ANALYSIS AND IMPLEMENTATION OF THE GAME ARIMAA

Christ-Jan Cox

Master Thesis MICC-IKAT 06-05

THESIS SUBMITTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE OF KNOWLEDGE ENGINEERING
IN THE FACULTY OF GENERAL SCIENCES
OF THE UNIVERSITEIT MAASTRICHT

Thesis committee:

prof. dr. H.J. van den Herik
dr. ir. J.W.H.M. Uiterwijk
dr. ir. P.H.M. Spronck
dr. ir. H.H.L.M. Donkers

Universiteit Maastricht
Maastricht ICT Competence Centre
Institute for Knowledge and Agent Technology
Maastricht, The Netherlands
March 2006

Preface

The title of my M.Sc. thesis is *Analysis and Implementation of the Game Arimaa*. The research was performed at the research institute IKAT (Institute for Knowledge and Agent Technology). The subject of research is the implementation of AI techniques for the rising game Arimaa, a two-player zero-sum board game with perfect information. Arimaa is a challenging game, too complex to be solved with present means. It was developed with the aim of creating a game in which humans can excel, but which is too difficult for computers.

I wish to thank various people for helping me to bring this thesis to a good end and to support me during the actual research. First of all, I want to thank my supervisors, Prof. dr. H.J. van den Herik for reading my thesis and commenting on its readability, and my daily advisor dr. ir. J.W.H.M. Uiterwijk, without whom this thesis would not have been reached the current level and the "accompanying" depth. The other committee members are also recognised for their time and effort in reading this thesis.

Finally, I would like to thank Omar Syed (inventor of the game Arimaa) for supplying the basic interface for playing Arimaa. Moreover, I would like to express my gratitude towards David Fotland, who provided me with relevant information on the game and how to include positions. Also, I want to thank my former student fellow Mark Winands for sharing his knowledge about the Java language. All in all, I am pleased that the research has been completed and the thesis has been written. I look back at a pleasant time with IKAT and forward to a future as Knowledge Engineer.

Christ-Jan Cox
Hoensbroek, March 2006

Abstract

Computers that play board games are still intriguing phenomena. For many years they are thoroughly studied with the aim to defeat the strongest humans. Of course, the most illustrious example of all in the game of Chess is where DEEP BLUE won against Kasparov in 1997. Computer-Chess research started in 1950 when Claude E. Shannon wrote the first article on computer Chess.

This thesis describes the problem domain and the intricacies of the game of Arimaa. Arimaa is a two-player zero-sum board game with perfect information. In this thesis Arimaa will be analysed and implemented. The name of the program is COXA. We begin to compute the complexities of the game Arimaa: (1) the state-space complexity of Arimaa is $O(10^{43})$ and (2) the game-tree complexity is $O(10^{300})$.

Then, an alpha-beta search with iterative deepening and a little knowledge of Arimaa is performed. This algorithm is refined with transposition tables and heuristics. Finally, some experiments are performed to fine-tune the basic and extended evaluation functions.

Material balance, piece-square tables, and mobility are important components in basic evaluation concerning Arimaa. The large weight of the elephant piece-square table is striking, indicating that it is important to keep the elephant in the centre. For the extended evaluation the rabbit evaluation is by far the most important one, with a weight of 50 for free files.

Principal variation search, transposition tables, and the killer-move heuristic are enhancements that have positive influence on the standard alpha beta with iterative-deepening.

For future research we recommend a further fine-tuning of the weights, the inclusion of more elaborate knowledge in the evaluation function, and investigating the usefulness of more search enhancements.

Contents

PREFACE ..... **III**

ABSTRACT ..... **V**

CONTENTS ..... **VII**

LIST OF FIGURES ..... **IX**

LIST OF TABLES ..... **X**

1 INTRODUCTION **1**

1.1 History of Computer-Game Playing 1
1.2 The Game of Arimaa 3
1.2.1 The Rules of the Game Arimaa *4*
1.2.2 Match Rules of Arimaa *6*
1.2.3 Notation *8*
1.3 Problem Statement and Research Questions 10
1.4 Thesis Outline 11

2 COMPLEXITY ANALYSIS **13**

2.1 State-space Complexity 13
2.2 Game-tree Complexity 17
2.3 Comparing Arimaa to Other Games 18
2.4 Chapter Conclusion 20

3 EVALUATION OF POSITIONS **21**

3.1 The Basic Evaluation 21
3.1.1 Material *21*
3.1.2 Piece-square Tables *22*
3.1.3 Mobility *23*
3.2 The Extended Evaluation 24
3.2.1 Rabbit Evaluation *24*
3.2.2 Goal-threat Evaluator *24*
3.2.3 Trap-control Evaluator *25*
3.3 Chapter Conclusion 26

4 SEARCHING THE TREE **27**

4.1 Building the Tree 27
4.1.1 Alpha-beta *27*
4.1.2 Iterative Deepening *28*
4.1.3 Principal Variation Search *28*
4.1.4 Null-move Search *28*
4.2 Transposition Table 29
4.2.1 Hashing *30*
4.2.2 Use of a Transposition Table *30*
4.2.3 Probability of Errors *31*
4.3 Move Ordering 32
4.3.1 Game-specific Heuristics *32*
4.3.2 Killer-move Heuristic *32*
4.3.3 History Heuristic *33*
4.4 Overall Framework 33

5 TUNING THE EVALUATION FUNCTION **35**

5.1 Setup of the Tuning 35
5.1.1 Random Factor *35*
5.2 Tuning the Basic Evaluation 36
5.2.1 Material Weight *37*
5.2.2 Piece-square Tables *38*
5.2.3 Mobility *44*

5.3 TUNING THE KNOWLEDGE EVALUATOR 45
5.3.1 Rabbit Evaluation *46*
5.3.2 Goal-threat Evaluator *48*
5.3.3 Trap-control Evaluator *49*
5.4 CHAPTER CONCLUSION 51

6 TESTING OF SEARCH ALGORITHM 53

6.1 WINDOWING TECHNIQUES 53
6.1.1 Principal Variation Search *53*
6.1.2 Null-move Search *54*
6.2 TRANSPOSITION TABLE 54
6.2.1 Only using Exact Table Hits *55*
6.2.2 Only using Upper and Lower Bound Pruning *55*
6.2.3 Only using Move Ordering *55*
6.2.4 Full use of the Table. *56*
6.3 MOVE ORDERING 56
6.3.1 Killer-move Heuristic *57*
6.3.2 History Heuristic *59*
6.4 CHAPTER CONCLUSIONS 59

7 CONCLUSIONS 61

7.1 PROBLEM STATEMENT AND RESEARCH QUESTIONS REVISITED 61
7.2 FUTURE RESEARCH POSSIBILITIES 62

REFERENCES 63

APPENDIX A: EXAMPLES OF TIME CONTROL IN ARIMAA 65

APPENDIX B: ALL UNIQUE PATTERNS AND THEIR HITS FOR EACH n 67

List of Figures

Figure 1.1: The Arimaa board. 3
Figure 1.2: A sample position with Gold to move. 5
Figure 1.3: Position after Gold's move. 5

Figure 2.1: Average branching factor per ply. 18
Figure 2.2: Estimated game complexities. 20

Figure 4.1: A starting position. 29
Figure 4.2: Position after moves e2-e3, e3-f3, d2-d3, d3-d4. 29

Figure 5.1: Influence of the random weight on the difference of moves for each depth. 36
Figure 5.2: Winning % of the random value. 36
Figure 5.3: Winning % of the material value. 37
Figure 5.4: Winning % of the material value (zoomed in). 37
Figure 5.5: Winning % of the rabbit piece-square table. 38
Figure 5.6: Winning % of the rabbit piece-square table (zoomed in). 38
Figure 5.7: Winning % of the cat piece-square table. 39
Figure 5.8: Winning % of the cat piece-square table (zoomed in). 39
Figure 5.9: Winning % of the dog piece-square table. 40
Figure 5.10: Winning % of the dog piece-square table (zoomed in). 40
Figure 5.11: Winning % of the horse piece-square table. 41
Figure 5.12: Winning % of the horse piece-square table (zoomed in). 41
Figure 5.13: Winning % of the camel piece-square table. 42
Figure 5.14: Winning % of the camel piece-square table (zoomed in). 42
Figure 5.15: Winning % of the elephant piece-square table. 43
Figure 5.16: Winning % of the elephant piece-square table (zoomed in). 43
Figure 5.17: Winning % of the frozen-weight value. 44
Figure 5.18: Winning % of the frozen-weight value (zoomed in). 44
Figure 5.19: Winning % of the partial-mobility weight value. 45
Figure 5.20: Winning % of the partial-mobility weight value (zoomed in). 45
Figure 5.21: Winning % of the solid-wall weight value. 46
Figure 5.22: Winning % of the solid-wall weight value (zoomed in). 46
Figure 5.23: Winning % of the free-file weight value. 47
Figure 5.24: Winning % of the free-file weight value (zoomed in). 47
Figure 5.25: Winning % of the goal-threat weight value. 48
Figure 5.26: Winning % of the goal-threat weight value (zoomed in). 48
Figure 5.27: Winning % of the trap-control weight value. 49
Figure 5.28: Winning % of the trap-control weight value (zoomed in). 49
Figure 5.29: Winning % of the bonus weight value. 50
Figure 5.30: Winning % of the bonus weight value (zoomed in). 50

List of Tables

Table 2.1: An overview of results needed for the State-Space Complexity..... 14
Table 2.2: Summation of the number of illegal positions where one gold piece is trapped. ... 15

Table 3.1: Material values. 21
Table 3.2: Material values for the Rabbits on the board. 21
Table 3.3: Rabbit piece-square table for Gold. 22
Table 3.4: Cat/Dog piece-square table for Gold..... 22
Table 3.5: Horse/Camel/Elephant piece-square table for Gold..... 23
Table 3.6: Mobility values according to the piece strength. 23
Table 3.7: Free files bonus according to the Rabbit rank..... 24
Table 3.8: Bonuses first and second pieces controlling a trap. 25

Table 5.1: Overview of the weight values of the tuned evaluation function..... 51

Table 6.1: Principal Variation Search (all steps) test results..... 53
Table 6.2: Principal Variation Search (last step) test results..... 54
Table 6.3: Null-move search test result..... 54
Table 6.4: TT (exact) test results. 55
Table 6.5: TT (upper / lower bound) test results. 55
Table 6.6: TT (ordering) test results..... 56
Table 6.7: TT (full) test results..... 56
Table 6.8: Killer-move heuristic (record one) test results. 57
Table 6.9: Killer-move heuristic (record two) test results..... 57
Table 6.10: Killer-move heuristic (record three) test results..... 58
Table 6.11: Killer-move heuristic (record four) test results. 58
Table 6.12: Killer-move heuristic (record five) test results. 58
Table 6.13: History-heuristic test results..... 59

1 Introduction

Games have always been one of humans' favourite ways to spend time, not only because they are fun to play, but also because of their competitive element. This competitive element is the most important impetus for people to excel in the games they play. In the past few centuries, certain games have gained a substantial amount of status, which resulted in even more people endeavouring to become the best in the game. Many games have evolved from just a way to spend some free time to a way of making a reputation. For some people games have even become their profession. This is an evolution, which will probably not stop in the foreseeable future. One of the trends which we have seen in the last few years is that people start to use computers to solve small game problems so that they can understand the problems and learn how to handle them. Computers are thus being used as a tool for people to help them in their quest for excellence.

In this chapter we will first give a chronological overview of the history of computer-game playing in section 1.1. Next we will introduce the game of Arimaa in section 1.2. This is largely adopted from the Arimaa official website (Syed and Syed, 1999). In section 1.3 we give the research questions and section 1.4 provides the outline of this thesis.

1.1 History of Computer-Game Playing

One of the consequences of the foreseeable computer use is that, in the long run, computers will be able to play a game superior to any human being. This has already happened in a number of games, with the most illustrious example of all being Chess where DEEP BLUE won against Kasparov in 1997 (Schaeffer and Plaat, 1997). Computer-Chess research started in 1950 when Claude E. Shannon wrote the first article on computer Chess (Shannon, 1950). Shannon noted the theoretical existence of a perfect solution to Chess and the practical impossibility of finding it. He described two general strategies that were both based on a heuristic *evaluation function* for guessing whether a position is in favour of one person or the other. Modern chess programs still follow the lines laid out by Shannon. In 1951 Alan Turing described and hand-simulated a computer chess program (Turing, 1953). Its play is best described as "aimless"; it loses to weak players.

In 1956, the first documented account of a running chess program was published, containing experiments on a Univac Maniac I computer (11,000 operations per second) at Los Alamos. The machine played Chess on a 6×6 chess board and a chess set without Bishops. It took 12 minutes to search 4 moves deep. Adding the two Bishops would have taken 3 hours to search 4 moves deep. Maniac I had a memory of 600 words, storage of 80 K, 11 KHz speed, and had 2,400 vacuum tubes. The team that programmed Maniac was led by Stan Ulam. Its play defeated weak players (Kister *et al.*, 1957). Alex Bernstein at MIT (Bernstein *et al.*, 1958) wrote a chess program for an IBM 704 that could execute 42,000 instructions per second and had a memory of 70 K. This was the first time that a full-fledged game of Chess was played by a computer. It did a 4-ply search in 8 minutes and its play was like a passable amateur.

The alpha-beta pruning algorithm for Chess was first published in 1958 by three scientists at Carnegie-Mellon (Newell, Shaw and Simon, 1958). It is the general game-search technique which effectively enlarges the length of move sequences one can examine in a considerable way. In 1959 Arthur Samuel wrote his first article on experimenting with automatic-learning techniques to improve the play of a checkers program (Samuel, 1959, 1967).

In 1962 the Kotok-McCarthy MIT chess program was written. It was the first chess program that played regular chess credibly. It was written by Alan Kotok for his B.Sc. thesis project (Kotok, 1962), assisted by John McCarthy who at that time moved to Stanford. Another MIT program called MACHACK by Greenblatt (1967) ran on an IBM 7090 computer, examining 1100 positions per second. Five years later in the spring of 1967, MACHACK VI became the first program to beat a human (1510 USCF rating), at the Massachusetts State Championship. By the end of that year, it had played in four chess tournaments. It won 3 games, lost 12, and drew 3. In 1967 MACHACK VI was made an honorary member of the US Chess Federation. The MACHACK program was the first widely distributed chess program, running on many of the PDP machines. It was also the first to have an opening chess book programmed with it.

The first Go program was written by Zobrist in 1968 (Zobrist, 1969). Its play was like a complete beginner.

The TECHNOLOGY chess program won 10 pts out of 26 in six tournaments in 1971. This was the first chess program written in a high-level programming language (Gillogly, 1972). It ran on a PDP-10 (1 MHz), and examined 120 positions/second.

Programmers Slate and Atkin revised their overly complicated chess program in preparation for the Computer Chess Championships in 1973. There their program CHESS 4.0 won. On a CDC 6400 a later version (CHESS 4.5) processed from 270 to 600 positions/second (Atkin and Slate, 1977).

The development of CRAY BLITZ started in 1975 by Robert Hyatt (Hyatt, Gower and Nelson, 1985). For a long time it was the fastest program and from 1983-1989 the World Computer Chess Champion. It was searching 40-50K positions/second in 1983, only a little slower than current programs on fast Pentiums. Hyatt is still very active today in computer Chess with his free program CRAFTY.

In 1977, BELLE was the first computer system to use custom-design chips to increase its playing strength. It increased its search speed from 200 positions per second to 160,000 positions per second (8 half moves or *ply*). Over 1,700 integrated circuits were used to construct BELLE (Thompson, 1982). The chess computer was built by Ken Thompson and Joe Condon. The program was later used to solve endgame problems. In the same year CHESS 4.6 beat a grandmaster (Stean) at speed chess.

IAGO played Othello at world-championship level (according to then human champion Jonathan Cerf) in 1982 but did not actually play against championship-level human competition.

In 1988 DEEP THOUGHT, predecessor of DEEP BLUE, was created by a team of Carnegie-Mellon University graduate students. The basic version of DEEP THOUGHT's chess engine contained 250 chips and two processors on a single-circuit board and was capable of analyzing 750,000 positions per second or 10 ply ahead. That same year DEEP THOUGHT became the first computer that defeated a Grandmaster in a tournament (Bent Larsen, who had at one time been a contender for world champion, being defeated by Bobby Fischer in a preliminary championship round). An experimental six-processor version of DEEP THOUGHT in 1989, searching more than two million positions/second, played a two-game exhibition match against Gary Kasparov, the reigning world champion, and was beaten twice (Hsu, 2004).

In Checkers the CHINOOK checkers program lost a match to the human world champion in 1992, Marion Tinsley, 4-2 (with 33 draws). CHINOOK became world checkers champion, in 1994, because Tinsley forfeited his match to CHINOOK due to illness (Schaeffer, 1997). Thereafter, CHINOOK has defended its world title several times successfully.

DEEP THOUGHT defeated Judit Polgar, at the time the youngest Grandmaster in history and still the strongest female player in the world (ranked in the top 20 grandmasters), in another two-game exhibition match in 1993.

DEEP BLUE, the new IBM's chess machine from 1996 onwards (a 32-processor parallel computer with 256 VLSI chess engines searching 2-400M moves/second), beat reigning world champion Gary Kasparov in the first game of a six-game match, but lost the match. And then there was the moment on May 11, 1997, when DEEP BLUE defeated Garry Kasparov in a six-game match held in New York. This was the first time a computer defeated a reigning world champion in a classical chess match. DEEP BLUE had 30 IBM RS-6000 SP processors coupled to 480 chess chips. It could evaluate 200 million moves per second (Schaeffer and Plaat, 1997).

It was clear that computers had finally become at a par with the strongest humans in the game of Chess. Millions of people around the world watched and wondered if computers were really getting to be as intelligent as humans. But have computers really caught up to the intelligence level of humans? Do they now have ***real* intelligence** (Syed and Syed, 1999)?

1.2 The Game of Arimaa

In an attempt to show that computers are not even close to matching the kind of *real* intelligence used by humans in playing strategy games, Omar and Aamir Syed created a new game called Arimaa.

Arimaa is a two-player zero-sum game with perfect information. The game can be played using the same board and pieces provided in a standard chess set. The game is played between two sides, Gold and Silver. To make the game easier to learn for someone who is not familiar with Chess the chess pieces are substituted with well-known animals. The substitution is as follows: Elephant for King, Camel for Queen, Horse for Rook, Dog for Bishop, Cat for Knight and Rabbit for Pawn, see figure 1.1. It shows the Arimaa board with on top the standard chess setup and at the bottom the corresponding Arimaa pieces (a possible Arimaa setup). The 4 traps are shaded (c3, c6, f3, f6).

Figure 1.1: The Arimaa board.

The rules of the game are a bit different from Chess. All of a sudden the computers are left way behind. How is this Possible? Even the fastest computers at this moment can not beat a good human player, according to Syed and Syed (1999). For humans the rules of Arimaa are quite easy to understand and more intuitive than Chess, but to a computer the game is more complex. To the best of their knowledge Arimaa is the first game that was designed intentionally to be difficult for computers to play. The rules of Arimaa were chosen to be as simple and intuitive as possible while at the same time it served the purpose of making the game interesting to play and yet difficult for computers. There are several reasons why Arimaa is difficult for computers to play. We will highlight them in Chapter 2. for this reason Syed made an Arimaa Challenge. A prize of $10,000 USD would be awarded to the first person, company or organization that developed a program that defeats a chosen human Arimaa player in an official Arimaa challenge match before the year 2020 (Syed, 1999). The official challenge match will be between the current best program and a top-ten-rated human player.

1.2.1 The Rules of the Game Arimaa

Goal of the game

The goal of the game Arimaa is to be the first to get one of your Rabbits to the other side of the board.

Setup of Arimaa

The game starts with an empty board. The player with the gold pieces (called Gold from now on) sets them on the first two rows. There is no fixed starting position, so the pieces may be placed in any arrangement. However, it is suggested that most of the stronger pieces be placed in front of the weaker Rabbits. Once Gold has finished, the player with the silver pieces (Silver) sets the pieces on the last two rows. Again the pieces may be placed in any arrangement within these two rows.

The play

The players take turns moving their pieces with Gold going first. All pieces move the same way: forward, backward, left and right (like Rooks in Chess but only one step at a time), but the Rabbits cannot move backward. On each turn a player can move the pieces a total of four steps. Moving one piece from its current square to the next adjacent square counts as one step. A piece can take multiple steps and also change directions after each step. The steps may be distributed among multiple pieces so that up to four pieces can be moved. A player can pass some of the steps, but at least one step must be taken on each turn to change the game position. There are no captures in Arimaa.

The stronger pieces are able to move adjacent opponent's weaker pieces. The Elephant is the strongest followed by Camel, Horse, Dog, Cat and Rabbit in that order. For example, in figure 1.2 the gold Dog (e4) can move the opponent's Cat (f4) or Rabbit (e5 or d4), but the Dog (a4) cannot move the opponent's Dog (a5) or any other piece that is stronger than it, like the Camel (b4). An opponent's piece can be moved by either pushing or pulling it. To push an opponent's piece with your stronger piece, first move the opponent's piece to one of the adjacent empty squares and then move your piece into its place, like the Dog (e4) in figure 1.2 pushes the Cat (f4) to g4 in figure 1.3 To pull an opponent's piece with your stronger piece, first move your piece to one of the adjacent empty squares and then move the opponent's piece into its place like the Cat (d5) in figure 1.2 pulls the Rabbit (d4) to d5 in figure 1.3. A push or pull requires two steps and must be completed within the same turn. Any combination

of pushing and pulling can be done in the same turn. However, when your stronger piece is completing a push it cannot pull an opponent's weaker piece along with it.

A stronger piece can also freeze any opponent's piece that is weaker than it. A piece which is next to an opponent's stronger piece is considered to be frozen and cannot move, like the Dog (a4) in figure 1.2 that is frozen by the Camel (b4) However, if there is a friendly piece next to it the piece is unfrozen and is free to move like the Horse (b3) in figure 1.2 with the friendly Cat (b2) next to it.

There are four distinctly marked trap squares on the board. Any piece that is on a trap square is immediately removed from the game unless there is a friendly piece next to the trap square to keep it safe. To give an example we take a look at figures 1.2 and 1.3. Here we see a Rabbit (f3) in figure 1.2 because of the friendly Cat (f4) next to the trap, but after the turn of Gold, resulting in the position of figure 1.3, we see that the Rabbit is gone because of the fact that there is no friendly piece next to the trap. Be careful not to lose your own pieces in the traps.

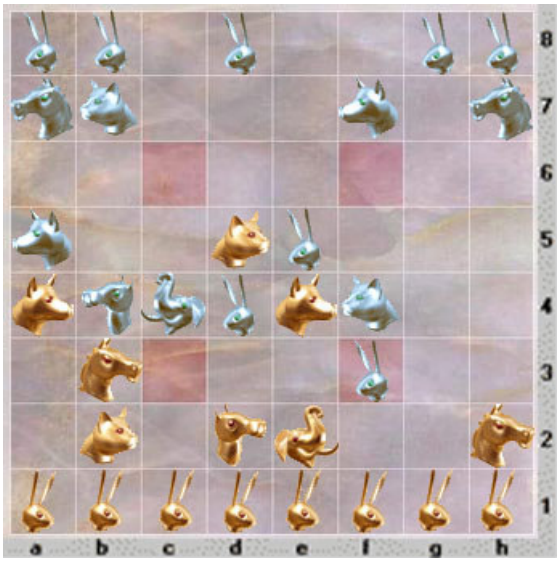

Figure 1.2: A sample position with Gold to move.

Figure 1.3: Position after Gold's move.

There are some special situations.

1. If both players lose all the Rabbits then the game is a draw.
2. If a player is unable to make a move because all the pieces are frozen or have no place to move, then that player loses the game.
3. If after a turn the same board position is repeated three times, then the player causing the position to occur the 3rd time loses the game.
4. A player may push or pull the opponent's Rabbit into the goal. If at the end of the turn the Rabbit remains there the player loses. However if the Rabbit is moved back out of the goal row before the end of the turn, the player does not lose.
5. If at the end of the turn Rabbits from both players have reached the goal, then the player who made the move wins. [*This rule is added after confronting the designer with the problem that there was no solution for this situation.*]

1.2.2 The Rules of Arimaa Matches

The following rules apply to official Arimaa games that are played for ranks, tournaments, contests, championships, etc. We will take a look at the general meaning of those rules and than turn to a tournament example for the Arimaa challenge.

Match Game Requirements

1) Players may not offer a draw.

An official Arimaa match is to be considered similar to a real sporting event. As such the players may not agree to end the match into a draw. But one of the players may resign at any time to end the match.

2) Time controls must be used.

An official Arimaa match must be played with some form of Arimaa time control. Details of time controls are given below.

3) If a game must be stopped then the Arimaa scoring system must be used.

When the preset time limit or move limit for the game expires and the game has not finished, the winner is determined by score. Details of the Arimaa scoring system are given below.

4) Games must be recorded.

All moves made in the game must be recorded using the notation for recording Arimaa games (see subsection 1.2.3).

Scoring System

If the amount of time which was set for the game runs out, then the following scoring system is used to determine a winner. The player with the higher score wins the game. In case the score of both players is the same, the game is a draw.

The score for each player is determined as follows:

$$Score = R + P \times (C + 1)$$

R denotes the points given for how far the player's Rabbits have progressed. The row to which each Rabbit has progressed is cubed (i.e., raised to the power of 3) and these values are summed up to determine R. The first row (from the player's perspective) is 1 and the goal row is 8.

C equals the number of Rabbits the player still has on the board.

P denotes the points given for the pieces the player still has on the board. The value of each piece on the board is summed. The value of each piece is:

1. Rabbit
2. Cat
3. Dog
4. Horse
5. Camel
6. Elephant

Time Controls

The Arimaa time controls were chosen to achieve the following.
1. Keep the game moving, by not allowing a player to take forever to make a move and bore the spectators.
2. Allow a great deal of flexibility in specifying the time controls.
3. Allow for a fixed upper limit on the total game time for practical reasons.
4. Attempt to prevent a player from losing the game due to time while imposing these time limits.
5. Preserve the quality of the game while imposing these time limits.
6. Allow for the most common time controls used in Chess. Thus the Arimaa time controls support all the common time controls used in Chess and more.

The time control used for Arimaa is specified as:

M/R/P/L/G/T

M is the number of minutes:seconds per move,
R is the number of minutes:seconds in reserve,
P is the percentage of unused move time that gets added to the reserve,
L is the number of minutes:seconds to limit the reserve,
G is the number of hours:minutes after which time the game is halted and the winner is determined by score. G can also be specified as the maximum number of moves.
T (optional) is the number of minutes:seconds within which a player must make a move.

On each turn a player gets a fixed amount of time per move (M) and there may be some amount of time left in the reserve (R). If a player does not complete the move within the move time (M) then the time in reserve (R) is used. If there is no more time remaining in reserve and the player has not completed the move then the player automatically loses. Even if there is time left in the move or reserve, but the player has not made the move within the maximum time allowed for a move (T) then the player automatically loses. If a player completes the move in less than the time allowed for the move (M), then a percentage (P) of the remaining time is added to the player's reserve. The result is rounded to the nearest second. This parameter is optional and if not specified, it is assumed to be 100%. An upper limit (L) can be given for the reserve so that the reserve does not exceed L when more time is added to the reserve. If the initial reserve already exceeds this limit then more time is not added to the reserve until it falls below this limit. The upper limit for the reserve is optional and if not given or set to 0 then it implies that there is no limit on how much time can be added to the reserve.

For practical reasons a total game time (G) may be set. If the game is not finished within this allotted time then the game is halted and the winner is determined by scoring the game. This parameter is optional and if not given (or set to 0) it means there is no limit on the game time. Also, instead of an upper limit for the total game time, an upper limit for the total number of turns each player can make may be specified. After both players have taken this many turns and the game is not finished, the winner is determined by scoring the game.

For games which use a time per move of less than 1 minute, both players are always given 1 minute of time to setup the initial position in the first move of the game. If the setup is not completed in 1 minute then the reserve time (R) is also used. The unused time from the setup move is not added to the reserve time unless the player completes the setup in less time than the time per move (M) set for the game. If so, then a percentage (P) of the unused time

after deducting the time used from the time per move set for the game is added to the reserve time.

Let us look at an example for the Arimaa challenge which also will be used in the experiments. In the tournaments the time control '3/3/100/15/8' will be used (Syed and Syed, 1999). This means that on each turn a player gets 3 minutes per move. If a move is not completed in this time then the reserve time may be used. There is a starting reserve of 3 minutes. If the reserve time is used up and the player has still not made a move then the player will lose on time. If the move is made in less than 3 minutes then 100% of the remaining move time is added to the reserve. The reserve may not exceed more than 15 minutes. If the game is not completed within 8 hours, it will be stopped and the winner determined by score.

In Appendix A more examples of time controls are given.

1.2.3 Notation

For reviewing games, the games have to be recorded. To do this recording we will use the notation for recording the Arimaa games and positions that is also used on the internet. We will outline these notations in this subsection.

Notation for recording Arimaa games

The pieces are indicated using upper or lower case letters to specify the piece colour and piece type. Upper case letters are used for the gold pieces and lower case letters are used for the silver pieces. For example, E means gold Elephant and h means silver Horse. The types of pieces are: Elephant, Camel, Horse, Dog, Cat and Rabbit. The first letter of each is used in the notation, except in the case of Camel the letter M (for Gold) or m (for Silver) is used.

Each square on the board is indicated by the column and row, like in Chess. The lower case letters a to h are used to indicate the columns and the numbers 1 to 8 are used to indicate the rows. The square a1 must be at the bottom left corner of the board for Gold.

Each player's move is recorded on a separate line. The line starts with the move number followed by the colour of the side making the move. For example 3w means move 3 for Gold; this would be followed by 3b which is move 3 for Silver. In the move numbers, 'w', White, is used for Gold, and 'b', Black, for Silver.

The initial placement of the pieces is recorded by indicating the piece and the square on which it is placed. For example Da2 means the gold Dog is placed on square a2.

The movement of the pieces is recorded by indicating the piece, the square from which it moves followed by the direction in which it moved. The directions are north, south, east and west with respect to the gold player. For example, Ea3n means the gold Elephant on a3 moves north (to square a4). The notation hd7s means that the silver Horse on square d7 moves south (to square d6).

Steps which are skipped are left blank. See the second move 3w (after the take back) in the example below where only three steps are taken and the last step is skipped.

When a piece is trapped and removed from the board it is recorded by using an x to indicate removal. For example cf3x means the silver Cat on square f3 is removed. When a piece is trapped as a result of a push, the removal is recorded before the step to complete the push. For example: rb3e rc3x Hb2n.

When a player resigns the word 'resigns' is used. If a player loses because an opponent's Rabbit has reached his first row then the word 'lost' is used. If the players agree to a draw then the word 'draw' is used.

If a move is taken back the word ‘takeback’ is used and the move count of the next move is that of the previous move.

The following example shows the Arimaa notation used to record the moves of a game.

```
1w Ra2 Rb2 Mc2 Dd2 ...
1b ra7 rb7 rc7 rd7 ...
2w Ra2n Ra3e Rb3n Rb4e
2b ra7s ra6s ra5e rb5e
3w Dd2n Dd3n Mc2e Rc4s Rc3x
3b rc7s rc5e rc6x rd5e re5s
4w takeback
3b takeback
3w Rb2n Rb3n Rb4n
3b ...
...
16b resigns
```

The seven tags from the PGN (Portable Game Notation) format (as well as other tags) used in Chess can be used prior to the recording of the moves. These are Event, Site, Date, Round, White, Black and Result. The format is simply the tag name followed by a ‘:’ and a space character followed by the tag value. A blank line separates the tags from the move list. All tags are optional.
Here is a sample recording of a game:

```
Event: Casual Game
Site: Cleveland, OH  USA
Date: 1999.01.15
Round: ?
White: Aamir Syed
Black: Omar Syed
Result: 1-0

1w ...
1b ...
2w ...
2b ...
...
16b resigns
```

A tag which requires multiple lines should have the string -=+=- after the tag name. All lines until a line that begins with this same string are considered to be the value of the tag. For example:

```
Chat: -=+=-
2b White: hi, how are u
2b Black: fine, thanks
-=+=-
```

The 2b just indicates that this chat was done when it was move 2b (Silver’s second move).

Notation for recording Arimaa positions

The gold pieces are shown in upper case letters and silver pieces are shown in lower case letters. The assignment of letters is the first letter of the piece name, except in the case of the Camel, when the letter m or M is used. The position file should simply be laid out as the board would appear with square a1 at the bottom left corner. The rows and columns of the board must be labelled and the board must be framed with '-' and 'I' characters. Spaces are used to indicate empty squares. X or x can be used to mark the trap squares when a piece is not on it. However, marking the trap squares is optional and not required. Here is a sample position file:

```
7w Da1n Cc1n
 +-----------------+
8|   r   r r   r   |
7| m   h     e   c |
6|   r x r r x r   |
5| h   d     c   d |
4| E   H         M |
3|   R x R R H R   |
2| D   C     C   D |
1|   R   R R   R   |
 +-----------------+
   a b c d e d g h
```

The first line of the file indicates the move number and which player to move. By default it is assumed that no steps have been taken and the player has 4 steps left. If any steps have already been taken to reach the position shown they are listed on the first line of the file using the notation for recording Arimaa games described earlier.

1.3 Problem Statement and Research Questions

In computer game-playing, the goal is to make a computer play a certain game as good as possible. So the problem statement for this thesis is the following:

Can we build an efficient and effective program to play the game of Arimaa?

There are three research questions that come up to answer this problem statement. The first research question is:

What is the complexity of Arimaa?

To answer this research question, the complexity of Arimaa needs to be computed.

The second research question is:

Can we use knowledge about the game of Arimaa in an efficient and effective implementation of an Arimaa program?

We will gather some specific knowledge about the game, and use it to implement different parts of an evaluation function for playing the game. This evaluation will be optimized using many tuning experiments.

The third research question is:

> *Can we apply and adapt relevant techniques, developed for computer-game playing, to the game of Arimaa?*

We will investigate how we can adapt several existing game-playing techniques to fit the game of Arimaa. These techniques will be tested for their effectiveness. We will also explore new techniques and ideas, which could be beneficial in playing the game of Arimaa.

1.4 Thesis Outline

The content of this thesis is as follows. Chapter 1 is an introduction to this thesis containing an overview of the matters under discussion including a presentation of the game Arimaa. This is also the chapter in which the problem statement and the research questions of this thesis are formulated.

In Chapter 2 we will go into detail on the complexity of Arimaa. Chapter 3 covers the evaluation function and the knowledge behind it. This chapter is closely related to chapter 4 where we will discuss the techniques investigated and implemented in our Arimaa program.

In Chapter 5 we will discuss the tuning of the evaluation functions described in Chapter 3. In Chapter 6 we will show some experimental results considering the techniques described in chapter 4. Its purpose is not only to find the best combination of techniques to play Arimaa but also to determine the influence of the individual techniques on the playing strength.

Chapter 7 will present the conclusions and will formulate answers to the research questions and problem statement. Moreover some ideas about future research on the subject will be presented.

2 COMPLEXITY ANALYSIS

The complexity of a game is measured by two different factors, the state-space complexity and the game-tree complexity (Allis, 1994). Together they provide some information on how difficult the game at hand is. The state-space complexity roughly indicates the number of different board positions that are possible in principle.

The game–tree complexity gives more insight into the decision complexity of the game. It is an indication of how many nodes there are in the solution tree. The number depends on the specific search techniques used. Below we discuss in more details these two measures of complexity and we will compare them with the complexities of other games.

2.1 State-space Complexity

The state-space complexity is an approximation of the number of different positions possible in Arimaa. The expected result for the state-space complexity must be close to that of Chess with $O(10^{46})$ (Chinchalkar, 1996), since a Chess-set is used for Arimaa. To compute the state-space complexity for Arimaa, we have to do a number of calculations. First we have to deal with how many pieces are on the board? There can be theoretically 1 piece on the board and this can go up to 32 pieces. When we put the number of pieces in a row we get the following pattern:

112228/112228

In this pattern the first part stands for the gold player and the second for the silver player. Each part is then built up with the number of Elephants, Camels, Horses, Dogs, Cats, and Rabbits. In this example we have used all 12 types of pieces of Arimaa together with their appearance on the board. So we have: 1 Elephant, 1 Camel, 2 Horses, 2 Dogs, 2 Cats, and 8 Rabbits of each colour. When we will create the other patterns we will start with pattern 112228/112228. By removing one piece of the board we can remove this on every position in the pattern. For example, we can remove the gold Elephant (012228/112228) or we can remove a silver Rabbit (112228/112227). We have 12 positions in the pattern were we can remove a piece but on every position we can remove a piece until there are 0 pieces left. E.g., we can remove one of the 8 Rabbits on position 6 in the pattern, but it does not matter which Rabbit we remove because in the pattern there still would remain 7 Rabbits. The total number of pieces we can remove is therefore the summation of the pieces on the 12 positions in the pattern, which will be 32. When we go on until we have removed 32 pieces from the board the remaining pattern will be 000000/000000.

In table 2.1 we see in column 1 the total number of pieces that we have on the board. In the second column is the result of the number of possibilities to remove $32-n$ pieces from the board.

When we remove a silver Rabbit from the number of pieces on the board it does not matter which silver Rabbit is removed for the pattern. When we remove silver Rabbit-1 we get pattern 112228/112227, and by removing silver Rabbit-2 we get also 112228/112227, because we hold the same set of pieces. This means that the numbers of possibilities in column 2 do not refer to unique patterns only, but also contain duplicates. These duplicate patterns have been filtered out and the numbers of unique patterns are given in column 3 of Table 2.1.

n	# possibilities	# unique patterns	# sorted unique patterns	$O(StateSpace\_n)$
32	1	1	1	4.63473×10^{42}
31	32	12	3	4.49428×10^{42}
30	496	74	7	2.30497×10^{42}
29	4960	310	13	8.31347×10^{41}
28	35960	987	22	2.35457×10^{41}
27	201376	2540	33	5.52581×10^{40}
26	906192	5499	48	1.10607×10^{40}
25	3365856	10314	63	1.92039×10^{39}
24	10518300	17163	80	2.92329×10^{38}
23	28048800	25866	94	3.92962×10^{37}
22	64512240	35955	108	4.6863×10^{36}
21	129024480	46818	116	4.96954×10^{35}
20	225792840	57774	123	4.68714×10^{34}
19	347373600	68022	123	3.92605×10^{33}
18	471435600	76569	122	2.91258×10^{32}
17	565722720	82314	114	1.90708×10^{31}
16	601080390	84348	107	1.09793×10^{30}
15	565722720	82314	94	5.536×10^{28}
14	471435600	76569	84	2.43527×10^{27}
13	347373600	68022	70	9.30991×10^{25}
12	225792840	57774	60	3.08072×10^{24}
11	129024480	46818	47	8.78575×10^{22}
10	64512240	35955	38	2.1486×10^{21}
9	28048800	25866	28	4.47876×10^{19}
8	10518300	17163	22	7.89728×10^{17}
7	3365856	10314	15	1.16634×10^{16}
6	906192	5499	11	1.4239×10^{14}
5	201376	2540	7	1.41124×10^{12}
4	35960	987	5	11066979168
3	4960	310	3	66079104
2	496	74	2	282240
1	32	12	1	768
0	1	1	1	1

Table 2.1: An overview of results needed for the State-Space Complexity.

In column 4 we see the numbers of unique patterns that occur when the patterns in column 3 are sorted. This sorting disregards the colour and type of the pieces. For example, the unique pattern 112228/112227 yields after sorting the pattern 111122222278. Every pattern has a number of hits, i.e., the summation that that pattern occurs when a pattern in column 3 is sorted. As another example, when we have only 1 piece on the board, we can create 12 different unique patterns (column 3), the summation of each type from each colour. But when we disregard the types and colours and sort the pattern there will be only one pattern (000000000001) left. The only difference is that this pattern occurs 12 times (hits)

when the patterns in column 3 with $n = 1$ are sorted. All patterns and their hits can be found in Appendix B. The sorting we used is allowed because the order of the pattern has no influence on the result of formula (1). To compute the results for each pattern we use formula (1).

$$O(StateSpace\_n) = \sum_{i=1}^{m} H_i \times \binom{64}{p_{i,1}} \times \prod_{k=2}^{12} \binom{64 - \sum_{j=1}^{k-1} p_{i,j}}{p_{i,k}} \tag{1}$$

In this formula H_i is the number of hits (Appendix B) from pattern i, $p_{i,j}$ is the value of the j^{th} element in pattern i, n is the number of pieces on the board, and m is the number of sorted unique patterns (column 4 of table 2.1) for n pieces. The results for each n are given in column 5 of table 2.1.

This results in a total of 2.51×10^{43} number of possible board positions according to formula (2). Here we also take into account the player to move.

$$O(StateSpace\_Positions) = 2 \times \sum_{n=0}^{32} O(StateSpace\_n) = 2.51 \times 10^{43} \tag{2}$$

Next we have calculated some number of illegal positions reducing this state-space complexity. The first group is when a piece comes on a trap without a friendly piece next to the trap, resulting in a piece less on the board. We have to calculate these situations and reduce the state-space complexity with this result. We only have to calculate when there is at least one piece on a trap that will be removed. This situation includes then also the other positions where two, three or four pieces are standing on a trap and will be removed.

To calculate this number of illegal positions we started with the pattern 822211/822211. Assume that there is one gold piece on a trap. We split the pattern into the patterns in table 2.2 where one gold piece is on a trap.

Possible Patterns		# positions
Gold	Silver	
722211	822211	7.50×10^{41}
812211	822211	1.87×10^{41}
821211	822211	1.87×10^{41}
822111	822211	1.87×10^{41}
822201	822211	9.37×10^{40}
822210	822211	9.37×10^{40}
	total	1.50×10^{42}

Table 2.2: Summation of the number of illegal positions where one gold piece is trapped.

Because there can be an unfriendly piece next to the trap we have to calculate these possibilities. These patterns look like $n_r n_c n_d n_h n_m n_e$, where n_r is the number of unfriendly rabbits adjacent to the trap, etc. The maximum value for each position in the pattern is 4, 2, 2, 2, 1, and 1.The maximum for n_r is 4 in stead of 8 is because there can only be 4 pieces next to the trap. We generate all patterns where there are no pieces next to the trap up to the

maximum of four pieces. When we take, e.g., the pattern 822210/822211 from table 2.2, we have to subtract each generated pattern for the unfriendly pieces that could stand next to the trap, from the silver part (822211), because a gold piece is standing on the trap. For example the silver part will be split into the pattern 722211/100000, where 100000 is one of the generated patterns, for an unfriendly piece (Silver) standing next to the trap. We take these two together with the gold part which results in 822210/722211/100000. Now we use formula (3) where m is the number of patterns we get from the splitting of the silver part.

$$O(StateSpace\_trap) = \sum_{i=1}^{m} 4 \times \binom{59}{p_{i,1}} \times \prod_{k=2}^{12} \binom{59 - \sum_{j=1}^{k-1} p_{i,j}}{p_{i,k}} \times \binom{4}{p_{i,13}} \times \prod_{k=14}^{18} \binom{4 - \sum_{j=13}^{k-1} p_{i,j}}{p_{i,k}} \quad (3)$$

In the last column of table 2.2 we see the summation of these results according to the gold part of the pattern. All these positions result in at least one gold piece and none or more silver pieces on the traps without a friendly piece next to the gold piece. When we also take into account which player has to move we have to apply a factor of two.

As we can see we have to do a great dead of calculations because we have 944,784 (summation column 3 table 2.1) possible starting patterns such as 822211/822211 for which we have to apply formula (3). We are going to make an assumption. When we have 32 pieces on the board there are $2 \times 4.63 \times 10^{42} \approx 9.26 \times 10^{42}$ possible positions and $2 \times 1.50 \times 10^{42} \approx 3.00 \times 10^{42}$ are illegal positions according to the trap rule, being a factor of 0.323. When there is only one piece left on the board there is only a factor of 0.0625 illegal. Now we assume that a factor of 0.323 is the upper bound of illegal positions according to the trap rule. The result will then be $0.323 \times 2.51 \times 10^{43} \approx 8.08 \times 10^{42}$ illegal trap positions.

The second group of illegal board positions are the board positions that will never be reachable. We distinguish three categories of illegal positions within this group. In the first category are those positions with at least one own Rabbit and two opponent Rabbits that have reached their goals at the end of turn, since such position would require at least five steps. To calculate this we generated the pattern from table 2.1 again but with a difference in the sorting. In stead of disregarding the colour and type of the piece we now put the Rabbits in the front two places and we sort the rest disregarding the colour and type again. Then we split every first two positions (Rabbits) of the pattern.

$$\mathrm{A,B,C,D,F,G,H,I,J,K,L,M} \Rightarrow \mathrm{A1,B1,A2(= A - A1),B2(= B - B1),C,D,E,F,G,H,I,J,K,L,M}$$

In this pattern A1 is two and B1 is one. Then we remove all patterns where A2 or B2 are below zero, because those patterns have not enough Rabbits to put in the goals. Now we can apply formula (4). Here we do not take into account which player is going to move because as we have seen the minimum condition to calculate this is depending on which player is to move. It results in 3.13×10^{41} illegal positions according to the number of Rabbits in the goal.

As a second category we have those positions where the player who moved last has at least three Rabbits of his own and at least one of his opponent Rabbits in the goal. Also here we apply the new patterns, but now A1 becomes three and B1 one. Again we remove all patterns where A2 and B2 are below zero. Using formula (4) we get 1.84×10^{41} illegal positions which are not reachable for the player to move.

$$O(StateSpace\_goal\_\mathrm{A1}\_\mathrm{B1}) = \sum_{n=0}^{32}\sum_{i=1}^{m} H_i \times \binom{8}{\mathrm{A1}} \times \binom{8}{\mathrm{B1}} \times \prod_{k=3}^{14} \binom{64 - \sum_{j=1}^{k-1} p_{i,j}}{p_{i,k}} \qquad (4)$$

The third category of unreachable positions is when we have five or more Rabbits from one player in the goal at the end of a turn. Here we must give A1 the value five and B1 the value zero. This yields 10.01×10^{39} illegal positions. A part of these positions we have already calculated above, where A1 is also five and B1 is one which results in 1.71×10^{39}. So there are $10.01\times10^{39} - 1.71\times10^{39} = 8.30\times10^{39}$ positions where five Rabbits of one player and no Rabbits of the other player are in the goal. These positions are unreachable disregarding which player finished his move so we have $2\times8.30\times10^{39} = 1.66\times10^{40}$ illegal positions.

We have 3.13×10^{41} and 1.84×10^{41} and 1.66×10^{40} unreachable positions for both players together at the end of their move. This gives in total 5.04×10^{41} unreachable positions.

Using the values above we can say that the state-space complexity can be approximated by $2.51\times10^{43} - 8.08\times10^{42} - 5.04\times10^{41} \approx 1.66\times10^{43}$.

2.2 Game-tree Complexity

The game-tree complexity can be approximated by the number of leaf nodes of the game tree with as depth the average game length (in ply), and as branching factor the average branching factor during the game. It is not hard to calculate the last. There is a database available with 6,075 Arimaa games played. These are not all usable, so we have selected games that have finished by:

1. a Rabbit reached the goal.
2. a player had no moves.
3. no Rabbit on the board.
4. a position repeated three times.

After this selection we have a database of 4,251 records. From the records we have determined the average branching factor at 276,386. In the beginning of the game the average branching factor is 23,026. The branching factor will increase for the first five ply, because we will have more moves per piece on average (figure 2.2). After that we see that the branching factor slowly goes down, but still stays extremely high.

With the same database we have determined the average game length. The result is an average of 55.2 ply (moves of four steps).

The formula to calculate the game-tree complexity is b^d, where b is the average branching factor and d the average depth. Using the values above, the game-tree complexity for Arimaa is calculated as $276{,}386^{55.2} \approx 3.1\times10^{300}$.

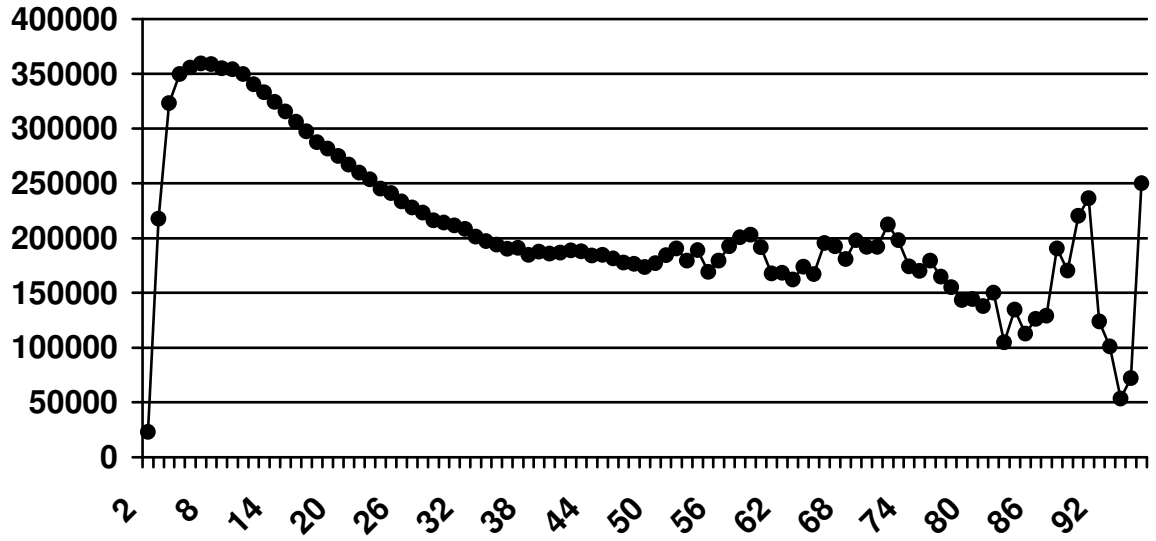

Figure 2.1: Average branching factor per ply.

2.3 Comparing Arimaa to Other Games

As mentioned in Chapter 1 there are at least four reasons why Arimaa is difficult for computers to play. First the branching factor runs into the twenty-thousand as compared to an average of about 35 for Chess. With a much bigger range of possibilities at each turn it becomes extremely hard for computers to use the brute-force look-ahead method to search through all of them to find the best moves. For example at the start of chess games White has 20 possible moves. In Arimaa a player has about 2,000 to 3,000 moves in the first turn depending on the way they chose to setup the pieces. During the mid-game the number of possible moves ranges from about 5,000 to 40,000. If we assume an average of 20,000 possible moves at each turn, looking forward just 2 moves (each player taking 2 turns) means exploring about 160 million billion positions. Even if a computer is 5 times faster than DEEP BLUE and could evaluate a billion positions per second it would still take more than 5 years to explore all those positions. However, modern game-playing programs use pruning algorithms to reduce significantly the number of positions that need to be explored. Even with such pruning the number of positions that need to be evaluated is still sufficiently large to make it extremely difficult for a program to search very deep.

Second, endgame databases have also significantly improved the performance of computers in many traditional strategy games. When the number of pieces remaining on the board is reduced to just a few, computers can play a perfect ending by simply looking up the best move for the current position from a precomputed database of endgame positions. However a typical Arimaa game can end with most of the pieces still on the board. It is not uncommon for a game to end without any pieces ever being exchanged. Thus building endgame databases for Arimaa will not be useful.

Third, important factor is that the starting position of Arimaa is not fixed as it is in Chess. The number of different ways in which each player can setup their pieces at the start of the game can be computed with formula (5).

$$O(StateSpace\_Setup, player1) = \binom{16}{8} \times \binom{8}{2} \times \binom{6}{2} \times \binom{4}{2} \times \binom{2}{1} \times \binom{1}{1} = 64{,}864{,}800 \quad (5)$$

This formula results from the placing of 8 Rabbits at 16 possible squares, 2 Cats at the remaining 8 squares, 2 Dogs at the remaining 6 squares, 2 Horses at the remaining 4 squares, 1 Camel at the remaining 2 squares and placing the Elephant at the last remaining square. As we can see there are almost 65 million different ways in which each player can setup his pieces at the start of the game. The total of the setup state-space complexity for both players is given by formula (6). It follows that it is very difficult to develop complete databases of opening moves. One of the difficulties that humans have when playing Chess against computers is that they can easily fall into an opening trap. To avoid this the human player must be extremely familiar with a very large number of chess openings. This basically boils down to a problem of memorization which computers are extremely good at and humans are not. Even Bobby Fischer, a former World Champion, has proposed allowing different starting positions in Chess to counter the problem of computers having an opening book advantage. Because of the many different ways in which a player can set up his pieces in Arimaa these databases of opening moves are not complete so the computer is also not aware of every opening trap like in Chess.

$$O(StateSpace\_Setup) = (64{,}864{,}800)^2 = 4.207 \times 10^{15} \quad (6)$$

Fourth, computers will have difficulty with Arimaa because it is much more a positional game and has much less tactics than Chess. Computers are great at spotting tactics and taking advantage of them, but they have a much harder time trying to determine if a materially equivalent position is more advantageous for one side or the other. Chess Grandmasters are continnously trying to bring their computer opponents into a positional disadvantage while trying to avoid tactical mistakes. After playing DEEP BLUE, Garry Kasparov wrote that the experience was like walking through a mine field. Arimaa tips the scale in favour of humans, by reducing tactics and giving more importance to positional features.

In figure 2.2 we have compared the complexity of Arimaa with other games. This figure is based on a table from Van den Herik et al. (2002). We have only added the complexity of Amazons, Lines of Action and Arimaa. If we have a closer look at figure 2.2, we see that Arimaa's state-space complexity is almost the same as that of Chess, but its game-tree complexity is more like that of Go. When we have a look on both complexities it is closely related to Amazons, where the established game-playing techniques do not work well due to a very high branching factor. In that game some other solutions have to be found. This is true for Arimaa as well and led to the current thesis

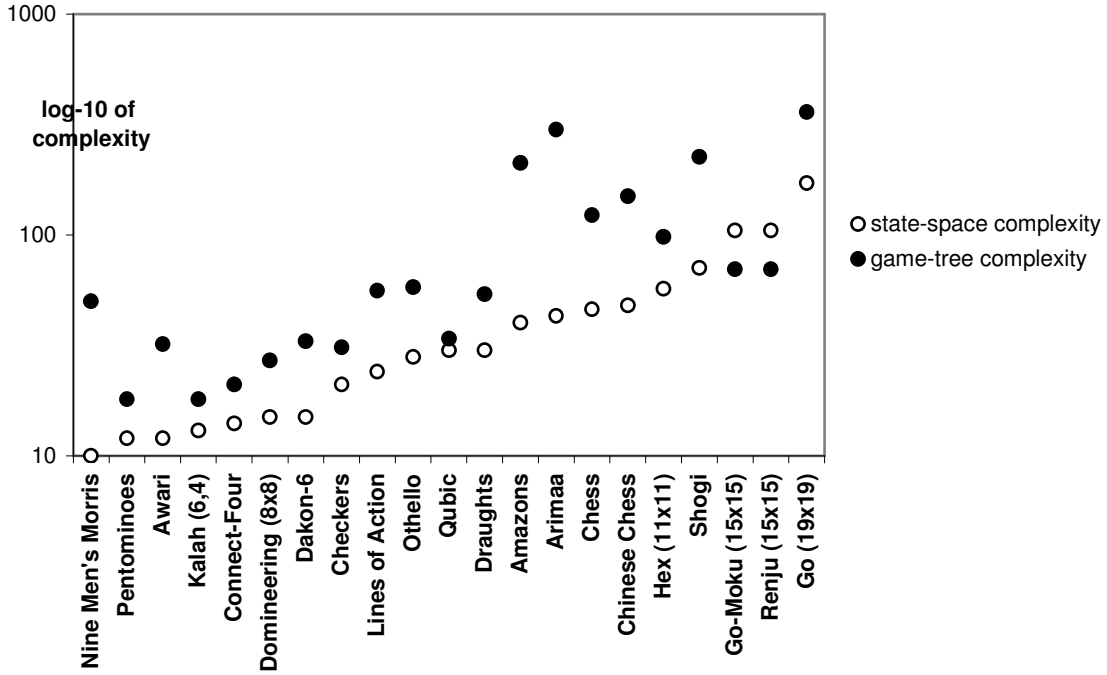

Figure 2.2: Estimated game complexities.

2.4 Chapter Conclusion

We may safely conclude that Arimaa is computationally an extremely complex game, especially since the game tree is huge. Because of the huge game-tree complexity, a conventional brute-force approach will not work. In chapter 4 we will therefore discuss a few search enhancements needed to make search a feasible approach in this game. In the next chapter we will see how we can use knowledge about the game in the evaluation.

3 Evaluation of Positions

Even the fastest search will not be useful in a complex game like Arimaa if it is not guided into the right direction. The ultimate goal of the search is to find a forced win for the player to move. This is usually impossible to achieve, because a search is only performed until a certain depth due the time limit. Therefore we will have to rely on an estimation of what is the most promising move in the current position. This estimation is based on an evaluator. The leaf nodes are valued using the evaluation function. It is therefore important that the evaluator gives a good estimation of a position. In this chapter we will examine "Arimaa knowledge" and discuss the construction of the evaluation function. In section 3.1 some basic evaluation techniques will be headed. Some advanced knowledge will then be discussed in section 3.2

3.1 The Basic Evaluation

According to many other games, Arimaa has some basic evaluations such as material values, piece-square tables, and mobility. We will discuss them below.

3.1.1 Material

Some Arimaa pieces are stronger than others. The element of material is concerned with this relative strength of the pieces. After some personal communication with David Fotland, we use a fixed set for the material values in the evaluation function (shown in table 3.1). The material values for the rabbits do not match with Fotlands part.

For the Rabbits the value is fluctuating and will be adjusted depending on how many Rabbits are left on the board, as we can see in table 3.2. Since we need at least one Rabbit for winning the game, we need to protect this piece to keep it on the board. Therefore we give the final Rabbit the highest material value. When we have more Rabbits on the board we can make more goal-threats to the opponent since he has more places to defend. Moreover we can defend better against a goal-threat by the opponent. So as the number of Rabbits on the board goes down, the material value for each Rabbit goes up.

Material	Value
Elephant	16
Camel	11
Horse	7
Dog	4
Cat	2
Rabbit	Variable, see table 3.2

Table 3.1: Material values.

Number of Rabbits on the board	Summation of rabbit values
1	40
2	47
3	53
4	58
5	62
6	65
7	67
8	68

Table 3.2: Material values for the Rabbits on the board.

Furthermore, we use a single weight for these values to determine how heavy the influence of the fixed material values must be to the total evaluation. This material weight will be tuned, which will be discussed in Chapter 5.

3.1.2 Piece-square Tables

Piece-square tables are used to indicate where each piece is located best on the board on average. We made three groups for the six different pieces. First, we have the rabbit piece-square table shown in table 3.3. On the one hand Rabbits have to be encouraged to stay at their own side to defend it. At the other hand, it is of course even more important for Rabbits to advance to the opponent side, but they best advance along the edges of the board and stay out of the centre.

8	8	8	8	8	8	8	8	8
7	7	6	5	4	4	5	6	7
6	6	2	-1	0	0	-1	2	6
5	5	1	0	0	0	0	1	5
4	4	1	0	0	0	0	1	4
3	3	1	-1	0	0	-1	1	3
2	2	2	1	1	1	1	2	2
1	2	2	2	2	2	2	2	2
	a	b	c	d	e	f	g	H

Table 3.3: Rabbit piece-square table for Gold.

Second, we have the group of Cats and Dogs. To have control over the traps and it is encouraged to use the Cats and Dogs. So they stay therefore close to the traps, which we see in table 3.4.

8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	-1	0	0	-1	0	0
5	0	0	1	0	0	1	0	0
4	1	1	2	1	1	2	1	1
3	3	5	-1	5	5	-1	5	3
2	2	4	5	4	4	5	4	2
1	1	2	3	2	2	3	2	1
	a	b	c	d	e	f	g	h

Table 3.4: Cat/Dog piece-square table for Gold.

Third, in table 3.5 we show the stronger pieces, i.e., Horse, Camel and Elephant. They are encouraged to stay close to the centre from where they can cross the board quickly.

Overall all pieces are encouraged to stay off the traps. The figures 3.3, 3.4 and 3.5 are the piece-square tables for Gold. For Silver, the piece-square tables are the opposite.

8	0	0	1	2	2	1	0	0
7	0	0	2	3	3	2	0	0
6	1	2	-1	4	4	-1	2	1
5	2	3	4	5	5	4	3	2
4	2	3	4	5	5	4	3	2
3	1	2	-1	4	4	-1	2	1
2	0	0	2	3	3	2	0	0
1	0	0	1	2	2	1	0	0
	a	b	c	d	e	f	g	h

Table 3.5: Horse/Camel/Elephant piece-square table for Gold.

For piece-square tables we also introduce a weight, but now with the difference that each piece type has a separate weight that must be tuned. So, whereas a Horse has the same piece-square table as the Elephant, it can be that it is less important for the Horse to be in the middle of the board than for the Elephant.

3.1.3 Mobility

How many different steps does each player have available? The idea is that if you have more moves to choose from, it is much more likely that at least one of them will lead to a good position. We made two ways of promoting mobility in COXA. First frozen pieces get penalized according to piece strength, the stronger the piece the bigger its penalty. The penalty for a frozen piece is the negative mobility value of the piece according to table 3.6. The source for the values from table 3.6 comes from a personal communication with David Fotland.

Material	Value
Elephant	0.4
Camel	0.2
Horse	0.15
Dog	0.1
Cat	0.1
Rabbit	0.02

Table 3.6: Mobility values according to the piece strength.

Second when the piece is not frozen we take a look on how mobile the piece is, i.e., we calculate its empty neighbour squares. The piece gets a bonus according to the strength and mobility of the piece. It is calculated as the mobility value from table 3.6 times the number of accessible neighbour squares. Both mobility types get an own weight that will be tuned.

3.2 The Extended Evaluation

Besides basic evaluations we have to put knowledge into the evaluation function to achieve a strong Arimaa playing program. Of course, there is the problem that when we put too much knowledge into the evaluator we can reach the point where the evaluation of a position takes so much time that we can search less deep than with the basic evaluation. So we have to decide in the balance between speed and knowledge. Our decision is on adding extended knowledge into our evaluation function regarding the rabbit evaluation (3.2.1), goal-threat (3.2.2) and having control of the traps on the board (3.2.3).

3.2.1 Rabbit Evaluation

In section 3.1.2 we said that the Rabbits are encouraged to stay in the back to defend the goal. With Rabbit Evaluation we go a little deeper on the rabbit part. First, to defend the goal we give a bonus of 0.1 multiplied by a weight to encourage Rabbits to have friendly pieces on all four neighbour squares. This tends to keep a solid wall of rabbits and pieces across the board to make it harder for the opponent to reach the goal. Second, there is a bonus if a Rabbit has no enemy Rabbits ahead on its file (column) or adjacent files ahead. Rabbits that are closer to the goal, according to table 3.7, get larger bonuses when these files are free from opponent Rabbits ahead.

Rabbit on rank	Bonus
1	0.1
2	0.2
3	0.3
4	0.4
5	0.5
6	0.65
7	0.8
8	1.0

Table 3.7: Free files bonus according to the Rabbit rank.

3.2.2 Goal-threat Evaluator

The basic idea behind our goal-threat evaluator is to see for each Rabbit if it has a free walk to the opponent's goal. Since our goal threat is not needed directly in the beginning of a game we use a point were we start using goal-threat knowledge in our evaluation. This point is the moment when a Rabbit reaches the fourth rank from his side.

Then we calculate how many steps a Rabbit needs to reach the goal without intervening moves from the opponent. We calculated the bonus as the number of steps needed times a standard bonus of 5 points times the tuning weight.

3.2.3 Trap-control Evaluator

Using trap control we decide which colour is controlling each trap. Therefore we determine if both colours are present near a trap. If no pieces are present then there is also no bonus for that trap. Else we have two possibilities. First when there is only one colour then we only give points to the two strongest pieces near the trap because only 2 pieces are needed to control a trap without that it can be taken over completely by the opponent in one turn. When there are for example 2 Dogs and 1 Cat near a trap we give points for 1 Dog and 1 Cat because we can use the second Dog maybe better somewhere else. But when there are only 2 Dogs then there is a bonus also for the second Dog. This is done in the evaluation by giving points to each piece. For the first piece of a type we give points according to column 1 of table 3.8. For the second piece of a type we use column 2 of table 3.8. Each bonus that occurs times a weight is added to the bonus for trap control.

Piece	First piece Bonus	Second Piece Bonus
Elephant	12	*
Camel	11	*
Horse	10	4
Dog	9	3
Cat	8	2
Rabbit	7	1

Table 3.8: Bonuses first and second pieces controlling a trap.

A second possibility is when both colours are near a trap. Here we have to determine which player is the strongest player and if the opponent has enough pieces to compromise against the pieces of the strongest player.

For every trap we loop for every piece, starting with the Elephant, if both colours have or not have present that piece a weaker piece is searched. When a colour has a piece present and the opponent has not, that is the stop-piece.

Having a friendly piece with the stop-piece next to the trap with the same piece-type, that colour is the strongest player for that trap. The opponent is not stronger there he only can have maximum two piece that are weaker than the stop-piece.

If it is not the same piece-type as the stop-piece, the opponent is the strongest player when he has two pieces that are stronger than the friendly piece of the stop-piece, but weaker than the stop-piece it self.

When there is not a friendly piece with the stop-piece, the opponent is the strongest player when he has at least two weaker pieces present at the trap.

When we determine the strongest player we add a bonus of 1, multiplied with a weight, for trap control.

Here we tune both weights for each possibility separated.

3.3 Chapter Conclusion

In COXA's final evaluator we use the basic evaluation together with the extended evaluation. Therefore the separated weights attached to every evaluation (basic and extended) part must be tuned. By tuning these weights we will see how important the separated weights will be. The tuning will be discussed in chapter 5.

4 Searching the Tree

In this chapter we introduce the search algorithms and heuristics tested, aiming to answer the first research question: How can we apply and adapt relevant techniques, developed for computer-game playing, to the game of Arimaa? The search implementation and the evaluation are the main fields of research for game playing. These two parts of the program can be compared to the heart and the brain in the human body. No human can do without either one of these, just like no game-playing program can do without them. This is also the reason why chapters 3 and 4 are tied together very closely and therefore cannot be read separately. However, before we dive into details concerning the search, we will describe how we build the tree which the program has to traverse.

4.1 Building the Tree

To search a search tree efficiently, our principal algorithm is the well-known alpha-beta algorithm, to be discussed below. Before we can find the best move using alpha-beta, we have to generate all the moves. This is done by the move generation code in the program. It is a straightforward procedure for all squares occupied by pieces of the player to move. All the possible moves are enumerated in a list. This takes a bit of time, but still the amount of time that is used to generate moves during search is on average less than 5% of the total search time for a typical Arimaa position. We therefore believe there is not much to gain in trying more efficient techniques like incremental move generation or move generation using pre-filled databases.

4.1.1 Alpha-beta

In the game Arimaa building the search tree is fairly straightforward. The root node corresponds to the current position, while every possible move in this position leads to a child node. The tree is constructed by recursively generating all the children of each position. Of course, we need a stopping criterion. The straightforward one is when we encounter a terminal position, which means that one of the players has won the game. If such a position is reached, we will backtrack and generate the remainder of the tree. A second stopping criterion is a maximum depth, set before the search started. If we reach this depth we will not generate any children but instead go on generating the remainder of the tree to this depth.

As can be deduced from the former paragraph we use a depth-first algorithm to traverse the game tree, a depth-first iterative-deepening algorithm to be more exact. The algorithm of our choice is alpha-beta. It has been conceived by McCarthy in 1956 (McCarthy, 1961), but the first person to use it in an actual program was Samuel (1959, 1967). When first proposed it was denoted a heuristic, since it was not clear whether it guarantees the best possible result. A few years later, when Knuth and Moore (1975) proved the correctness of alpha-beta, it was no longer seen as a heuristic, but promoted to the status of an algorithm.

By using the alpha-beta algorithm, we implicitly assume that both players play optimally according to the evaluation function in use. The idea behind this is that if the opponent does not play the best move, the current player will often be able to achieve more profit in terms of evaluation scores. Of course, it then is of the utmost importance that a position is evaluated as accurately as possible.

The success of alpha-beta in game playing is mainly achieved by cut-offs it enables. It effectively prunes away large uninteresting chunks of the game tree. These cut-offs are realised by the alpha and beta bounds, where alpha is a lower and beta is an upper bound on the evaluation score. The reduction of the game tree can be as high as 99% when the moves are perfectly ordered. For a detailed description of the alpha-beta algorithm we refer to (Marsland, 1986).

4.1.2 Iterative Deepening

Iterative deepening is a method of consecutively searching to increasing depths 1…*N*. The evaluations in the previous search depth can be used to sort the moves of the next iteration. The overhead when using this method is minimal since the game-tree complexity is $O(b^d)$, where b is the branching factor and d is the depth. This means that in each search there are always more leaf nodes than there are internal ones. So the majority of the nodes encountered in each search are leaf nodes. Therefore the number of nodes examined during a search process is only a fraction of the nodes to be examined in the next iteration. Moreover, using results from previous iterations can enhance the move ordering and thus efficiency. The killer move, history heuristic and transposition tables are examples of techniques which use the concept of iterative deepening to optimise the search. They do this by sorting the moves from best to worst using the evaluations found in the previous search. As an effect, node counts with iterative deepening can be even less than searching to the same depth without iterative deepening.

4.1.3 Principal Variation Search

It seems unreasonable to set the initial bounds of alpha and beta at $-\infty$ and ∞, respectively. If we use some initial bounds (called a window), it is possible to get more cut-offs. But the disadvantage is that sometimes those bounds do not enclose the minimax value, and a re-search is needed. The problem is to find a window that provides overall more cut-offs including possible re-searches. There are several methods to do this. We will take a look at principal variation search (PVS).

Principal variation search uses the concept to close the window as much as possible. This means that the beta value equals alpha + 1. The basic idea behind this method is that it is cheaper to prove a subtree inferior, instead of determining its exact value. It has been shown that this method does well for trees like in chess. Because the branching factor of Arimaa (276,386) for four steps comes down to 23 for each step, in the same range as in chess (35), we can safely assume that PVS will work fine in Arimaa too. Provided we have a good move-ordering mechanism, PVS reduces the size of the search tree. For a more detailed description of the algorithm, see again (Marsland, 1986).

4.1.4 Null-move Search

The null move means changing who is to move without any other change to the game state. It is thus different to passing, which is legal. Passing can happen on any step except the first in a turn, and causes a pass for all the remaining steps in that turn. The null move is

illegal because it passes already on the first step of a turn. The reason for doing a null move in our search is that we hopefully can narrow our window resulting in a smaller search tree.

The null move is performed before any other move and it is searched to a lower depth than we would do for other moves. The search depth of the null move is reduced by the factor R, which we have set at 8 (4 steps for each player) in Arimaa. If the null move produces no cut-off or improvement of alpha, some unnecessary search has been done. This has to be avoided and in the following situations the null move is not performed:

1. the previous move was a null move.
2. the search is at the root.
3. the side to move is at a considerable disadvantage, the chances for a cut-off or an improvement of alpha being low.

For a more detailed description of the algorithm see (Donninger, 1993).

We have assumed that doing a null move is a poor move: there is always a move that is better than the null move. That is the reason why we can safely use a null move. Problems occur when the null move is the best move. It is possible then that we get different outcomes for the search when doing null moves or not. This situation is called zugzwang (Uiterwijk and Van den Herik, 2000) and occurs mostly in endgames. However, in Arimaa zugzwang is unlikely, since the game usually ends with many pieces still on the board.

4.2 Transposition Table

When we are searching for a good move, programs build large trees. Since a position can sometimes be arrived at by several distinct move sequences, the size of the search tree can be reduced considerably if the results of a position are stored. The position showed in figure 4.2 can be reached by 1. e2-e3, e3-f3, d2-d3, d3-d4, but also by 1. d2-d3, d3-d4, e2-e3, e3-f3, or by many other move sequences, all starting in the position showed in figure 4.1. Assume this position appears in a search tree. After exploring this position, we know its score and the best move. Because this position exist somewhere else in the tree it is beneficial to save the relevant information in a transposition table.

Figure 4.1: A starting position.

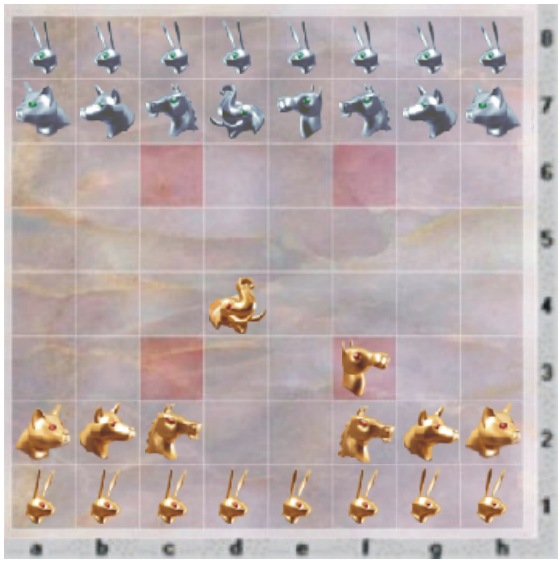

Figure 4.2: Position after moves e2-e3, e3-f3, d2-d3, d3-d4.

When we encounter one of these positions again somewhere else in our search, we can use this information to narrow down the rest of the search. This heuristic has proven to be very successful, for example in Chess where the search tree can be reduced up to 90% depending on the game phase. In the next subsection we explain how we use transposition tables in Arimaa.

4.2.1 Hashing

We would like to save every position encountered in the search tree, but unfortunately this is not possible due to memory restrictions of most nowadays computers. Therefore a transposition table is implemented as a hash table using some hashing method.

In Arimaa there are six different pieces (Elephant, Camel, Horse, Dog, Cat, and Rabbit), two colours (Gold and Silver) and 64 squares. For any combination of a piece and a square a random number is generated. Thus, in total 768 ($64 \times 6 \times 2$) random numbers are available and there is one generated for the player that has to move in the position. The hash value for a position is computed by doing an XOR operation on the numbers associated with the piece combination of that position. This method is called Zobrist Hashing (Zobrist, 1970). It is not only fast, but it can also be done incrementally.

$$\begin{aligned} \text{hashvalue}(new\_position) = {} & \text{hashvalue}(old\_position) && \text{XOR} \\ & \text{hashvalue}(from\_square\_of\_piece\_moved) && \text{XOR} \\ & \text{hashvalue}(to\_square\_of\_piece\_moved) \end{aligned}$$

Normally we have to do only two XOR operations in Arimaa. For push and pull moves, we have to do this for both pieces.

$$\begin{aligned} \text{hashvalue}(new\_position) = {} & \text{hashvalue}(old\_position) && \text{XOR} \\ & \text{hashvalue}(from\_square\_of\_piece1\_moved) && \text{XOR} \\ & \text{hashvalue}(to\_square\_of\_piece1\_moved) && \text{XOR} \\ & \text{hashvalue}(from\_square\_of\_piece2\_moved) && \text{XOR} \\ & \text{hashvalue}(to\_square\_of\_piece2\_moved) \end{aligned}$$

If in a move a piece gets lost on a trap, then we have also to XOR the square of the trapped piece.

If the transposition table consist of 2^n entries, the n lower-order bits of the hash value are used as a hash index. We used n = 24 in COXA. The remaining bits (the hash key) are used to distinguish among different positions mapping on the same hash index. In Arimaa we use a 64-bits hash value.

4.2.2 Use of a Transposition Table

Each entry in the transposition table is 64 bits long. The following traditional components are stored in an entry:

key contains the hash-key part of the hash value. There are 34 bits reserved for the key.

move contains the best move in the position obtained from the search. There are 12 bits reserved for the best move.

score contains the value of the position obtained from the search. The score can be an exact value, an upper bound or a lower bound. There are 11 bits reserved for the score.

flag the flag indicates whether the score is an exact value, an upper bond or a lower bound. There are 2 bits reserved for the flag.

depth contains the depth of the searched subtree. There are 5 bits reserved for the depth.

The key value is used to check if the retrieved board position is the same as the current. Sometimes using the key value cannot detect that the retrieved board position is different. Therefore we always check if the move stored in the transposition table is valid in the position concerned.

We use the transposition table at three different levels:

1. The depth still to be searched is less than or equal to the depth retrieved from the table and the retrieved value is an exact value. In this case the position is not further analysed and the value and move retrieved are returned. This is the main raison why transposition tables are used.

2. The depth still to be searched is less than or equal to the depth retrieved from the table and the retrieved value is not an exact value. The retrieved value can be used to adjust either the alpha value (if the retrieved value is a lower bound) or the beta value (if the retrieved value is an upper bound). Thus, we can use this value to adapt the alpha-beta window. A cut-off is possible, otherwise the retrieved move can be used as a first candidate, since it was considered best previously.

3. The depth still to be searched is greater than the depth retrieved from the table. The retrieved move is used as first move in the move ordering. Because iterative deepening is used, this situation occurs often. Iterative deepening and transposition tables are used in combination to speed up the alpha-beta search.

For a more detailed description of the algorithm, see (Marsland, 1986).

4.2.3 Probability of Errors

Using a transposition table as a hash table introduces two types of errors. The first type of error is a type-1 error (Breuker, 1998). A type-1 error occurs when two different positions have the same hash value. This mistake will not be recognised and can lead to wrong evaluations in the search tree.

Let N be the number of distinguishable positions, and M be the number of different positions to be stored. The probability that this error will happen is given by the following equation (Gillogly, 1989):

$$P(no\_errors) \approx e^{\frac{-M^2}{2N}} \tag{4.1}$$

As an example we consider COXA, which searches 20000 nodes per second (nps). If it plays Arimaa using a 3-minute search, the number of nodes investigated is 3.6×10^6. Assume that an attempt is made to store them all in the transposition table. If the hash value consists of 58 bits (index + key), the probability of at least one type-1 error is:

$$1-e^{\frac{-3600000^2}{2\times2^{58}}} \approx 2.2\times10^{-5}$$

Because the transposition table is small, we cannot store all the positions occurred in the search. It happens that a position is to be stored in an entry, which is already occupied by another position. This is called a type-2 error or a collision. A choice has to be made which of the two involved positions should be stored in the transposition table. There are several replacement schemes (Breuker *et al.*, 1994), which tackle the collision problem. We use the familiar Deep replacement scheme. If a collision occurs, the one with the greatest depth is kept. Notice that for every new search process the transposition table is cleared.

4.3 Move Ordering

By sorting the moves generated in the alpha-beta search we can maximize the number of cut-offs in the process. Therefore we have to create an ordering function, which tries plausible moves first. The idea behind this is that if the best move is evaluated first, then the alpha and beta bounds will leave only a small window in which moves are not pruned. In effect every move which violates these bounds will be pruned. However, we do not know which the plausible moves are, since that is why we are searching in the first place. We can solve this problem by determining those by heuristics. There are two kinds of heuristics that exist: game-specific and game-independent. Game-specific heuristics are based on characteristics of the game and are discussed in section 4.3.1. Game-independent heuristics do not take characteristics of the game into consideration, but their applicability can depend on the kind of game. Besides the transposition move we will use two other game-independent move-ordering techniques, discussed in sections 4.3.2 and 4.3.3.

4.3.1 Game-specific Heuristics

The characteristics of Arimaa we used are that push and pull move will be investigated before single steps. This is done because this can give a material advantage and can open or close ways for other pieces. Another heuristic we used for the push and pull moves is that the moves of the stronger pieces will be tried first. The single steps will be generated also in order of the strongest pieces first. These single steps will be reordered when we use the history heuristic, as discussed in section 4.3.3.

4.3.2 Killer-move Heuristic

The killer-move heuristic (Huberman, 1968) is one of the most efficient move-ordering heuristics. It has proved to be quite successful and reliable in all sorts of games. The logic behind this heuristic is based on the assumption that the best move for a given position could well be a very good or even the best move in another position encountered at the same depth during the search process. Normally the last pruning or best move is stored, but more

killer moves can be stored. Pruning by the killer move is mostly due to information gathered at the same depth in the same subtree. The heuristic is cheap: for k moves at n plies, the memory cost is $k \times n$ entries in a move table. For more information on the killer-move heuristic we refer to (Akl and Newborn, 1977).

4.3.3 History Heuristic

The history heuristic was invented by Schaeffer (1983). The working of the history heuristic is in some way similar to that of the killer moves. Unlike the killer heuristic, which only maintains a history of one or a few best killer moves at each ply, the history heuristic maintains a history for every legal move seen in the search tree. The best move at an interior node is the move which either causes an alpha-beta cut-off, or which causes the best score. With only a finite number of legal moves, it is possible to maintain a score for each move in two (Gold and Silver) tables. At every interior node in the search tree the history-table entry for the best move found is incremented by 2^d where d is the depth of the subtree searched under the node. When a new interior node is examined, moves are re-ordered by descending order of their history scores.

The logic behind the history heuristic is that a move, often preferred over other moves, has a high likelihood of causing a large number of cut-offs. Moreover, a move that is good at this depth might very well also be good at other depths. A well-known problem that we stumble upon here is that the history table keeps building up and what appear to be good moves now can still be overshadowed by the good moves earlier in the game. To counterbalance this we divide all the counters by 2 at the start of a new search process.

This heuristic does not cost very much memory. The history tables are defined as two tables with 4096 entries ($64\ from\_squares \times 64\ to\_squares$), where each entry is 4 byte large. The total memory cost is only 32.7 kilobytes.

4.4 Overall Framework

In this chapter we have discussed several methods which are used in the experiments to create COXA. The pseudo code of Marsland (1986) and Donninger (1993) are combined. In which way alpha-beta, PVS, transposition tables, null move and tree construction are used is described by them. Before the null move is tried, the transposition table is used to prune a subtree or to narrow the window. As far as move ordering is concerned, the transposition move, if applicable, is always tried first. Next, the killer moves are tried. After that the push and pull moves are tried in the order as generated (i.e., according to the piece strength). All the other moves are ordered decreasingly to their scores in the history table.

5 Tuning the Evaluation Function

For optimal performance of the evaluation function we have to tune the weights that we added to the different parts of the evaluation function. In this chapter we will discuss the tuning of the evaluation function.

5.1 Setup of the Tuning

There are 14 weights and 1 bonus value that we have to tune. We choose for two type of players. One will be the standard player, were as the other (the variable player) will be the player were we vary the weights every time. The standard player will get the value 1.0 for all weights. The variable player will get also all weights set to 1.0 except the weight to be tuned. Each weight will be tuned separately. For the weight to be tuned we try a large negative value and then we play a tournament against the standard player were each player will be 50 times playing Silver and 50 times playing Gold. We also do this with a large positive value and some weight values in between. From the results we determine an optimum value.

When the optimum is reached we give both players the optimum value for that weight and we go on to the next weight.

5.1.1 Random Factor

Before the tuning can start, we need a random value that we add to every evaluation of a position, since otherwise we get the same result for every game. This random value makes the games every time a little bit different. The impact of the random value may not be too big to prevent determining the outcome of a search. Therefore we optimize this value first.

Every evaluation we start with a random number between 0 and 10. This number will be multiplied with a factor ‘x’ to be tuned. The outcome of the two numbers will then be added to the evaluation value.

For tuning this weight, we set the search depth to 4 without any time limit. This gives us a better look on how big the influence of the random factor is. For the standard player we used the value 0.0 for the weight so he will not have any influence on the results of the variable player. We made steps of 0.1 and investigated the number of different steps for move 2. Like we have discussed before also here we had to do with transpositions. These transpositions are deleted and the results are shown in figure 5.1.

As we can see in figure 5.1 for depth 4 the influence of a very small random factor has already much influence on the number of different moves. This is stabilized around 0.2. For depth 3 this is around 0.3. For depths 2 and 1 there is no influence at all in the beginning, and a random factor of at least 0.7 is needed for enough variation.

We chose for a random weight of 0.3, because there it has almost a stable influence on depth 3 and, as we can see in figure 5.2, 0.3 is the point were the random variable still has not too much influence on the outcome of the games.

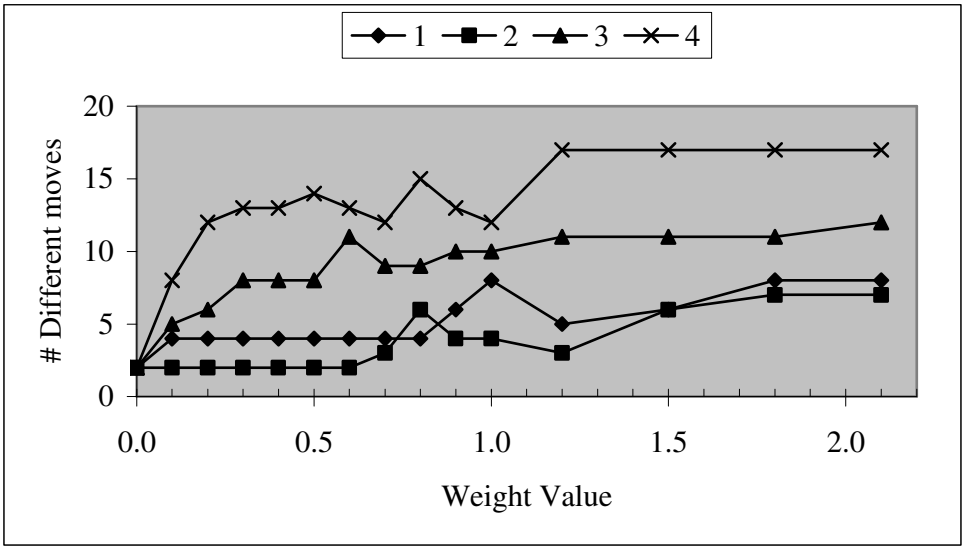

Figure 5.1: Influence of the random weight on the difference of moves for each depth.

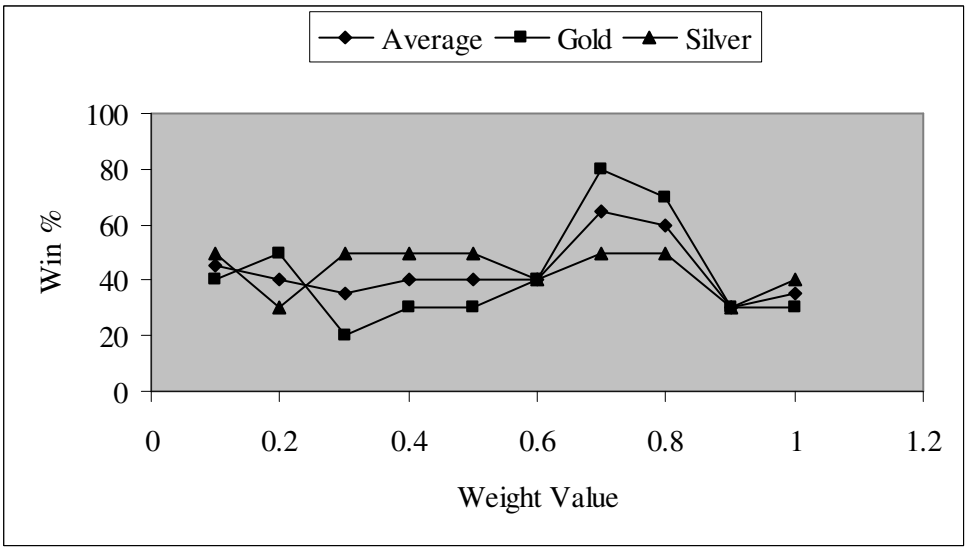

Figure 5.2: Winning % of the random value.

5.2 Tuning the Basic Evaluation

After determining the random variable we will tune the other weights of our evaluator according to the setup described. Now we take a look at the weights of the basic evaluation parts.

5.2.1 Material Weight

In this experiment we set for both players the random value at 0.3 and the remaining weights at 1.0; only the material weight of the variable player receives the values -20, -10, 0, 10, 20, 30, and 40. From the result depicted, figure 5.3, we see a top around 10. This is the area were we have used more values.

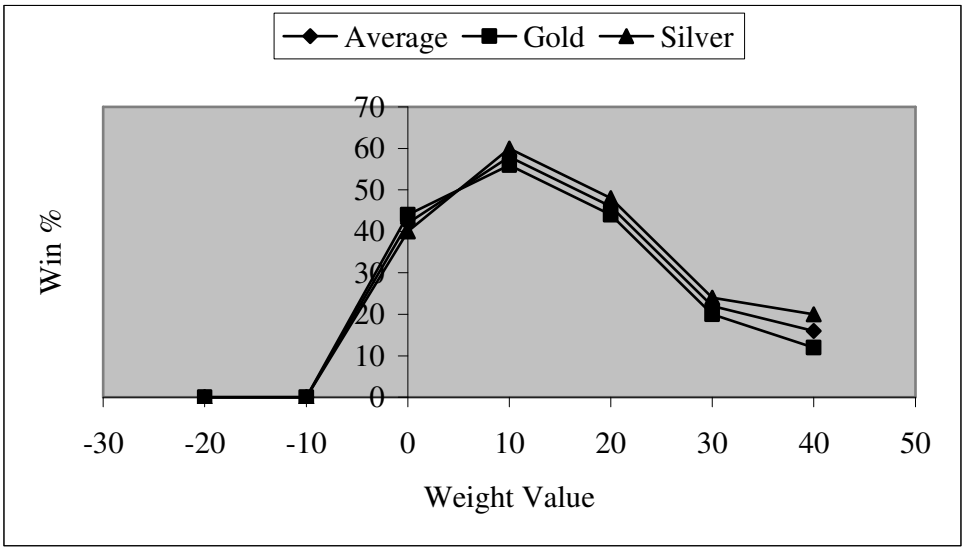

Figure 5.3: Winning % of the material value.

By zooming in step by step to the top we get the results shown in figure 5.4. As optimum we settled to use the value of 4 for the material weight.

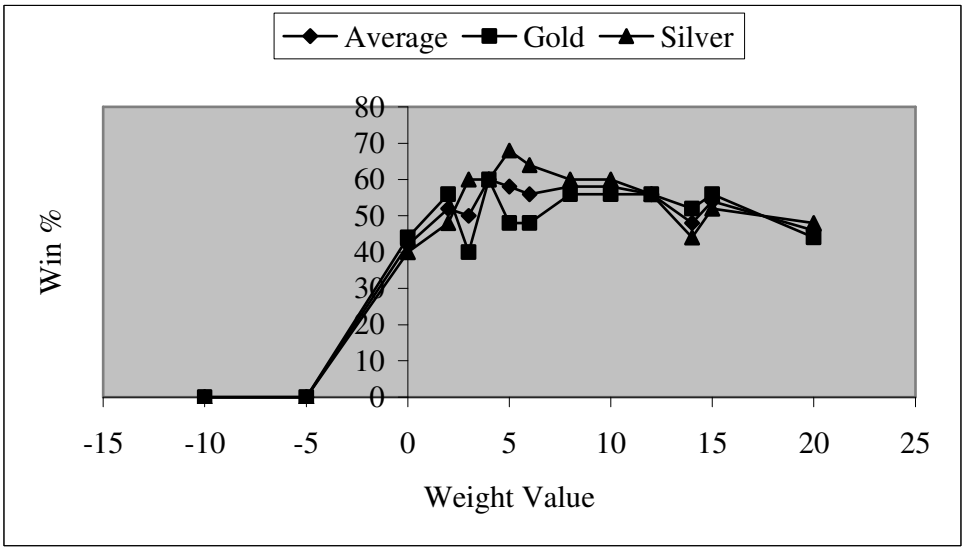

Figure 5.4: Winning % of the material value (zoomed in).

5.2.2 Piece-square Tables

As said in chapter 3, we have three different piece-square tables for all six pieces, but six weights that we have to tune. We start with the piece-square table weight for the rabbit.

Here we also started with some sparsely distributed values to locate the top. These values ranged from -10 to 40 with steps of 10 and plus the value of 60. the results are shown in figure 5.5.

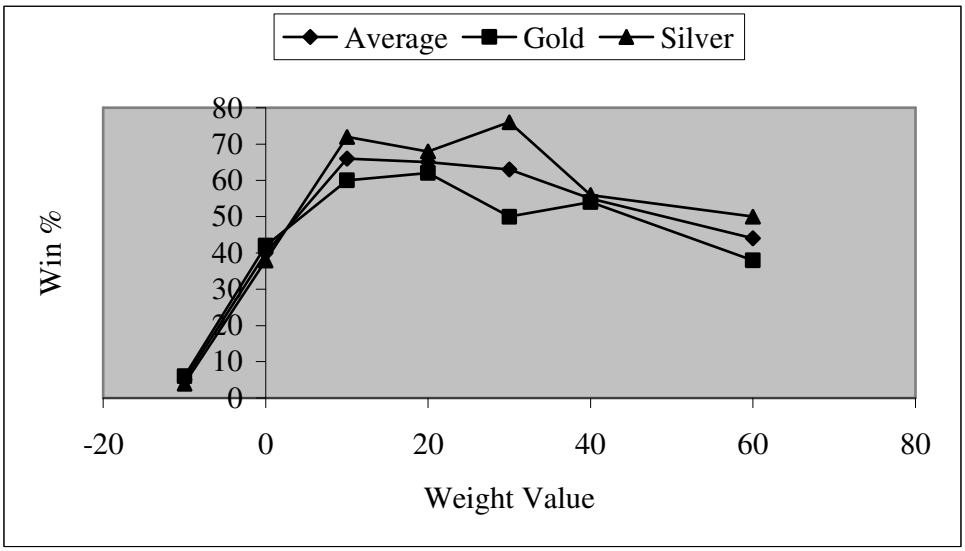

Figure 5.5: Winning % of the rabbit piece-square table.

We see a top around 10 and that is the point where we zoomed in. From the results shown in figure 5.6 we decided for the value 6 as the optimum.

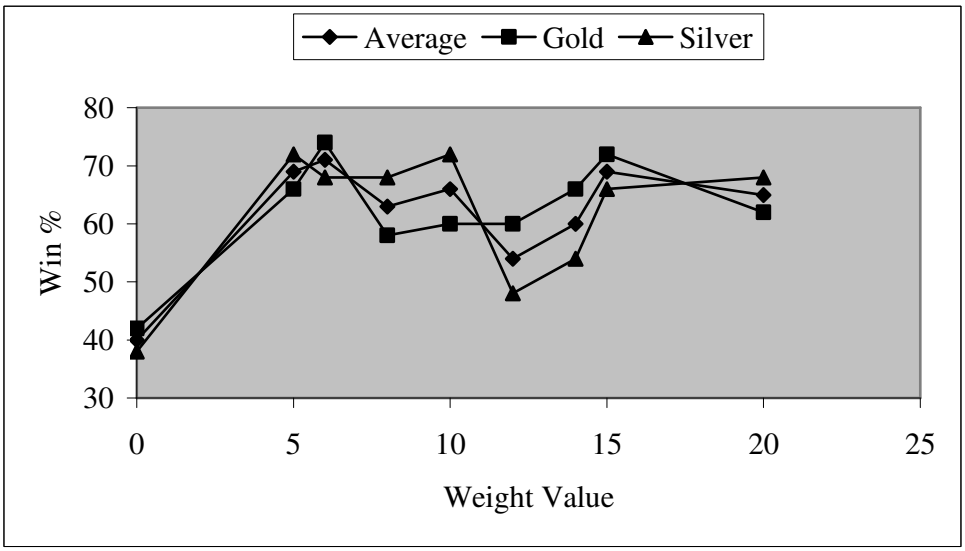

Figure 5.6: Winning % of the rabbit piece-square table (zoomed in).

After the weight of the rabbit piece-square table, we tuned the weight of the cat piece-square table. The results are shown in figure 5.7, were we see that the top is located around 0. Zooming in on 0 we obtain figure 5.8 where we see that the optimum is at 0 or 1. We decided to use the value 1 as the best value for the cat piece-square table weight, since for that value the difference between the gold and silver winning percentage is smaller.

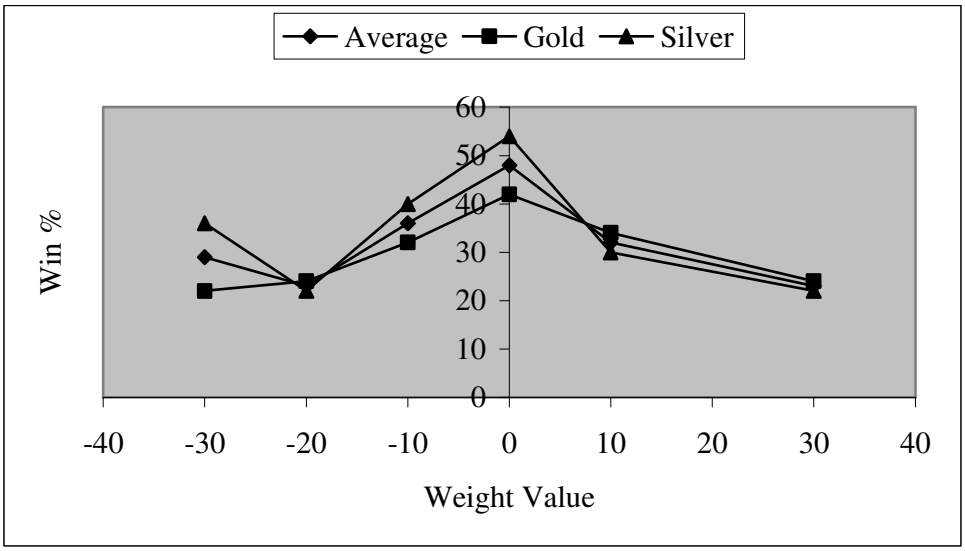

Figure 5.7: Winning % of the cat piece-square table.

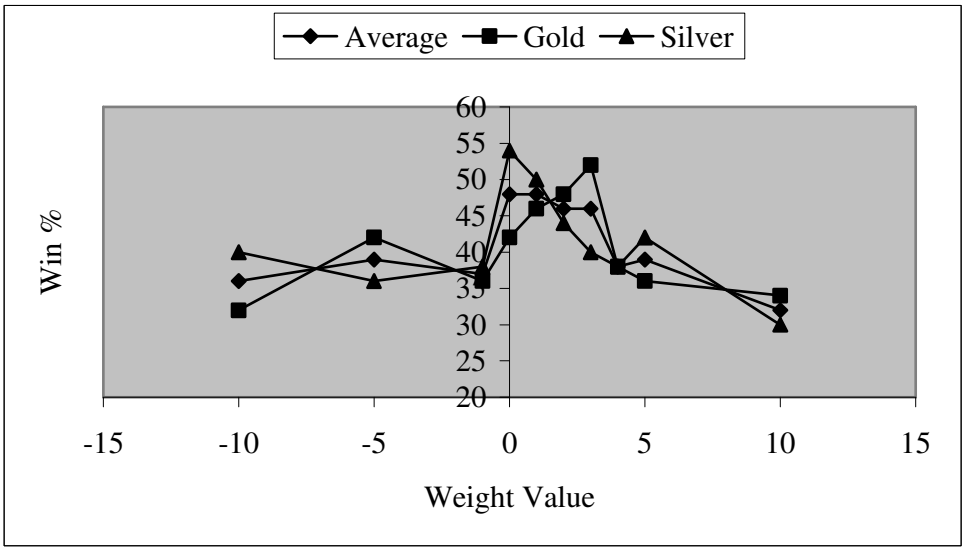

Figure 5.8: Winning % of the cat piece-square table (zoomed in).

The next weight is that of the piece-square table of the Dog. Also here we did many test runs. We found a best value of 0, as we can see in figures 5.9 and 5.10. This means that the dog piece-square table that we use at the moment is no improvement to the player.

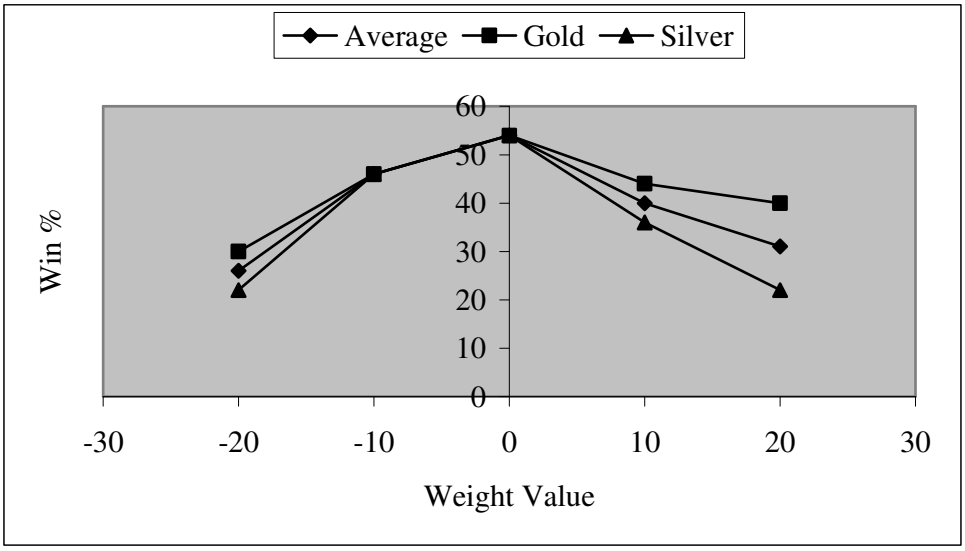

Figure 5.9: Winning % of the dog piece-square table.

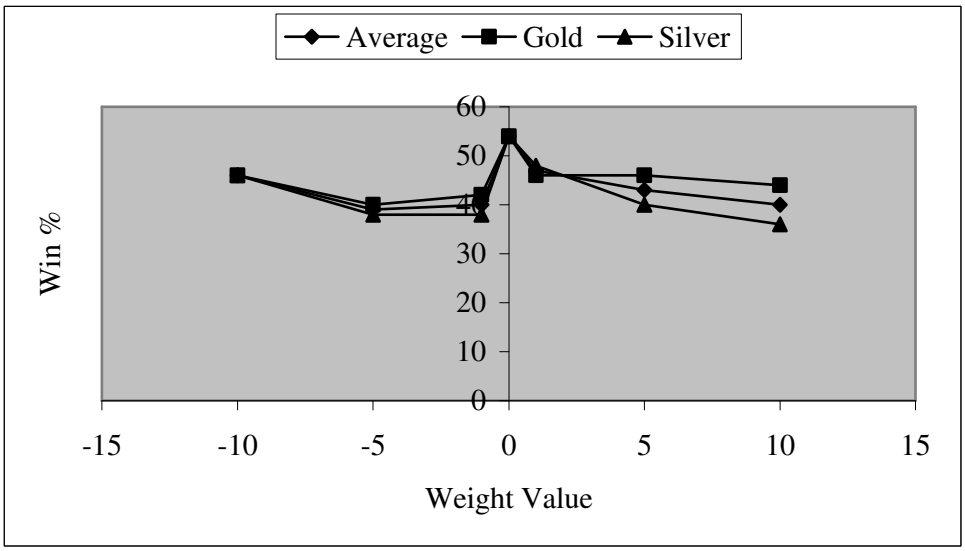

Figure 5.10: Winning % of the dog piece-square table (zoomed in).

After training the weight for the dog piece-square table we similarly tuned the weight for the Horse. Just like by the Dog, this piece-square table did not give any improvement to the player, as we can see in figures 5.11 and 5.12.

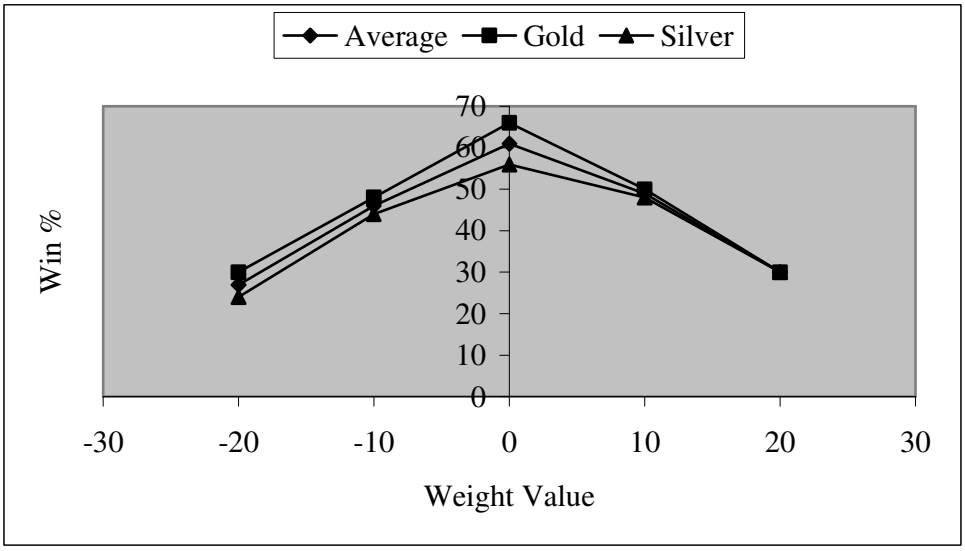

Figure 5.11: Winning % of the horse piece-square table.

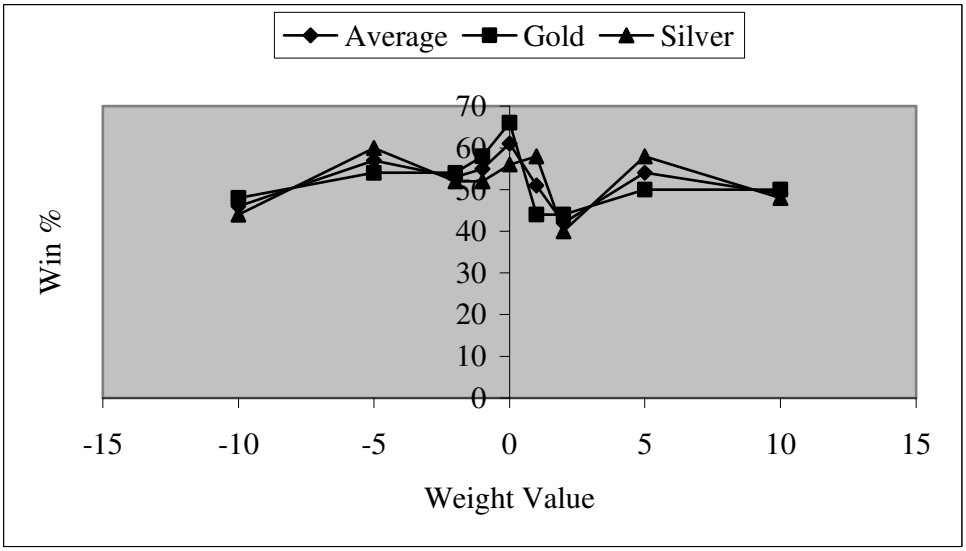

Figure 5.12: Winning % of the horse piece-square table (zoomed in).

Next we tuned the value of the weight of the camel piece-square table. This table seems to be an improvement to our player because, as we can see in figures 5.13 and 5.14, we find the value 3 to be the best value.

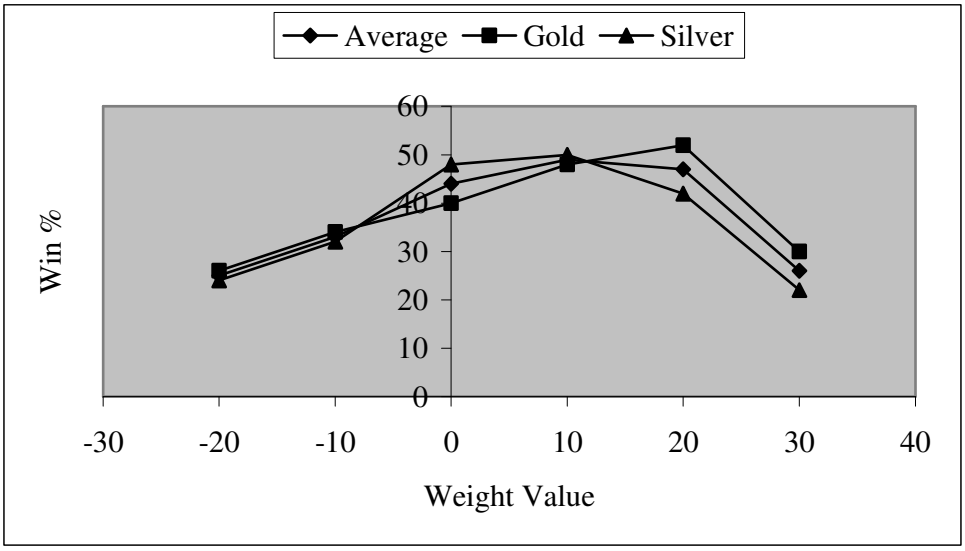

Figure 5.13: Winning % of the camel piece-square table.

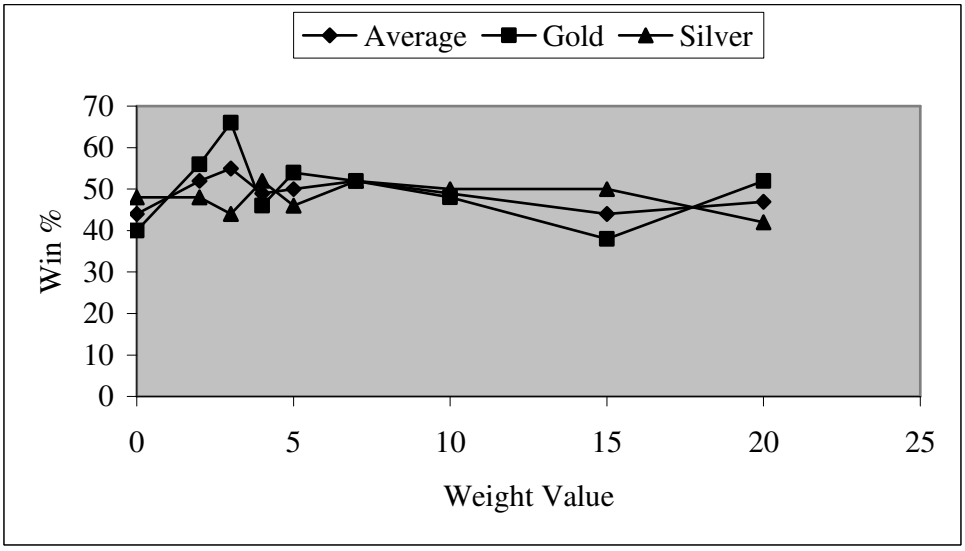

Figure 5.14: Winning % of the camel piece-square table (zoomed in).

For the last weight of the piece-square table, that of the Elephant, we performed the test runs and found the value of 15 to be the best for the player. This is a large value, which is explainable since the Elephant is the strongest piece on the board and has to get access to each side as quick as possible. Therefore it strongly wants to stay in the middle of the board just as the piece-square table indicates.

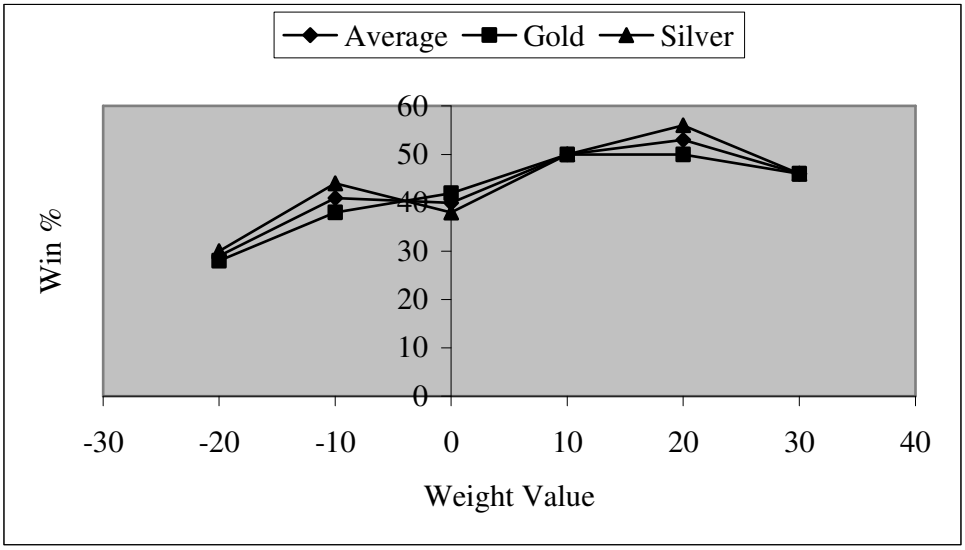

Figure 5.15: Winning % of the elephant piece-square table.

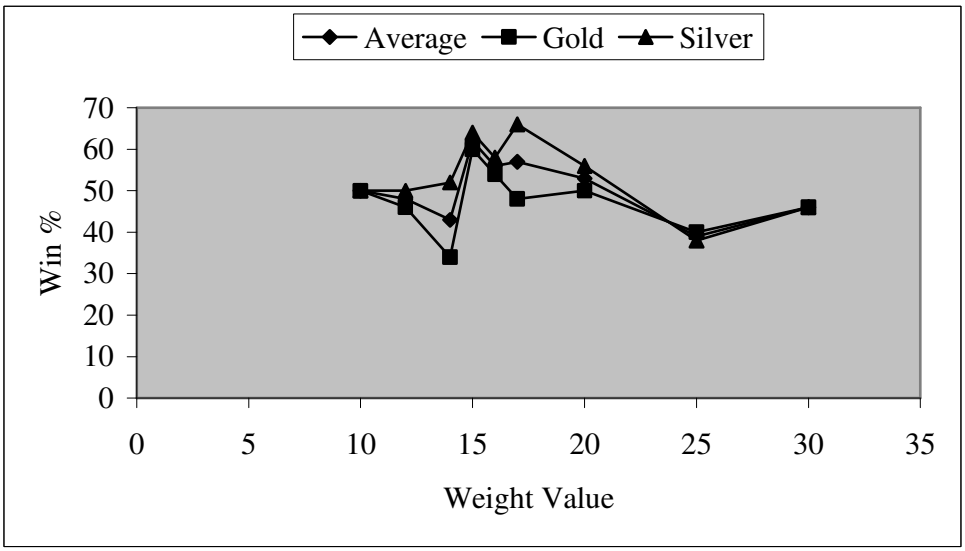

Figure 5.16: Winning % of the elephant piece-square table (zoomed in).

5.2.3 Mobility

As said in chapter 3 we split mobility in two, namely a weight for the frozen part, and one for the partial mobility.

First we tuned the frozen part. Here we find a negative weight, which is correctly indicated that a piece is penalized for being frozen. A can be seen in figures 5.17 and 5.18 we found the value -8 for this weight.

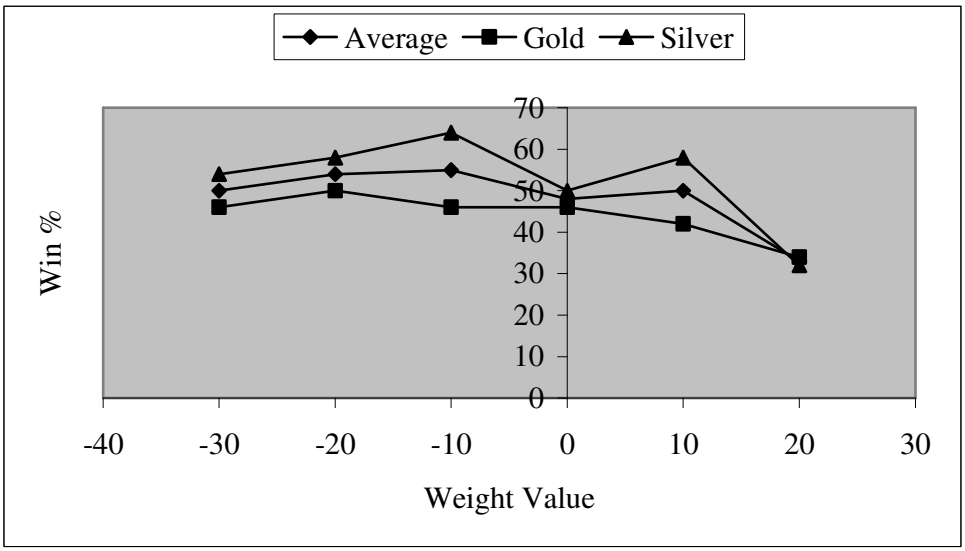

Figure 5.17: Winning % of the frozen-weight value.

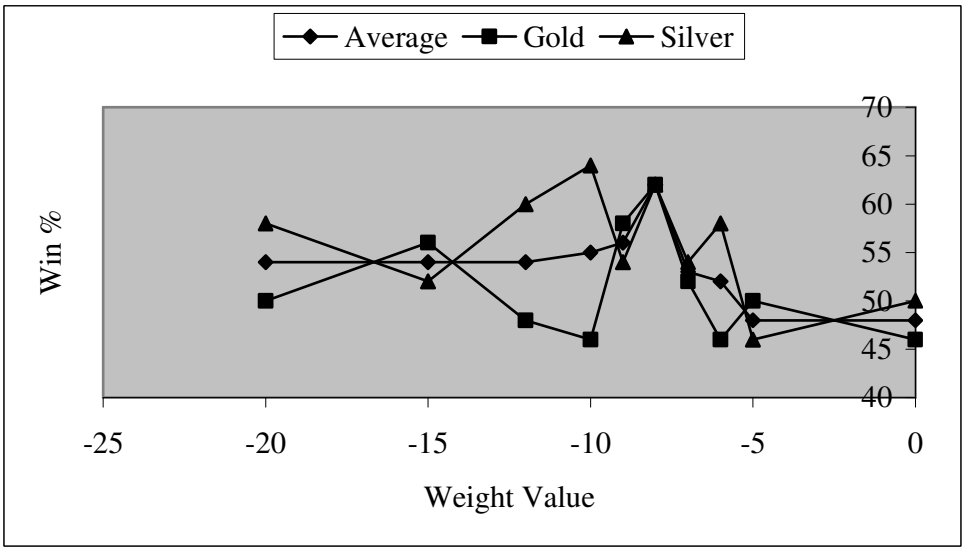

Figure 5.18: Winning % of the frozen-weight value (zoomed in).

Second we tuned the partial-mobility part. Here we see that the effect of this weight hardly depends on its value, except for the optimum value 0. See figures 5.19 and 5.20.

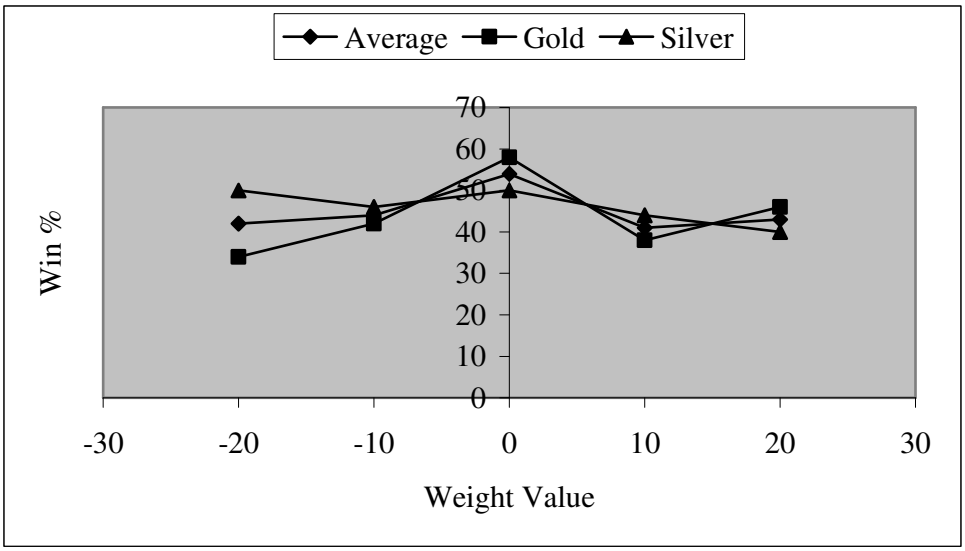

Figure 5.19: Winning % of the partial-mobility weight value.

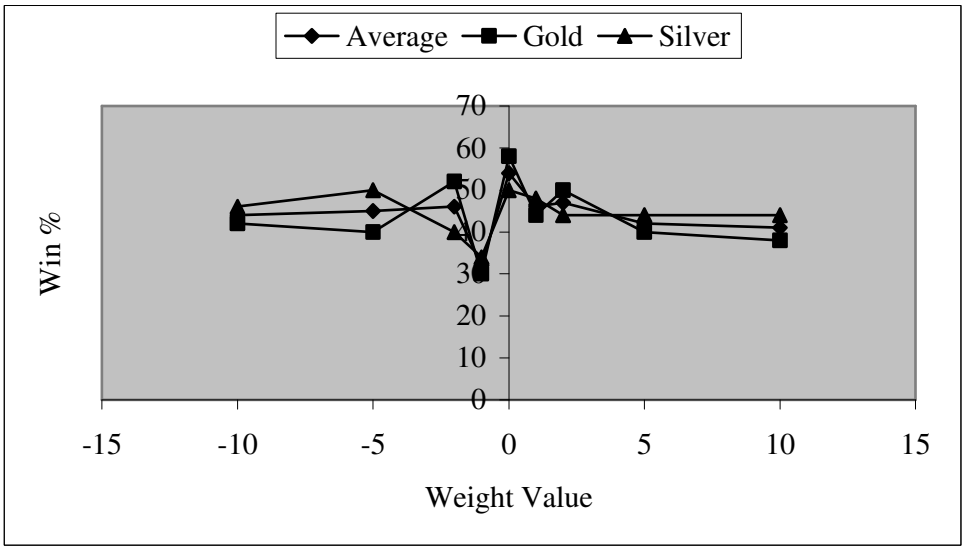

Figure 5.20: Winning % of the partial-mobility weight value (zoomed in).

5.3 Tuning the Knowledge Evaluator

Besides the basic evaluator we also had the knowledge-evaluation part which we described in section 3.2. The tuning of this part will be described in this section, similarly as we did for the basic evaluator.

5.3.1 Rabbit Evaluation

Like we have seen in chapter 3 we had two parts in the rabbit evaluation. First we had a weight to encourage the Rabbits to from solid walls. The results for this weight can be seen in figures 5.21 and 5.22, where it is tuned to the value 10.

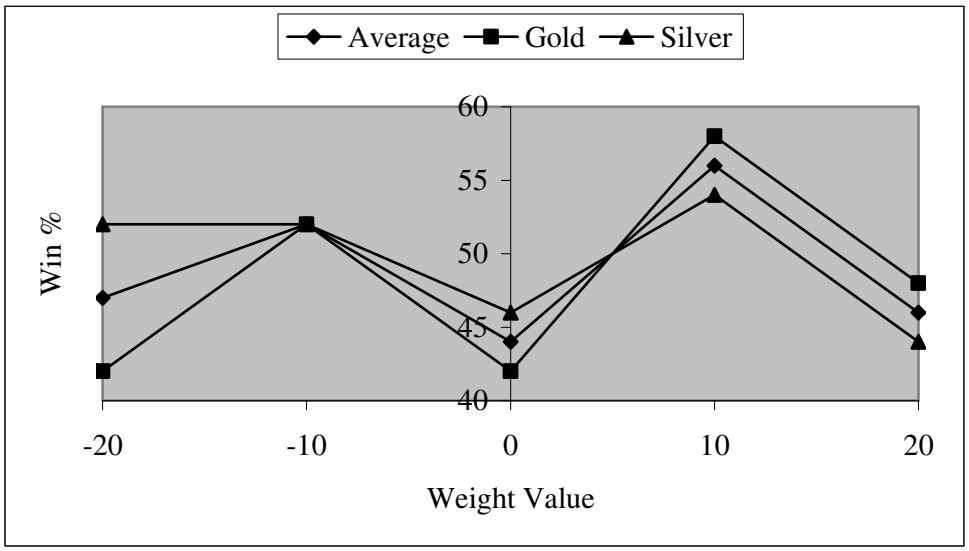

Figure 5.21: Winning % of the solid-wall weight value.

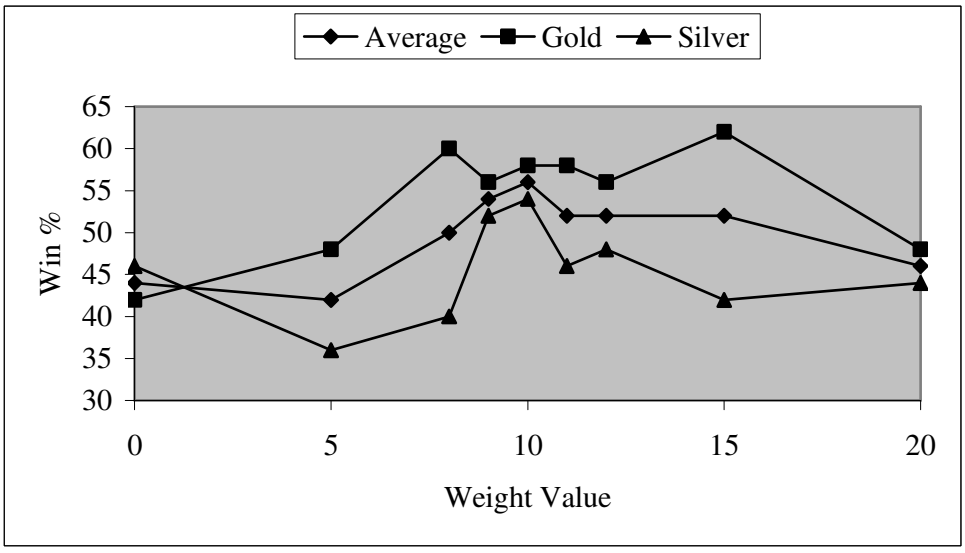

Figure 5.22: Winning % of the solid-wall weight value (zoomed in).

Second we had to tune the weight rewarding Rabbits on free files to the goal. This weight is tuned in figures 5.23 and 5.24 and resulted in the value 50.

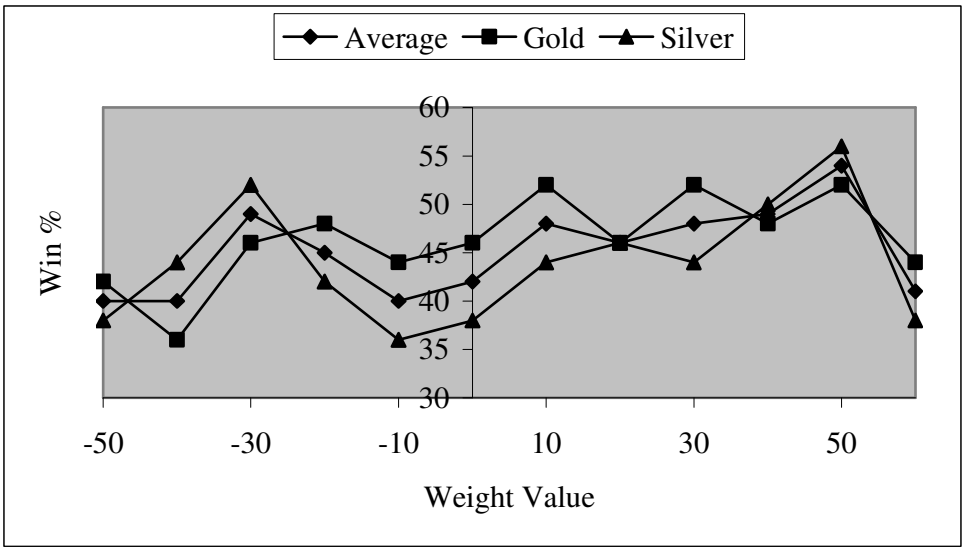

Figure 5.23: Winning % of the free-file weight value.

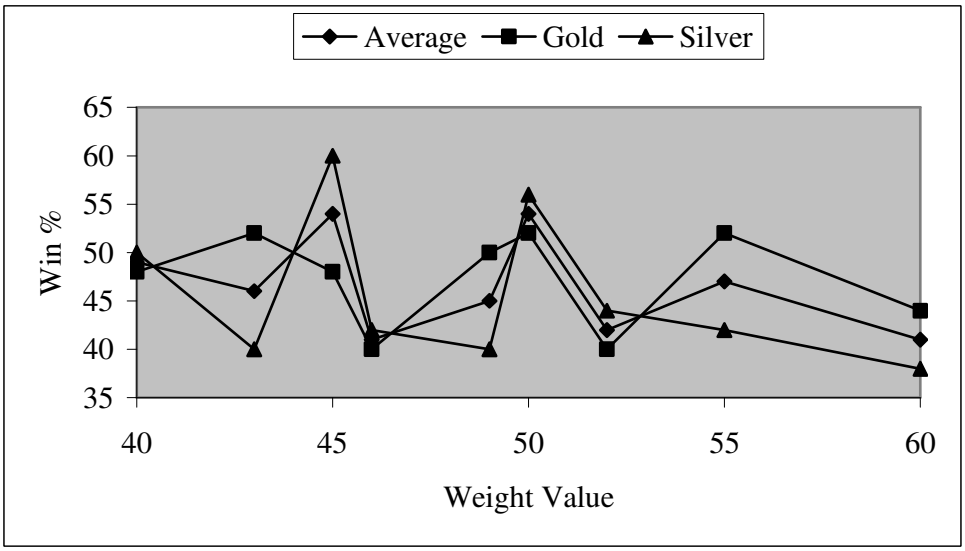

Figure 5.24: Winning % of the free-file weight value (zoomed in).

5.3.2 Goal-threat Evaluator

Like the other tuning experiments we did a rough investigation first (figure 5.25), and then zoomed in around value 10 (figure 5.26). The value 5 was found to be the best for the goal-threat evaluator weight.

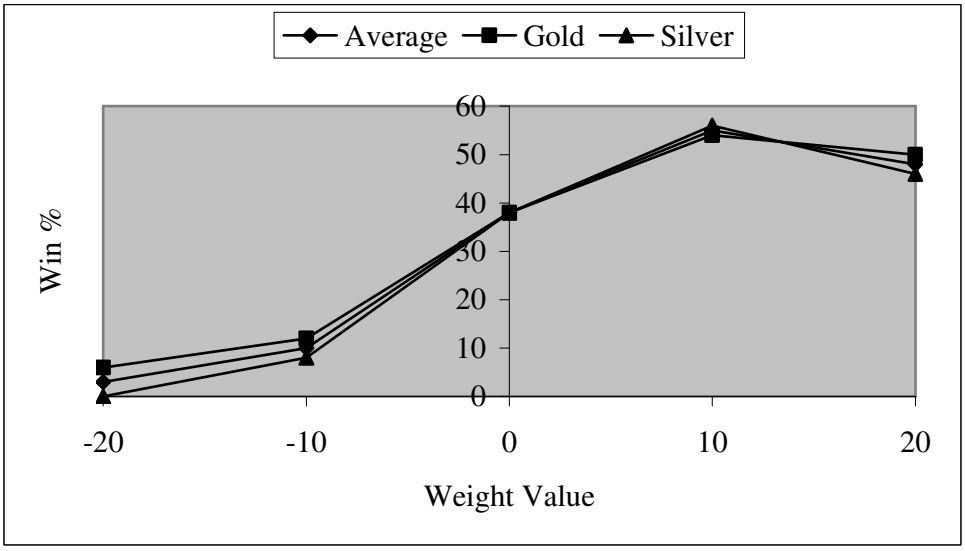

Figure 5.25: Winning % of the goal-threat weight value.

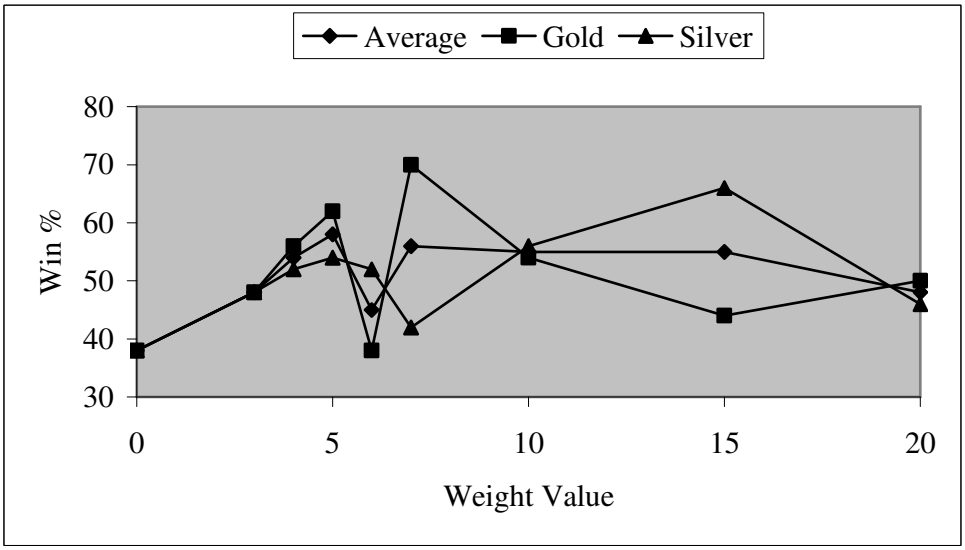

Figure 5.26: Winning % of the goal-threat weight value (zoomed in).

5.3.3 Trap-control Evaluator

For the trap control evaluator we had two ways to determine the bonus. First we had a weight when there was only one colour present at the goal. This weight we tuned according to figures 5.27 and 5.28. Here we see that the best value is 0. This means that the current trap control for one colour present is no improvement to our player.

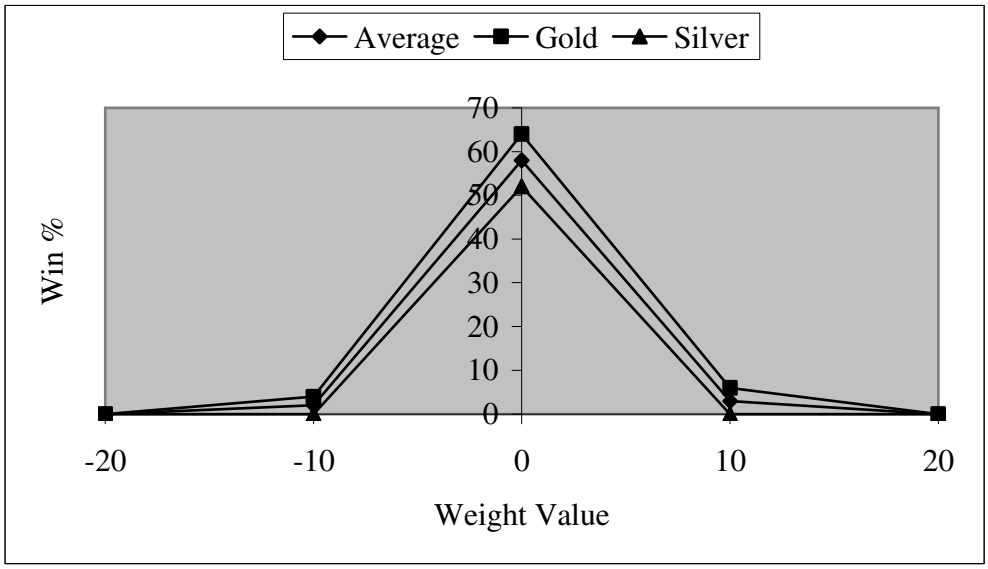

Figure 5.27: Winning % of the trap-control weight value.

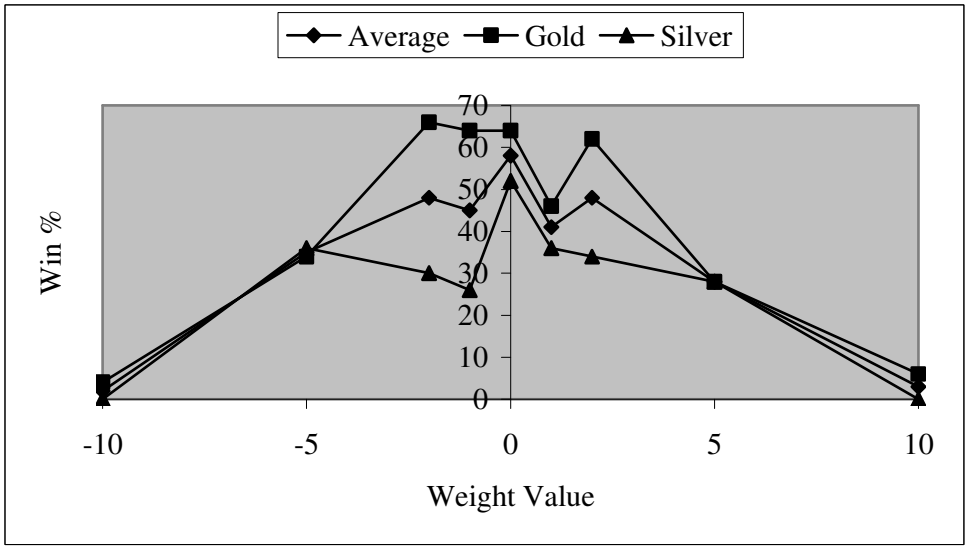

Figure 5.28: Winning % of the trap-control weight value (zoomed in).

Second we had the possibility that both colours are present near a trap. For this we determine how big the weight should be for the strongest player. This tuning can be seen in figures 5.29 and 5.30. Here we see again that the best value is 0. This means also that this brings no improvement to our player.

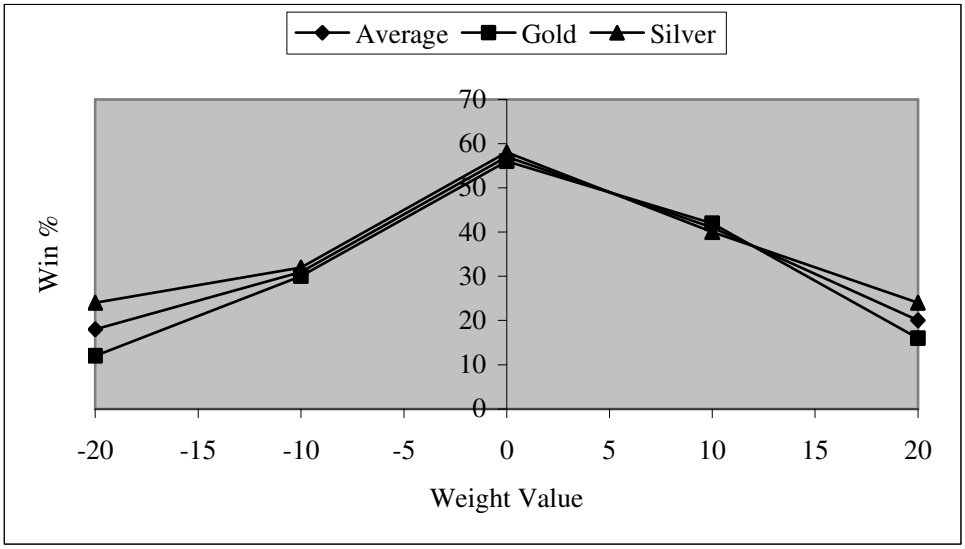

Figure 5.29: Winning % of the bonus weight value.

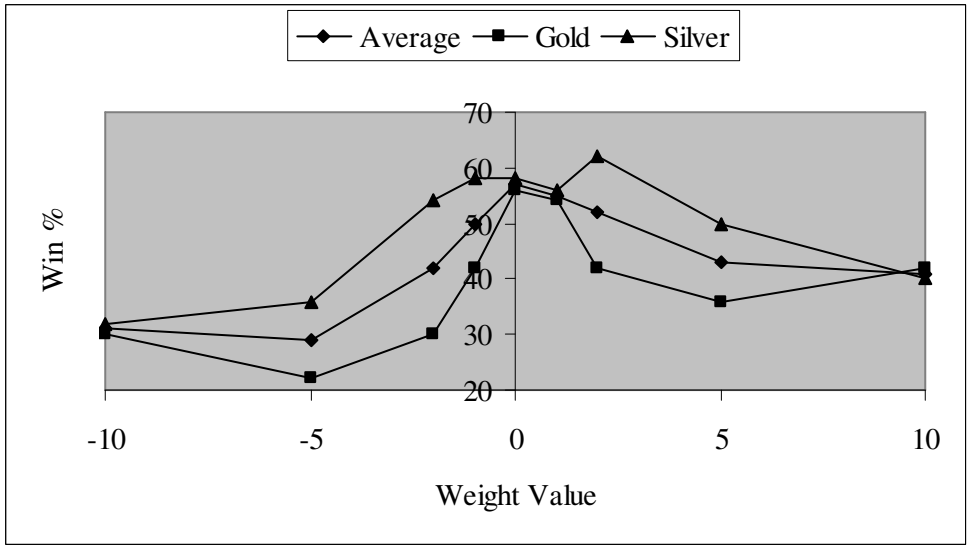

Figure 5.30: Winning % of the bonus weight value (zoomed in).

5.4 Chapter Conclusion

We have tuned the weights of the evaluation function, the results of which are gathered in table 5.1. We see that some parts of the evaluation (viz. with weight 0) are no improvement to COXA. So we removed the dog piece-square table, horse piece-square table, and the partial mobility for the basic evaluator, and trap control for single and two player(s) for the extended evaluator.

Furthermore, we see that the piece-square table for the Elephant is very important with a weight of 15, meaning that it is important for the Elephant to be in the middle of the board.

Another point that gets our attention is the rabbit evaluation. Here we see that solid walls with weight 10 are important. We also see that free files become very important when they are present (weight 50).

Overall we see that the basic evaluation does good work to give improvement. The extended evaluation also gives improvement except the trap control. This needs more attention before it can become part of a better playing program.

	Weight	Value
Basic	material balance	4
	rabbit piece-square table	6
	cat piece-square table	1
	dog piece-square table	0
	horse piece-square table	0
	camel piece-square table	3
	Elephant piece-square table	15
	frozen pieces	-8
	partial mobility	0
Extended	Rabbit's solid walls	10
	Rabbit's free files	50
	goal threat	5
	trap control single player	0
	trap control two players	0

Table 5.1: Overview of the weight values of the tuned evaluation function.

6 Testing of Search Algorithm

This chapter describes some numerical tests about the search techniques described in chapter 4. All results in this chapter are taken from a series of 100 games of Arimaa. Each algorithm played 50 times as Gold, and 50 times as Silver. The small random factor of 0.3 was also added to the evaluation function to avoid playing the same game 100 times. For each experiment, the search depth of the algorithm was fixed at depth 5, and iterative deepening was enabled.

First, the results for the various windowing techniques are described in section 6.1. The best algorithm is then extended with the transposition tables (see section 6.2) and then the move ordering techniques are added (see section 6.3). Finally, in section 6.4 a short conclusion is given about all search-algorithm experiments.

In all tests, the results are displayed as the average number of nodes searched per search depth, the time in seconds to reach that depth and the percentage of nodes gained when using two different search algorithms.

6.1 Windowing Techniques

Two windowing techniques were implemented: principal variation search (PVS), and the null move. First PVS is tested in two ways against plain Alpha Beta in subsection 6.1.1, and then the best algorithm is tested against itself with the null move enabled in subsection 6.1.2. These tests were run at a fixed search depth of 5 plies, as it almost takes three days to complete a test run of 100 games at this depth.

6.1.1 Principal Variation Search

Principal variation search was tested in two ways. In the first test, plain Alpha Beta played against PVS, where PVS is applied on every step of a move.

	Alpha Beta		PVS all steps		
depth	# sec	# nodes	# sec	# nodes	% gain
1	0.0	20	0.0	23	-15.0
2	0.0	483	0.0	494	-2.3
3	1.7	10943	1.6	10825	1.1
4	41.7	251166	36.8	244461	2.7
5	65.6	352949	61.1	351604	0.4

Table 6.1: Principal Variation Search (all steps) test results.

As table 6.1 shows, PVS performs much like Alpha Beta, with only a small enhancement at depth 4.

In the second test, Alpha Beta was compared to PVS, where PVS was only used on the last step of a move.

depth	Alpha Beta		PVS only last step		
	# sec	# nodes	# sec	# nodes	% gain
1	0.0	19	0.0	19	0.0
2	0.0	438	0.0	431	1.6
3	1.5	9383	1.4	9206	1.9
4	33.6	203392	32.7	196009	3.6
5	60.0	322094	58.7	316579	1.7

Table 6.2: Principal Variation Search (last step) test results.

As shown in table 6.2, principal variation search, only on the last step, performs slightly better than PVS on all steps. Here we see an improvement on depth 4, which is where the last step of the first player occurs.

6.1.2 Null-move Search

In the last test of this series, two different versions of principal variations search played against each other. One used the null move heuristic, and the other did not. For the null move, R (the reduced- depth parameter) was set to 0, because here the null move is only applied at depth 5. When the null move is used the depth springs from 5 to 9 directly because switching from player means omitting 4 steps. On the others steps it is allowed to pass anyway and adding a null move.

depth	Null move Disabled		Null move enabled		
	# sec	# nodes	# sec	# nodes	% gain
1	0.0	20	0.0	20	0.0
2	0.0	479	0.0	475	0.8
3	1.7	10782	1.7	10625	1.5
4	40.2	240242	40.3	241485	-0.5
5	64.4	345210	66.4	356405	-3.2

Table 6.3: Null-move search test result

As table 6.3 clearly shows, when the null move is used (see depth 5) it is bad for our search. To test the null move really good we have to go much deeper in the search, since it only can be used at depths like 5, 9, 13, 17, etc. But depths 9 and above are unreachable at the moment.

6.2 Transposition Table

To test the effect of transposition tables on Arimaa, each of its components was enabled separately and tested against the reference player from now on: PVS on the last step without null move. In subsection 6.2.1 the transposition table is only used when an exact hit has been found. In subsection 6.2.2 only upper/lower bound hits in the table are used. In subsection 6.2.3 the table is only used as a move ordering mechanism, and finally in subsection 6.2.4 all components are enabled and the table is fully used.

6.2.1 Only using Exact Table Hits

In this test, the transposition table pruning was only enabled when the value stored in the table was an exact value. The number of tthits is therefore defined as the number of times that exact values in the tables were used to cut off the search.

depth	TT disabled		TT enabled (exact)			
	# sec	# nodes	# sec	# nodes	# tthits	% gain
1	0.0	20	0.5	20	0	0.0
2	0.0	454	0.6	257	205	43.4
3	1.6	10053	1.4	2821	7442	71.9
4	38.1	223364	15.1	20127	211669	91.0
5	60.3	323719	38.8	38889	411411	88.0

Table 6.4: TT (exact) test results.

As table 6.4 shows, only pruning the tree when an exact hit is found in the transposition table reduces the number of nodes with about 90% at search depth 5.

6.2.2 Only using Upper and Lower Bound Pruning

In this test, only upper and lower bound hits in the table were used to speed up the search process. These values are obtained from previous Alpha and Beta values and can be used to adjust the search window. In this case, the tthits count is the number of upper and lower bound hits that were used to prune the tree.

depth	TT disabled		TT enabled (upper/lower bound)			
	# sec	# nodes	# sec	# nodes	# tthits	% gain
1	0.0	20	0.5	20	0	0.0
2	0.0	479	0.6	484	0	-1.0
3	1.7	10708	1.1	6146	348	42.6
4	39.8	241324	6.1	58676	3872	75.7
5	64.9	353936	12.2	78206	43241	77.9

Table 6.5: TT (upper / lower bound) test results.

As shown in table 6.5, using upper and lower bound reduces the number of nodes with about 78% at search depth 5.

6.2.3 Only using Move Ordering

In this test, the transposition table was only used as a move ordering mechanism. If the current position has a value and a move stored in the table, that move is tried first when searching the position, even if the value stored in the table could have been used to cut off the tree. The tthits value stands for each time the transposition table is used to re-order the moves.

depth	TT disabled		TT enabled (ordering)			
	# sec	# nodes	# sec	# nodes	# tthits	% gain
1	0.0	20	0.5	21	0	-5.0
2	0.0	471	0.6	486	217	-3.2
3	1.6	10489	1.4	10902	8664	-3.9
4	39.3	235967	15.9	248376	236015	-5.3
5	63.2	341181	41.6	488458	686605	-43.2

Table 6.6: TT (ordering) test results.

As the results in table 6.6 shows, only using the move ordering feature of the transposition table yields a negative reduction in nodes searched of about 43% at search depth 5.

6.2.4 Full use of the Table.

In the last test of this series, the transposition table was fully enabled, combining all three features tested in the previous tests. This time, tthits is the number of times some information stored in the table was used in the search. This can either be an exact hit, an upper/lower bound hit or a re-ordering hit.

depth	TT disabled		TT enabled (full)			
	# sec	# nodes	# sec	# nodes	# tthits	% gain
1	0.0	20	0.5	21	0	-5.0
2	0.0	452	0.6	273	221	39.6
3	1.6	9896	1.1	2508	4143	74.7
4	36.2	217874	6.3	21143	43759	90.3
5	62.7	340407	12.5	40531	84999	88.1

Table 6.7: TT (full) test results.

As table 6.7 shows, using the full transposition table lowers the number of nodes searched with about 90% at depth 5. This is the same as in table 6.4 where we only used the exact feature. However, the full feature searches uses for a similar number of nodes considerably less time, a gain of almost 68% at search depth 5. The reason is that the number of times information is retrieved from the table, as indicated by the number of hits, is considerably lower when the full feature is used. Consequently, we opted to use the full feature from now on.

6.3 Move Ordering

As said in chapter 4 we put the game-specific heuristics directly into the code. Two different game-independent move-ordering techniques were implemented for Arimaa: the killer-move heuristic, and the history heuristic. (The use of the transposition move as move ordering was already tested in subsection 6.2.3.) The best algorithm so far, PVS only on the last step of a move with the null move disabled and the transposition table fully used, will play against its counterpart with the killer move enabled in subsection 6.3.1, where a few variants of the killer-move heuristic are tested.

Furthermore, in subsection 6.3.2 the history heuristic is tested. In all these tests, most settings remain unchanged, but the search depth was fixed at ply 6 since use of the transposition table enabled a considerable speed-up in search time.

6.3.1 Killer-move Heuristic

To test the impact of the killer-move heuristic, five tests were run. In test 1, the best algorithm so far, PVS only on the last step of a move with the null move disabled and the transposition table fully used, played against its counterpart with killer-move heuristic enabled. For this first test, only one killer was recorded per search depth. For the other four tests each time one killer extra will be recorded per search depth, so in the last test we have 5 killers recorded.

depth	Killer disabled		Killer enabled (record one)		
	# sec	# nodes	# sec	# nodes	% gain
1	0.5	19	0.5	20	-5.3
2	0.6	230	0.6	244	-6.1
3	1.0	1983	1.1	2144	-8.1
4	4.9	15783	5.4	17358	-10.0
5	9.7	30685	10.7	34060	-11.0
6	15.1	43201	16.8	49181	-13.8

Table 6.8: Killer-move heuristic (record one) test results.

As shown in table 6.8. using the killer heuristic as move ordering on top of PVS and transposition table yields an increase of up to 14 % at depth 6.

In test 2, we record two killers for one player and none for the other.

depth	Killer disabled		Killer enabled (record two)		
	# sec	# nodes	# sec	# nodes	% gain
1	0.5	20	0.5	19	5.0
2	0.6	247	0.6	242	2.0
3	1.1	2194	1.1	2124	3.2
4	5.5	18029	5.3	17198	4.6
5	11.0	34918	10.4	33262	4.7
6	16.9	47045	16.2	45547	3.2

Table 6.9: Killer-move heuristic (record two) test results.

Table 6.9 shows a little improvement of around 4-5 % at depth 5.

In test 3, again we recorded one extra killer, i.e., three for one player and none for the other.

depth	Killer disabled		Killer enabled (record three)		
	# sec	# nodes	# sec	# nodes	% gain
1	0.5	19	0.5	19	0.0
2	0.6	239	0.6	240	-0.4
3	1.1	2093	1.1	2115	-1.1
4	5.2	17023	5.2	17112	-0.5
5	10.3	33031	10.4	33410	-1.1
6	15.9	45483	16.4	47763	-5.0

Table 6.10: Killer-move heuristic (record three) test results.

As shown in table 6.10 it gives no improvement in node reduction. It searches throughout more nodes.

In test 4, the fourth killer will also be recorded for one player and none for the other.

depth	Killer disabled		Killer enabled (record four)		
	# sec	# nodes	# sec	# nodes	% gain
1	0.5	19	0.5	20	-5.3
2	0.6	236	0.6	243	-3.0
3	1.0	2061	1.1	2143	-4.0
4	5.1	16521	5.3	17388	-5.2
5	10.1	32062	10.5	33584	-4.7
6	15.9	46557	16.2	45050	3.2

Table 6.11: Killer-move heuristic (record four) test results.

Once again, as table 6.11 shows, more recording of killers gives no reduction of nodes searching. Only a little improvement at depth 6 of some 3%.

Finally in test 5, we recorded five killers for one player.

depth	Killer disabled		Killer enabled (record five)		
	# sec	# nodes	# sec	# nodes	% gain
1	0.5	20	0.5	20	0.0
2	0.6	242	0.6	244	-0.8
3	1.1	2126	1.1	2162	-1.7
4	5.3	17186	5.4	17646	-2.7
5	10.4	33306	10.7	33977	-2.0
6	16.4	47658	16.4	45445	4.6

Table 6.12: Killer-move heuristic (record five) test results.

As shown in table 6.12, we see here no improvement in reducing nodes, except for a small gain at depth 6 of almost 5%. This could mean that deeper searches will give more reducing of nodes. Further research is needed to answer this question.

6.3.2 History Heuristic

In the history test, PVS only on the last step of a move with the null move disabled, the transposition table fully used and the killer move enabled using two killers per depth was equipped with the history heuristic enabled. For this test, the values in the history tables were incremented with 2^d.

depth	History disabled		History enabled		
	# sec	# nodes	# sec	# nodes	% gain
1	0.5	19	0.5	20	-5.3
2	0.6	230	0.6	251	-9.1
3	1.0	2009	1.1	2256	-12.3
4	5.0	16281	5.7	18645	-14.5
5	9.9	31639	11.4	35768	-13.1
6	15.3	43205	16.8	41972	2.9

Table 6.13: History-heuristic test results.

As table 6.13 shows, the history heuristic brings no improvement to our search. Once again, like with the killer-move heuristic, it gives however a small improvement at depth 6.

6.4 Chapter Conclusions

As seen throughout this chapter, most search enhancements yield a positive result in nodes reduction. So far, only null move search, some killer-move heuristic variations and the history heuristic had a negative influence. Therefore, the recommended search algorithm will use the following enhancements:

- principal-variation search but only at the last step;
- the transposition table fully used;
- the killer-move heuristic with 2 killers per depth.

7 Conclusions

This chapter contains the final conclusions on our research. Section 7.1 revisits the problem statement and research questions, and section 7.2 lists possibilities for future research.

7.1 Problem Statement and Research Questions Revisited

In section 1.3 we have defined the following research questions:

What is the complexity of Arimaa?

In chapter 2, we have seen that the state-space complexity of Arimaa is $O(10^{43})$, and the game-tree complexity is $O(10^{300})$. These numbers are only an approximation, since the state-space complexity includes some unreachable positions as shown, and the game-tree complexity is based on games between humans and is subject to change once more Arimaa programs are created. Both the state-space complexity and the game-tree complexity of Arimaa are comparable to that of the game of Amazons. Completely solving Arimaa is, probably impossible in a foreseeable future.

The second research question was:

Can we use knowledge about the game of Arimaa in an efficient and effective implementation of an Arimaa program?

The evaluation function described in chapter 3 implements different strategies. The basic evaluation, i.e., material balance, piece-square tables, and mobility, are useful in Arimaa, with the exception of the dog and horse piece-square tables. Also the partial mobility did not work.

For the extended evaluation the rabbit evaluation and the goal threats appeared to be useful in Arimaa, but trap control not.

The third research question was:

Can we apply and adapt relevant techniques, developed for computer-game playing, to the game Arimaa?

As described in chapter 4, many known techniques can be used for Arimaa. Compared to standard Alpha Beta search, PVS on the last step, transposition tables and killer move heuristic all cause a more or less substantial reduction in nodes searched.

Now that the research questions have been answered, we can also formulate an answer to the problem statement:

Can we build an efficient and effective program to play the game of Arimaa?

To build a program to play Arimaa, known search algorithms like Alpha Beta search, PVS on the last step, transposition tables and killer move heuristic can be used. Game knowledge in the form of a static evaluation function based on game strategies also has to be used.

A program with these features plays reasonably well against amateur human players. This at least is indicative for a positive answer to the problem statement.

7.2 Future Research Possibilities

The search algorithms and enhancements can be more fine-tuned. The null move is not used in the game, but probably when the search depth is large enough it will give better results. More research can be done with the history tables, a different method for upgrading or reducing the counts. They can be used on more steps besides the single steps, like the push and pull moves. And maybe other techniques that are not described here can be applied to Arimaa. Furthermore, it can be possible to create a more efficient or faster implementation of Arimaa.

The evaluator is still a crude estimate of the value of a board position. A more elaborate (and perhaps faster) evaluator might possibly be created. Since besides basic evaluation so far only the rabbit evaluation and the goal-threat strategy worked, it is likely that a more elaborate evaluation function can be created.

References

Akl, S.G. and Newborn, M.M. (1977). The Principal Continuation and the Killer Heuristic. *1977 ACM Annual Conference Proceedings*, pp. 466-473. ACM, Seattle.

Allis, L.V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ph. D. thesis, Rijksuniversiteit Limburg, Maastricht, The Netherlands.

Atkin, L.R. and Slate, D.J. (1977). CHESS 4.5 – The Northwestern University Chess Program. Frey P. W. (ed.), *Chess Skill in Man and Machine*, pp. 82-118. Springer-Verlag, New York.

Bernstein, A., Roberts, M. De V., Arbuckle, T. and Belsky, M.A. (1958). A Chess Playing Program for the IBM 704. *Proceedings of the Western Joint Computer Conference*, pp. 157-159

Breuker, D.M., Uiterwijk, J.W.H.M. and Herik, H.J. van den (1994). Replacement schemes for Transposition Tables. *ICCA Journal*, Vol. 17, No.4, pp. 183-193.

Breuker, D.M. (1998). *Memory versus Search in Games*. Ph. D. thesis, University of Maastricht, The Netherlands.

Chinchalkar, S. (1996). An Upper Bound for the Number of Reachable Positions. *ICCA Journal*, Vol. 19, No. 3, pp. 181-183.

Donninger, C. (1993). Null Move and Deep Search: Selective-Search Heuristics for Obtuse Chess Programs. *ICCA Journal*, Vol. 16, No.3, pp. 137-143.

Gillogly, J.J. (1972). The TECHNOLOGY Chess Program. *Artificial Intelligence*, vol. 3, pp. 145-163.

Gillogly, J.J. (1989). Transposition Table Collisitions. Workshop on New Directions on Game-Tree Search (pre prints) (ed. T.A. Marsland), p.12. Printing Services, University of Alberta, Edmonton.

Greenblatt, R.D., Eastlake, D.E. and Crocker, S. D. (1967). The Greenblatt Chess Program, *Proceedings of the Fall Joint Computer Conference*, pp. 801-810.

Herik, H.J. van den, Uiterwijk, J.W.H.M. and Rijswijck, J van (2002). Games Solved: Now and in the future. *Elsevier, Artificial Intelligence*, Vol. 134, pp. 277-311.

Huberman, B. J. (1968). *A Program to Play Chess End Games*. Ph. D. thesis, Stanford University, Computer Science Department, USA. Technical Report no.CS-106.

Hyatt, R.M., Gower, A. and Nelson, H. (1985). CRAY BLITZ. Beal, D. F. (ed.), *Advances in Computer Chess 4*, pp. 8-18. Pergamon Press, Oxford.

Hsu, F-h. (2004). *Behind Deep Blue: Building the computer that defeated the world Chess champion*. Princeton university press.

Kister, J., Stein, P., Ulam, S., Walden, W. and Wells, M. (1957). Experiments in Chess. *Journal of the ACM*, Vol. 4, pp. 174-177

Knuth, D.E. and Moore, R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293-326.

Kotok, A. (1962). A Chess Playing Program for the IBM 7090, B.S. Thesis, MIT. *Computer Chess Compendium*, pp. 48-55.

Marsland, T.A. (1986). A Review of Game-Tree Pruning. *ICCA Journal*. Vol. 9, No.1, pp.3-19

McCarthy, J. (1961) A Basic for a Mathematical Theory of Computation. *In Proc. Western Joint Computer Conference*, pp. 225-238.

Newell, A., Shaw, J.C. and Simon, H.A. (1958). Chess Playing Programs and the Problem of Complexity, *IBM Journal of Research and Development*, Oct 1958, pp. 320-335.

Samuel, A.L. (1959). Some studies in machine learning using the game of Checkers. *IBM Journal of Research and Development*, Vol. 3, No. 3, pp. 210-229.

Samuel, A.L. (1967). Some Studies in Machine Learning using the Game of Checkers. II-Recent Progress. *IBM Journal of Research and Development*, Vol. 2, No. 6, pp. 601-617.

Schaeffer, J. (1983). The History Heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16-19.

Schaeffer, J. (1997). *One Jump Ahead*. Springer-Verlag, New York.

Schaeffer, J. and Plaat, A. (1997). Kasparov versus Deep Blue: The Rematch. *ICCA Journal*, vol. 20, No. 2, pp. 95-101.

Shannon, C. (1950). Programming a Computer for playing Chess. *Philosophical Magazine*, vol. 41, pp. 256-275.

Syed, O. and Syed, A. (1999). Arimaa Official Homepage. http://www.Arimaa.com/Arimaa/.

Thompson, K. (1982). Computer Chess Strength. *Advances in Computer Chess 3*, Clarke, M.R.B. (ed.), pp. 55-56. Pergamon Press, Oxford.

Turing, A.M. (1953). Chess. Digital Computers Applied to Games. *Faster than Thought*, Bowden B.V. (ed.) pp. 286-310. Pitman

Uiterwijk, J.W.H.M. and Herik, H.J. van den (2000). The Advantage of the Initiative. *Information Sciences*, Vol. 122, No.1, pp. 43-58.

Zobrist, A.L. (1969). A Model of Visual Organisation for the Game of Go. *Proceedings of the Spring Joint Computer Conference 69*, pp 103-112.

Zobrist, A.L. (1970). A New Hashing Method with Application for Game Playing. Technical Report 88, Computer Science Department, The University of Wisconsin, Madison, WI, USA. Reprinted (1990) in *ICCA Journal*, Vol. 13, No. 2, pp.69-73.

APPENDIX A: Examples of time control in Arimaa

Example 1: 0/5 means 0 minutes per move with 5 minutes in reserve (per player). This is equivalent to G/5 in Chess; it means each player has a total of 5 minutes of time to play. If a player runs out of time before the game is over, the player loses. This is known as Blitz or "Sudden Death" time control in Chess.

Example 2: 0:12/5 means 12 seconds per move with 5 minutes in reserve and all of the unused time from each move is added to the reserve time. It is similar to "5 12" in Chess which means "Game in 5 minutes with a 12 second increment". After each move 12 seconds is added to the remaining time. This is known as Incremental time control in Chess.

Example 3: 3/0 means 3 minute per move and no reserve time, but 100 percent of the unused time for each move is added to the reserve. This guarantees that each player will make at least 40 moves in 2 hours. This is similar to the "40/2" Quota System time control used in Chess.

Example 4: 0:30/5/100/3 means 30 seconds per move with 5 minutes in reserve and 100% of the unused time from each move is added to the reserve time. When the reserve already exceeds the limit, more time is not added to it. When the reserve falls below 3 minutes more time can be added to it, but the reserve is capped at 3 minutes.

Example 5: 4/2/50/10/6 this means 4 minutes per move with a starting reserve of 2 minutes. After the move 50% of the time remaining for the move (rounded to the nearest second) is added to the reserve such that it does not exceed 10 minutes. There is a limit of 6 hours for the game after which time the game is halted and the winner is determined by score.

Example 6: 4/4/100/4/6 this means 4 minutes per move and a starting reserve of 4 minutes. 100% percent of the unused move time gets added to the reserve such that it does not exceed 4 minutes. There is a time limit of 6 hour for the game after which the winner is determined by score.

Example 7: 4/4/100/4/90t this is the same as above, but the game ends after both players have made 90 moves. Thus it ends after move 90 of Silver is completed.

Example 8: 4/4/100/4/90t/5 This is the same as above, but the players may not take more than 5 minutes for each turn even if there is still time remaining in reserve.

Different time units for any of the time control fields can be specified by adding one of the following letters after the numbers. In such cases the letter serves as the separator and : should not be used.
s - seconds
m - minutes
h - hours
d - days
For example: 24h5m10s/0/0/0/60d means 24 hours, 5 minutes and 10 seconds per move and the game must end after 60 days. Such a time control may be used in a postal type match.
The game time parameter (G) can also be specified in terms of maximum number of turns each player can make by adding the letter t after the number.

APPENDIX B: All unique patterns and their hits for each *n*

N	sorted unique pattern	# hits	$O(StateSpace\_n)$ only the unique pattern	$O(StateSpace\_n)$
0	1 1 1 1 2 2 2 2 2 2 8 8	1	4.63473E+42	4.63473E+42
1	0 1 1 1 2 2 2 2 2 2 8 8	4	5.61785E+41	4.49428E+42
	1 1 1 1 1 2 2 2 2 2 8 8	6	1.68536E+42	
	1 1 1 1 2 2 2 2 2 2 7 8	2	2.24714E+42	
2	1 1 1 1 2 2 2 2 2 2 7 7	1	2.64369E+41	2.30497E+42
	0 1 1 1 2 2 2 2 2 2 7 8	8	2.64369E+41	
	1 1 1 1 1 2 2 2 2 2 7 8	12	7.93108E+41	
	1 1 1 1 2 2 2 2 2 2 6 8	2	4.62647E+41	
	1 1 1 1 1 1 2 2 2 2 8 8	15	2.47846E+41	
	0 1 1 1 1 2 2 2 2 2 8 8	30	2.47846E+41	
	0 0 1 1 2 2 2 2 2 2 8 8	6	2.47846E+40	
3	0 0 0 1 2 2 2 2 2 2 8 8	4	4.72088E+38	8.31347E+41
	0 1 1 1 2 2 2 2 2 2 7 7	4	3.02136E+40	
	0 0 1 1 2 2 2 2 2 2 7 8	12	1.13301E+40	
	1 1 1 1 1 1 1 2 2 2 8 8	20	1.88835E+40	
	0 0 1 1 1 2 2 2 2 2 8 8	60	1.41626E+40	
	0 1 1 1 1 1 2 2 2 2 8 8	90	4.24879E+40	
	1 1 1 1 1 2 2 2 2 2 7 7	6	9.06409E+40	
	1 1 1 1 1 1 2 2 2 2 7 8	30	1.13301E+41	
	0 1 1 1 1 2 2 2 2 2 7 8	60	1.13301E+41	
	1 1 1 1 2 2 2 2 2 2 6 7	2	1.05748E+41	
	0 1 1 1 2 2 2 2 2 2 6 8	8	5.28739E+40	
	1 1 1 1 1 2 2 2 2 2 6 8	12	1.58622E+41	
	1 1 1 1 2 2 2 2 2 2 5 8	2	7.93108E+40	
4	1 1 1 1 2 2 2 2 2 2 6 6	1	1.0281E+40	2.35457E+41
	0 0 0 0 2 2 2 2 2 2 8 8	1	3.27839E+36	
	0 0 1 1 2 2 2 2 2 2 7 7	6	1.2589E+39	
	0 0 0 1 2 2 2 2 2 2 7 8	8	2.09817E+38	
	0 1 1 1 2 2 2 2 2 2 6 7	8	1.17498E+40	
	1 1 1 1 1 1 2 2 2 2 7 7	15	1.2589E+40	
	0 1 1 1 1 2 2 2 2 2 7 7	30	1.2589E+40	
	1 1 1 1 1 2 2 2 2 2 6 7	12	3.52493E+40	
	1 1 1 1 2 2 2 2 2 2 5 7	2	1.76246E+40	
	0 1 1 1 2 2 2 2 2 2 5 8	8	8.81231E+39	
	1 1 1 1 1 2 2 2 2 2 5 8	12	2.64369E+40	
	1 1 1 1 2 2 2 2 2 2 4 8	2	1.10154E+40	
	1 1 1 1 1 1 2 2 2 2 6 8	30	2.20308E+40	
	0 1 1 1 1 2 2 2 2 2 6 8	60	2.20308E+40	
	0 0 1 1 2 2 2 2 2 2 6 8	12	2.20308E+39	