Arimaa: from rules to bitboard analysys

Knowledge Representation Thesis

of

Stefano Carlini

December 2008

Supervisor:
Prof. S. Bergamaschi

University of Modena and Reggio Emilia

Abstract

This paper will go through the Arimaa rules analizing why is so difficult to develop a computer program able to defeat strong human players. After an overview of the bitboards and their technical advantages, the report will deeply detail how to apply this data structure according to the rules of Arimaa.

Contents

List of Figures

Chapter 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. Computers are thus being used as a tool for people to help them in their quest for excellence [2].

In this chapter we will discover Arimaa, a game "easy" for the human players, but "hard" for the computers [6].

1.1 Introduction to Arimaa

When playing strategy games such as Chess, the computer is actually exploring all the move combinations to look ahead as far as possible, so that it can pick out the move that leads to the most favorable position. This brute-force approach of examining each move as deep as possible is quite different than the way used by humans. Humans typically do little search, but use lots of knowledge. This was taken to the extreme in the 1997 Deep Blue versus Kasparov chess match: man 2 positions per sec; machine 200,000,000 positions per second [4].

Omar Syed, a former NASA computer engineer, believed that even though brute-force

searching computers had made such impressive achievements, they were still even not close to matching the kind of real intelligence used by humans in playing strategy games. To prove his point, he and his son designed a 2-player game called Arimaa in 1997, which was intentionally made "easy" for the human players, but "hard" for the computers.

This objective was reached by having a very large set of legal moves in a position (the so-called branching factor). Since the size of the search tree is based on b^d (b for branching factor and d for search depth), the large branching factor of Arimaa results in the search tree size growing rapidly, making a deep search impractical even on the most advanced computer. By contrast, for the human players, the branching factor does not influence the difficulty of the game as much. The challenge is to build a computer program capable of playing a strong game of Arimaa [6].

1.2 Rules of Arimaa

This section describes the rules of Arimaa, but there is also a visual flash tutorial in Arimaa web page (`http://www.arimaa.com/arimaa/learn/flash/vs/new/`). I suggest to look there instead of read this section (1.2) and continue to the next one (1.3) in page 7.

Arimaa is a game for two players, Gold and Silver, played on an 8x8 board with a standard chess set. 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. The elephant is the strongest followed by camel, horse, dog, cat and rabbit.

The game starts with an empty board. The player with the gold pieces sets them on the first two rows closest to that player. There is no fixed starting position so the pieces may be placed in any arrangement. But it is suggested that most of the stronger pieces be placed in front of the weaker rabbits. Once the gold player has finished the silver player sets the pieces on the two closest rows. Again the pieces may be placed in any arrangement within the first two rows (see Figure 1.1) [6].

The goal of this game is to get any one of the 8 weakest pieces (Rabbit) across the board to the other side. All pieces have the same mobility: they can move forward, backward, left and right. The Rabbit (weakest piece) cannot move backwards. 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.

The stronger pieces can move opponent's weaker pieces. For example your dog can

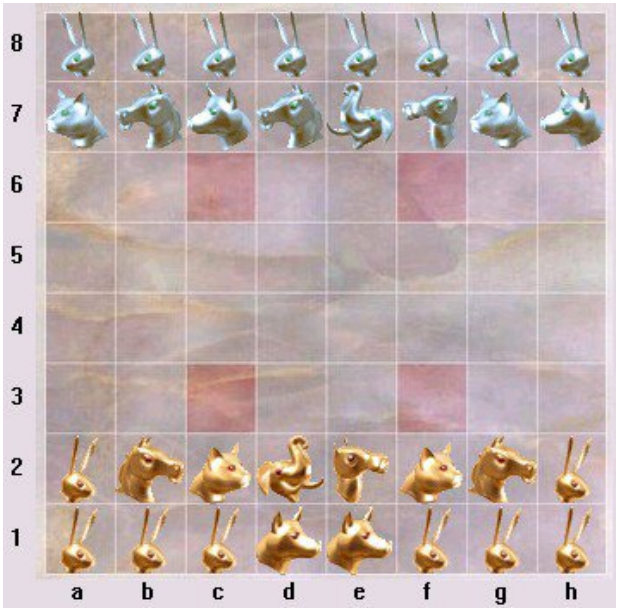

Figure 1.1: Setting Up

move the opponent's cat or rabbit, but not the opponent's dog or any other piece that is stronger than it. 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 squares and then move your piece into its place. To pull an opponent's piece with your stronger piece, first move your piece to one of the unoccupied adjacent squares and then move the opponent's piece into the square that was just vacated. 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. However if there is a friendly piece next to it the piece is unfrozen and is free to move.

There are four distinctly marked traps squares on the board (c3, f3, c6 and f6 in standard notation). 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. Be careful not to lose your own pieces in the traps [6].

1.3 Challenge

The main reason why this game is difficult for a computer is the large branching factor. Compared to an average of about 35 moves in a typical Chess position, or roughly 200 in

a Go position, a player has to choose between 2,000 to 3,000 moves for their first move in Arimaa, and about 5,000 to 40,000 moves in the mid-game. 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 was 5 times faster than Deep Blue and could evaluate a billion positions per second it would still take it more than 5 years to explore all those positions [6].

There are 64 million different possible initial arrangements, so it is almost impossible for a computer to use pre-computed databases of opening analysis [1].

Another important reason why Arimaa is difficult compared to Chess is that it is a challenge to build a function that can assess who has the better Arimaa position. For the computer, spotting tactics is easier than evaluating a materially equivalent position. Arimaa is much more of a positional game and has much less tactics than Chess [10].

Compared to the capture rule in Chess, pushing, pulling, freezing and trapping in Arimaa is more difficult for the computer to handle (see Figure 1.2). Some of the successful heuristics used in games like Chess do not work well in this game [6].

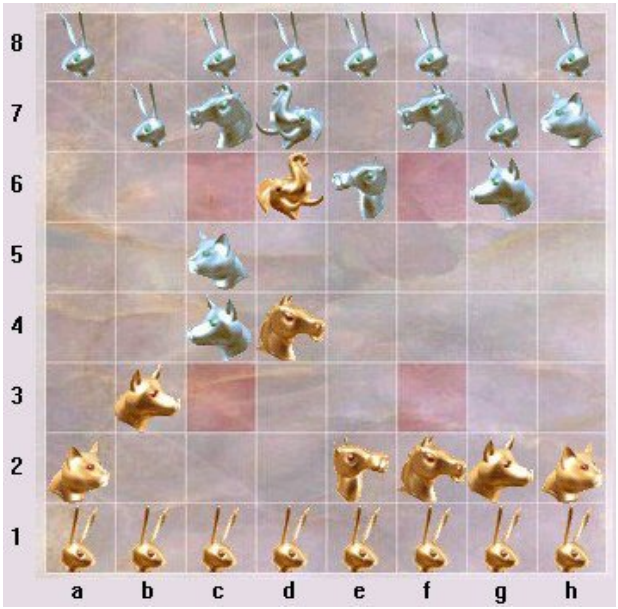

Figure 1.2: A Typical Situation

For those reasons 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. The official challenge match will be between the current best program and a top-ten-rated human player [2].

1.4 Notation

For reviewing games, the games have to be recorded. To do this recording we will use the notation for recording the Arimaa games(1.4.1) and positions(1.4.2) that is also used on the internet. We will outline these notations in those subsection [6].

1.4.1 Notation for Recording Arimaa Games

The first player to move has the gold colored pieces. The second player to move has the silver colored pieces.

The pieces are indicated using upper or lower case letters to specify the piece color 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 silver) or M (for gold) is used.

Each square on the board is indicated by the column and row. The lower case letters a to h are used to indicate the column 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 players move is recorded on a separate line. The line starts with the move number followed by the color of the side making the move. For example 3g means move 3 for gold; this would be followed by 3s which is move 3 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 move 3w 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 placed for the players move. If a player

loses because the opponents weakest piece has reached their first row then the word 'lost' is placed for the players move.

If a move is taken back the word 'takeback' is placed for the move 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:

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

Figure 1.3: Sample Position Image

1.4.2 Notation for Recording Arimaa Positions

The gold color pieces are shown in upper case letters and silver color 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 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 labeled and the board must be framed with - and | 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. But marking the trap squares is optional and not required.

Here is a sample position file equavalent to Figure 1.3 :

```
 +-----------------+
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
```

Chapter 2

Bitboard

A bitboard, often used for boardgames such as chess, checkers and othello, is a specialization of the bitset data structure, where each bit represents a game position or state, designed for optimization of speed and/or memory or disk use in mass calculations. Bits in the same bitboard relate to each other in the rules of the game often forming a game position when taken together. Other bitboards are commonly used as masks to transform or answer queries about positions. The "game" may be any game-like system where information is tightly packed in a structured form with "rules" affecting how the individual units or pieces relate [8].

In this chapter we will see why and how we adopted a 64-bit bitboard to represent the Arimaa board. This technique is actually used by *Bomb* developed by *David Fotland* that is currently the best Arimaa program, winner of every edition of Arimaa's Computer Championship (from 2004 to 2008) [3] but still unable to beat the strong human players.

2.1 Bitwise Operation

A *Bitwise Operation* operates on one or two binary numbers at the level of their individual bits. On most microprocessors, bitwise operations are sometimes slightly faster than addition and subtraction operations and usually significantly faster than multiplication and division operations [9].

2.1.1 Binary Number

A binary number can be represented by any sequence of bits (binary digits), which in turn may be represented by any mechanism capable of being in two mutually exclusive states.

The following sequences of symbols could all be interpreted as the same binary numeric value of 667:

```
1 0 1 0 0 1 1 0 1 1
| - | - - | | - | |
x o x o o x x o x x
y n y n n y y n y y
```

When spoken, binary numerals are usually read digit-by-digit, in order to distinguish them from decimal numbers. For example, the binary numeral 100 is pronounced *one zero zero*, rather than *one hundred*, to make its binary nature explicit, and for purposes of correctness. Since the binary numeral 100 is equal to the decimal value four, it would be confusing, and numerically incorrect, to refer to the numeral as *one hundred* [7].

2.1.2 NOT

The *bitwise NOT*, is a unary operation that performs logical negation on each bit, forming the ones' complement of the given binary value. Digits which were 0 become 1, and vice versa. For example:

```
NOT 0111
  = 1000
```

In many programming languages (including those in the C family), the bitwise NOT operator is '~' (tilde). This operator must not be confused with the 'logical not' operator, '!' (exclamation point), which treats the entire value as a single Boolean — changing a true value to false, and vice versa. The 'logical not' is not a bitwise operation.

From now on we will use ~ to indicate a logical NOT.

2.1.3 OR

A *bitwise OR* takes two bit patterns of equal length, and produces another one of the same length by matching up corresponding bits (the first of each; the second of each; and so on) and performing the logical inclusive OR operation on each pair of corresponding bits. In each pair, the result is 1 if the first bit is 1 OR the second bit is 1 (or both), and otherwise the result is 0. For example:

```
   0101
OR 0011
 = 0111
```

In the C programming language family, the bitwise OR operator is '|' (pipe). Again, this operator must not be confused with its Boolean 'logical or' counterpart, which treats its operands as Boolean values, and is written '||' (two pipes).

From now on we will use | to indicate a logical OR.

2.1.4 AND

A *bitwise AND* takes two binary representations of equal length and performs the logical AND operation on each pair of corresponding bits. In each pair, the result is 1 if the first bit is 1 AND the second bit is 1. Otherwise, the result is 0. For example:

```
    0101
AND 0011
  = 0001
```

In the C programming language family, the bitwise AND operator is '&' (ampersand). Again, this operator must not be confused with its Boolean 'logical and' counterpart, which treats its operands as Boolean values, and is written '&&' (two ampersands).

From now on we will use & to indicate a logical AND.

2.1.5 XOR

A *bitwise eXclusive OR* takes two bit patterns of equal length and performs the logical XOR operation on each pair of corresponding bits. The result in each position is 1 if the two bits are different, and 0 if they are the same. For example:

```
    0101
XOR 0011
  = 0110
```

In the C programming language family, the bitwise XOR operator is '^' (caret), so from now on we will use ^to indicate a logical XOR.

2.1.6 Bit Shifts

In a *Logical Shift*, the digits are moved, or *shifted*, to the left or right. The bits that are shifted out of either end are discarded, and zeros are shifted in (on either end). This example uses an 8-bit register:

```
   00010111 LEFT-SHIFT
=  00101110

   00010111 RIGHT-SHIFT
=  00001011
```

In C-inspired languages, the left and right logical shift operators are '<<' and '>>>', respectively and the number of places to shift is given as the second argument to the shift operators.

From now on we will use << k and >>> k to indicate a left and right logical shift by k values.

2.2 Introduction to Bitboard

Chess used to be a hot research area in computer science during the height of the Cold War. Apparently, independently of one another, both Soviet and American teams came up with a new data structure called a bitboard. The American team, Slate and Atkin, seem to have been first to print with a chapter in Chess Skill in Man and Machine on Chess 4.x. The Soviet team, including Mikhail Donskoy, among others, wrote a bitboard-enabled program called Kaissa. Both programs competed successfully internationally [5].

Before looking at bitboards, let's first look at the standard way to represent a chess board in Java (and many other languages).

```
//declare an integer enum for the possible
//state of the 64 individual squares.

final int EMPTY        = 0;
final int WHITE_PAWN   = 1;
final int WHITE_KNIGHT = 2;
final int WHITE_BISHOP = 3;
final int WHITE_ROOK   = 4;
final int WHITE_QUEEN  = 5;
```

```
final int WHITE_KING   = 6;
final int BLACK_PAWN   = 7;
final int BLACK_KNIGHT = 8;
final int BLACK_BISHOP = 9;
final int BLACK_ROOK   = 10;
final int BLACK_QUEEN  = 11;
final int BLACK_KING   = 12;

//And we have an array of 64 squares.

int[] board= new int[64];
```

The array method is extremely straightforward. In contrast, the bitboard structure is made up of twelve 64-bit bitsets; one for each type of piece. They are visualized as being stacked on top of each other (see Figure 2.1).

```
//Declare twelve 64-bit integers for one board:
long WP, WN, WB, WR, WQ, WK, BP, BN, BB, BR, BQ, BK;
```

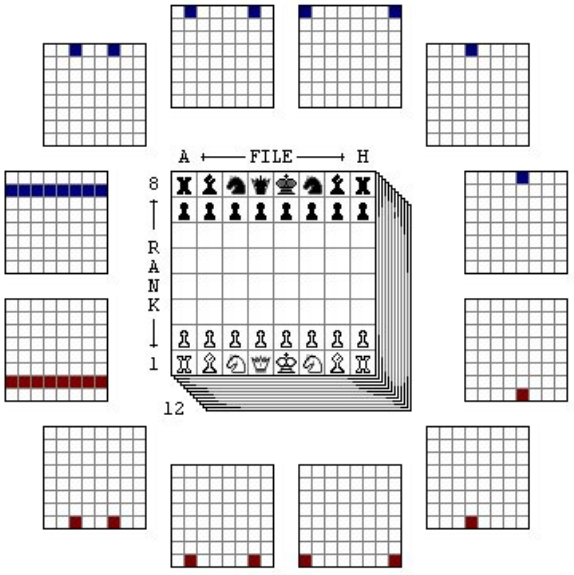

Figure 2.1: the Bitboard Data Structure

Since EMPTY is a function of the other twelve layers, including it would create redundant data. To calculate EMPTY, we just join the 12 layers and negate:

```
long NOT_EMPTY = WP | WN | WB | WR | WQ | WK |
                 BP | BN | BB | BR | BQ | BK ;
long EMPTY = ~NOT_EMPTY;
```

2.3 General Technical Advantages and Disadvantages

2.3.1 Processor Use

Pros

The advantage of the bitboard representation is that it takes advantage of the essential logical bitwise operations available on nearly all CPUs that complete in one cycle and are full pipelined and cached etc. Nearly all CPUs have AND, OR, NOR, and XOR. Many CPUs have additional bit instructions, such as finding the 'first' bit, that make bitboard operations even more efficient. If they do not have instructions well known algorithms can perform some 'magic' transformations that do these quickly [8].

Furthermore, modern CPUs have instruction pipelines that queue instructions for execution. A processor with multiple execution units can perform more than one instruction per cycle if more than one instruction is available in the pipeline. Branching (the use of conditionals like if) makes it harder for the processor to fill its pipeline(s) because the CPU can't tell what it needs to do in advance. Too much branching makes the pipeline less effective and potentially reduces the number of instructions the processor can execute per cycle. Many bitboard operations require fewer conditionals and therefore increase pipelining and make effective use of multiple execution units on many CPUs.

CPUs have a bit width which they are designed toward and can carry out bitwise operations in one cycle in this width. So, on a 64-bit or more CPU, 64-bit operations can occur in one instruction. There may be support for higher or lower width instructions. Many 32-bit CPUs may have some 64-bit instructions and those may take more than one cycle or otherwise be handicapped compared to their 32-bit instructions.

If the bitboard is larger than the width of the instruction set, then a performance hit will be the result. So a program using 64-bit bitboards would run faster on a real 64-bit processor than on a 32-bit processor.

Cons

Some queries are going to take longer than they would with perhaps arrays, but bitboards are generally used in conjunction with array boards in chess programs.

2.3.2 Memory Use

Pros

Bitboards are extremely compact. Since only a very small amount of memory is required to represent a position or a mask, more positions can find their way into registers, full speed cache, Level 2 cache, etc. In this way, compactness translates into better performance (on most machines anyway). Also on some machines this might mean that more positions can be stored in main memory before going to disk.

Cons

For some games writing a suitable bitboard engine requires a fair amount of source code that will be longer than the straight forward implementation. For limited devices (like cell phones) with a limited number of registers or processor instruction cache, this can cause a problem. For full sized computers it may cause cache misses between level one and level two cache. This is a potential problem–not a major drawback. Most machines will have enough instruction cache so that this isn't an issue.

2.3.3 Source Code

Bitboard source code is very dense and sometimes hard to read. It must be documented very well.

Chapter 3

Bitboards in Arimaa

3.1 Representation Examples

To represent a bitboard layer we can use one of those methods:

- The Figure:

Figure 3.1: Gold Elephant Layer (Ed2)

- The 64-bit array, formatted in to 8 rows of 8 bits each, like a chessboard:

```
long GE; // Gold Elephant Position
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0
```

- The Arimaa notation (see Chapter 1.4.2):

```
long GE; // Gold Elephant Position
 +-----------------+
8|                 |
7|                 |
6|     x     x     |
5|                 |
4|                 |
3|     x     x     |
2|       E         |
1|                 |
 +-----------------+
   a b c d e d g h
```

3.2 Bitboard Problem

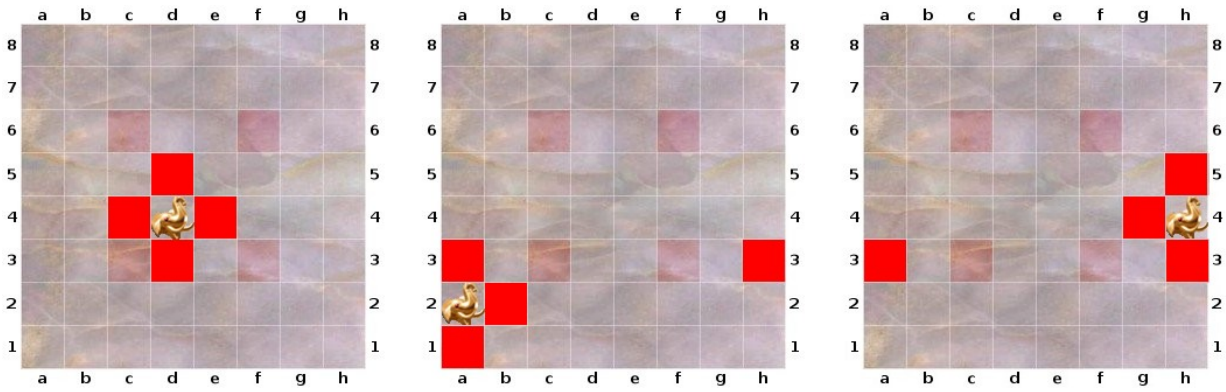

Figure 3.2: Finding Gold Elephant steps

In Figure 3.2 we see three representation of the Gold Elephant (E) possible steps bitboard, obtained shifting the E bitboard table with a Bit Shift:

```
long GE; // the Gold Elephant bitboard

long GE_step = // the bitobard elephant step
   (GE <<  1) | // the East square
   (GE <<  8) | // the North square
   (GE >>> 1) | // the West square
   (GE >>> 8) ; // the South square
```

In second and third images of Figure 3.2 we see an undesired step of the Gold Elephant; in fact we formatted the 64-bit array in 8 rows, so we have to modify the step algorithm. If we shift a layer visually left (right) by one, we know that the rightmost (leftmost) file, file H (file A), should always be zeros [5]. We needA vertical row is called a file, let's code that:

```
final long FILE_H =
   1L        | (1L << 8)  | (1L << 16) | (1L << 24) |
  (1L << 32) | (1L << 40) | (1L << 48) | (1L << 56);

final long FILE_A =
  (1L << 7)  | (1L << 15) | (1L << 23) | (1L << 31) |
  (1L << 39) | (1L << 47) | (1L << 55) | (1L << 63);
```

```
long FILE_H;             long FILE_A;
 +-----------------+      +-----------------+
8|               * |     8| *               |
7|               * |     7| *               |
6|     x     x   * |     6| *   x     x     |
5|               * |     5| *               |
4|               * |     4| *               |
3|     x     x   * |     3| *   x     x     |
2|               * |     2| *               |
1|               * |     1| *               |
 +-----------------+      +-----------------+
   a b c d e d g h          a b c d e d g h
```

```
long near(long bitboard){
   long tempNear =
     ((bitboard <<  1) & ~FILE_H) | // East
      (bitboard <<  8)            | // North
     ((bitboard >>> 1) & ~FILE_A) | // West
      (bitboard >>> 8)            ; // South

   return tempNear;
}
```

We didn't define any restriction to the Northern and Southern square because it's not necessary. This algorithm works very good also with bitboards with more than one piece; in the following example we see a Gold Cat bitboard and how works the function **near**.

```
long GC; // Gold Cat
 +-----------------+
8|                 |
7|                 |
6|     x     x     |
5|                 |
4|                 |
3|     x     x   C |
2|                 |
1|   C             |
 +-----------------+
   a b c d e d g h
```

```
long near(GC); // correct squares near Gold Cat
 +-----------------+
8|                 |
7|                 |
6|     x     x     |
5|                 |
4|               * |
3|     x     x *   |
2|   *           * |
1| *   *           |
 +-----------------+
   a b c d e d g h
```

3.3 Freeze

In chapter 1.2 we saw an important Arimaa Rule:

> 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. However if there is a friendly piece next to it the piece is unfrozen and is free to move.

A rule easy to understand for a human, but difficult for a computer. With bitboards we write it as follows:

```
long NOT_EMPTY = GE | GM | GH | GD | GC | GR |
                 SE | SM | SH | SD | SC | SR ;
long EMPTY = ~NOT_EMPTY; // All the empty spaces in the board

// bitboard where a piece can legally move
long move(long bitboard){
   long not_frozen = bitboard & ~frozen(bitboard);
   long tempMove = EMPTY & near(not_frozen);

   return tempMove;
}
```

Where the moving action is limited by being frozen or not. We already seen the **near** function, but in rabbit pieces it will be replaced by another function (the rabbit can't move backwards). Let's see how the **frozen** function works for a Gold Cat.

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

long GOLD = GE | GM | GH | GD | GC | GR ;
long STRONG_SILVER = SE | SM | SH | SD ;

// the GC with a stronger enemy near them
long strongEnemyNear(long bitboard){
   long tempStrongEnemyNear = near(STRONG_SILVER) & bitboard;

   return tempStrongEnemyNear;
}
```

```
long STRONG_SILVER;
 +-----------------+
8|                 |
7|                 |
6| *   x     x     |
5|                 |
4|     *     *     |
3|     x     x     |
2|   *             |
1|                 |
 +-----------------+
   a b c d e d g h
```

```
long strongEnemyNear(GC);
 +-----------------+
8|                 |
7|                 |
6|     x     x     |
5|                 |
4|                 |
3|     x   * x     |
2|                 |
1|   *             |
 +-----------------+
   a b c d e d g h
```

```
// the GC with a friend near them
long friendNear(long bitboard){
   long tempFriendNear = near(GOLD) & bitboard;

   return tempFriendNear;
}
```

```
long GOLD;
 +-----------------+
8|                 |
7|                 |
6|     x     x   * |
5| *               |
4|                 |
3|     x   * x     |
2|             *   |
1|   * *     * *   |
 +-----------------+
   a b c d e d g h
```

```
long friendNear(GC);
 +-----------------+
8|                 |
7|                 |
6|     x     x     |
5|                 |
4|                 |
3|     x     x     |
2|                 |
1|   *             |
 +-----------------+
   a b c d e d g h
```

```
// the GC frozen
long frozen(long bitboard){
   long tempFrozen = strongEnemyNear(bitboard) & ~friendNear(bitboard);

   return tempFrozen;
}
```

```
long ~friendNear(GC);
 +-----------------+
8| * * * * * * * * |
7| * * * * * * * * |
6| * * * * * * * * |
5| * * * * * * * * |
4| * * * * * * * * |
3| * * * * * * * * |
2| * * * * * * * * |
1| *   * * * * * * |
 +-----------------+
   a b c d e d g h
```

```
long frozen(GC);
 +-----------------+
8|                 |
7|                 |
6|     x     x     |
5|                 |
4|                 |
3|     x   * x     |
2|                 |
1|                 |
 +-----------------+
   a b c d e d g h
```

```
long not_frozen(GC);
 +-----------------+
8|                 |
7|                 |
6|     x     x     |
5|                 |
4|                 |
3|     x     x     |
2|                 |
1|   *             |
 +-----------------+
   a b c d e d g h
```

```
long EMPTY;
 +-----------------+
8| * * * * *   * * |
7|   * * * * * * * |
6|   * * * * * *   |
5|   * * * * * * * |
4| * *   *   * * * |
3| * * * *   * * * |
2| *   * * * *   * |
1| *     * *     * |
 +-----------------+
   a b c d e d g h
```

```
long move(GC);
 +-----------------+
8|                 |
7|                 |
6|     x     x     |
5|                 |
4|                 |
3|     x     x     |
2|                 |
1| *               |
 +-----------------+
   a b c d e d g h
```

From this long example we see how we can easily obtain all the legal steps for a piece represented with a bitboard. The code is slightly different from piece to piece except for the Elephant, which can never become *frozen* because there aren't stronger pieces.

```
// bitboard where an Elephant can legally move
long move(long bitboard){
   long tempMove = EMPTY & near(bitboard);

   return tempMove;
}
```

3.4 Pushing and Pulling

The stronger pieces can move opponent's weaker pieces. For example your dog can move the opponent's cat or rabbit, but not the opponent's dog or any

other piece that is stronger than it. 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 squares and then move your piece into its place. To pull an opponent's piece with your stronger piece, first move your piece to one of the unoccupied adjacent squares and then move the opponent's piece into the square that was just vacated. A push or pull requires two steps and must be completed within the same turn.

A push or pull move can be easily translated in bitboard terms. We have only to define another type of move, that cost 2 steps instead of one ($step^2$), and apply the rules given above. This move is again dependent by the piece that wants to perform it, where a rabbit will never push or pull another piece.

```
long move2(long bitboard){
   long tempMove2 = push(bitboard) | pull(bitboard);

   return tempMove2;
}
```

Let's see how **push** and **pull** functions work for a Gold Horse. A weaker piece need a free space near itself to be pushed away (**pushFull**). Every weaker piece can be pulled away (**pullFull**) until the pulling piece is not frozen.

```
// Board situation
 +-----------------+
8|   c       r     |
7| r   r           |
6| d   x     x   d |
5| M             H |
4|     h   m       |
3|     x r H x     |
2|   e         R   |
1|   C E     R C   |
 +-----------------+
   a b c d e d g h

long GOLD = GE | GM | GH | GD | GC | GR ;
long WEAK_SILVER = SD | SC | SR ;

// the weaker pieces that can be pushed away
long pushFull(){
```

```
   long tempPushFull = near(near(WEAK_SILVER) & EMPTY) & WEAK_SILVER;

   return tempPushFull;
}

long push(long bitboard){
   long tempPush = near(not_frozen(bitboard)) & pushFull();

   return tempPush;
}
```

```
long pushFull();          long push(GH);
 +-----------------+       +-----------------+
8|   *       *     |      8|                 |
7| *   *           |      7|                 |
6| *   x     x   * |      6|     x     x   * |
5|                 |      5|                 |
4|                 |      4|                 |
3|     x *   x     |      3|     x     x     |
2|                 |      2|                 |
1|                 |      1|                 |
 +-----------------+       +-----------------+
   a b c d e d g h           a b c d e d g h
```

```
// the weaker pieces that can be pulled away
long pullFull(long bitboard){
   long tempPullFull = near(not_frozen(bitboard)) & WEAK_SILVER;

   return tempPullFull;
}

long pull(long bitboard){
   long tempPull = near(not_frozen(bitboard) & pullFull(bitboard))
      & EMPTY;

   return tempPull;
}
```

```
long pullFull(GH);
 +-----------------+
8|   *       *     |
7| *   *           |
6| *   x     x   * |
5|                 |
4|                 |
3|     x     x     |
2|                 |
1|                 |
 +-----------------+
   a b c d e d g h
```

```
long pull(GH);
 +-----------------+
8|                 |
7|                 |
6|     x     x     |
5|             *   |
4|               * |
3|     x     x     |
2|                 |
1|                 |
 +-----------------+
   a b c d e d g h
```

We should notice that the $step^2$ movement generated bitboard is relative only to the gold piece, and don't show where a silver piece could be pushed/pulled. While when pulling we know exactly where the piece will be pulled, to know the right moving positions bitboard of a pushed piece we need another function:

```
long pushed(long bitboard){
   long tempPushed = near(push(bitboard)) & EMPTY;
}
```

```
long push(GH);
 +-----------------+
8|                 |
7|                 |
6|     x     x   * |
5|                 |
4|                 |
3|     x     x     |
2|                 |
1|                 |
 +-----------------+
   a b c d e d g h
```

```
long pushed(GH);
 +-----------------+
8|                 |
7|               * |
6|     x     x *   |
5|                 |
4|                 |
3|     x     x     |
2|                 |
1|                 |
 +-----------------+
   a b c d e d g h
```

Chapter 4

Conclusions

In fine we have learned how to play arimaa and why it's so difficult to build a strong computer player. We also learned how we could represent the pieces positions and correct movents positions with bitboards.

Using techniques like alpha-beta pruning and bitboards we can reduce the gap between man and machines.

The bitboards were initially developed for chess, but them are even more efficient for Arimaa thanks to the restricted set of moves of Arimaa pieces.

Considering that only a very small amount of memory is required to represent a board with bitboards (96 byte), their use for data representation can help the data mining, being easier to manage huge set of boards, facilitating machine learning techniques.

Bibliography

[1] M. Buro. Toward opening book learning. *ICGA Journal*, 22(2):98–102, 1999.

[2] C.-J. Cox. Analysis and implementation of the game arimaa. Master's thesis, University of Maastricht, 03 2006.

[3] D. Fotland. Building a world-champion arimaa program. *Lecture Notes in Computer Science*, 3846:175–186, 2006.

[4] F. Hsu. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, 2002.

[5] G. Pepicelli. *Bitwise Optimization in Java: Bitfields, Bitboards, and Beyond*. O'Reilly, 2005. `http://www.onjava.com/pub/a/onjava/2005/02/02/bitsets.html`.

[6] O. Syed and A. Syed. Arimaa: A new game designed to be difficult for computers. *ICGA Journal*, 26(2):138–139, 2003. `http://www.arimaa.com/arimaa/`.

[7] Wikipedia. Binary numeral system. `http://en.wikipedia.org/wiki/Binary_numeral_system`.

[8] Wikipedia. Bitboard. `http://en.wikipedia.org/wiki/Bitboard`.

[9] Wikipedia. Bitwise operation. `http://en.wikipedia.org/wiki/Bitwise_operation`.

[10] H. Zhong. Building a strong arimaa-playing program. Master's thesis, University of Alberta, 2005.