and also a set random function that is a requirement of the XBoard interface. This set random function adds the ability for the evaluator to evaluate a board and then also add another random value between -100 and 100 to give a slightly different game every time. This interface was initially implemented with a random evaluator, the Marmoset. Before explaining the Marmoset, a note on the naming conventions. Each of the evaluators was named after a type of monkey, for no other reason that the names were interesting and unique. These names provided memorable evaluators for the team to use. Monkey names were chosen as they are below the intelligence of human beings, and the state of the evaluator at this time was on par with the intelligence of a monkey. Returning to the functionality of the Marmoset, it simply chooses a random integer number and returned this through the evaluate function. This meant that the moves performed were completely randomly choosen from the list of generated moves, but it would allow simple testing of the Analyser algorithms. The next step was to implement the basic evaluator, one that relied on Material evaluation only, which was called Capuchin. This was also a very useful evaluator for the team to have, as its behaviour was extremely predictable, and allowed the team to easily discover bugs in other parts of the project. The next evaluator to be created was Sifaka, which would extend Capuchin. Sifaka calculated pawn heuristics, and was a good step up on the predictive behaviour of Capuchin. The evaluator now knew how to defend pawns and keep good pawn formations. The last evaluator was Macaque, which extended Sifaka. Macaque added the much needed functionality of giving the scores of positive or negative infinity for checkmates, whether the king was vunerable to attack, and also how good a positional advantage each side had. With this last evaluator, the evaluation functions were completed, but had one major flaw.

With the good object orientated design and the breaking up of major functionality into these three seperate evaluators, code was repeated and reused in each of the three evaluators. Considering the evaluator can be run as much as five million times during a single Analysis of the game tree (to a depth of six ply), this duplicate code was inefficient although well designed. The design decision was taken to merge all of these evaluators into one big evaluator, aptly named Mantis after the program name. This new Evaluator Mantis combined much of the duplicate functionality of the previous three Evaluators at a cost of good object orientated design.

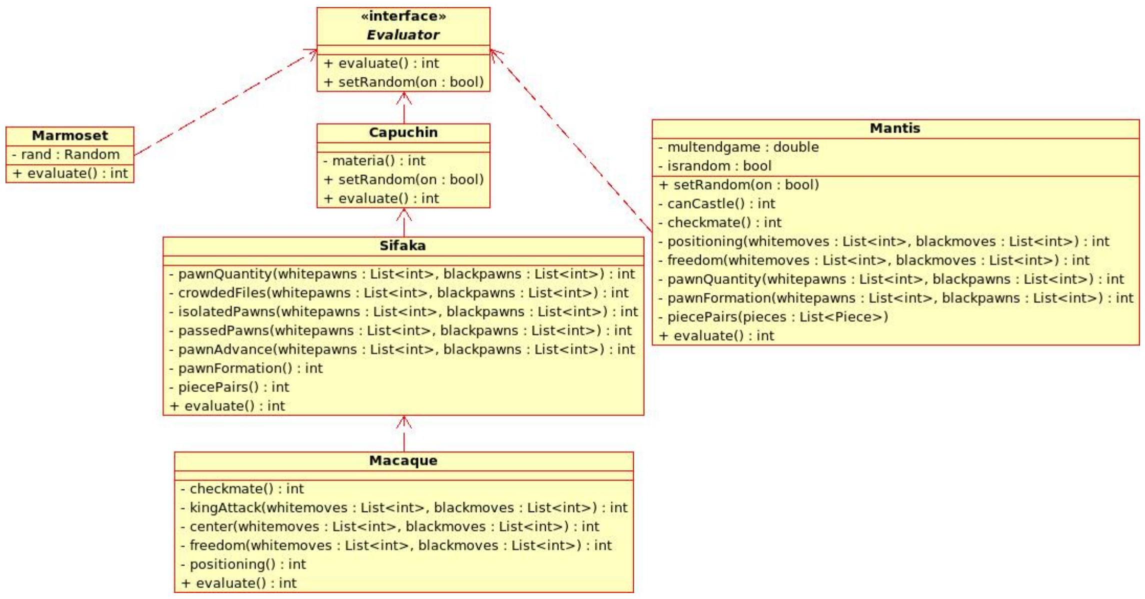

«interface»
Evaluator
+ evaluate() : int
+ setRandom(on : bool)
Marmoset
- rand : Random
+ evaluate() : int
Capuchin
- materia() : int
+ setRandom(on : bool)
+ evaluate() : int
Sifaka
- pawnQuantity(whitepawns : List<int>, blackpawns : List<int>) : int
- crowdedFiles(whitepawns : List<int>, blackpawns : List<int>) : int
- isolatedPawns(whitepawns : List<int>, blackpawns : List<int>) : int
- passedPawns(whitepawns : List<int>, blackpawns : List<int>) : int
- pawnAdvance(whitepawns : List<int>, blackpawns : List<int>) : int
- pawnFormation() : int
- piecePairs() : int
+ evaluate() : int
Macaque
- checkmate() : int
- kingAttack(whitemoves : List<int>, blackmoves : List<int>) : int
- center(whitemoves : List<int>, blackmoves : List<int>) : int
- freedom(whitemoves : List<int>, blackmoves : List<int>) : int
- positioning() : int
+ evaluate() : int
Mantis
- multendgame : double
- israndom : bool
+ setRandom(on : bool)
- canCastle() : int
- checkmate() : int
- positioning(whitemoves : List<int>, blackmoves : List<int>) : int
- freedom(whitemoves : List<int>, blackmoves : List<int>) : int
- pawnQuantity(whitepawns : List<int>, blackpawns : List<int>) : int
- pawnFormation(whitepawns : List<int>, blackpawns : List<int>) : int
- piecePairs(pieces : List<Piece>)
+ evaluate() : int

Figure 4.8: Breakdown of Evaluator design

The Mantis evaluator later grew to include checks for castling also, and implementing the end game multiplier (see 3.4.13). Mantis cut the running time of each evaluation in half, and thus improved the depth that the Analyser could reach in a given time. This had the knock-on effect of greatly increasing the skill of our chess AI, and thus our ability to play a good game at an intermediate level.

The rest of this Evaluator section consists of each of the evaluation heuristics considered by the Evaluator, and the pseudocode showing how they are implemented. It also discusses any implementation issues encountered with implementing each of these heuristics and any interesting other notes on each heuristic.

The following heuristics use pseudocode samples to describe the functionality of how each of these heuristics are calculated. Originally in the heirarchal structure the heuristics were seperated out into individual functions, to take advantage of the object orientated features. Once Mantis was made and the three previous evaluators were combined into one bigger one, any common functionality that could be abstracted out from each of the functions, to increase the speed of each evaluation, was abstracted. Therefore, the pseudocode examples of the heuristic are not necessarily accurate in terms of how Mantis actually computes each of these functions, however the result from each of these functions is added to the total in the exact same way. This guarantees that Mantis and Macaque will get the same evaluation score from the same board, but Mantis gets the result much faster.

4.4.1 Material Evaluation

Algorithm 21 Material Evaluation

```
$total \leftarrow 0$
for each Piece $p$ on Board do {Run through every piece on board}
    if $p$ is black then
        $total \leftarrow total +$ value of $p$ {Add value of piece to total}
    else {$p$ is white}
        $total \leftarrow total -$ value of $p$ {Subtract value of piece from total}
    end if
end for
return $total$
```

The pseudocode shows the simplicity of the function which will calculate Material Evaluation, which was described eariler (see 3.4.1). The pseudocode here is being used to elaborate the functionality of the Material Evaluation function. Summarising the pseudocode, it takes the total value of all black pieces and subtracts the total value of all white pieces.

4.4.2 Passed Pawns

Algorithm 22 Passed Pawns

```
Ensure: $whitepawns$ is a collection of all squares on the board occupied by white pawns
Ensure: $blackpawns$ is a collection of all squares on the board occupied by black pawns
  $total \leftarrow 0$
  for each Piece $p$ in $blackpawns$ do {Run through every square occupied by a black pawn on board}
    $passed \leftarrow true$
    for each Piece $q$ in $whitepawns$ do {Compare against all squares occupied by white pawns}
      if $p$ passed every Piece $q$ in $whitepawns$ then
        $passed \leftarrow true$
      else
        $passed \leftarrow false$
      end if
    end for
    if $passed$ equals true then
      $total \leftarrow total +$ (value of $p$) {Bonus equal to one pawn}
    end if
  end for
  for each Piece $p$ in $whitepawns$ do {Run through every white pawn on board}
    $passed \leftarrow true$
    for each Piece $q$ in $blackpawns$ do {Compare against all black pawns}
      if $p$ passed every Piece $q$ in $blackpawns$ then
        $passed \leftarrow true$
      else
        $passed \leftarrow false$
      end if
    end for
    if $passed$ equals true then
      $total \leftarrow total -$ (value of $p$) {Bonus equal to one pawn}
    end if
  end for
  return $total$
```

To aid in ensuring efficiency, this function requires a list of the positions of all the white pawns on the board and a similar list for the black pawns. The algorithm itself, for the list of pawns of a particular colour, compares each pawn in that list in turn, against all the pawns in list of the opposite colour. If that pawn has passed all the opposition's pawns, then that pawn is a passed pawn and therefore, the bonus, equal to the (hard-coded) value of one pawn, is applied to the total.

4.4.3 Crowded Files

Algorithm 23 Crowded Files

```
Ensure: $whitepawns$ is a collection of all squares on the board occupied by white pawns
Ensure: $blackpawns$ is a collection of all squares on the board occupied by black pawns
  $total \leftarrow 0$
  $pawncount \leftarrow 0$
  for each Piece $p$ in $blackpawns$ do {Run through every black pawn on board}
    for each Piece $q$ in $blackpawns$ do {Compare against all other black pawns}
      if $q$ in same file to $p$ then
        $pawncount \leftarrow pawncount + 1$ {Increment total number of pawns found in that file}
      end if
      if $pawncount$ equal to 2 then
        $total \leftarrow total -$ (value of $p$) {Bonus is equal to the value of a pawn}
      end if
      if $pawncount$ greater than 2 then
        $total \leftarrow total -$ ((value of $p$) + ((value of $p$)/2) + (($valueof$p) / 4)) {Bonus is equal to one and three quarters of a pawn}
      end if
    end for
    $pawncount \leftarrow 0$
  end for
  for each Piece $p$ in $whitepawns$ do {Run through every white pawn on board}
    for each Piece $q$ in $whitepawns$ do {Compare against all other white pawns}
      if $q$ in same file to $p$ then
        $pawncount \leftarrow pawncount + 1$ {Increment total number of pawns found in that file}
      end if
      if $pawncount$ equal to 2 then
        $total \leftarrow total +$ (value of $p$) {Bonus is equal to the value of a pawn}
      end if
      if $pawncount$ greater than 2 then
        $total \leftarrow total +$ ((value of $p$) + ((value of $p$)/2) + (($valueof$p) / 4)) {Bonus is equal to one and three quarters of a pawn}
      end if
    end for
    $pawncount \leftarrow 0$
  end for
  return $total$
```

To aid in ensuring efficiency, this function requires a list of all the white pawns on the board and a similar list for the black pawns. For each of the two lists of pawns in turn, the algorithm looks at each of the pawns in that list. It counts the number of pawns present in each file. If it finds there to be two pawns present in the same file a penalty, equivalent to one pawn, is applied to the total. Should it find there to be more than two pawns in any particular file, a penalty equivalent to one and three quarter pawns is applied.

4.4.4 Pawn Quantity

Algorithm 24 Quantity

```
Ensure: $whitepawns$ is a collection of all squares on the board occupied by white pawns
Ensure: $blackpawns$ is a collection of all squares on the board occupied by black pawns
  $total \leftarrow 0$
  if there are 8 pawns present in $blackpawns$ then
    $total \leftarrow total -$ (value of pawn $p$) {Bonus is equal to that of a pawn}
  else if there are 7 pawns present in $blackpawns$ then
    $total \leftarrow total -$ (value of pawn $p)/2$ {Bonus is equal to half a pawn}
  else {less than 7 pawns}
    apply no bonus
  end if
  if there are 8 pawns present in $whitepawns$ then
    $total \leftarrow total +$ (value of pawn $p$) {Bonus is equal to that of a pawn}
  else if there are 7 pawns present in $whitekpawns$ then
    $total \leftarrow total +$ (value of pawn $p)/2$ {Bonus is equal to half a pawn}
  else {less than 7 pawns}
    apply no bonus
  end if
  return $total$
```

To aid in ensuring efficiency, this function requires a list of all the white pawns on the board and a similar list for the black pawns. The algorithm itself is fairly straightforward. Firstly, for the particular side in question, it finds the number of pawns in the list. If that side is found to have eight pawns in the list, a penalty equivalent to the value of one pawn is applied to the total. Should the list be found to contain seven pawns, a penalty of equal to half the value of one pawn is applied to the total. When the list contains a number of pawns which is less than seven no change to the total is made.

4.4.5 Crippled Pawns

Algorithm 25 Crippled Pawns

```
Ensure: $whitepawns$ is a collection of all white pawns
Ensure: $blackpawns$ is a collection of all black pawns
  $total \Leftarrow 0$
  for each Piece $p$ in $blackpawns$ do {Run through every black pawn on board}
    if $p$ in file 0 or file 7 then
      $total \Leftarrow total -$ (value of $p)/2$ {Bonus is one half of a pawn}
    end if
  end for
  for each Piece $p$ in $whitepawns$ do {Run through every white pawn on board}
    if $p$ in file 0 or file 7 then
      $total \Leftarrow total +$ (value of $p)/2$ {Bonus is one half of a pawn}
    end if
  end for
  return $total$
```

To aid in ensuring efficiency, this function requires a list of the positions of all the white pawns on the board and a similar list for the black pawns. The algorithm itself is very simple in its solution. For each of the two lists of pawns, it runs through each list checking to see if any of the pawns in that list are located on either of the two outer files. If a crippled pawn should happen to be found, a bonus of one half of a pawn is applied to the total, in such a manner that it favours the opposition side. In other words, having a crippled pawn has a reduction effect on the total, for that colour.

4.4.6 Isolated Pawns

Algorithm 26 Isolated Pawns

```
Ensure: $whitepawns$ is a collection of all white pawns
Ensure: $blackpawns$ is a collection of all black pawns
  $total \leftarrow 0$
  for each Piece $p$ in $blackpawns$ do {Run through every black pawn on board}
    for each Piece $q$ in $blackpawns$ do {Compare against all other black pawns}
      if $q$ in neighbouring file to $p$ then
        $total \leftarrow total +$ (value of $p)/10$ {Bonus is a tenth of a pawn}
      end if
    end for
  end for
  for each Piece $p$ in $whitepawns$ do {Run through every white pawn on board}
    for each Piece $q$ in $whitepawns$ do {Compare against all other white pawns}
      if $q$ in neighbouring file to $p$ then
        $total \leftarrow total -$ (value of $p)/10$ {Bonus is a tenth of a pawn}
      end if
    end for
  end for
  return $total$
```

This function will require a list of all the white pawns on the board, and all the black pawns, to ensure an optimal efficiency concious solution. The actual algorithm itself is quite simplistic. It runs through the collection of pawns, for each colour at a time, comparing the current pawn it is looking at against all other pawns of the same colour. Then, if one of the pawns(denoted by q) happens to reside in a neighbouring file, the bonus of a tenth of a pawn is applied to the total. This means in real terms that, at the start of the game, each pawn, except the pawns in the outermost files, will gain a bonus of 20, but will start to lose this as soon as the first pawn is removed from the board.

4.4.7 King Pawns

Algorithm 27 King Pawns

```
Ensure: $whitepawns$ is a collection of all white pawns
Ensure: $blackpawns$ is a collection of all black pawns
Ensure: $whiteking$ is the square the white King resides on
Ensure: $blackking$ is the square the black King resides on
  $total \leftarrow 0$
  if $blackking$ has pawn south-east then
    $total \leftarrow total + (\text{value of } p)/10$ {Bonus is a tenth of a pawn}
  end if
  if $blackking$ has pawn south then
    $total \leftarrow total + (\text{value of } p)/10$ {Bonus is a tenth of a pawn}
  end if
  if $blackking$ has pawn south-west then
    $total \leftarrow total + (\text{value of } p)/10$ {Bonus is a tenth of a pawn}
  end if
  if $whiteking$ has pawn north-west then
    $total \leftarrow total - (\text{value of } p)/10$ {Bonus is a tenth of a pawn}
  end if
  if $whiteking$ has pawn north then
    $total \leftarrow total - (\text{value of } p)/10$ {Bonus is a tenth of a pawn}
  end if
  if $whiteking$ has pawn north-east then
    $total \leftarrow total - (\text{value of } p)/10$ {Bonus is a tenth of a pawn}
  end if
  return $total$
```

The function simply checks whether the black King has pawns to the south-east, south and south-west of its current position, and then whether the white King has pawns to the north-west, north and north-east of its position. If any of these positions contains a pawn, the bonus of 10 points is added or subtracted from the total and then returned.

4.4.8 Piece Freedom

As discussed in section 3.4.8, piece freedom heuristic concerns itself with rewarding the side, that has got a higher number of legal moves it can resort to. As the number of legal moves varies throughout the game, the reward should be given for having more moves available as compared to the opponent, and not as an absolute value.

The following algorithm summarises this:

Algorithm 28 Algorithm to calculate the piece freedom bonus

```
Require: blackmoves is a list of all possible moves the black side can make
Require: whitemoves is a list of all possible moves the white side can make
Require: FM is a number of bonus points a side awarded per move it can make
  RETURN $(blackmoves.size() - whitemoves.size()) \times FM$
```

4.4.9 Center Control

Algorithm 29 Center control

```
Require: INNERCENTER is an array of inner center squares
Require: OUTERCENTER is an array of outer center squares
Require: blackmoves is a list of all possible moves, the black side can make
Require: whitemoves is a list of all possible moves, the white side can make
Require: CIC is a bonus a piece receives for being able to move into an inner center square
Require: COC is a bonus a piece receives for being able to move into an outer center square
Ensure: total is the total bonus center control score, which is positive is black has got a better
  position
  for all squares in INNERCENTER do
    if Piece is black then
      $total \leftarrow total + 2 \times CIC$
    end if
    if Piece is white then
      $total \leftarrow total - 2 \times CIC$
    end if
  end for
  for all squares in OUTERCENTER do
    if Piece is black then
      $total \leftarrow total + 2 \times COC$
    end if
    if Piece is white then
      $total \leftarrow total - 2 \times COC$
    end if
  end for
  for all moves in blackmoves do
    for all squares in INNERCENTER do
      if move is to square then
        $total \leftarrow total + CIC$
      end if
      for all squares in OUTERCENTER do
        if move is to square then
          $total \leftarrow total + COC$
        end if
      end for
    end for
  end for
  for all moves in whitemoves do
    for all squares in INNERCENTER do
      if move is to square then
        $total \leftarrow total - CIC$
      end if
      for all squares in OUTERCENTER do
        if move is to square then
          $total \leftarrow total - COC$
        end if
      end for
    end for
  end for
```

As discussed in section 3.4.9, a side should receive a bonus for controling the center of the board.

Two areas on the board should be considered:

i) inner center, which consists of squares d4, e4, d5 and e5, is the most attractive position on the board for the majority of pieces as it improves their mobility

ii) outer center, which consists of squares c3, d3, e3, f3, c4, f4, c5, d5, e5, f5, which is also attractive, however not as much as the inner center.

Also a figure may either be able to move to a position in the center or occupy on of such square, when it should receive a higher bonus.

Overall the above is summarised in Algorithm 29.

4.4.10 King Attack

Algorithm 30 King Attack

```
Ensure: $whitepieces$ is a collection of all white pieces
Ensure: $blackpieces$ is a collection of all black pieces
Ensure: $whiteking$ is the white King
Ensure: $blackking$ is the black King
  $total \leftarrow 0$
  for each Piece $p$ in $blackpieces$ do {Run through every black piece on board}
    if $p$ has a view on $whiteking$ then
      $total \leftarrow total +$ (value of $p$)$/50$ {Bonus is a fiftieth of a pawn}
    end if
  end for
  for each Piece $p$ in $whitepieces$ do {Run through every white piece on board}
    if $p$ has a view on $blackking$ then
      $total \leftarrow total -$ (value of $p$)$/50$ {Bonus is a fiftieth of a pawn}
    end if
  end for
  return $total$
```

The function requires that there are two collections of pieces, one containing the white pieces, and the other the black. The algorithm works by running through the collection of black pieces, and checks to see if any of these pieces has a view of the white King. If this is true, it applies the bonus to the total. Then it does the same for the white pieces, and checks them against the white King. A view on the white King is when a piece has the ability to move to the square the white king resides on, thus allowing these pieces to open up squares surrounding that king and exposing it for further attack.

4.4.11 Center King

Algorithm 31 Center King

```
Ensure: $whiteking$ is the white King
Ensure: $blackking$ is the black King
Ensure: $endgamemult$ is the end game multiplier (see 3.4.13)
  $total \leftarrow 0$
  if $blackking$ in center then
    $total \leftarrow total + (0.2 - endgamemult) * 100$ {Bonus is a pawn}
  end if
  if $whiteking$ in center then
    $total \leftarrow total - (0.2 - endgamemult) * 100$ {Bonus is a pawn}
  end if
  return $total$
```

The function uses some strange logic that requiers a bit of explaining, noteably the formula it uses to calculate the bonus applied to each side. The basis of the formula is that endgamemult is a value between 0.0 and 1.0, thus by subtracting from 0.2 we get a value that ranges between -0.8 and 0.2. This means that in the start game, when endgamemult is closer to 1.0, the formula will return a negative result if the King is in the center, thus making this a bad thing to do. Conversely though in the end game, when the value is closer to 0.0, the formula will return a positive result, making this a good thing to do in the end game. So the formula does what was needed in 3.4.11 and uses an efficient solution to calculate this heuristic. One of the best features of this formula is that it gradually changes this value from start to end game, giving this heuristic a gradually changing bonus. This is useful as it gradually becomes more important for the King to head for the center later on in the game, rather than getting to a point in the game and the King instantly heading for the center thereafter. The rest of the algorithm is simple, checking for both sides if their respective Kings reside in the center, and applying the bonus discussed before to each side.

4.4.12 Has Castled

Algorithm 32 Material Evaluation

```
Ensure: $whiteking$ is the white King
Ensure: $blackking$ is the black King
  $total \Leftarrow 0$
  if $blackking$ has castled then
    $total \Leftarrow total + 10$ {Bonus is a tenth of a pawn}
  end if
  if $whiteking$ has castled then
    $total \Leftarrow total - 10$ {Bonus is a tenth of a pawn}
  end if
  return $total$
```

The function is entirely trivial. It checks whether either King has castled, then applies the bonus of a tenth of a pawn, a positive bonus for the black King, and a negative bonus for the white King.

4.4.13 End Game Multiplier

Algorithm 33 End Game Multiplier

```
$total \Leftarrow 0$
**for** each Piece $p$ on Board **do** {Run through every piece on board}
    $total \Leftarrow total +$ value of $p$ {Add value of piece to total}
**end for**
$total \Leftarrow total/8000$ {Divide total by value of eighty pawns}
**return** $total$
```

The end game multiplier is a simple calculation that runs through every piece that is present on the board, and adds the value of that piece to the total. Once this has completed, it divides this total by eighty pawn values to get us a result that lies between 0.0 and 1.0, and returns this result.

Chapter 5

Evaluation

5.1 Testing Rules

5.1.1 Outline

The purpose of this portion of the testing section is to ensure the chess engine adheres to the standard rules of chess. The rules adopted by the engine conform to the rules as stated by the World Chess Federation [9]. It should be noted however, that some of the rules, as stated by the chess governing body, F.I.D.E, have not been implemented. The reason for omitting these rules was simply due to the fact that they are very obscure and in most cases, are only considered at a professional level during tournament competition. An example of such a rule, is the fifty move rule, which states that should a game go fifty moves without either a pawn being moved, or a capture occuring, then that game is drawn.

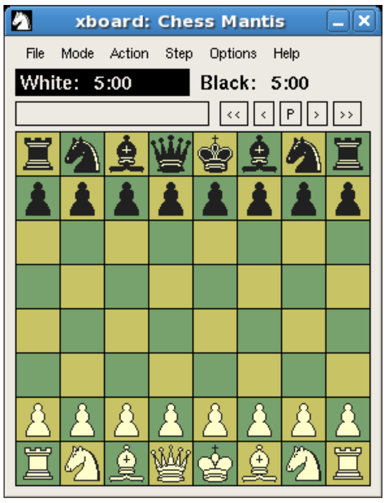
This section outlines the various test cases, chosen in order to prove the correctness of particular rules, documenting the results of each of the test cases and providing a conclusion, in terms of conformity to each particular rule. The various test cases tested the movement of each piece type, more complex piece movement, the ability to capture and promote pieces, as well as the chess engine's capability of recognising those situations which signal the end of a game.

5.1.2 Piece Movement

This section will provide, for each of the six types of piece, test cases for each move that piece can make. The expected result will be stated, followed by the actual result produced.

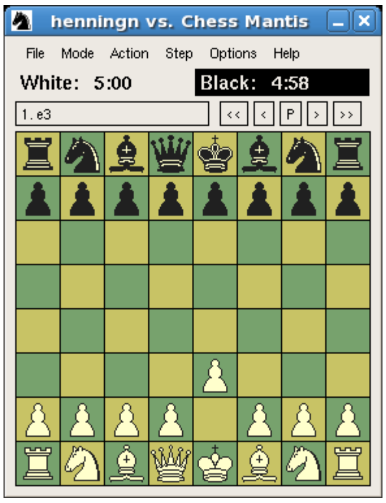
Test Case T1: Pawn Movement

There were various test cases created in order to ensure the engine allowed correct pawn movement. Test cases were set up to ensure that a pawn can move by one square, in a forward direction only and by two squares, from their initial position only. All the results matched the expected results.



White: 5:00 Black: 5:00

(a) White pawn initially on e2



White: 5:00 Black: 4:58

1. e3

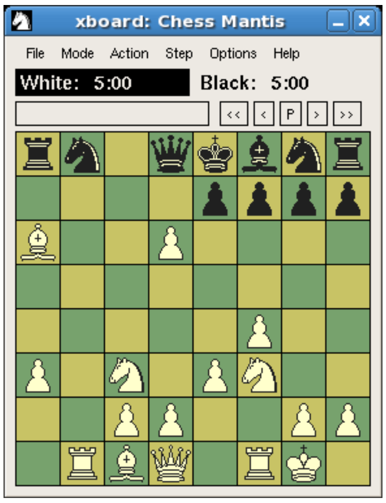
(b) Pawn successfully moves forward to e3

Figure 5.1: Single square pawn movement from e2 to e3.

Test Case T2: Rook Movement

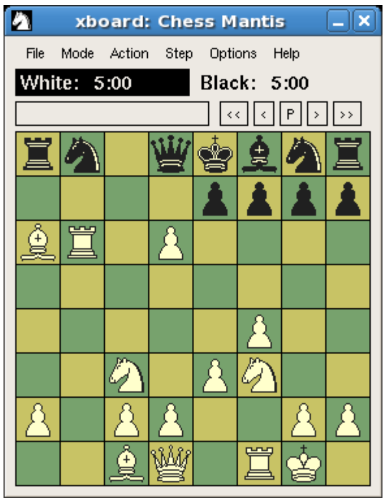
As a rook can move, in four directions, an arbitrary number of squares, each of the test cases looked at movement by one square, as well as movement by more than one square.

Test cases was designed to test the eigtht possible legal rook movements i.e forward, backward, left and right, by one square and by more than one square. Each of the moves were deemed as legal by the chess engine. Another set of tests were also created to ensure a rook could not move diagonally, the only direction of movement illegal for such a piece. As expected, an attempt at a diagonal move by a rook, was rejected by the engine.



White: 5:00 Black: 5:00

(a) White Rook initially on b1



White: 5:00 Black: 5:00

(b) White Rook successfully moves in file b from b1 to b6

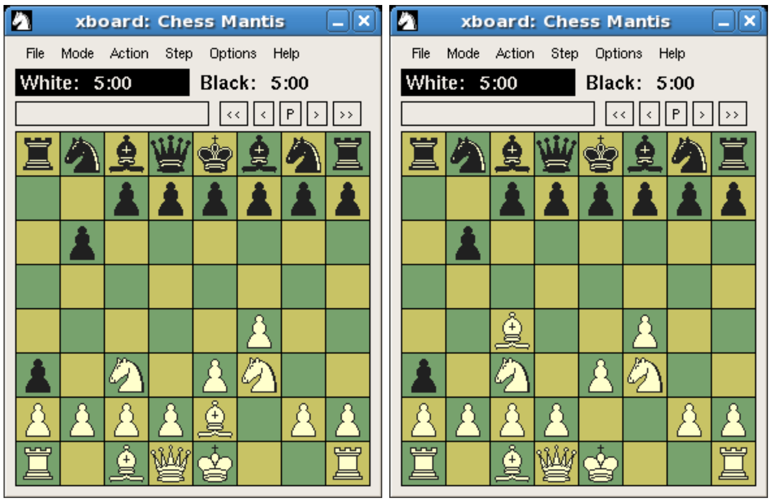
Figure 5.2: Rook movement from b1 to b6.

Test Case T3: Bishop Movement

As a bishop can move, in four directions, an arbitrary number of squares, each of the test cases looked at movement by one square, as well as movement by more than one square.

A set of test cases were created to ensure that the engine allowed all eight types of movement i.e diagonally in all directions, by both one square and more than one square. As expected each of the moves were deemed legal by the engine.

Another test case was created to ensure that the engine would reject any illegal bishop movement i.e movement vertically in a file, or horizontally in a rank. As expected, both types of illegal movement were rejected by the engine.



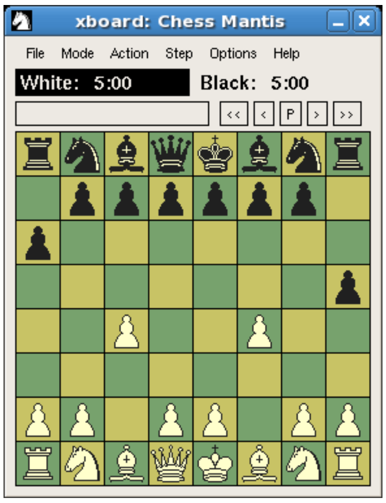
(a) White bishop initially on e2 (b) White bishop successfully moved to c4

Figure 5.3: Bishop movement from e2 to c4.

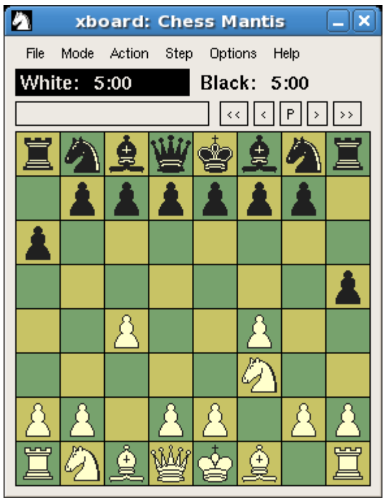
Test Case T4: Knight Movement

For knight movement, a test case was designed to ensure all eight possible knight movements would be accepted as legal by the engine. As expected, all four types of movement were were accepted by the engine as legal.

A further test case was created to ensure that the chess engine would not allow the knight to perform any illegal movements. As expected, the engine rejected any attempted illegal moves, made by the knight.



(a) White knight initially on g1



(b) White knight successfully moved to f3

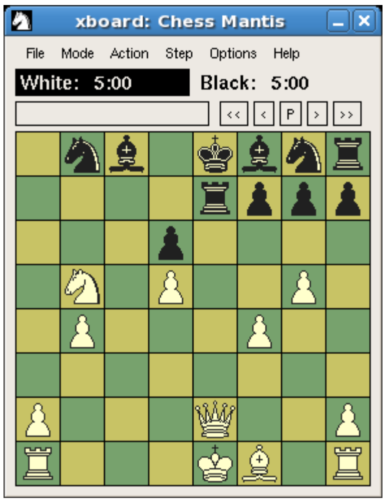
Figure 5.4: Knight movement from g1 to f3.

Test Case T5: Queen Movement

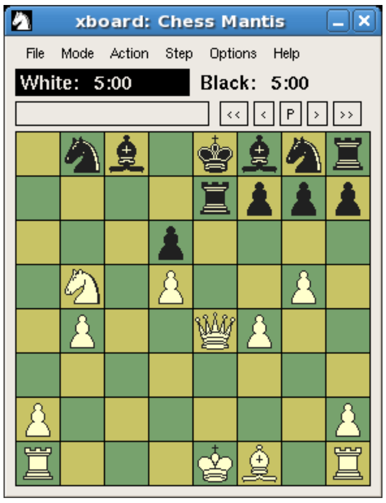
As a queen can move, in all directions, an arbitrary number of squares, each of the test cases looked at movement by one square, as well as movement by more than one square.

T5.1

A set of test cases were designed to ensure that the each of the queen's sixteen types of movement i.e both one square and mutiple squares, in any direction, were deemed legal by the chess engine. As expected, the queen was allowed to perform all of its movements legally.



(a) White queen initially on e2



(b) White queen successfully moved to e4

Figure 5.5: Queen movement from e2 to e4.

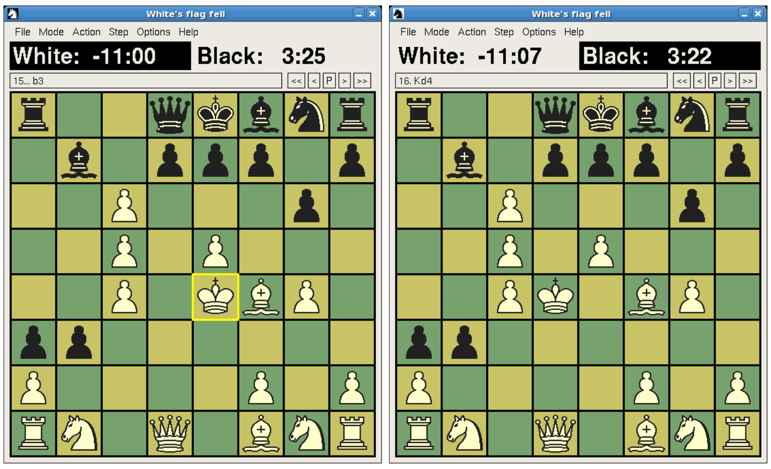
As there is no direction in which the queen cannot move, there was no need for a test case which checked for illegal, directional movement.

Test Case T6: King Movement

T6.1

Test cases were created to ensure that the king could move one square in any direction. The results produced show the engine accepted the king's movements as being legal, which conformed with the expected results.

A further set of test cases were created to ensure that, whilst the king can move in any direction, it can only move by one square in that particular direction, during any one move. The results showed that the engine rejected any attempted king moves, which travelled over more than one square.



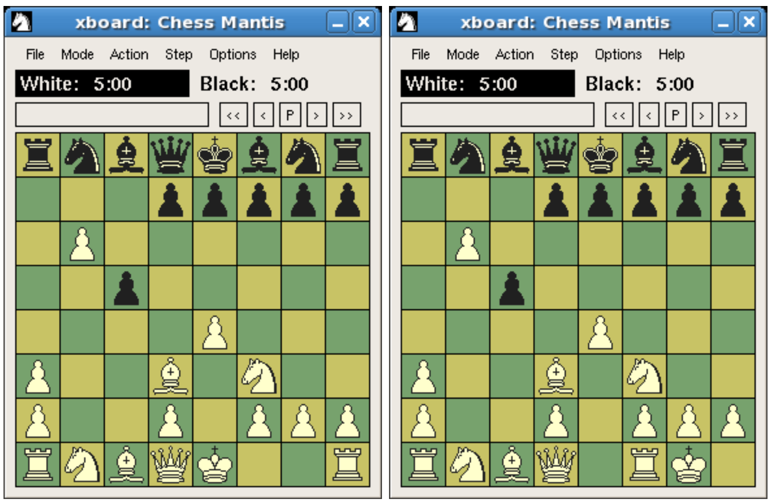
(a) White king initially on e4 (b) White king successfully moved to d4

Figure 5.6: King movement from e4 to d4.

Test Case T7: Castling Movement

A set of test cases were created for the purpose of ensuring that the castling manoever could be both recognised and deemed as a legal move, by the chess engine. Castling can happen both kingside and queenside, with tests for both movements performed. The results showed that both types of castling were performed legally by the chess engine.

It was necessary to create a test case which ensured that castling would only be performed under the correct circumstances, as explained in 4.2.6. The engine was able to recognise illegal castling attempts and rejected them accordingly.



(a) White king initially on e1, white rook on h1

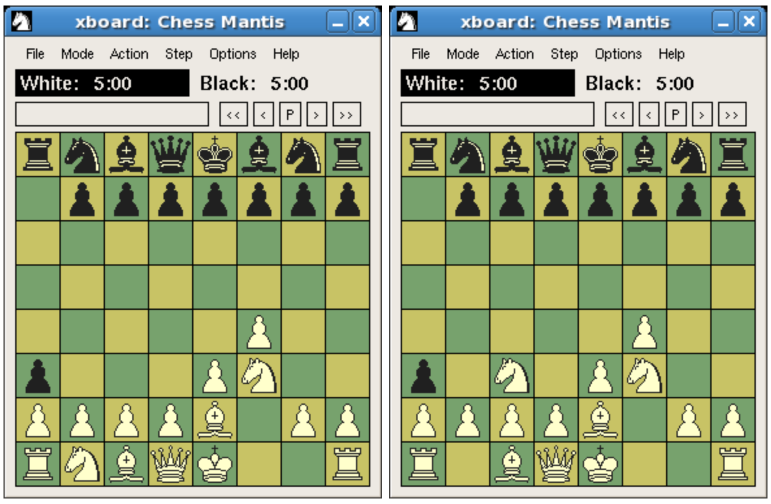
(b) White king successfully moved to g1, white rook moved to f1

Figure 5.7: Successful castling movement

Test Case T8: Blocked Paths

These test cases were created in order ensure legal piece movements would be rejected by the chess engine, should the piece in question's path be blocked. The one piece which is exempt from this rule is the knight, which doesn't have a path of movement.

The results produced conformed with the results expected. All legal piece movements on blocked paths were rejected, with the exception of the knight, which was able to travel 'over' other pieces in its path.



(a) White knight initially on a2

(b) White knight successfully moved to c3

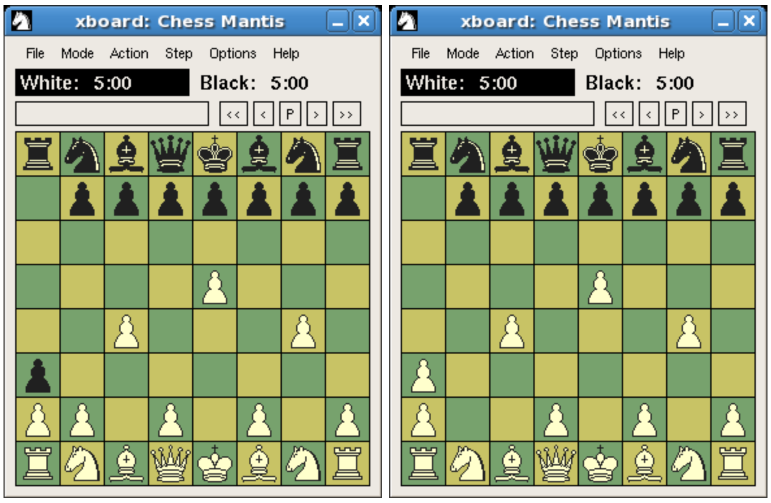
Figure 5.8: Successful blocked path knight movement from a2 to c3

5.1.3 Captures

Test Case T1: Basic Captures

A set of test cases were created in order to ensure the chess engine would allow each of the six piece types to perform their basic respective captures of opposition pieces. A basic capture is one in which a piece makes a standard legal move onto a square occupied by an opposition piece, thus capturing it. The only exception to this is the pawn piece, which cannot capture while performing its basic forward movement, instead it captures diagonally.

The results produced matched the expected results, with all pieces being able to perform their respective basic captures. Pawns were not allowed to capture in a forward movement, but were able to successfully capture diagonally, as desired.



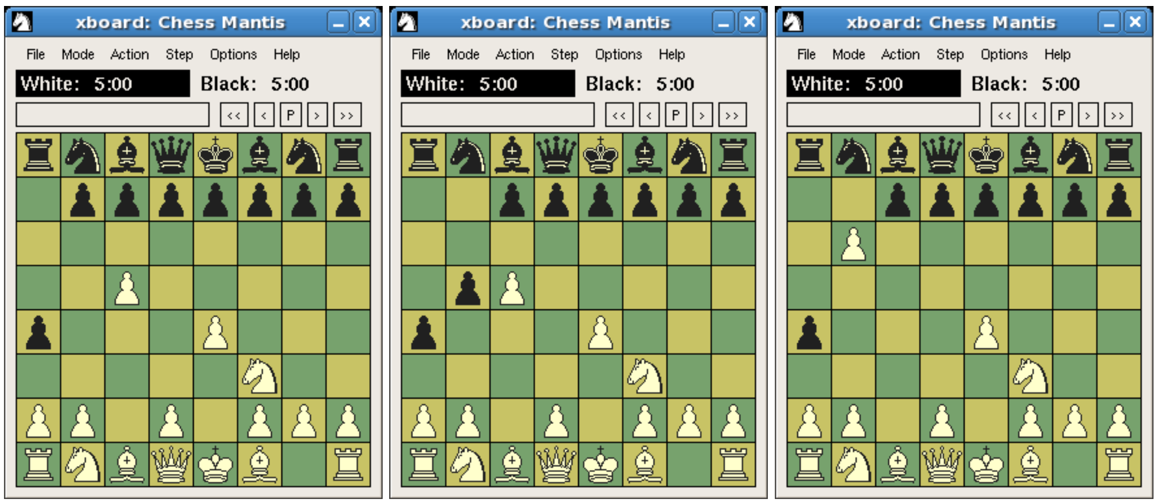
(a) White pawn initially on b2

(b) White pawn successfully moved to a3, capturing the black pawn in the process

Figure 5.9: Successful capture of black pawn on c3

Test Case T2: En Passant Capture

This test case was designed to ensure the chess engine would allow legal en passant captures to be performed as stated in 4.2.8. As expected, before testing was conducted, the engine was able to recognise and perform legal en passant captures, whilst also rejecting illegal attempts made by a pawn to capture en passant.



(a) Black pawn initially on b7 (b) Black pawn double jumps to b5 (c) White pawn moved to b6, successfully capturing the black pawn on b5

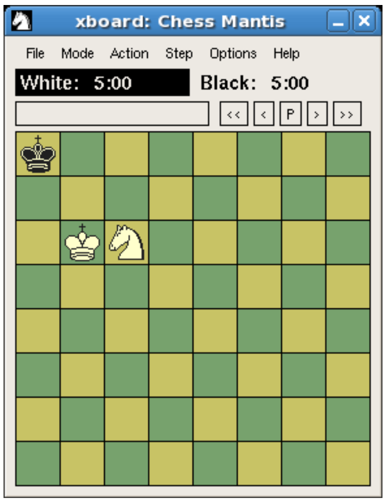
Figure 5.10: Successful capture of black pawn en passant

5.1.4 End of Game Recognition

The purpose of the following test cases is to determine whether the chess engine is able to recognise specific situations which signal the end of the game. These cases are particularly important, as every game must be able to come to a clear, unambiguous conclusion.

Test Case T1: Stalemate

This test case was created to ensure the chess engine would recognise a board which is in a state of stalemate, see 4.2.5. It was expected that the checks the engine performed on the board were sufficient for recognising such a situation. The results produced concurred with this assumption.

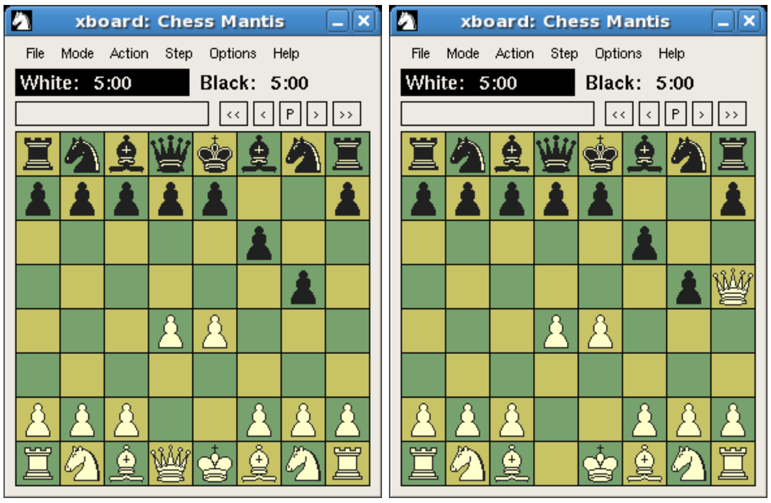


(a) With Black king to move, stalemate results

Figure 5.11: Board in a state of stalemate

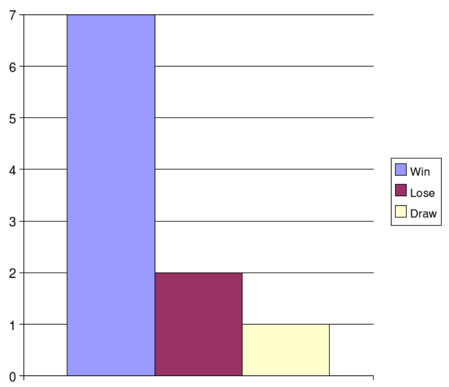
Test Case T2: Checkmate

This test case created to ensure the chess engine was able to correctly identify a board which is in a state of checkmate, see 4.2.5. It was expected that the checks, performed by the engine on the board, were sufficient for recognising such a situation. The results produced concurred with this assumption.

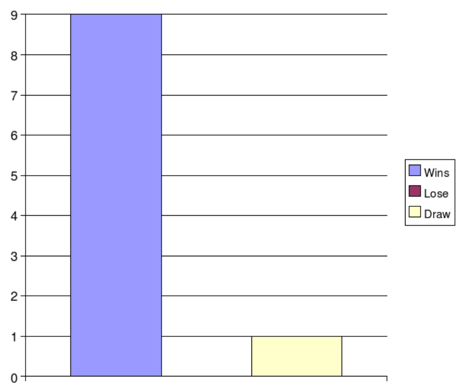


(a) White queen is on d1, with white to move (b) White queen moves to h5, resulting in checkmate

Figure 5.12: White wins by checkmate



(a) minimax games



(b) alpha beta games

Figure 5.13: Game results from the engine's perspective

5.2 Player vs Computer

As well as testing the various rules of chess for compliance, it is important to test the engine against human opponents. Our requirements specification states that our program should give a good game to an intermediate chess player, and so this requirement must be tested thoroughly. The manner in which humans attempt to work out a good move is also drastically different from the brute force strategy employed by a computer chess engine, thus potentially resulting in a slightly different style of play.

In order to asses our engine's performance against human opponents, we devised a standard experiment, to be carried out by a variety of human subjects. This experiment involved the human playing the engine using standard blitz chess rules, 5 minutes per 20 moves, three times, once with each of the analysers we implemented, in order to test the advantages of being able to go deeper into the game tree with alpha beta, in comparison to mini max. As this experiment was deemed to comply with the university rules on ethics, ethics approval was not sought further than our mentor for these experiments.

The participant was not informed of the difference between the three engines until after the experiment, to avoid bias. During the games, the experimenter observed the play of the engine, and after each game recorded its results and the time taken by each side. After all three games were finished, the participant was handed a questionnaire that asked them how many chess games they had played, as a rough gauge of human skill, and which game they felt they had played best in. They were also asked which game they thought the engine had played best in, and whether or not they found it enjoyable to play.

In total ten students of varying experience and ability played the engine. The results for the minimax and alpha beta algorithms are presented in figure 5.13. Clearly simply recording wins and losses is not sufficient to properly evaluate our engine's strength against human players, as it cannot capture information about how the game was played out, however it can serve as a rough guide to engine strength.

We found that the estimate of games played metric for human players was not generally a useful indication of strength, as some players who had played a lot of games had played them at a very low level, and a long time ago. Also, if the engine won within a few moves, as it did in several cases due to error on the part of the human player, then the game could not be treated as a fair test of the computer evaluator. However, the numerous occurrences of this situation do serve to one of the key advantages of any computer chess program, in that it should never make such simple mistakes.

The most significant problem that became apparent with the user testing was getting a checkmate when the human opponents king was free to move about the board. The engine would typically only utilise one or two pieces to check the king, largely failing to trap it, even if other useful pieces were available on the board. The problem would best be solved by creating a separate end game evaluator , that would be simpler and thus capable of evaluating to deeper depths, discarding factors that matter little at endgame.

We discovered early on in the experiments that several other factors were coming into play with our testing, such as human players losing interest and so not concentrating. We also observed during early tests that the participants generally increased their skill against the analyser after the first game (this was especially true of more experienced players). In order to compensate for this effect we randomised the order in which different analysers were played in future human tests, so that our data would be less skewed.

Some human players exhibited a tendency to ignore the time constraints placed upon them, and so some more skilled players lost to the engine through running out of time. We also found one player attempting to use the threefold repetition rule, which is unimplemented by our engine due to its relative difficulty in implementation.

Most players claimed they enjoyed playing the engine, and most of the comments received were about the user interface (Xboard) rather than the play of the engine itself. There are many flaws in the Xboard interface and so in future we might hook our engine up to a different compatible interface.

5.3 Computer vs Computer

One of the best features of using the standard GNU XBoard protocol is that it gives us the ability to play our chess engine against other chess engines that also implement the same interface. This means we can test the engine against other formidable engines. These engines are the result of years of work and some of them are competitors in the Computer Chess Championship, although the three we compared against have never been winners. The following three subsections discuss our engine's performance against all three of these chess engines and have quantitive data showing our performance against them in terms of moves played, outcome of game, and notes on the standard of chess that Chess Mantis played. Considering each of the three chess engines, that have been chosen, can play at well above intermediate level we fully consider our engine to be beaten in every game it plays.[13]

We played each engine on three different timing controls. The first of these is bullet chess, where each side has only 1 minute to make 40 moves, and is generally where computers excel in chess playing as only they can consider a decent move fast enough to win a game. The second of these is blitz chess, a well known type of chess that is used in some tournament matches. Each player is required to make 40 moves in 5 minutes, which roughly works out at 8 seconds per move. This is a good test also because it is the standard timing controls that XBoard uses and are the most likely time controls that will be used when players play our engine normally. The last timing control that we will test is a form of chess called rapid chess. This form of chess requires each side to make 40 moves in a 25 minute time frame. It gives each side 5 times the amount of time as blitz chess to calculate a move, and thus is a form of chess that players will use when playing a computer. We played 6 games against each of these three time control options, playing 3 games as white and 3 as black.

5.3.1 GNUChess vs Chess Mantis

GNUChess is an interesting opponent to play against. It is the standard program that is provided if you download the XBoard interface through Linux, and also is the standard provided if you download WinBoard for windows. GNUChess is a very competitive engine that has a well documented history of playing well in matches against human and computer players.

The following table shows the results of the games, and the number of moves we lasted against this engine. The results shown are in terms of moves lasted each time we played the competitor engine. A first note on the results is that we did lose all of the games we played against the GNUChess.

Bullet Chess	Blitz Chess	Rapid Chess
19	18	17
20	25	22
22	25	23
22	32	23
27	36	29
26	38	31

	Bullet Chess	Blitz Chess	Rapid Chess
Average	23	29	24
Minimum	19	18	17
Maximum	27	38	31

A first note on the results is that we did lose all of the games we played against the GNUChess.

The results clearly show that we are not capable of winning against GNUChess, but the results are surprising none-the-less. The fact that we lasted as long as we did in some cases, and the range in them is particularly impressive.

This shows that on average we play better during Blitz Chess than in the other two types of time controls, which leads us to believe that perhaps that is the optimum time for Chess Mantis to play games in. This could be because we gain at least an extra 2 ply on the analyser search depth over the Bullet Chess game, which would explain the average jump of 6 moves from Bullet Chess to Blitz Chess. Althought GNUChess will receive this same advantage as ourselves, it seems to be the case that we gain the ability to spot more of GNUChess's tactics that is uses when playing us. The sharp decrease from Blitz Chess to Rapid Chess is interesting as it aludes to some deeper happenings within our engine, and GNUChess. Having five times longer to calculate a good move seems to aid GNUChess more than Chess Mantis. This is most likely caused by the fact that our engine can only gain 1 or 2 extra ply in the mid-game from having this extra time. This seems a small increase in the search depth for a huge increase in time, but it was to be expected as the time taken to perform a search increases exponentially as you attempt a deeper search.

Another interesting point is the range of the results. On Bullet Chess time control options the range of 8 moves is to be expected, as the extremely fast speed of the game means both engines resort to moves that have had less search depth behind their searches, and thus may turn out to be bad moves. The lower bound of 18 moves shows though that having 5 times more time aids GNUChess also, allowing it to spot a promising checkmate then executing this move. When using the Rapid Chess time controls we see a range of 14 moves, which in itself is not very interesting, but the maximum amount of moves we achieved, 31 moves, clearly shows that when given more time to calculate a good move, GNUChess can checkmate extremely quickly.

One of the things noticed when watching these games taking place was the fact that GNUChess was extremely adept at trapping the Chess Mantis, forcing it to give up one of its pieces for no gain.

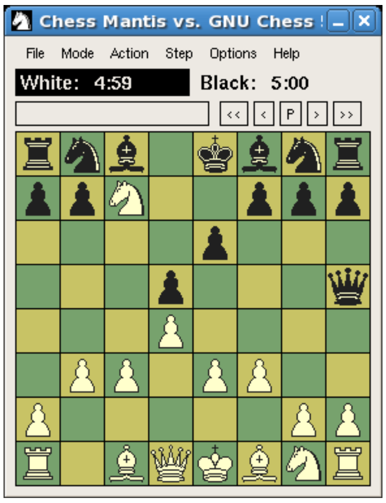


Figure 5.14: White Knight checking the King

The figure above shows that the black King is being checked by the white knight, forcing the King to move from its position, and then allowing the white knight to take the black rook at position a8. This is a typical move that the Chess Mantis fails to spot, along with another few moves of this nature.

In summary, the result we got was entirely expected before we commenced with the testing, and this testing was only to show us problems with our own engine by testing it against a far superior engine that has many years of work in making it as good as it possibly can be, and then seeing the results it provided. It was particularly useful as it shows us exactly where we are struggling, like the example noted above with the knight trap.

5.3.2 Crafty vs Chess Mantis

Crafty was chosen as the second opponent in our computer versus computer testing purely because the XBoard interface website[18] contains a link to this chess engine. On further examination though it turned out to be a competitive engine to play against that was comparable to GNUChess in terms of its playing ability.

The following table shows the results of the games, and the amount of moves we lasted against this engine. The results are in terms of moves lasted each time we played the competitor engine.

Bullet Chess	Blitz Chess	Rapid Chess
18	17	17
21	20	21
24	20	23
25	22	24
26	23	31
26	42	33

	Bullet Chess	Blitz Chess	Rapid Chess
Average	23	24	25
Minimum	18	17	17
Maximum	26	42	33

A first note on the results is that we did lose all of the games we played against the Crafty.

The results are similar to the results obtained from GNUChess in many ways, but also have some noticeable changes. Crafty did not suffer from the same dip in form when playing Blitz Chess as GNUChess did, but it did suffer from games where it played particularly badly. Crafty seemed to be more likely to get itself into worse situations, but as always with engines with as many man hours behind them as Crafty has, it always managed to manipulate the Chess Mantis into an eventual checkmate.

The most noteable thing that Crafty showed up in Chess Mantis was our over aggresiveness. Chess Mantis contains a simple check for piece freedom (see 3.4.8) and controlling the center of the board (see 3.4.9), which gives bonus points for each of those heuristics. This often means that the perceived *'best'* move is moving the Queen to the center of the board, giving us many more moves (as the Queen is the most manoeuverable piece on the board) and thus controls more of the center squares. This can sometimes be a good move as it allows the Queen to become a much more useful piece, but it also greatly increases the chances of losing the most valuable piece (beside the King) to a very simple move. These heuristics seem to have worked against us when playing Crafty because it was very adept at forcing us into a situation where it would be able to remove our Queen from the field of play. What is noteable though is that when playing under Blitz Chess time controls in the game we lasted 42 moves, our Queen did avoid a carefully placed trap from Crafty, and went on to remove some of Crafty's best pieces. Crafty ultimately set up a better trap and the second time it managed to checkmate us, but the important note is that although Chess Mantis does suffer from failing to spot some of these traps, it has the capability to spot others.

The conclusion to draw from playing Crafty is that Chess Mantis needs to be a lot more conservative when dealing with the Queen. This was a known issue from doing preliminary testing when the team was coding the engine, but was not regarded to be as bad a problem as first thought. A simple fix could be made to the evaluator to change this, and thus make the Queen less likely to wander out to gain control of the center of the board, but this would be a hardcoded fix for the Queen, and would not address the main issue that was first noticed when Chess Mantis played GNUChess, the fact that it cannot readily spot traps that are placed by an opponent.

5.3.3 Beowulf vs Chess Mantis

Beowulf was chosen as it is also similar to GNUChess and Crafty in terms of its playing ability, and it is therefore good as a third playing engine to see if we can expose any other issues with the Chess Mantis.

The following table shows the results of the games, and the amount of moves we lasted against this engine. The results are in terms of moves lasted each time we played the competitor engine.

Bullet Chess	Blitz Chess	Rapid Chess
19	24	16
21	25	20
23	28	23
30	32	25
32	32	26
34	36	31

	Bullet Chess	Blitz Chess	Rapid Chess
Average	27	30	24
Minimum	19	24	16
Maximum	34	36	31

A first note on the results is that we did lose all of the games we played against the Beowulf.

The most noticeable thing from the quantitative tables shown above is that the Chess Mantis performed better in general against the Beowulf than the other two engines. Beowulf strangely had no interesting moves or positions that it managed to manoeuver the Chess Mantis into, but it still played solid chess. Chess Mantis suffered from the same setbacks to its playing ability as it did with the previous two engines, suffering from letting the Queen wander too early and also from getting trapped and losing a valuable piece.

Chapter 6

Conclusion

On reflection, one of the first design decisions taken upon embarking on this project, to use the Java language for implementation, was a good one. Though coding in a lower level language such as C could have resulted in greater performance, the choice of Java kept our code reletively clean, maintainable and uncomplex. This allowed us to develop our implementation at a great pace and coordinate easily between team members.

Our first aim was to ensure that the engine gave an an interesting game against a casual chess player. Tests against human players (see 5.2) showed that most players claimed they found the game an enjoyable experience. The personal experience of the developers also reflects this as the engine played at a level that was good enough to provide a serious, though not insurmountable, challenge.

The engine is capable of opening chess games in a strategic manner, without foolishly sacrificing pieces early in the game. This is due to evaluation functions such as the pawn isolation function (see 3.4.6) and the king pawns function (see 3.4.10). However the engine fares worse in the endgame scenarios where it has the advantage. The horizon effect (see 2.2) is especially crippling at this stage. Were more time to be allotted to this project, then resources could be allocated to speeding up both the board representation and the analyser thus allowing searches to greater depths. An alternative, or complementary solution, would be to implement a simpler endgame evaluator (see 5.2).

The heuristics contained within the evaluator encourage the engine to adopt strategic formations of pieces. There are further heuristics that could be added to increase the engine's ability, such as giving knights and rooks a bonus if the engine brings them into play.

The engine successfully integrates with the Chess Engine Communication Protocol (see 3.1) allowing it to interface with numerous clients, including the standard XBoard interface. As only version 2 of this protocol is implemented some older interfaces cannot be used with the engine. Ideally we would also implement the Universal Chess Interface, to allow us to connect to a wider range of clients. Our support for CECP allows us to easily load and save games through client interfaces. The engine uses its own move generation code in order to check whether or not an opponents moves are legal.

The engine, thanks to its implementation of iterative deepening (see 3.3.1), supports various time controls. These include any time limit per move, and any number of moves per unit of time. This aspect of our chess engine can clearly be deemed a success.

The place where the engine is clearly found wanting is its lack of implementation of certain more obscure chess rules, for example the threefold repetition rule[10] and the fifty move rule[7]. Also, as was outlined above, its inability to finish certain games should be considered a serious flaw.

However despite these flaws, the engine can be deemed an overall success. It implements the vast majority of chess rules and plays a game that is considered challenging and enjoyable by most to beginner and intermediate players who have faced it. Though the engine is certainly no match for other computer engines (see 5.2), given the developement time invested, its results are impressive.

Appendix A

Contributions

A.1 Paul Dailly

- Investigated many of the concepts behind the implementation of computer chess engines, such as game trees, board representations etc.
- Worked with Alec Macdonald, setting up and contributing to the maintenace of the Game class.
- Researched various chess heuristics, paricularly pawn formations, for use in evaluator.
- Worked with Neil Henning on the Sifaka evaluator, implementing various pawn heuristics.
- Implemented checkmates and stalemates for the board.
- Worked with Alec Macdonald implementing and debugging the castling rule.
- Implemented and debugged the initial version of the en passant rule.
- Worked with Alec Macdonald and others debugging his version of the en passant rule.
- Worked, mainly with Alec Macdonald, maintaining, debugging and evolving many features of the board.
- Contributed to all chapters of the report, excluding the introduction and the conclusion.

A.2 Dominik Gotojuch

- Investigated different board representations, MiniMax and Alpha Beta search algorithms and examined existing chess engines
- Implemented the abstract GenericPiece class and piece subclasses and continued extending and improving these
- Implemented the BoardArray class with Tamerlan Tajaddinov and continued extending and improving it

- Worked on integrating the BoardArray class with the Move and Piece classes, as well as engine's XBoard interface support
- Implemented Zobrist hashing of the board representation
- Rewrote the Alpha Beta analyser for testing and debugging purposes and helped with improving the original implementation of Alpha Beta
- Investigated possible enhacements of the analyser and worked on initial implementation of the NegaMax Alpha Beta, Principal Variation Search, transposition tables and history heuristics
- Integrated the engine with WinBoard, allowing Chess Mantis to work in Microsoft Windows environment
- Contributed to the Introduction, Design and Implementation sections of the report
- Took part in all of the team debugging sessions, focusing on the board representation and the analysers
- Participated in human testing and in numerous, team testing sessions

A.3 Neil Henning

- Investigated chess heuristics for future work on the chess engine
- Designed the Class Diagram and interface layout for the project
- Implemented the interfaces from the Class Diagram design and created the basic classes in a primative form
- Worked on a DebugWriter swing application that would be used to show the debug output from our engine (was later deprecated in favour of XBoard's own -debug flag)
- Work with Keir Lawson on CECP support
- Implemented minimax analyser
- Implemented Marmoset, a random evaluator
- Implemented Capuchin, an evaluator for Material evaluation
- Implemented Sifaka with Paul Dailly, an extension of Capuchin that considered various pawn heuristics
- Implemented Macaque with Tamerlan Tajaddinov, an extension of Sifaka that considered various positional heuristics
- Implemented Mantis, a combination of the three previous evaluators, to increase efficiency
- Added in checkmate heuristic to Mantis to allow engine to close games
- Added in has castled heuristic to Mantis to allow it to perform castling

- Worked with Alec Macdonald, Paul Dailly, Dominik Gotojuch on debugging board implementation
- Contributed to various sections of the report

A.4 Keir Lawson

- Set up and oversaw Subversion repository
- Worked with Neil Henning on CECP support
- Helped debug board implementation
- Worked on minimax analyser
- Implemented alpha beta analyser
- researched and implemented iterative deepening, time control
- implemented aditional xboard commands for time control, depth limitation
- helped implement promotion
- implemented dynamic loading of analyser and evaluator at runtime, via reflection
- oversaw user testing
- contributed to all sections of the report

A.5 Alec Macdonald

- Investigated Chess and Computing Chess history
- Picked up Neil's basic Game and BoardArray classes and developed
- Implemented support for xboard commands such as force, go and setboard
- Implemented various rules including movement, capture, promotion, castling and en passant with Paul, Tam and Dom
- Investigated alternative CECP user interfaces with 3D support
- Worked on user testing with Keir
- Debugged with the rest of the team
- Wrote various sections of the report

A.6 Tamerlan Tajaddinov

- Implemented the Move class and added initial support for promotions
- Implemented the BoardArray class with Dominik Gotojuch and provided an ongoing support and maintainance to the move, board and piece representations
- Implemented support for promotions
- Optimised the entire algorithm reducing the time taken to evaluate the board to a given number of moves by a factor of two.
- Implemented a number of drivers, most importantly BoardArrayDemo, that enabled to view the internal states of the board
- Worked with Neil Henning on the implementation of Sifaka, an evaluator, providing heuristics for evaluating the positioning
- Worked with Neil Henning on the implementation of Mantis evaluator
- Contributed to Background, Design and Implementation sections of the report
- Oversaw the background section of the report
- Worked with the rest of the team on the ongoing debugging of the engine and conducted numerous test sessions

Appendix B

Status Report

B.1 Product

The product plays chess using the XBoard standard protocol and gives a decent game of chess at intermediate level. It is fully working and has been tested to ensure all of the standard and special chess rules are working correctly. It has also been tested against other computer chess playing engines and players to give as broad a testing set as is possible.

B.2 Deficiencies

B.2.1 XBoard Commands

The engine does not implement the variant command, which allows for other small variants of chess to be used, for example suicide chess that needs to throw away all its pieces first. Was not deemed to be an important feature. We do not implement the st command, which allows the board to set the time controls. We do however implement the tc which performs the exact same functionality. Do not implement the time or the otim as these give different times for each player, and we want to make it fairer and keep the time constant for each player. We also require the XBoard to send us the move with usermove before the actual move, to allow it to be easier for us to parse the incoming commands. We also do not implement the hint command, although it would be easy to implement as the background functionality is in place to calculate a good move for the player to make. We also do not support the ability that XBoard has to be able to play internet chess games. This was beyond the scope of our engine in the time given.

B.2.2 Features

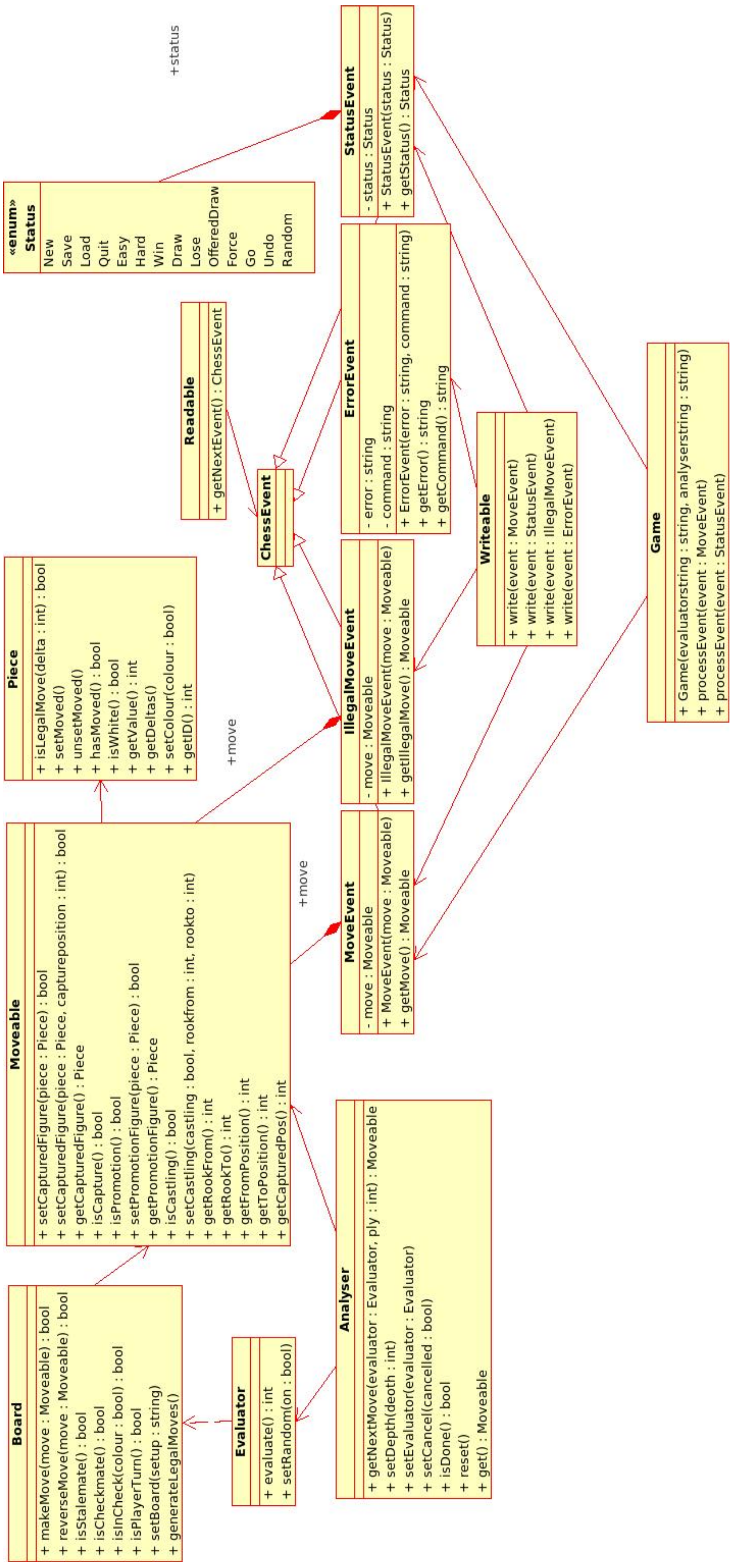
We implemented the start of a feature that would allow us to speed up the time taken for each analysis of the game tree. We managed to implement Zoborist Hashing, which is the basis for further optimisations of the Analyser, the ability to do transposition tables and also killer/history heuristic. The transposition tables are used in Principal Variation Search and MTD(f) search algorithms, one of which was planned to become the search algorithm of the engine's analyser. These features are much more complex and harder to implement than we first perceived, so we decided to work on other features instead and leave these as a potential, future extension of the engine.

B.2.3 Rules

The fifty move rule and the three fold repetition rule remain unimplemented as their implementation would be difficult within the time constraints.

Appendix C

Class Diagram


«enum»
Status
New
Save
Load
Quit
Easy
Hard
Win
Draw
Lose
OfferedDraw
Force
Go
Undo
Random
+status
StatusEvent
- status : Status
+ StatusEvent(status : Status)
+ getStatus() : Status
Readable
+ getNextEvent() : ChessEvent
ChessEvent
ErrorEvent
- error : string
- command : string
+ ErrorEvent(error : string, command : string)
+ getError() : string
+ getCommand() : string
Writeable
+ write(event : MoveEvent)
+ write(event : StatusEvent)
+ write(event : IllegalMoveEvent)
+ write(event : ErrorEvent)
Game
+ Game(evaluatorstring : string, analyserstring : string)
+ processEvent(event : MoveEvent)
+ processEvent(event : StatusEvent)
Piece
+ isLegalMove(delta : int) : bool
+ setMoved()
+ unsetMoved()
+ hasMoved() : bool
+ isWhite() : bool
+ getValue() : int
+ getDeltas()
+ setColour(colour : bool)
+ getID() : int
+move
IllegalMoveEvent
- move : Moveable
+ IllegalMoveEvent(move : Moveable)
+ getIllegalMove() : Moveable
+move
MoveEvent
- move : Moveable
+ MoveEvent(move : Moveable)
+ getMove() : Moveable
Moveable
+ setCapturedFigure(piece : Piece) : bool
+ setCapturedFigure(piece : Piece, captureposition : int) : bool
+ getCapturedFigure() : Piece
+ isCapture() : bool
+ isPromotion() : bool
+ setPromotionFigure(piece : Piece) : bool
+ getPromotionFigure() : Piece
+ isCastling() : bool
+ setCastling(castling : bool, rookfrom : int, rookto : int)
+ getRookFrom() : int
+ getRookTo() : int
+ getFromPosition() : int
+ getToPosition() : int
+ getCapturedPos() : int
Board
+ makeMove(move : Moveable) : bool
+ reverseMove(move : Moveable) : bool
+ isStalemate() : bool
+ isCheckmate() : bool
+ isInCheck(colour : bool) : bool
+ isPlayerTurn() : bool
+ setBoard(setup : string)
+ generateLegalMoves()
Evaluator
+ evaluate() : int
+ setRandom(on : bool)
Analyser
+ getNextMove(evaluator : Evaluator, ply : int) : Moveable
+ setDepth(deoth : int)
+ setEvaluator(evaluator : Evaluator)
+ setCancel(cancelled : bool)
+ isDone() : bool
+ reset()
+ get() : Moveable

Appendix D

Requirements Specification

D.1 Functional requirements

- Ability to play chess
- Should be multiplatform, being capable of running on at least Linux and Windows
- Has to implement and enforce all chess rules except the fifty move rule and the three fold repetition rule
- Must be able to play against both human and computer opponents
- Must be a graphical application

D.2 Non-functional requirements

- Ability to generate a move in short time controls, under 10 second on a lab computer, so that the player does not need to wait excessive periods of time.
- Ability to play chess at an intermediate level, so a moderately skilled player can have an interesting game.
- The user interface must be easy to use

Appendix E

Coding Style

Case: ObjectName

methodName

variablename

Structure:

```
functionName (variables)
{
<– Tab –>Line1
<– Tab –>loop
<– Tab –>{
<– Tab –><– Tab –>lineinloop
<– Tab –>}
}
```

Comments:

Include javadoc in every method

Every other comment should start with initials

Bibliography

[1] Alpha-beta pruning. http://en.wikipedia.org/wiki/Image:AB_pruning.svg.

[2] Brains in bahrain. http://en.wikipedia.org/wiki/Brains_in_Bahrain.

[3] Castling. http://en.wikipedia.org/wiki/Castling.

[4] Chess check. http://chess.vpst.org/check.html.

[5] Computer chess. http://en.wikipedia.org/wiki/Computer_chess.

[6] El ajedrecista. http://en.wikipedia.org/wiki/El_Ajedrecista.

[7] Fifty move rule. http://en.wikipedia.org/wiki/Fifty_move_rule.

[8] Game over: Kasparov and the machine. http://en.wikipedia.org/wiki/Game_Over:_Kasparov_and_the_Machine.

[9] Laws of chess. http://www.fide.com/official/handbook.asp?level=EE101.

[10] Threefold repetition rule. http://en.wikipedia.org/wiki/Threefold_petition.

[11] Debian. Computer language banchmarks. http://shootout.alioth.debian.org/debian/benchmark.php?test=all.

[12] J. Gallagher. *Starting Out: the Caro-Kann*. Everyman Chess, 2002.

[13] GNUChess. Gnuchess rating. http://www.gnu.org/software/chess/chess_faq.html.

[14] R. Hyatt. Chess program board reprsentations.

[15] Petterssonm J. Medicore chess. Technical report, Ume University, 2007.

[16] L. Kaufman. *The Evaluation of Material Imbalances*. Chess Life, 1999.

[17] F. Laramee. Chess programming part 2: Data structures. 2004.

[18] T. Mann. Chess engine communication protocol. http://www.tim-mann.org/xboard/engine-intf.html.

[19] B. Moreland. Principal variation search. http://www.seanet.com/ brucemo/topics/pvs.htm.

[20] B. Moreland. 0x88 move generation. 2001.

[21] P. Rayson. Iterative deepening. http://www.comp.lancs.ac.uk/computing/research/aai-aied/people/paulb/old243prolog/subsection3_6_4.html.

[22] A. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):211–229, 1959.

[23] J. Stanback. Gnu's heuristics. Technical report, Free Software Foundation, Inc., 1987.

[24] CCRL Team. Ccrl 40/40 rating list. http://www.computerchess.org.uk/ccrl/4040/.

[25] J. Van der Steen. Introduction to the elo rating system. http://gobase.org/studying/articles/elo/.

[26] J. Von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[27] M. Wichitill. Is rykba a derivative of fruit. http://rybkaforum.net/cgi-bin/rybkaforum/topic_show.pl?pid=19118#pid19118.

[28] N. Zibaldone. How to test a chess engine. http://lozibaldonedinicola.blogspot.com/2007/08/how-to-test-chess-engine.html.

[29] Albert L. Zobrist. A hashing method with applications for game playing, tech. rep. 88. Technical report, The Univeristy of Wisconsin, Madison, Wisconsin, 1970.