Universita' degli Studi di Modena e Reggio Emilia
Facolta' di Ingegneria di Modena

Corso di Laurea Specialistica in Ingegneria Informatica

Arimaa: Una Nuova Sfida per l'Intelligenza Artificiale

Relatore:
Chiar.mo Prof. Sonia Bergamaschi

Correlatore:
Dott.ssa Serena Sorrentino

Candidato:
Stefano Carlini

Anno Accademico 2008/2009

Sintesi

Questa tesi rappresenta il frutto di un lungo lavoro, cominciato nell'ottobre del 2008. Lo scopo della tesi e' lo studio di tecniche e strumenti dell'Intelligenza Artificiale (IA), nel campo dei **Giochi da Tavolo** [11], una tipologia di giochi che include anche Scacchi, Go, Forza Quattro, e Tris.

Il gioco a cui ci siamo dedicati si chiama **Arimaa**, ed e' relativamente nuovo; questo gioco da tavolo e' stato ideato da Omar Syed nel 2003, ispirato dal match di Deep Blue[1] contro Gary Kasparov. Arimaa e' stato creato in modo da essere giocato su una scacchiera standard, facile da imparare per gli esseri umani, ma difficile per i computer.

Gli **obbiettivi** della tesi sono: (1) studiare lo *stato dell'arte* nel campo dell'IA per giochi da tavolo; (2) realizzare un motore per Arimaa, cioe' un sistema in grado di rappresentare una situazione di gioco e generare le possibili mosse dei giocatori, nel modo piu' veloce ed efficiente possibile.

Arimaa e' un gioco da tavolo per due giocatori, ideato, come abbiamo detto, per essere difficile per i computer. Ogni anno dal 2004, viene organizzato un campionato, in cui i migliori giocatori 'umani' affrontano i migliori programmi per computer; fino ad oggi, nessun programma e' riuscito a battere un giocatore umano in una partita alla pari.

Una partita di arimaa comincia con una scacchiera vuota, contenente 8x8 caselle. Esistono sei tipi di pezzi, ognuno di essi piu' forte o piu' debole dell'altro; dal piu' forte al piu' debole abbiamo: (1) un **Elefante**; (2) un **Cammello**; (3) due **Cavalli**; (4) due **Cani**; (5) due **Gatti**; (6) otto **Conigli**. Il primo giocatore (color Oro), comincia la partita disponendo i suoi sedici pezzi, a suo piacimento, nelle prime due righe della scacchiera; dopodiche', il secondo giocatore (color Argento) dispone i propi pezzi nelle due ultime righe della scacchiera. L'obiettivo del gioco e' portare uno dei propri Conigli nell'ultima riga della parte opposta della scacchiera. Tutti i pezzi si muovono nello stesso modo, cioe' avanti, indietro, a sinistra, o a destra e di un solo passo, in una casella vuota; l'unica eccezzione e' il Coniglio, che non puo' muoversi all'indietro. Durante ogni fase, un giocatore e' obbligato a fare almeno un **passo** (spostare un pezzo dalla sua posizione in una casella adiacente). Un giocatore puo' effettuare fino a quattro passi per turno; questi possono essere effettuati con un unico pezzo (che puo' anche cambiare direzione dopo ogni passo) o con pezzi diversi. Un pezzo piu'

[1] Deep Blue era un computer prodotto dall'IBM progettato per giocare a scacchi.Deep Blue è stato il primo computer a vincere una partita a scacchi contro un Campione del Mondo in carica, (Garry Kasparov) [23].

forte puo' spostare i pezzi avversari piu' deboli: per esempio un Cavallo puo' spostare un Cane, un Gatto, o un Coniglio, ma non puo' spostare un Elefante, un Cammello, o un altro Cavallo. Per spostare un pezzo avversario, e' necessario che questo si trovi su una casella ortogonalmente adiacente al pezzo che eseguira' lo spostamento; il pezzo nemico puo' essere **spinto** o **tirato**. Per spingere un pezzo nemico e' necessario prima spostarlo in una casella vuota ad esso adiacente, per poi muovere il pezzo 'spingente' nella casella precedentemente occupata dal pezzo spostato. Per tirare un pezzo nemico, e' necessario muovere prima il pezzo 'tirante' in una casella vuota ad esso adiacente, per poi muovere il pezzo nemico nella casella precedentemente occupata dal pezzo tirante. Spingere o tirare un pezzo nemico 'costa' due passi. E' vietato spingere un pezzo mentre si tira un altro pezzo. Quando un pezzo piu' debole si trova accanto ad un pezzo nemico piu' forte, questo si considera **congelato**, e non puo' essere mosso a meno che sia presente un pezzo alleato accanto ad esso. Un pezzo congelato puo' ancora essere spinto o tirato. Nella scacchiera sono presenti quattro trappole, posizionate nelle casselle C3, F3, C6 ed F6; se un pezzo giace su una su una trappola, e non ci sono pezzi alleati accanto ad esso, questo viene rimosso dal gioco. E' importante sottolineare che questo e' l'unico modo per catturare pezzi dell'avversario.

Per realizzare un programma in grado di giocare ad Arimaa, e' necessario occuparsi di una particolare branca dell'IA nota come **Problem Solving**. Un problema puo' essere definito da quattro componenti: (1) *lo stato iniziale*; (2) *una funzione di successione*, che restituisce i possibili stati successivi, tramite una coppia di valori (azione, stato successore); (3) *un test di goal*, che determina se uno stato e' lo stato finale; (4) *una funzione di costo*, che assegna un valore numerico per raggiungere uno stato successivo. Per effettuare una ricerca, e' utile rappresentare lo spazio degli stati (state space) tramite un albero di ricerca. Ogni nodo dell'albero rappresenta una struttura dati contenente cinque componenti: (1) *Stato*, lo stato corrente del nodo; (2) *Stato Genitore*, il nodo nell'albero di ricerca, che ha generato il nodo corrente; (3) *Azione*, l'azione applicata al genitore per ottenere lo stato corrente; (4) *Costo*; (5) *Profondita'* (Depth), cioe' il numero di passi effettuati, a partire dallo stato iniziale, per raggiungere lo stato corrente.

Per usare correttamente un albero degli stati, dobbiamo fare tre passi, partendo dallo stato iniziale: (1) controllare se lo stato corrente e' lo stato finale; (2) espandere lo stato corrente, applicando la funzione di successione; (3) scegliere uno degli stati non ancora espansi, e ripartire dal punto 1. La scelta di quale stato espandere e' determinato da una **strategia di ricerca**; le strategie di ricerca possono essere **senza informazioni** o **con informazioni** (euristiche). Le principali strategie di ricerca senza informazioni

sono: *Breadth-First Search*, che espande prima lo stato iniziale, poi tutti gli stati generati, e cosi' via (in generale vengono espansi tutti i nodi ad una certa profondita', prima di passare alla profondita' successiva); *Depth-First Search*, che espande sempre il nodo piu' in profondita' del nostro albero di ricerca, fino a che i nodi esauriscono i successori (evento non sempre possibile); *Depth-Limited Search*, che funziona in modo simile alla 'depth-first search', ma espande i nodi solo fino ad una certa profondita' **l**; *Iterative Deepening Depth-First Search*, che funziona come la 'depth-limited search', imponendo **l** inizialmente a 0, poi ad 1, poi a 2, ecc., fino a che non viene trovato uno stato finale.

Per usare una strategia con informazioni, e' necessario usare una **funzione euristica** (**h(n)**), la quale rappresenta il costo stimato della soluzione piu' economica (in relazione al costo per raggiungere uno stato successivo) per raggiungere lo stadio finale. Le principali strategie di ricerca con informazioni sono: *Greedy Best-First Search*, che espande il nodo che si trova piu' vicino alla soluzione, cioe' con h(n) piu' basso; *A* Search*, che somma il costo per raggiungere il nodo (**g(n)**), con h(n), per poi espandere il nodo con valore sommato piu' basso (in questo modo si ha la garanzia di trovare la soluzione ottimale).

Nei giochi da tavolo con due o piu' giocatori non cooperanti, la ricerca di una soluzione non e' lineare, come presentato nei casi precedenti, poiche' ciascuno dei giocatori cerca di raggiungere un obiettivo contrapposto. Se consideriamo un tipico gioco per due giocatori (che chiameremo MAX e MIN), dal punto di vista del 'Problem Solving', i giocatori modificano a turno lo stato del gioco (espandendo uno dei nodi), fino alla fine del gioco (stato finale). Al termine della partita viene assegnata una **utility function**: +1, se il vincitore e' MAX; -1, se il vincitore e' MIN; 0, nel caso di un pareggio. In generale, MAX cerca di massimizzare l'utility function, mentre MIN cerca di minimizzarla. Per trovare una stragegia ottimale, e' necessario esaminare il valore **minmax** di ciascun nodo. Per ricavare questo valore, dobbiamo partire dai possibili stati finali (di cui conosciamo l'utility function), e proseguire fino ad arrivare alla radice dell'albero: (1) il valore minmax degli stati finali e' semplicemente uguale al valore dell'utility function; (2) il valore minmax di uno stato genitore MIN(MAX) e' dato dal valore minmax piu' basso(alto) fra i valori dei possibili stati figli. In questo modo, e' possibile identificare la mossa che consente al giocatore corrente di ottenere il miglior risultato. L'algoritmo che trova il valore di minmax, esegue una ricerca completa dell'intero albero di tipo depth-first; se indichiamo con **b** il numero medio di mosse possibili, e con **m** il valore massimo di mosse, il numero di nodi visitati sara' nell'ordine di $O(b^m)$, impraticabile in un gioco reale. Per questo motivo vengono usate tecniche di **pruning**, che riducono il numero di

nodi che e' necessario visitare. Una delle tecniche piu' utilizzate, l'**Alpha-Beta Pruning**, permette di generare correttamente il valore di minmax, riducendo il numero dei nodi da visitare a circa $O(b^{\frac{m}{2}})$-$O(b^{\frac{3m}{4}})$ a seconda dell'ordinamento delle mosse. Anche se, usando tecniche di pruning, riusciamo a ridurre il numero di stati visitati, rimane comunque impraticabile riuscire a generare una mossa in un tempo ragionevole. Per questo motivo, la ricerca viene di solito fermata ad una certa profondita', per poi usare una **funzione di valutazione** sullo stato corrente, che ne calcola il valore tramite una funzione euristica, che sostituisce l'utility function. E' evidente che la bonta' di un programma dipende fortemente dalla funzione euristica utilizzata.

Le tecniche fino ad ora presentate hanno permesso a Deep Blue di battere Kasparov, ma non sono pienamente utilizzabili in Arimaa, per due motivi: (1) il numero medio di mosse possibili per ogni giocatore e' circa 17'000; (2) e' difficile produrre una buona funzione di valutazione, in quanto le 'catture' (il principale metro di giudizio di una funzione di evalutazione) avvengono molto meno spesso che negli scacchi.

Per utilizzare queste tecniche in Arimaa (ed in generale, in qualunque gioco da tavolo), e' necessario costruire una rappresentazione della situazione di gioco, funzioni capaci di generare correttamente le mosse possibili e funzioni in grado di controllare se un certo stato e' uno stato finale. Generalmente viene utilizzata una rappresentazione ad array, ma noi abbiamo deciso di utilizzare una rappresentazione basata su **bitboard**. In Arimaa, una rappresentazione basata su bitboard e' costituita da dodici numeri a 64-bit (bitboard), ognuno dei quali rappresenta uno dei dodici pezzi del gioco (sei pezzi d'Oro e sei pezzi d'Argento). Una bitboard e' costituita da 64 elementi, ognuno dei quali identifica una specifica casella della scacchiera; gli '1' identificano la presenza di un pezzo in quel punto della scacchiera, mentre gli '0' indicano gli spazi dove quel pezzo non e' presente. E' possibile fare operazioni sulle bitboard, tramite operatori binari: *NOT* (`!`); *OR* (`|`); *AND* (`&`); *XOR* (`^`); *Bit Shifts* (`<<` e `>>`). E' inoltre necessario costruire degli operatori non primitivi, per agire sull"1' meno significativo della bitboard (LS1B), data una bitboard (**x**): *Isolamento* ($LS1B(x) = x\& - x$); *Reset* ($RESET(x) = x\&(x-1)$), che rimuove lo LS1B dalla bitboard. Introdotte queste operazioni, e' possibile, per esempio, ottenere la bitboard che rappresenta tutti gli spazi vuoti della scacchiera, ottenere l'indice del LS1B, sapere quali pezzi in campo sono congelati ed, in generale, rappresentare qualsiasi situazione di gioco, tipologia di movimento e cattura dei pezzi. La parte relativa alla rappresentazione tramite bitboard e' stata iniziata per l'esame di 'Rappresentazione della Conoscenza' (sostenuto nel Dicembre del 2008) e completata con questa tesi.

Giunti a questo punto, e' stata completata la parte di ricerca relativa all'IA per giochi da tavolo. La fase successiva del progetto richiedeva di sviluppare un motore per Arimaa, che integrasse la teoria sul Problem Solving, con un sistema di rappresentazione (bitboard, nel nostro caso). Bisogna innanzitutto premettere che, negli Scacchi, un motore basato su array si e' dimostrato piu' efficiente di uno basato su bitboard. Avendo avuto l'intuzione che, in Arimaa, un motore basato su bitboard sarebbe stato piu' efficiente di uno basato su array, sono stati impiegati tempo e risorse per verificare questa ipotesi. Per fare cio', e' stata introdotta un'unita' di misura, basata sul numero dei nodi visitati diviso il tempo impiegato per visitarli (Nodi al Secondo[NS]), e usato un motore di Arimaa basato su array come campione di riferimento. Quest'ultimo e' stato capace di visitare circa 1.8 milioni di nodi al secondo ($1.8 * 10^6 NS$). Il nostro motore, basato su bitboard, e' stato sviluppato partendo da un motore degli sacchi chiamato 'Gray Matter'; una volta inserite le operazioni per generare correttamente le mosse, partendo da una possibile situazione, si e' rivelato decisamente lento: $76 * 10^3 NS$. Per migliorare le prestazioni del motore sono state effettuate modifiche di ottimizzazione al codice software, il quale era stato concepito per giocare a scacchi. Alla conclusione delle operazioni di ottimizzazione, solo il 5%-10% del nostro codice faceva parte di Gray Matter; l'ultima versione del nostro motore e' stata capace di visitare piu' di 2.5 milioni di nodi al secondo ($2.5 * 10^6 NS$), risultando cosi' piu' veloce del 40% rispetto al motore di riferimento basato su array.

Come gia' premesso, e' stato raggiunto il primo obiettivo della tesi studiando le tecnologie alla base dello *stato dell'arte* nell'IA per giochi da tavolo. Inoltre e' stato raggiunto anche il secondo obiettivo, ovvero realizzare un motore che sia veloce ed affidabile. Il lavoro su questo motore puo' essere ulteriormente ottimizzato, introducendo una tecnica che permetta di evitare di visitare una mossa piu' di una volta. In Arimaa, e' molto facile muovere i pezzi in modo diverso, ottenendo ogni volta lo stesso risultato finale (che incrementa decisamente il numero medio di mosse possibili). Un modo per evitare queste ripetizioni e' salvare ogni stato visitato e, prima di espandere un nuovo stato, controllare se questo non e' gia' presente nel nostro database degli stati. Per fare questo abbiamo cominciato ad implementare una funzione di hash, chiamata 'Zobrist Hash', che e' stata sviluppata specificatamente per giochi da tavolo. Sfortunatamente, non siamo riusciti a finire l'implementazione del codice in tempo per questa tesi.

Contents

List of Figures

Chapter 1

Introduction

Board games have been played by humans for five thousands of years. The Chinese 'Go', for example, is one of the oldest board games, and is still played nowadays.

Computers have been playing board games only for the last five decades. However, game playing was one of the first tasks of Artificial Intelligence (AI); people like Konrad Zuse, Claude Shannon, Norbert Wiener and Alan Turing applied themselves to Chess's AI [11].

Since then, there has been continues progresses, to the point that machines have surpassed humans in Checkers and Connect Four and even defeated the human world champion in Chess in 1997. There are some exceptions, like Go and Arimaa, where humans are still stronger than the machines.

1.1 Goal Definition

The Arimaa's creator, Omar Syed, leads a yearly challenge: the best Arimaa playing programs against the best human players, and, until today, the humans always won.

In order to develop a program capable to beat a professional human player in an Arimaa match we have to reach two main goals: **first goal**, study the *state of the art* in AI game playing; **second goal**, develop new techniques to allow our program to compete with professional human players.

The **goal** of this thesis is to illustrate some of the research results of the

first goal. Moreover, we want to achieve a **tangible goal**: realize an Arimaa engine, capable to represent a board situation, and generate the possible moves as much fast as possible.

1.2 Thesis Overview

In this thesis, we will discuss two branches of AI, Problem Solving and Knowledge Representation, to develop a fast and reliable engine for Arimaa, a board game designed to be difficult for computers. In chapter 2, we provide an introduction on Arimaa game and his rules.

The *state of the art* in AI game playing searches through the possible moves as much ahead as possible, trying to choose the best move and ignoring the bad ones (pruning). To understand how those techniques work, we studied a branch of Artificial Intelligence called Problem Solving; in chapter 3, we will present how to define correctly a problem, how to represent it using a search tree, a summary on the principal search strategies, and how them do apply on Arimaa.

Besides, every engine needs a board representation, in order to maintain the piece positions and generate the possible moves. The standard way to represent a board is through arrays, but we decide to use bitboards: in Chess, array representation engines has proven to be more efficient than bitboard based ones [10], but we had the intuition that, in Arimaa, a bitboard based engine would be more efficient. In chapter 4, we will go through bitboard theory, explaining how it works and it is applied on Arimaa. Most of this chapter was studied as part of the 'Knowledge Representation' exam, discussed in December 2008.

In chapter 5, we will show how we implemented the problem solving techniques with a bitboard representation system, producing an Arimaa engine. We will also take a reference Arimaa array based engine, organize a performance test method, and measure which one is faster. Then, we will discuss a few code optimization of the bitboard engine.

Finally, in chapter 6, we will discuss the results relatively to the *tangible subgoal*, the uses of our Arimaa engine we developed, the future work, and propose a new technique to reach the *second goal*.

Chapter 2

Arimaa

Arimaa is a two players board game created by Omar Syed, designed to be easy to learn, difficult for computers and playable on a standard Chess set [14].

Since 2004, there has been an annual Championship where the stronger humans play against the stronger computer players; up until now the humans have always defeated the computer, even with a handicap of several pieces.

In the following chapter, we will display and explain the rules of Arimaa, and it is strongly based on the flash tutorial on Arimaa website [13]. In particular, the wording of the rules are maintained as much as possible similar to the original versions [14, 3].

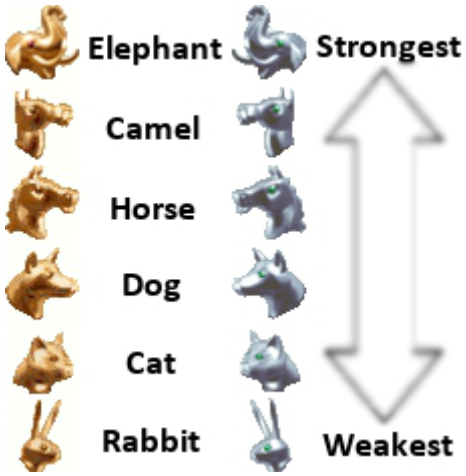

Figure 2.1: Arimaa Pieces.

2.1 Board and Pieces

A game of Arimaa starts with an empty 8x8 board.

There are six types of pieces, each one of them stronger or weaker than the others; the elephant is the strongest followed by camel, horse, dog, cat and rabbit(see Figure 2.1). Each player (gold and silver) starts the game with one elephant, one camel, two horses, two dogs, two cats and eight rabbits.

2.2 Setup

The gold player places all his pieces within the first two rows, followed by the silver player.

There is no fixed starting position, so the pieces may be placed in any arrangement (see Figure 2.2).

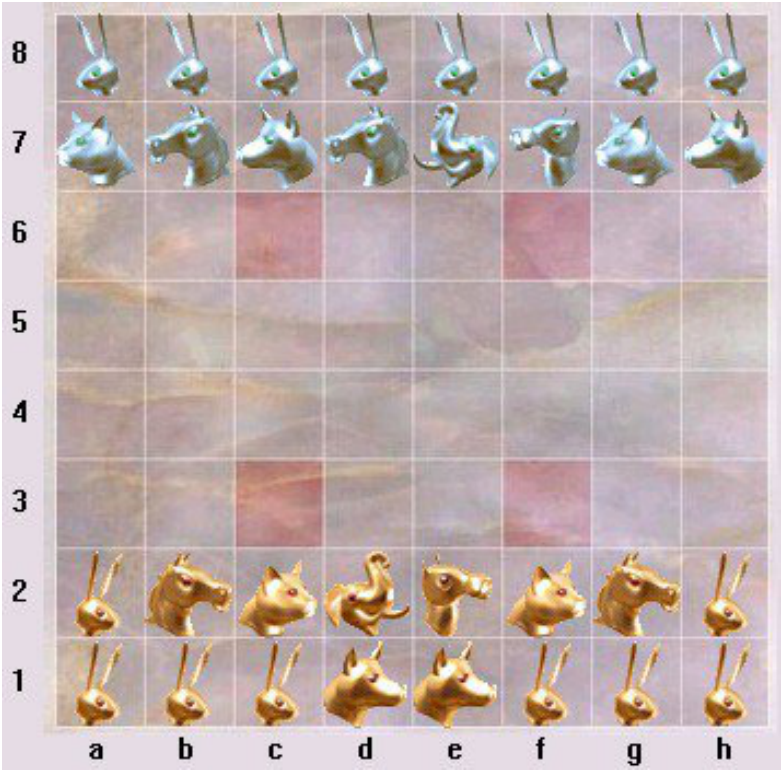

Figure 2.2: Setting up

2.3 Goal

The goal is to get any of the eight rabbits to the eighth row, on the other side of the board.

2.4 Movement

All the pieces in Arimaa can be moved in the same direction. They can move forward, backward, left and right(see Figure 2.3).

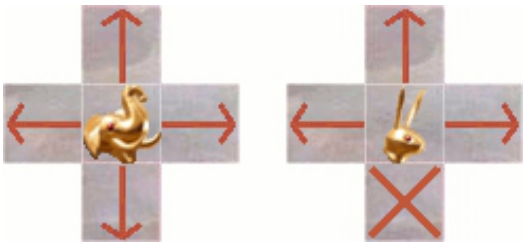

Figure 2.3: Pieces Movement

Only the rabbits cannot move backwards: they can only move forward, left and right(see Figure 2.3).

We will indicate this type of movement as 'one step' move.

2.4.1 Four Steps

At each turn, a player must take at least one step. Moving a piece from its position to the next square counts as one step. A player can take up to four steps per turn. A piece can take multiple steps and also change direction after each step. All steps do not have to be taken by the same piece and can be divided among them so that more than one piece can be moved.

For example, considering the Figure 2.2, the gold player could move up to four times the elephant, or two times the camel and two times one horse, or four different pieces.

2.4.2 Pushing and Pulling

The stronger pieces can move opponent's weaker pieces. For example the dog can move the opponent's cat or rabbit, but not the opponent's dog or any other piece that is stronger than it. The opponent's piece must be orthogonally adjacent to yours, and can be moved by either pushing or pulling it.

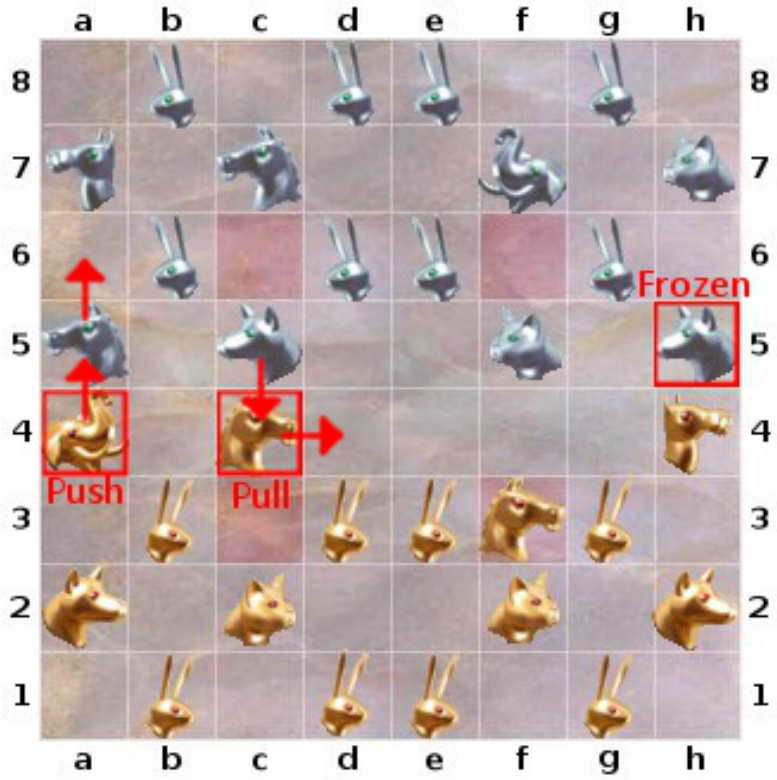

Figure 2.4: Movement Example

To push an opponent's piece with your stronger piece, first move the opponent's piece to one of the empty orthogonally adjacent squares and then move your piece into its place 2.4). 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 2.4).

A push or pull requires two consecutive 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.

The pieces are not strong enough to pull one enemy piece while pushing another at the same time. Note that even though the rabbits cannot move backwards on their own, they can still be pushed or pulled back.

We will indicate those types of movement as 'two steps' moves.

2.4.3 Freezing

When a weaker piece is next to a stronger enemy piece, it becomes frozen and cannot move unless there is a friendly piece next to it(see Figure 2.4). The frozen piece can still be pushed or pulled by stronger enemy pieces.

2.5 Trapping

There are four trap squares on the board, placed on C3, F3, C6 and F6. Any piece that is on a trap square and does not have friendly piece next to it is removed from the game.

2.6 Special Situations

A player may push or pull the opponent's rabbit into the goal row it is trying to reach. If at the end of the turn the rabbit remains there, the player loses. However if the opponent's rabbit is moved back out of the goal row before the end of the turn, the player does not lose.

If a player is unable to make a move because all the pieces are frozen or have no place to move, then that player has lost the game.

If after a turn the same board position and side to move would be repeated three times, then that move is considered illegal and the player must select a different move. If in the rare case the only moves a player has are not allowed then the player loses due to being unable to make a move.

If a player loses all the rabbits then that player loses the game. If in the rare case both players lose all rabbits on the same move then the player making the move wins the game.

2.7 Notation

For reviewing games, the games have to be recorded. To do this recording we will use the notation for recording the Arimaa positions [14].

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 equivalent to Figure 2.4:

```
 +-----------------+
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 f g h
```

Chapter 3

Problem Solving

The whole chapter is based on the Russel-Norvig book 'Artificial Intelligence, a modern approach' [11]. In the following pages, we will present: (1)how to define correctly a problem, (2)how to represent it using a search tree, (3)a summary on the principal search strategies, and (4)how to apply them on Arimaa.

3.1 Well Defined Problem

Before analyzing complex problems, we should start with a simple one. In Figure 3.1, we see an eight elements sliding-block puzzle; (a) shows an example of a starting position, and (b) is the solution we want to achieve.

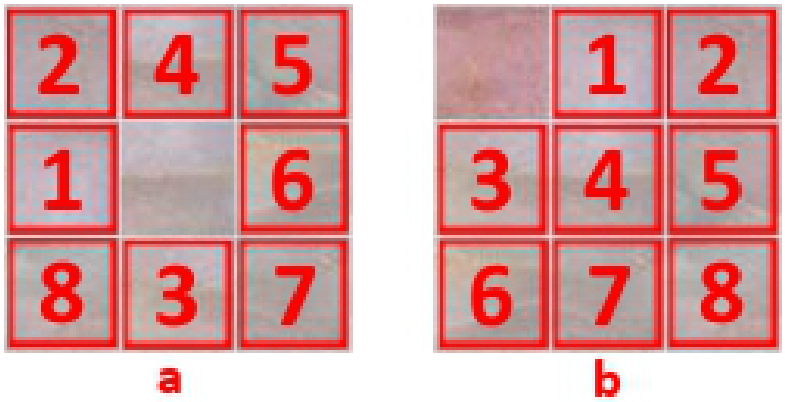

Figure 3.1: The 8 Sliding-Block Puzzle

We can also represent the sliding-block puzzle as a 3x3 matrix or a 9-dimensional array of integers, using '0' as the empty tale:

```
a:{2, 4, 5,    b:{0, 1, 2,
   1, 0, 6,       3, 4, 5,
   8, 3, 7}       6, 7, 8}
```

A problem can be defined by four components [11]:

- *The initial state*: in our sliding-block puzzle 3.1(a) is the initial state.
- *A successor function*: given a state x, $SUCCESSOR(x)$ returns a set of (action, successor) where each action is a legal action in x and each successor is a state that can be reached from x by applying the action. For example, in our sliding-block puzzle, the possible actions are moving the empty tile $Left$, $Right$, Up, or $Down$; thus, using the initial state (a), the successor function would return:

```
       {2, 4, 5,          {2, 4, 5,
{(Left, 0, 1, 6,), (Right, 1, 6, 0,),
        8, 3, 7}           8, 3, 7}

       {2, 0, 5,          {2, 4, 5,
 (Up,   1, 4, 6,), (Down,  1, 3, 6,)}
        8, 3, 7}           8, 0, 7}
```

- *The goal test*: determines if a give state is the goal state. This checks whether the state matches the goal configuration 3.1(b).
- *A path cost function*: assigns a numeric cost to each path. In our sliding-block puzzle each step cost 1, so the path cost is the number of steps in the path.

3.2 Search Tree

To solve a problem we have to search through the state space [11]. To represent it, we can use a search tree, with the initial state as the root, and the branches as the products of the successor function.

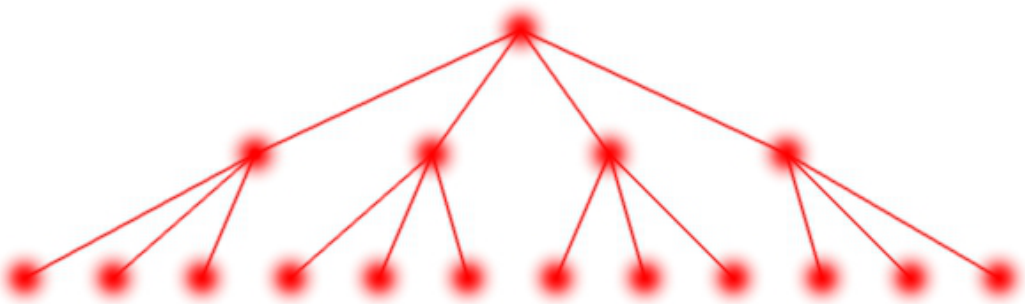

Figure 3.2: Search Tree

For example, the Figure 3.2 can be used to represent the first two successor functions of the sliding-block puzzle presented in 3.1. From the initial state (root) we have the four branches already discussed, and from each one of them, three more branches; we have only three branches for each node because, after the first step, the empty tile is laying on a side, where only three types of actions are possible.

There are many ways to represent each node, and Russel-Norvig give an example of a data structure with five components [11]:

- *State*: the current state of the node.
- *Parent-Node*: the node in the search tree that generated this node.
- *Action*: the action that was applied to the parent to generate the node.
- *Path-Cost*: the cost from the initial state to the current node.
- *Depth*: the number of steps along the path from the initial state.

For our example, we can use a much simpler version; we do not need to remember the parent node (not now, at least), neither the action that was performed, and the *Path-Cost* has the same value as *Depth*, since in our sliding-block puzzle the cost of each step is 1.

To use correctly a search tree, we have to perform three steps, starting from the root node [11]:

1. Check if the current state is the goal state.
2. Expand the current state, applying the successor function to the current state, thereby generating a new set of states.

3. Choose one of the not expanded state and restart from point 1.

The choice of which state to expand is determined by the *search strategy*. The collection of nodes that have been generated but not yet expanded is called *fringe*. Each element of the fringe is a leaf node.

3.3 Uninformed Search Strategies

When we have no additional information about states beyond that provided in the problem definition, we have to perform an uninformed search. We have several types of uninformed search, and they can generate successors and distinguish a goal state from a non-goal state. We will examine briefly few of them.

To analyze the complexity of the search strategies, we have to introduce the branching factor value b. For example, in our sliding-block puzzle, the branching factor is the average of the number of successor functions for each location where the empty tile could be; thus, for each one of the four corners we have two successor functions, three for each one of the four sides and four for the center position:

$$b = \frac{((4 \times 2) + (4 \times 3) + (1 \times 4))}{9} = 2.\bar{6} \cong 3$$

3.3.1 Breadth-First Search

When we use breadth-first search first we expand the root node, then all the successors of the root node, then their successors, and so on. In general, all the nodes are expanded at a given depth before any node at bigger depths are expanded [11].

For example, the Figure 3.3 shows the first four steps of the breadth-first search for the sliding-block puzzle. The bright red nodes are the generated ones, while the out-of-focus ones still have to be generated. The triangle points at the node we are currently examining, generate the successor nodes if the current one is not the goal one. The obscured nodes are the already visited ones.

If the solution is at depth d, in the worst case we would expand all the nodes until the last node at level d (since the goal itself in not expanded),

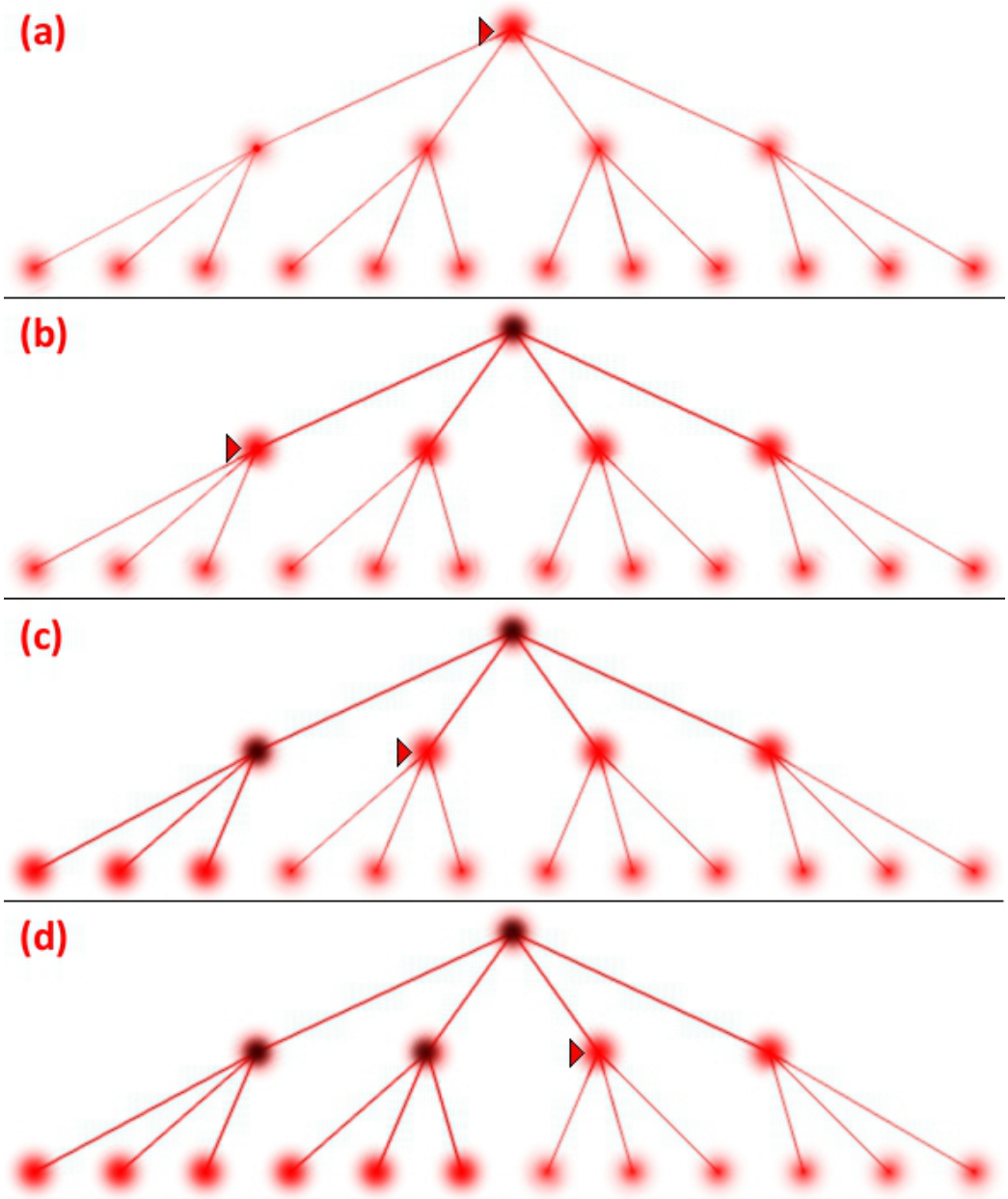

Figure 3.3: Breadth-First Search

generating $b^{d+1} - b$ nodes at level $d + 1$. Then the total number of nodes generated is [11]:

$$b + b^2 + b^3 + ... + b^d + (b^{d+1}) = O(b^{d+1})$$

3.3.2 Depth-First Search

The depth-first search always expand the deepest node in the current fringe of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors [11].

This strategy can be used only for finite (bounded) type of trees. Since our sliding-block puzzle produce an infinite (unbounded) tree, we can't use this technique with our problem. How the sliding-block puzzle produce infinite tree is easily demonstrated: from the initial state, we can slide the empty tile to any position (i.e. to the $Left$) and after slide it back to the original position (i.e. to the $Right$). Therefore, repeating the $Left - Right$ actions, we can obtain an infinite number of states, where none of them is the goal state.

In the worst case, the depth-first search will generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node. Note that m can be much larger than d (the depth of the shallowest solution) [11].

3.3.3 Depth-Limited Search

The depth-limit search is a strategy, similar to the depth-first search, that uses a pre-determined depth limit l; nodes at depth l are, in fact, treated as if they have no successors. This approach alleviates the problem of treating unbounded trees. Unfortunately if we choose $l < d$, the goal will not be reached. This is not unlikely when d is unknown [11].

The Figure 3.4 shows the first eight steps of a depth-limited search on our sliding-block puzzle with $l = 2$. The steps are arranged from left to right, and from top to bottom. The bright red nodes are the generated ones, while the out-of-focus ones still have to be generated. The triangle points at the node we are currently examining, generate the successor nodes if the current one is not the goal one, or if we do not reach the depth limit l. The obscured nodes are the already visited ones.

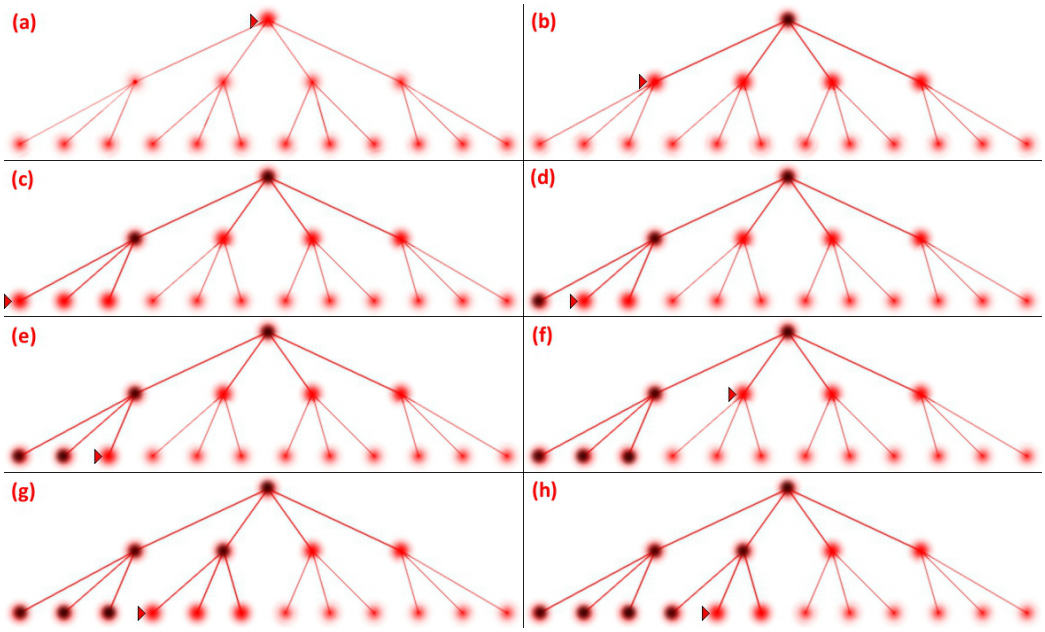

Figure 3.4: Depth-Limited Search

The complexity of depth-limited search is $O(b^l)$. Depth-first search can be viewed as a special case of depth-limited search with $l = \infty$ [11].

3.3.4 Iterative Deepening Depth-First Search

The iterative deepening depth-first search (or iterative deepening search) is a strategy that finds the best depth limit. It does this by gradually increasing the depth limit (first 0, then 1, then 2, and so on) until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node [11].

Iterative deepening search may seem wasteful, because states are generated multiple times. However, it turns out this is not very costly. The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times. In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next to bottom level are generated twice, and so on, up to the children of the root, which are generated d times. So the total number

of nodes generated is:

$$N(IDS) = (d)b + (d-1)b^2 + ... + (1)b^d$$

which gives a complexity of $O(b^d)$ [11].

3.4 Informed (Heuristic) Search Strategies

When we uses problem-specific knowledge beyond the definition of the problem itself, we can use an informed search strategy, finding solution more efficiently than a uninformed strategy.

A key component of those type of strategies is a heuristic function, denoted $h(n)$, that represents the estimated cost of the cheapest path from node n to a goal node. If n is a goal node, then $h(n) = 0$ [11].

For example, if we consider our sliding-blocks puzzle, there are two commonly used heuristic functions [11]:

- h_1 = number of misplaced tiles. For figure 3.1(a), all eight tiles are out of position, so the start state would have $h_1 = 8$.
- h_2 = the sum of the distances of the tile from their goal position. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is called *Manhattan distance* [28]. Tiles from 1 to 8 in the figure 3.1(a) have a Manhattan distance of:

$$h_2 = 2 + 2 + 2 + 1 + 1 + 3 + 1 + 2 = 14$$

We will now analyze few heuristic search strategies.

3.4.1 Greedy Best-First Search

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function: $f(n) = h(n)$ [11].

In Figure 3.5 we can see how the greedy best-first search works for our sliding-block puzzle. We use the heuristic function h_1 presented in section 3.4.

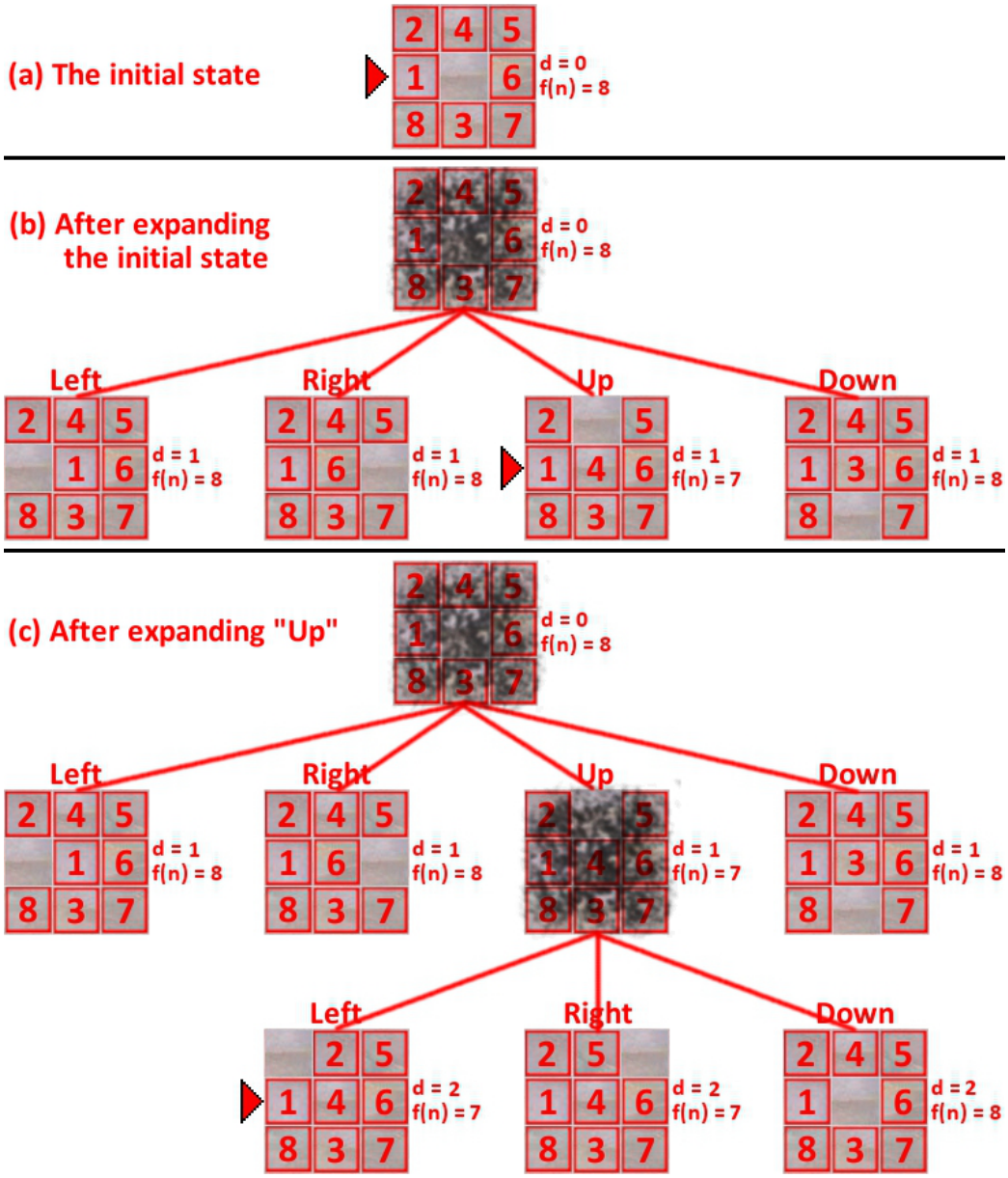

Figure 3.5: Greedy Best-First Search

For each node, we indicate the depth d and the calculated value of the heuristic function $f(n) = h(n)$. The triangle points at the node we are currently examining, generate the successor nodes if the current one is not the goal one. The obscured nodes are the already visited ones.

The first node expanded after the initial state will be Up, because it is the closer to the goal state ($f(n) = 7$, while $f(n) = 8$ in all the other nodes). When the algorithm reaches depth 2, it could choose indifferently to continue from node $Left$ or $Right$, having them the same and smallest heuristic value in the search tree. We could also observe how at depth 2 the $Down$ action returns the initial state.

Even if the heuristic function is not particularly adapt to show it, the previous behavior suggests why the algorithm is called 'greedy', as at each step tries to get as close to the goal as it can. The complexity is $O(b^m)$, where m is the maximum depth of the search space. With a good heuristic function, however, the complexity can be reduced substantially [11].

3.4.2 A* Search

A* search is the most widely-know form of best-first search. It evaluates the node combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal:

$$f(n) = g(n) + h(n)$$

Thus, $f(n)$ is the estimated cost of the cheapest solution through n. A* gives an optimal solution; an optimal solution has the lowest path cost among all solutions [11].

The Figure 3.6 shows how works the A* search for our-sliding block puzzle. We use the heuristic function h_1 presented in section 3.4. For each node we indicate the depth d, that is also equal to the value $g(n)$ (as presented in section 3.2), the calculated value of the heuristic function $h(n)$, and the value of $f(n) = d + h(n)$. The triangle points at the node we are currently examining, generate the successor nodes if the current one is not the goal one. The obscured nodes are the already visited ones.

The first node expanded after the initial state it will be Up, because it has the smaller value of the nodes on the fringe ($f(n) = 8$, while $f(n) = 9$ in all the other nodes). When the algorithm reaches depth 2, it could choose

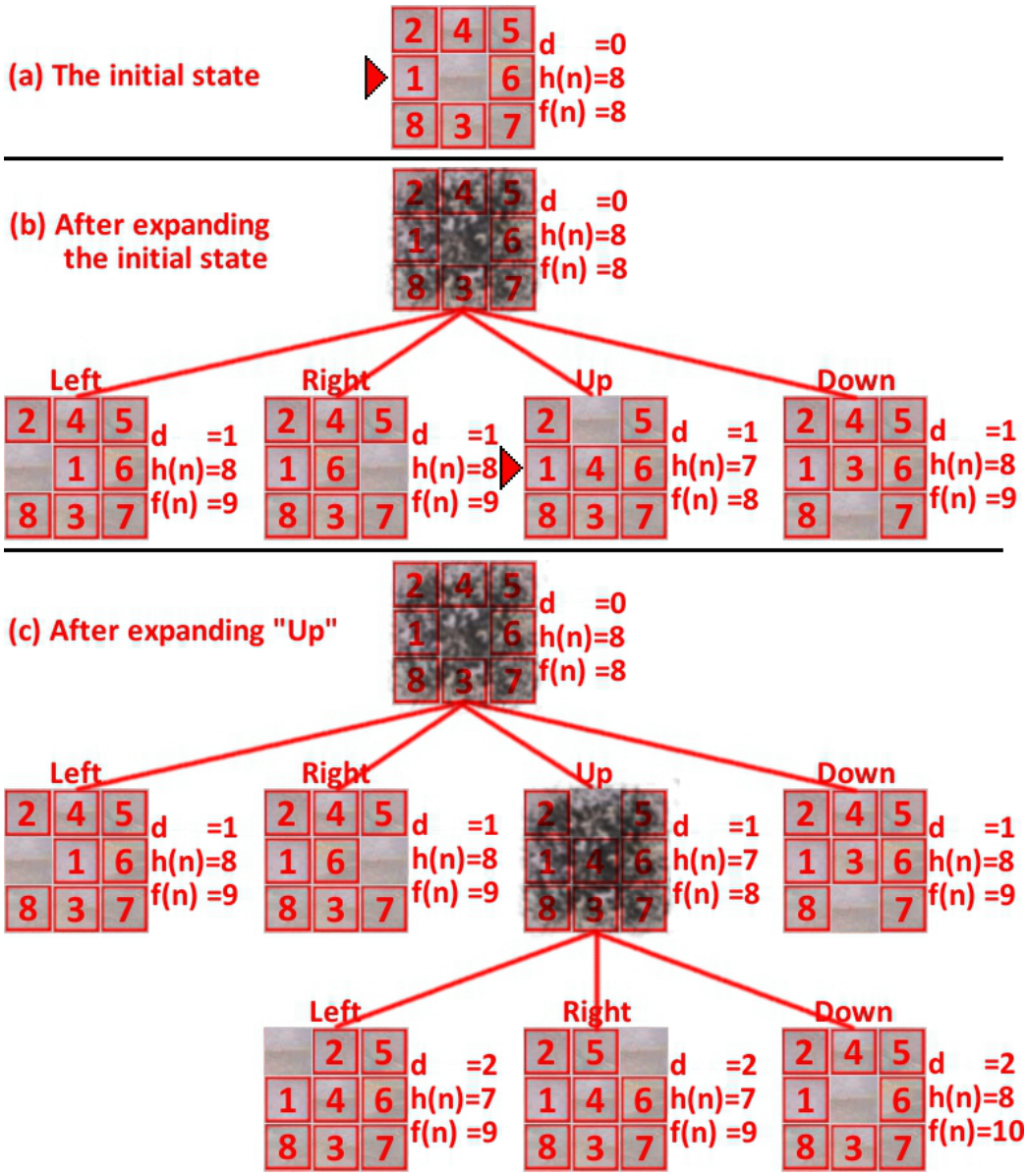

Figure 3.6: A* Search

indifferently any node on the fringe with $f(n) = 9$, excluding only the node at depth 2 that has a value $f(n) = 10$: as in the greedy best-first search, this node has the same shape as the initial state, but, differently, it will not be expanded immediately. Thus, A* can limit (or even avoid, using a good heuristic) the problems caused by unbounded trees.

Among optimal algorithms that extend search paths from the root, A* is optimally efficient for any given heuristic function. No other optimal algorithm is guaranteed to expand fewer nodes than A* [11].

3.5 Adversarial Search

The following problem solving strategies are for zero-sum abstract strategy games. The Wikipedia definition for abstract strategy game is 'a board or card game with perfect information, no chance and (usually) two players or teams' [19].

A game has perfect information when all players know all moves that have taken places [15]. Chess, Go and Arimaa are examples of games with perfect information, while Poker, Stratego and Cluedo are games with imperfect information, since you do not know what cards your opponents have, what pieces is your opponent hiding or who is the murder.

A game has no chance when there is not a random element involved. Chess, Go and Arimaa are games with no chance, while Backgammon, Poker and Monopoly have a chance element, that could be the roll of a die or the card shuffling.

Finally, a game is zero-sum when a player gain or loss is exactly balanced by the losses or gains of the other player(s) [29]. The zero-sum concept is studied in game theory and economic theory: generally, any game where all strategies give a gain to a player and a loss to the other is called a conflict(zero-sum) game [2]. Again, Chess, Go and Arimaa are examples of zero-sum games.

Games are interesting because they are hard to solve. For example, Chess has an average branching factor of about 35, and games often go to 50 moves by each player, resulting a search tree of 35^{100} nodes. Moreover, games require to make a decision in a limited amount of time. Thus, being not feasible to calculate the optimal solution exploring the whole search tree, we need to look at techniques for choosing a good move [11].

3.5.1 Optimal Decision in Games

We will consider games with two players, whom we will call MAX and MIN. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are given to the winning player and penalties are given to the loser. We can define a game as a search problem with the following components [11]:

- *The initial state*: include the board position and identifies the player to move.
- *A successor function*: returns a list of (moves, state) pairs, each indicating a move and the resulting state.
- *A terminal test*: determines when the game is over. States where the game has ended are called terminal states.
- *A utility function*: gives a numeric value for the terminal states. In chess, the outcome is a win, loss or draw, with values +1, -1 or 0.

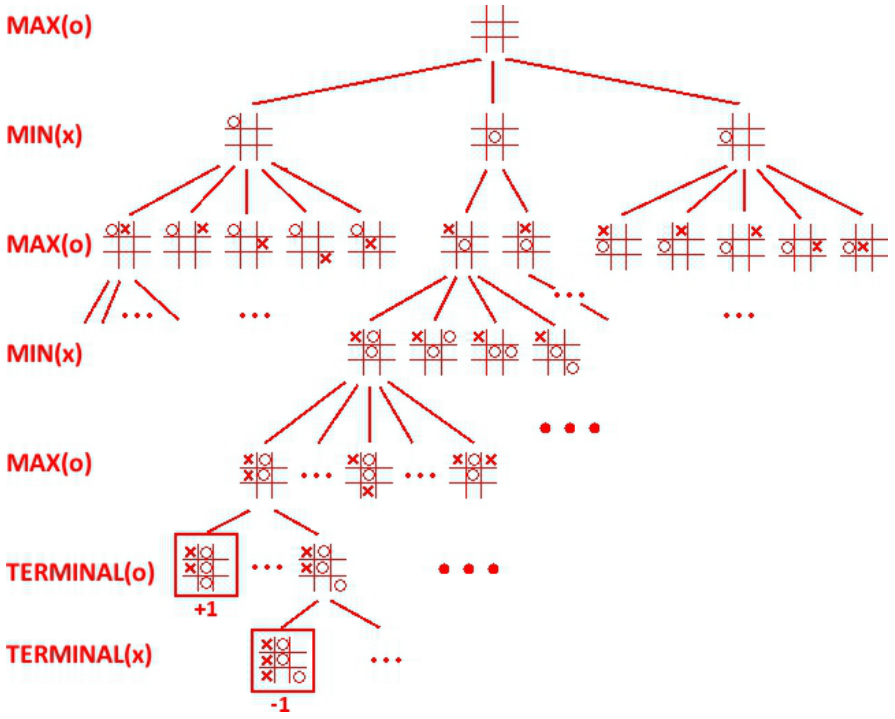

Figure 3.7: Tic-Tac-Toe

The initial state and the legal moves for each side define the game tree for the game. The Figure 3.7 shows part of the game tree for Tic-Tac-Toe. From the initial state, MAX has three possible moves (we eliminated the symmetric ones). Play alternate MAX's placing an O and MIN's placing an X until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled. The number on the each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names) [11]. MAX's work is to maximize the utility function, while MIN has to minimize it.

3.5.2 Optimal Strategies

In a normal search problem, the optimal solution would be a sequence of moves leading to a goal state, or a terminal state that is a win. In a game both players try to reach a goal state, that is different for each one of them. We will begin by showing how to find an optimal strategy [11].

To illustrate how to proceed, we will use the Figure 3.8, that shows a trivial game using a square to represent the MAX and a circle to represent the MIN. On the bottom are shown the utility values for each terminal value, in MAX prospective.

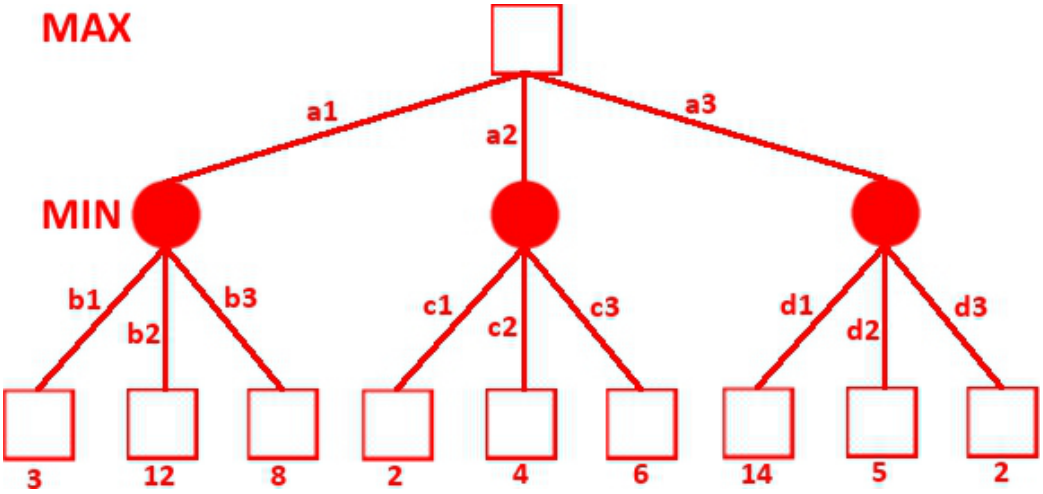

Figure 3.8: A Simple Game Tree

The possible moves for MAX are $a1$, $a2$ and $a3$. The possible replies to $a1$ for MIN are $b1$, $b2$ and $b3$, to $a2$ are $c1$, $c2$ and $c3$, to $a3$ are $d1$, $d2$ and

$d3$. This game ends after one move per player.

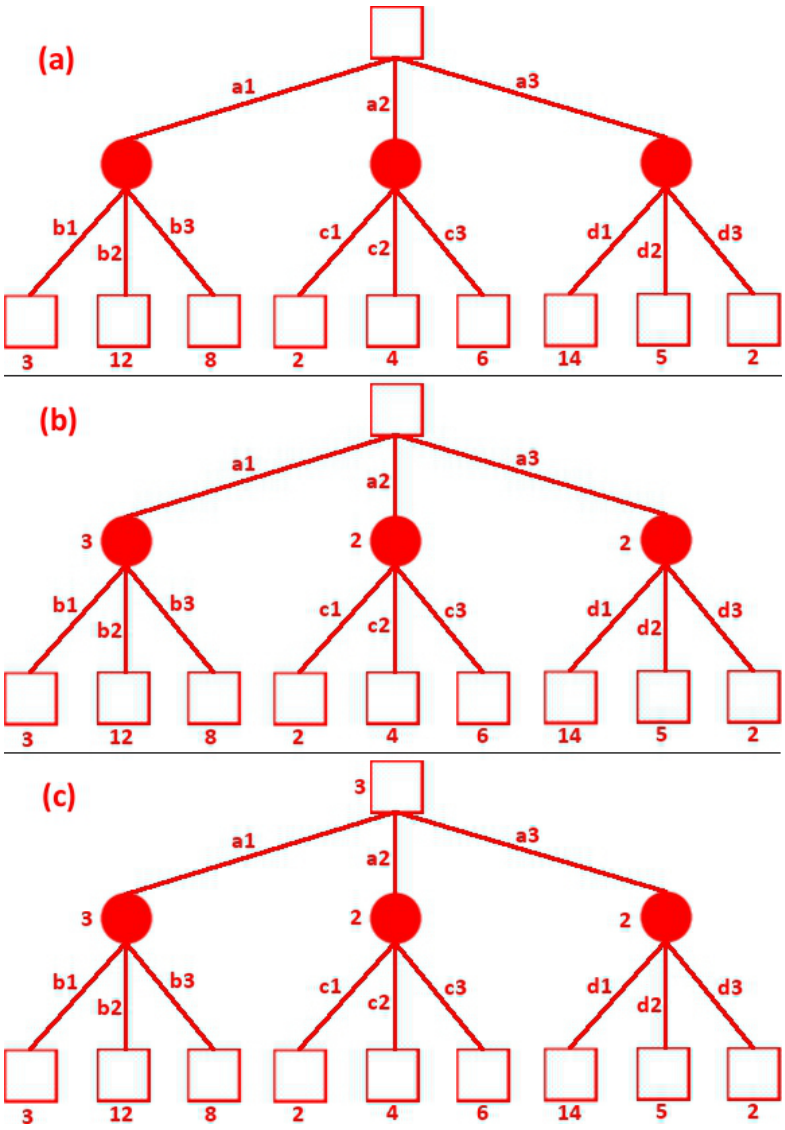

Figure 3.9: Minimax

Given a game tree, the optimal strategy can be determined by examining the minimax value of each node. We start from the terminal states; the minimax value of a terminal state is just its utility value (see Figure 3.9(a)). Then is MIN turn to choose the move, and from each possibility, he/she will choose the one with smaller minimax value, as he/she will try to minimize the value (see Figure 3.9(b)). Then is MAX turn to choose the best move, so he will choose the higher minimax value of the one chosen by MIN, as he

tries to maximize the value(see Figure 3.9(c)).

Thus, if both players play optimally, MAX will choose $a1$ and after MIN will choose $b1$. We observe how the best absolute move for MAX is in the branch $a3$, but if MAX would choose this one instead, MIN would maximize his outcome choosing $d3$ instead of $d1$.

The algorithm that computes the minimax decision from the current state is called 'minimax algorithm'. It uses a simple recursive computation of the minimax values of each successor state. The recursion proceeds all the way down the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds [11].

The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m, and there are b average legal moves, then the complexity of the minimax algorithm is $O(b^m)$. For a real game, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms [11].

3.5.3 Alpha-Beta Pruning

Alpha-beta pruning is a useful technique to reduce the number of state spaces that has to be examined, computing the correct minimax decision without looking at every node in the game tree. In a standard minimax tree, this is obtained pruning the branches that cannot possibly influence the final decision [11].

To illustrate a simple example, let's consider the same trivial game as in section 3.5.2. In Figure 3.10 we can see the stages to calculate the optimal decision for the game tree in Figure 3.8. At each point, we show the range of possible values for each node [11] (we used the € symbol to represent ∞, as we could not find it on our graphic editor):

- a) The first leaf obtained from B with action $b1$ has the value 3. Hence, B, which is a MIN node, has a value of at most 3.
- b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3.
- c) The third leaf below B has a value of 8; we have seen all B's successors, so the value of B is exactly 3. Now, we can infer that the value of

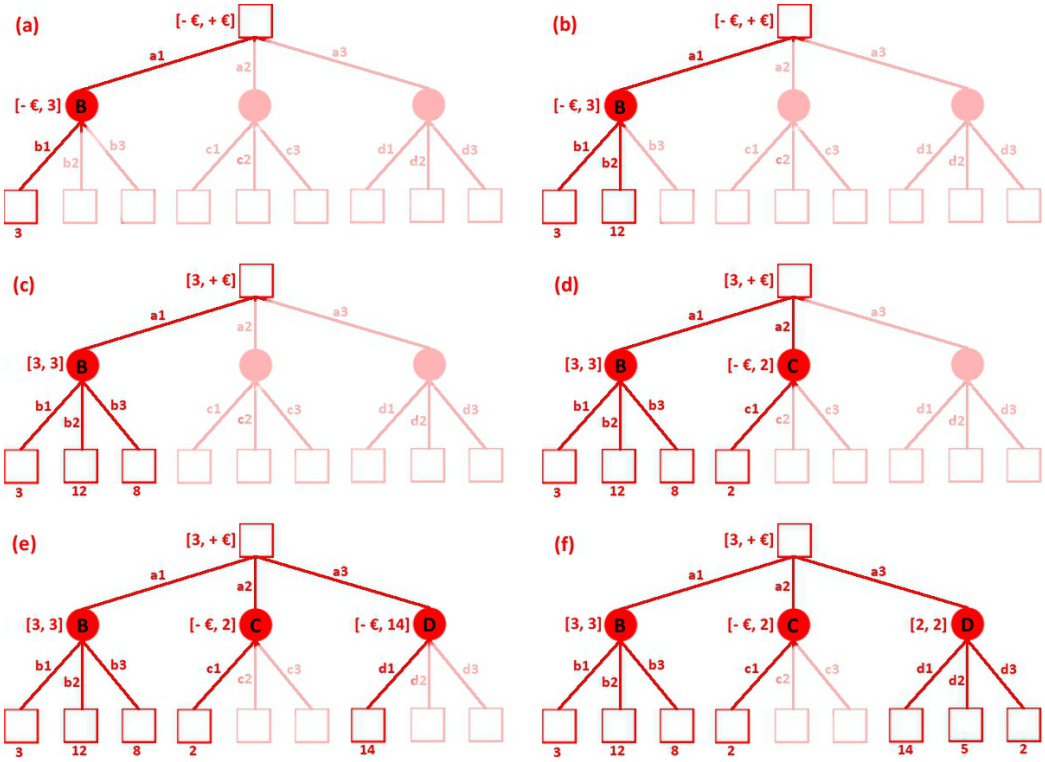

Figure 3.10: Alpha-Beta Pruning

the root is at least 3, because MAX has a choice worth 3 at the root.

- d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successors of C. This is an example of alpha-beta pruning.

- e) The first leaf below D has the value 14, so D is worth at most 14. This is still higher than MAX'S best alternative (i.e., 3), so we need to keep exploring D's successors. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14.

- f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX'S decision at the root is to move to B, giving a value of 3.

Alpha-beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves, but the effectiveness

of alpha-beta pruning is highly dependent on the order in which the successors are examined. If successors are examined in random order, the total number of nodes examined will be roughly $O(b^{\frac{3m}{4}})$ [11].

3.5.4 Imperfect, Real-Time Decisions

The minimax algorithm generates the entire game search space; even if the alpha-beta algorithm allows us to prune large parts of it, it still has to search all the way to the terminal states, for at least a portion of the search space. This depth is usually not practical, because the moves must be made in a reasonable amount of time [11].

Instead of search all the way to the terminal states, we could cut off the search earlier, and use a heuristic evaluation function replacing the utility function [12]. Of course we can still use the alpha-beta algorithm to further reduce the search space.

An evaluation returns an estimate of the expected utility of the game from a given position. It should be clear that the performance of a game-playing program is dependent on the quality of its evaluation function. An inaccurate evaluation function will guide an agent toward losing the game [11].

For example, introductory chess books give an approximate material value for each piece: each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the queen 9. Other features such as 'good pawn structure' and 'king safety' might be worth half a pawn, say. These feature values are then simply added up to obtain the evaluation of the position [11].

3.6 Problem Solving in Arimaa

To obtain a valid behavior from an computer playing Arimaa, we could rely on the search strategies described in the previous sections, and more specifically on the adversarial search techniques. In the other hand, now we have the basis to understand why Arimaa is so difficult for computers.

First of all, Arimaa has a huge branching factor; the average branching factor is 17'281[5]. This was obtained giving the possibility of up to four steps for each players' move.

Considering that the complexity is exponential (b^d, where b is the branch-

ing factor, and d is the depth we want to reach), this limits considerably the portion of state space a computer can look in a time constraints game.

Even using alpha-beta pruning ($O(b^{\frac{3m}{4}})$), Deep Blue (capable to explore $200'000'000$ nodes per second[23]) would be able to look only three moves ahead (so at depth 3), in four minutes.

Second, there are no opening books. An 'Opening Book' is a database of openings that guides computer chess programs for approximately the first ten moves. As result, it places the computer in a stronger position using considerably less resources than if it had to calculate the move itself [26]. This was obtained giving no fixed starting position.

Lastly, it is very difficult to build a valid evaluation function. The material value of the pieces is still valid, but the capture in an Arimaa game happens lesser often than in Chess, thanks to the presence of only four traps, and keeping a strong piece around them provides a very good safety net. It also happened, during the annual Championship, that a player won the game without losing any piece.

3.6.1 Strategy and Tactic

It is necessary to illustrate the difference between strategy and tactic in a game like Arimaa. Both those terms have military origin, and every game, being a simulation of a battle (military, economic or politic), have a balance of those components.

In the Dictionary of Military Terms [8], the tactic is 'the level of war at which battles and engagements are planned and executed to achieve military objectives assigned to tactical units or task forces'. The activities at this level focus on the ordered arrangement and maneuver of combat elements in relation to each other and to the enemy to achieve combat objectives.

The strategy is the overall campaign plan, which may involve complex operational patterns, activities, and decision-making that lead to the tactical execution [27].

At a different level, the tactic is more similar to science, where strategy is more similar to art. The machines are pretty good in all the games where the tactic is prevalent, and/or the strategy is simple, like Chess or Connect Four, while they suffer in games where the strategy has larger impact, like Go or Arimaa.

In conclusion, we cannot rely on traditional techniques to build a good 'all-around' AI for Arimaa. The strategical emphasis of Arimaa requires an innovative problem solving approach. On the other hand, the tactical movements, like a Rabbit running to the goal or the capture of an enemy piece, benefit largely of the search techniques we have illustrated in this chapter.

Chapter 4

Bitboard

A bitboard, often used for boardgames such as Chess, Checkers and Othello, is a specialization of the bitset data structure. Each bit represents a game position or state, designed to optimize speed and/or memory. 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 [21].

To build a fast engine, we have to start by studying its knowledge representation system. In this chapter, we will give an overview of the basic operation for binary numbers, we will explain how a bitboard works and why it is better than a standard array approach. Finally, we will see how we modeled the rules of Arimaa using the bitboards.

4.1 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 [22].

In the following section, we will explain what is a binary number and the basic operation we will use on them.

4.1.1 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 [20]:

```
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
```

4.1.2 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[1] 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.

4.1.3 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 pattern; the second of each pattern; and so on) and performing the logical inclusive OR operation on each pair of corresponding bits. In each pair, the

[1] C is a general-purpose computer programming language

result is 1 if the first bit is 1 OR the second bit is 1 (or both), 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.

4.1.4 AND

A bitwise AND takes two binary patterns 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.

4.1.5 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.

4.1.6 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 and performs one step 'left/right shift':

```
   00010111 LEFT-SHIFT
=  00101110

   00010111 RIGHT-SHIFT
=  00001011
```

In C-inspired languages, the left and right logical shift operators are '<<' and '>>', respectively, followed by the number of places to shift.

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

4.1.7 Least Significant One

The least significant one bit (LS1B) in a binary number is the right-most one bit. Sometimes it is required to isolate this LS1B, or to reset it, obtaining the same binary number without the last significant one bit.

To isolate the LS1B of a binary number x [18]:

```
LS1B(x) = x & -x;

 x:  01011100 &
-x:  10100100 =
LS1B: 00000100
```

To reset the LS1B of a binary number x [18]:

```
RESET(x) = x & (x-1);

  x:  01011100 &
(x-1): 01011011 =
RESET: 01011000
```

4.2 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 [9].

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

```
// Possible states of the 64 individual squares.
#define EMPTY               0
#define GOLD_RABBIT         1
#define GOLD_CAT            2
#define GOLD_DOG            3
#define GOLD_HORSE          4
#define GOLD_CAMEL          5
#define GOLD_ELEPHANT       6
#define SILVER_RABBIT       7
```

```
#define SILVER_CAT          8
#define SILVER_DOG          9
#define SILVER_HORSE       10
#define SILVER_CAMEL       11
#define SILVER_ELEPHANT    12

// And we have an array of 64 squares.
int board[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 4.1).

```
//Declare twelve 64-bit unsigned integers for one board:
typedef unsigned __int64  bitboard;

bitboard GR, GC, GD, GH, GM, GE, SR, SC, SD, SH, SM, SE;
```

In the Figure 4.1, we can see how the board situation in the middle can be represented as the twelve boards around it. As indicated in section 2.7, upper case letters are for gold pieces, lower case letters are for silver pieces. In particular, R/r:Rabbit, C/c:Cat, D/d:Dog, H/h:Horse, M/m:Camel, E/e:Elephant.

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:

```
bitboard NOT_EMPTY = GR | GC | GD | GH | GM | GE |
                     SR | SC | SD | SH | SM | SE ;
bitboard EMPTY = ~NOT_EMPTY;
```

4.3 Mapping

A bitboard is constituted of 64 elements, but we have to identify for each bit which rank/file of a board associate with it. There are several mappings, but we decided to use the 'Little-Endian Rank-File' mapping.

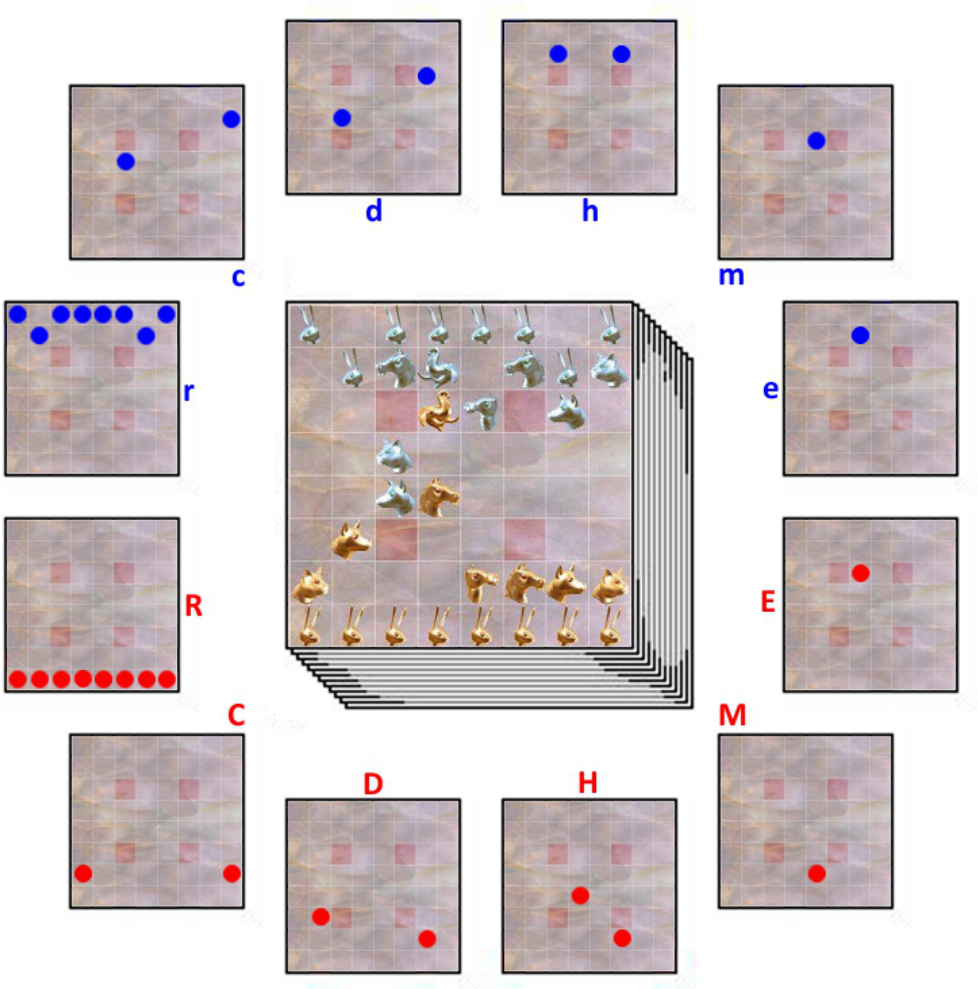

Figure 4.1: The Bitboard Data Structure

The representation is:

```
    A    B    C    D    E    F    G    H
  +----+----+----+----+----+----+----+----+
8 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |  8th rank
  +----+----+----+----+----+----+----+----+
7 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |  7th rank
  +----+----+----+----+----+----+----+----+
6 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |  6th rank
  +----+----+----+----+----+----+----+----+
5 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |  5th rank
  +----+----+----+----+----+----+----+----+
4 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |  4th rank
  +----+----+----+----+----+----+----+----+
3 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |  3rd rank
  +----+----+----+----+----+----+----+----+
2 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 |  2nd rank
  +----+----+----+----+----+----+----+----+
1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  1st rank
  +----+----+----+----+----+----+----+----+
    A    B    C    D    E    F    G    H - file(s)
```

For example, the square $D5$ is identified by the 35th bit. Hypothesizing we have the location of a square by its bit, how to get the square directly over, under, at left, or a right of it? The 'Compass Rose' of the little-endian rank-file mapping gives us such information:

```
            nort
             +8
              |
west  -1 <-   0 -> +1  east
              |
             -8
            sout
```

Let's notice how north and south are shorten in *nort* and *sout*. This was made to have all direction functions (see Section 4.6) of the same length (four char).

4.4 Bitscan

A bitscan is used to find the index of the LS1B. Many implementation have been devised since the advent of bitboards[16].

For our purpose, we will use the 'De Bruijn Multiplication', a minimal perfect hashing to determine the LS1B index in a 64-bit number[7]:

```
const int index64[64] = {
   63,  0, 58,  1, 59, 47, 53,  2,
   60, 39, 48, 27, 54, 33, 42,  3,
   61, 51, 37, 40, 49, 18, 28, 20,
   55, 30, 34, 11, 43, 14, 22,  4,
   62, 57, 46, 52, 38, 26, 32, 41,
   50, 36, 17, 19, 29, 10, 13, 21,
   56, 45, 25, 31, 35, 16,  9, 12,
   44, 24, 15,  8, 23,  7,  6,  5
};

// De Bruijn Multiplication.
const bitboard debruijn64 = 0x07EDD5E59A4E28C2ULL;

int bitscan(bitboard LS1B)
   return index64[(LS1B * debruijn64) >> 58];
```

Here is the explanation of how it works from Chess Programming Wiki[17]:

> A 64-bit De Bruijn sequence contains 64-overlapped unique 6-bit sequences, thus a circle of 64 bits, where five leading zeros overlap five hidden 'trailing' zeros. There are $2^{26} = 67108864$ odd sequences with 6 leading binary zeros and 2^{26} even sequences with 5 leading binary zeros, which may be calculated from the odd ones by shifting left one. A multiplication with a power of two value (the isolated LS1B) acts like a left shift by it's exponent. Thus, if we multiply a 64-bit De Bruijn sequence with the isolated LS1B, we get a unique six bit subsequence inside the most significant bits. To obtain the bit-index we need to extract these upper six bits by shifting right the product, to lookup an array.

4.5 General Technical Advantages and Disadvantages

4.5.1 Processor Use

Pros

The bitboard 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, and XOR.

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 [21].

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 [21].

4.5.2 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 [21].

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 is not an issue [21].

4.6 Bitboard in Arimaa

In this section, we will show how we will represent a bitboard in Arimaa and how we will model the Arimaa rules using the bitboards.

4.6.1 Representation

A bitboard is an unsigned integer of 64 bits. It can be represented in many different ways; for example, let's consider the bitboard of silver rabbits in Figure 4.2.

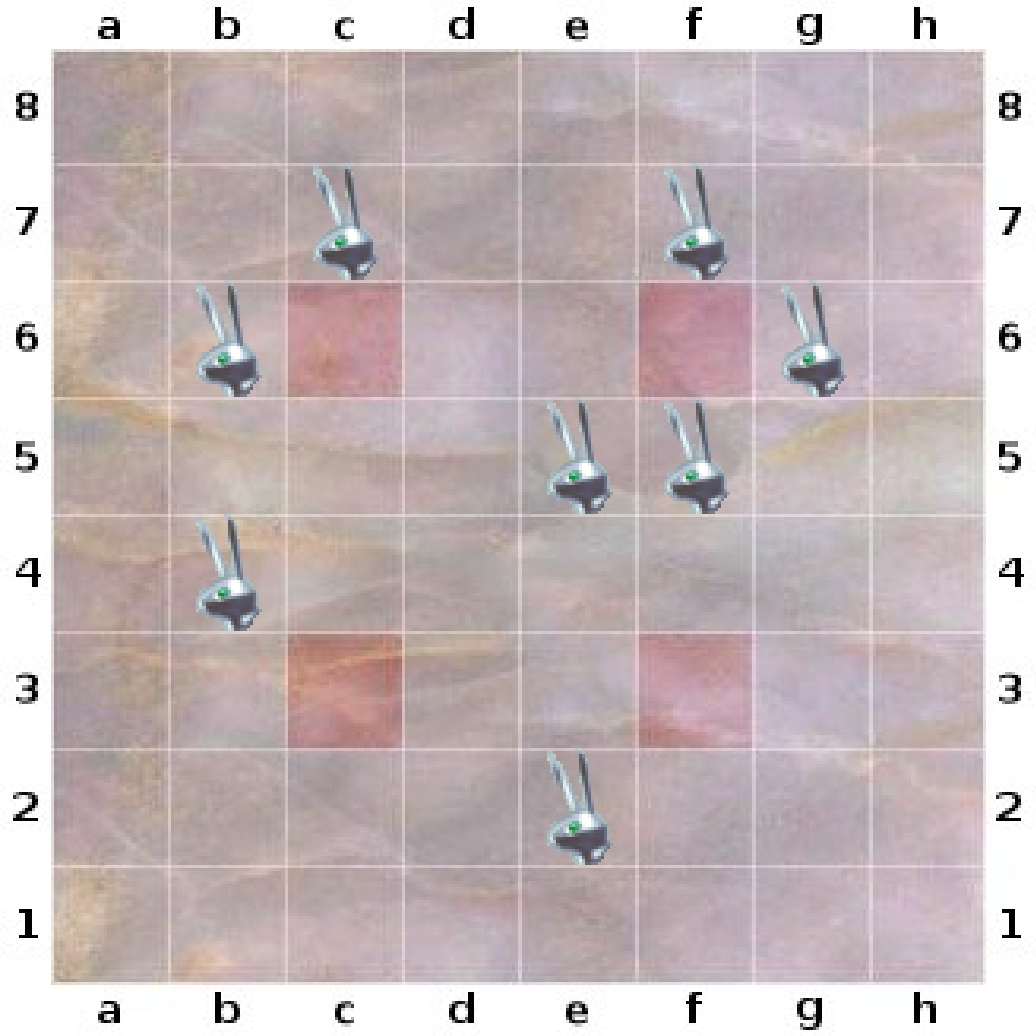

Figure 4.2: Silver Rabbit Bitboard

We could represent it as a binary number:

```
0000000000100100010000100000110001000000000000000000100000000000
```

However it's not very readable. We could use hexadecimal notation to make the line shorter:

```
0x0024420C400800
```

But it is still not easy to understand at first glance what is what. For this reason, we could format the binary number in to 8 rows of 8 bits each, like the board:

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

The last one is much better, but it is still difficult to recognize immediately the row and column of a certain bit. For this reason, we will use the Arimaa notation, as presented in section 2.7:

```
bitboard SR: // Silver Rabbit.
 +-----------------+
8|                 |
7|     r     r     |
6|   r x     x r   |
5|         r r     |
4|   r             |
3|     x     x     |
2|         r       |
1|                 |
 +-----------------+
   a b c d e f g h
```

4.6.2 Bitboard Movement

A bitboard makes easy to obtain the possible square near a piece, using the bit shift function:

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

bitboard NEAR(GE) =   // GE movement squares.
```

```
  return ((GE << 8) | // the Nort square.
          (GE << 1) | // the East square.
          (GE >> 1) | // the West square.
          (GE >> 8)); // the Sout square.
```

In Figure 4.3(a) we see the result of the previous function. Unfortunately, when a piece lies on the first or last column, it will have some weird behavior, as shown in Figure 4.3(b) and Figure 4.3(c).

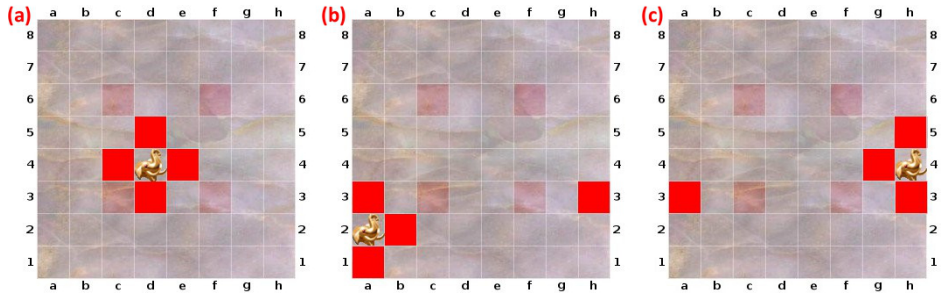

Figure 4.3: Finding Gold Elephant steps

To avoid this, first we have to define two new constants, the column A and the column H as follows:

```
static const bitboard COLUMN_H =
   1ULL         | (1ULL << 8)  | (1ULL << 16) | (1ULL << 24) |
  (1ULL << 32) | (1ULL << 40) | (1ULL << 48) | (1ULL << 56) ;

static const bitboard COLUMN_A =
  (1ULL << 7)  | (1ULL << 15) | (1ULL << 23) | (1ULL << 31) |
  (1ULL << 39) | (1ULL << 47) | (1ULL << 55) | (1ULL << 63) ;
```

```
bitboard COLUMN_H:
 +-----------------+
8|               * |
7|               * |
6|     x     x   * |
5|               * |
4|               * |
3|     x     x   * |
2|               * |
1|               * |
 +-----------------+
   a b c d e f g h
```

```
bitboard COLUMN_A:
 +-----------------+
8| *               |
7| *               |
6| *   x     x     |
5| *               |
4| *               |
3| *   x     x     |
2| *               |
1| *               |
 +-----------------+
   a b c d e f g h
```

And then modify the $NEAR$ function as follows:

```
bitboard  NORT(bitboard b) {return ((b) << 8);}
bitboard  EAST(bitboard b) {return ((b & ~COLUMN_H) << 1);}
bitboard  WEST(bitboard b) {return ((b & ~COLUMN_A) >> 1);}
bitboard  SOUT(bitboard b) {return ((b) >> 8);}

bitboard  NEAR(bitboard b) {return ((NORT(b)) | (EAST(b)) |
                                   (WEST(b)) | (SOUT(b)));}
```

We did not define any restriction to the northern and southern square because it is 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 the output of the $NEAR$ function.

```
bitboard GC: //Gold Cat.      bitboard NEAR(GC):
 +-----------------+           +-----------------+
8|                 |          8|                 |
7|                 |          7|                 |
6|     x     x     |          6|     x     x     |
5|                 |          5|                 |
4|                 |          4|               * |
3|     x     x   C |          3|     x     x *   |
2|                 |          2|   *           * |
1|   C             |          1| *   *           |
 +-----------------+           +-----------------+
   a b c d e f g h               a b c d e f g h
```

All the pieces can generate their movement using the $NEAR$ function, except the rabbits that cannot move backwards. For this reason they use a modified version of the $NEAR$ function:

```
bitboard  GR_NEAR(bitboard b) // Gold Rabbits cannot move NORT.
   {return ((EAST(b)) | (WEST(b)) | (SOUT(b)));}

bitboard  SR_NEAR(bitboard b) // Silver Rabbits cannot move SOUT.
   {return ((NORT(b)) | (EAST(b)) | (WEST(b)));}
```

Finally, a piece can move only in an empty space. If we indicate as $EMPTY$ the bitboard containing all the empty squares on the board, the one step movements of a piece in the bitboard b, are obtained as:

```
bitboard one_step = NEAR(bitboard b) & EMPTY;
```

4.6.3 Bitboard Freezing

In section 2.4.3, we reported an important Arimaa Rule:

> When a weaker piece is next to a stronger enemy piece, it becomes frozen and cannot move unless there is a friendly piece next to it.

To represent this rule, we have to define two new functions:

```
// Returns the bitboard of all the pieces of one 'color'.
bitboard GROUP(boolean color);

// Returns the bitboard of all the pieces of one 'color'
// limited to the pieces stronger than 'piece'.
bitboard STRONGER(boolean color, int piece);
```

In this way we can indicate the frozen pieces as:

```
// Returns the bitboard of frozen pieces of a piece with
// color value of 'color' and piece value of 'piece'.
bitboard FROZEN(bitboard b)
           // All the pieces near a stronger enemy.
   {return ((NEAR(STRONGER(!color, piece)) & b)
           // All the pieces without an ally near them.
            & ~NEAR(GROUP(color)));}
```

Consequently, the not frozen pieces, and the modified one step movements, are:

```
// Return the not frozen pieces.
bitboard NOT_FROZEN(bitboard b)
   {return (b & ~FROZEN(bitboard b));}

// The modified one_step movement bitboard.
bitboard one_step = NEAR(NOT_FROZEN(bitboard b)) & EMPTY;
```

For example, let's consider what are the one step movements of the Golden Cat in the following board:

```
// 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 f g h
}

bitboard GC; // The Golden Cat.

// The silver pieces stronger than a cat.
bitboard stronger = STRONGER(SILVER, CAT);
// The stronger pieces near the gold cat.
bitboard near_stronger = NEAR(stronger) & GC;

bitboard stronger:            bitboard near_stronger:
 +-----------------+           +-----------------+
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 f g h               a b c d e f g h

// All the gold pieces on the board.
bitboard gold = GROUP(GOLD);
```

```
bitboard gold:                bitboard NEAR(gold):
 +-----------------+           +-----------------+
8|                 |          8|                 |
7|                 |          7|               * |
6|     x     x   * |          6| *   x     x *   |
5| *               |          5|   *           * |
4|                 |          4| *       *       |
3|     x   * x     |          3|     x *   * *   |
2|             *   |          2|   * *   * * * * |
1|   * *     * *   |          1| * * * * * * * * |
 +-----------------+           +-----------------+
   a b c d e f g h               a b c d e f g h
```

```
// The frozen golden cat.
bitboard frozen = near_stronger & ~NEAR(gold);
// The not frozen golden cat.
bitboard not_frozen = GC & ~frozen;
```

```
bitboard frozen:              bitboard not_frozen:
 +-----------------+           +-----------------+
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 f g h               a b c d e f g h
```

```
// The empty spaces the not frozen golden cat can move.
bitboard one_step = NEAR(not_frozen) & EMPTY;
```

```
bitboard EMPTY:              bitboard one_step:
 +-----------------+          +-----------------+
8| * * * * *   * * |         8|                 |
7|   * * * * * * * |         7|                 |
6|   * * * * * *   |         6|     x     x     |
5|   * * * * * * * |         5|                 |
4| * *   *   * * * |         4|                 |
3| * * * *   * * * |         3|     x     x     |
2| *   * * * *   * |         2|                 |
1| *     * *     * |         1| *               |
 +-----------------+          +-----------------+
   a b c d e f g h              a b c d e f g h
```

From this long example, we see how we can easily obtain all the legal one step movements for a piece represented with a bitboard. The code is similar for all pieces, except for the Elephant, which cannot become frozen, as there are not stronger pieces. For this reason, for the Elephant we use the simpler one step movement generator:

```
bitboard one_step = NEAR(bitboard b) & EMPTY;
```

4.6.4 Two Step Moves

There are two types of moves that give the possibility to move an enemy piece. Those are called *pull* and *push*, and count as two steps. For this reason, we will refer to them as two step moves.

First of all, a piece can move only a weaker piece, so we need a function to select only weaker enemy pieces:

```
// Returns the bitboard of all the pieces of one 'color'
// limited to the pieces weaker than 'piece'.
bitboard WEAKER(boolean color, int piece);

// All the weaker enemy pieces near the current bitboard with
// color value of 'color' and piece value of 'piece'.
bitboard two_step = NEAR(bitboard b) & WEAKER(!color, piece);
```

Whit this function, it is easy to represent all the two step moves.

Pull

In a pull move, a piece will move as it was doing a one step move, and the enemy weaker piece will move in its location. This move is feasible only if the pulling piece have an empty square near it.

For example, let's see how the pull move works for a Gold Horse:

```
// 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 f g h

bitboard GH; // The Golden Horse.

// The silver pieces weaker than a horse.
bitboard weaker = WEAKER(SILVER, HORSE);
//
bitboard two_step = NEAR(GH) & weaker;

bitboard weaker:              bitboard two_step:
 +-----------------+           +-----------------+
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 f g h               a b c d e f g h
```

```
// The pulling piece.
bitboard pulling = NOT_FROZEN(GH) & NEAR(two_step);
// The empty squares the pulling piece can move to.
bitboard one_step = NEAR(not_frozen) & EMPTY;
```

```
bitboard pulling:          bitboard one_step:
 +-----------------+          +-----------------+
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 f g h              a b c d e f g h
```

Push

In a pull move, a piece will take the place of a weaker enemy, and the enemy will be moved in an empty square near it.

For example, let's see how the push move works for a Gold Horse:

```
// 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 f g h

bitboard GH; // The Golden Horse.
```

```
// The silver pieces weaker than a horse.
bitboard weaker = WEAKER(SILVER, HORSE);
//
bitboard two_step = NEAR(GH) & weaker;
```

```
bitboard weaker:
 +-----------------+
8|   *       *     |
7| *   *           |
6| *   x     x   * |
5|                 |
4|                 |
3|     x     x     |
2|                 |
1|                 |
 +-----------------+
   a b c d e f g h
```

```
bitboard two_step:
 +-----------------+
8|                 |
7|                 |
6|     x     x   * |
5|                 |
4|                 |
3|     x     x     |
2|                 |
1|                 |
 +-----------------+
   a b c d e f g h
```

```
// The pushing piece.
bitboard pushing = NOT_FROZEN(GH) & NEAR(two_step);
// The empty squares the pushed piece can move to.
bitboard push_full = NEAR(two_step) & EMPTY;
```

```
bitboard pushing:
 +-----------------+
8|                 |
7|                 |
6|     x     x     |
5|               * |
4|                 |
3|     x     x     |
2|                 |
1|                 |
 +-----------------+
   a b c d e f g h
```

```
bitboard push_full:
 +-----------------+
8|                 |
7|               * |
6|     x     x *   |
5|                 |
4|                 |
3|     x     x     |
2|                 |
1|                 |
 +-----------------+
   a b c d e f g h
```

4.6.5 Bitboard Trapping

At the end of each step, if a piece is over a trap and does not have any ally near it, it will be removed from the play.

To find and remove the trapped pieces, first we have to define the location of the traps as a new constant:

```
static const bitboard TRAP_SQUARES = 0x0000240000240000ULL;
```

```
bitboard TRAP_SQUARES:
 +-----------------+
8|                 |
7|                 |
6|     *     *     |
5|                 |
4|                 |
3|     *     *     |
2|                 |
1|                 |
 +-----------------+
   a b c d e f g h
```

Then we can remove any piece on a trap and not near a piece of the same color; the following function has to be checked for each bitboard.

```
// Check for every bitboard 'b' of color 'color'.
bitboard trapped_piece =
   // Pieces over a trap square.
   (b & TRAP_SQUARES) &
   // Pieces not near an ally.
   ~(NEAR(GROUP(color)))

// Remove the trapped piece from the bitboard.
b &= ~trapped_piece;
```

Let's see how the trapping function works for the Gold Dog in the following example:

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

bitboard GD; // The Golden Dog.

// The bitboard of the dogs in danger.
bitboard piece_in_danger = GD & TRAP_SQUARES;
// The bitboard of all the squares near an ally.
bitboard safe_places = NEAR(GROUP(GOLD));

bitboard piece_in_danger:     bitboard safe_places:
 +-----------------+           +-----------------+
8|                 |          8|                 |
7|                 |          7|     *           |
6|     *     x     |          6| * * x *   x     |
5|                 |          5|   * *           |
4|                 |          4| *       * *     |
3|     x     *     |          3|     x * * * *   |
2|                 |          2|   * *   * * * * |
1|                 |          1| * * * * * * * * |
 +-----------------+           +-----------------+
   a b c d e f g h               a b c d e f g h

// The bitboard of the dog in trap.
bitboard trapped_piece = piece_in_danger & ~safe_places;
// The updated golden dog bitboard.
GD &= ~trapped_piece;
```

```
bitboard trapped_piece:
 +-----------------+
8|                 |
7|                 |
6|     *     x     |
5|                 |
4|                 |
3|     x     x     |
2|                 |
1|                 |
 +-----------------+
   a b c d e f g h
```

```
bitboard GD:
 +-----------------+
8|                 |
7|                 |
6|     x     x     |
5|                 |
4|                 |
3|     x     *     |
2|                 |
1|                 |
 +-----------------+
   a b c d e f g h
```

4.6.6 Bitboard Goal

A player wins when one of his rabbit reaches the opposite side. This is the goal the computer player has to reach, and it is easy to implement it using the bitboards.

First we have to define two new constants; the 1st row and the 8th one:

```
static const bitboard ROW_1 = 0x00000000000000FFULL;
static const bitboard ROW_8 = 0xFF00000000000000ULL;
```

```
ROW_1:
 +-----------------+
8|                 |
7|                 |
6|     x     x     |
5|                 |
4|                 |
3|     x     x     |
2|                 |
1| * * * * * * * * |
 +-----------------+
   a b c d e f g h
```

```
ROW_8:
 +-----------------+
8| * * * * * * * * |
7|                 |
6|     x     x     |
5|                 |
4|                 |
3|     x     x     |
2|                 |
1|                 |
 +-----------------+
   a b c d e f g h
```

Then we can define the victory situation for each player:

```
bitboard GR; // The Golden Rabbit.
```

```
if (GR & ROW_8)
   // GOLD WON!

bitboard SR; // The Silver Rabbit.
if (SR & ROW_0)
   // SILVER WON!
```

The example is really straightforward:

```
// Board situation:
 +-----------------+
8|   c     R 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 f g h

bitboard GR; // The Golden Rabbit.

bitboard GR:                  GR & ROW_8:
 +-----------------+           +-----------------+
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 f g h               a b c d e f g h

// if (GR & ROW_8) contains at least one '1', gold player won.
```

Chapter 5

Implementation

In this chapter, we will see how we implemented the bitboard knowledge representation system with problem solving techniques.

Fritz Reul, in his Ph.D thesis 'New Architectures in Computer Chess'[10], showed how, in Chess, his array based architecture (LOOP LEIDEN) is faster than the Rotated Bitboards one; in particular, in a 32-bit mode, LOOP LEIDEN is from 32% to 87% faster than the Rotated Bitboards, while in 64-bit mode it is only from 3% to 5% faster.

In Chess, the bitboard architecture uses several techniques to generate faster the movement of the pieces, and in particular of the sliding one; those techniques include flipping, mirroring and rotating the bitboard. In Arimaa we do not need of those techniques, as every piece is moved of only one step at a time. For this reason, we had the intuition that a bitboard architecture would perform better than an array based one.

5.1 The Test

To control the performance of the engines, we decided to use a brute-force approach. In this type of approach, we perform a depth-limited search, as described in section 3.3.3, count the number of nodes visited and divide it by the time spent in searching (in seconds). In this way, we obtain a new unit

of measurement called 'Nodes Per Second' (NPS):

$$NPS = \frac{nodes}{time(s)}$$

The NPS indicates the speed an engine can reach: bigger is this value, better is the engine. For our purposes, we will use the kNPS, indicating 1'000 NPS.

To test the engines, we used a real board situation; the board we used, is in appendix A. All the tests were made on the same machine, equipped with (in brackets the relative Windows 7 rate):

- OS : Windows 7 Professional 64-bit.
- CPU: i7 720QM. (6.9)
- RAM: 4GB DDR3 1333Mhz. (7.4)
- HD : 500GB 5'400rpm. (5.9)
- GPU: GTX 280M 1GB GDDR3. (6.8)
- Visual Studio 2008 Professional Edition.

The tests were made turning off all the programs, and were repeated at least five times, discarding the highest and lowest values and then, calculating the average. This was necessary because of the i7 'Turbo' feature, that automatically overclocks the CPU when needed.

5.2 The Array Based Engine

The array based engine was found on Arimaa website. We chosen 'botFairy' (light version) developed by Ola Hannson and written in *C* [4]. Considering that we wanted to use a brute-force approach, without using any evaluation function and consequently, any kind of pruning (like the alpha-beta pruning). We decided to modify the 'Fairy_light' version disabling those parts of the program.

Even the testing board we used (showed in appendix A) is one of the 'Fairy_full' test boards.

The modified version of 'botFairy' has been proven to perform an average of 1'755 kNPS, with a peak of 1'794 kNPS. Our goal was to build a bitboard engine capable to beat those results.

5.3 The Bitboard Based Engine

To build an Arimaa bitboard based engine, we decided to not start from scratch, but to take a Chess engine based on bitboard, and modify it. We chose 'Gray Matter' [1]; this Chess engine is written in $C++$ with readable, understandable and hackable code.

We eliminated the useless parts of the code, like the rotated bitboards, and modified most of the rest. Our Arimaa engine includes roughly 5%-10% of the original Gray Matter code.

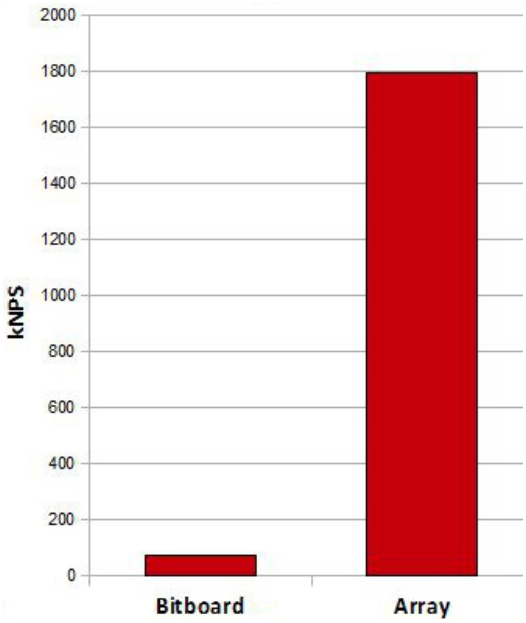

Figure 5.1: Bitboard and Array Engine Compared Speed at Beginning

After modifying the code, implementing the bitboard representation and rules presented in section 4.6, we started to test our engine. The results were disappointing, as it was capable to perform only 76 kNPS. We were not expecting such bad results, considering that the array based engine was more than 200 times better(see Figure 5.1).

Thus, we decided to optimize the code. In the following section, we will discuss only few of the many code improvement we made.

5.3.1 Pre-Generated Array of Moves

The main recursive function was structured in five main parts:

1. *Create* an array to save the possible moves.
2. *Generate* the available moves.
3. *Make* one of the moves previously generated.
4. *Call* the recursive function (with decreased depth) and restart from point 1 if the depth is bigger than 0.
5. *Unmake* the previously move.

Then, we measured the percentage of time spent on the following three main parts: 'Create', 'Generate' and 'Make/Unmake' (together). Generating a move was taking 7.30% of the total time, Making/Unmaking them 12.45%, and Creating the array 80.25% (see Figure 5.2).

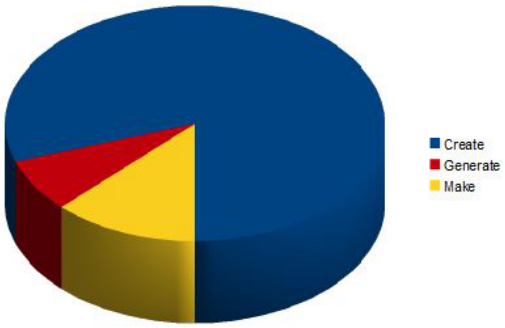

Figure 5.2: Create, Generate and Make(/Unmake) in Percentage

Consequently, we decided to work first on the 'Create' part. We decided to create all the necessaries arrays at program startup, and to populate them when needed. This boosted the speed of the program by ten times, reaching a speed of 780 kNPS (see Figure 5.3).

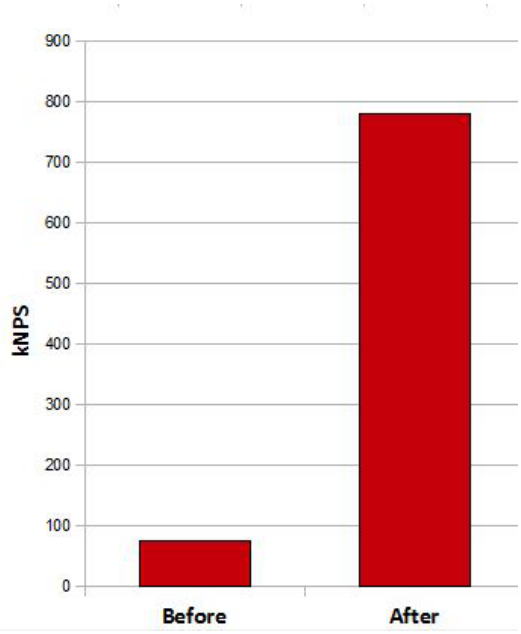

Figure 5.3: Before and After Pre-Generated Arrays

5.3.2 Make Improvement

Thus, we eliminated one of the parts of the main recursive function, gaining a considerable performance boost, but we were far from approaching a speed comparable with the array based engine. For this reason, we started to analyze the second biggest cause of slowness; in fact, generating the moves was taking 37,95% of the processing time, while making/unmaking them the complementary 62,05%.

The code in this part was incredibly chaotic and dirty, so we decided first to modify the structure of the 'Move' class, where we were saving a generated move, as follows:

```
/// Move class.

bitboard np;        ///< Piece ON_MOVE move.
bitboard ne;        ///< Piece OFF_MOVE move [=0 if 1step].
unsigned piece : 3; ///< Piece ON_MOVE id [0-5].
unsigned enemy : 3; ///< Enemy piece (OFF_MOVE) id [0-4].
```

The 3-bit *piece* variable contains the 'on move' piece ID (from $RABBIT = 0$ to $ELEPHANT = 5$), while the 3-bit *enemy* variable contains the pushed/pulled

enemy ID (from $RABBIT = 0$ to $CAMEL = 4$). Just saving the pieces ID in the 'Move' class gave an average performance boost of 200 kNPS.

The bitboard *np* contains the piece 'on move' current position and the position he will be after the movement. The bitboard *ne* is similar to *np*, but contains the positions of the pushed/pulled enemy piece, or is empty if it is a one step move.

Consequently, we were able to write a make function very clean and ordered:

```
/// Make a move.

// ON_MOVE piece.
state.piece[ON_MOVE][m.piece] ^= m.np;
// OFF_MOVE piece.
if (m.ne)  // not empty.
   state.piece[OFF_MOVE][m.enemy] ^= m.ne;


/// Update the number of steps.
if(!m.ne)  /// 1 step move.
   state.steps ++;
else       /// 2 step move.
   state.steps += 2;


/// Check the number of steps.
if(state.steps == 0)
{
   // Set the other color on move.
   state.on_move = !state.on_move;
   // Update number of moves.
   state.number_moves++;
}
```

In fact, the XOR (^) operation works as a piece shifter; for example:

```
// Given the
bitboard GC: //Gold Cat.
 +-----------------+
8|                 |
7|                 |
6|     x     x     |
5|                 |
4|                 |
3|     x     x   C |
2|                 |
1|   C             |
 +-----------------+
   a b c d e f g h

bitboard np:                bitboard GC ^= np:
 +-----------------+         +-----------------+
8|                 |        8|                 |
7|                 |        7|                 |
6|     x     x     |        6|     x     x     |
5|                 |        5|                 |
4|                 |        4|                 |
3|     x     x     |        3|     x     x   C |
2|   *             |        2|   C             |
1|   *             |        1|                 |
 +-----------------+         +-----------------+
   a b c d e f g h             a b c d e f g h
```

The above code is the final step of many intermediate trials, where only some of them produced a real speedup in the engine. At this stage, our engine was able to reach a speed of 1'698 kNPS: a total increase of 118% (see Figure 5.4).

5.3.3 Generate Improvement

The last part where we were able to make some improvements were the move generator. One of the benefits of the bitboard approach is the possibility to generate all the moves for a set of pieces: for example, we were able

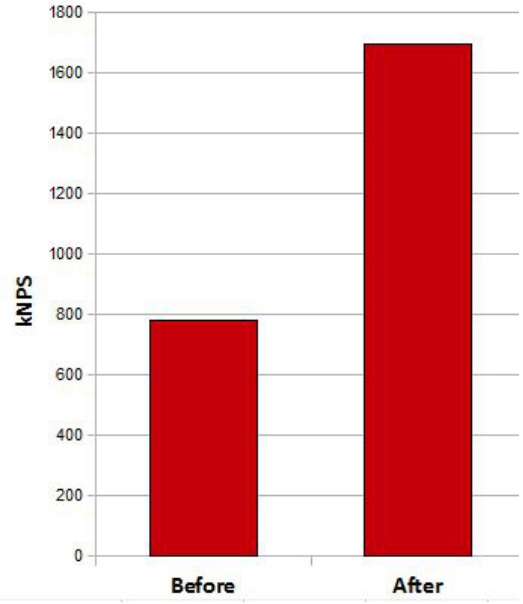

Figure 5.4: Before and After Make Improvements

to generate all the moves of the eight rabbits in one passage, as seen in section 4.6.2:

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

bitboard GR; // The Gold Rabbits.
GR = NOT_FROZEN(GR); // Remove the frozen rabbits.
bitboard one_step = (NORT(GR) | EAST(GR) | WEST(GR)) & EMPTY;
```

```
bitboard one_step:
 +-----------------+
8|                 |
7|                 |
6|     x     x     |
5|                 |
4|                 |
3|     x     x   * |
2|   *         *   |
1|       *   *     |
 +-----------------+
   a b c d e f g h
```

The problems were risen when we had to isolate each move for each piece, saving the initial and final positions. In the above example, we have also a move overlap of the two Rabbits in $B1$ and $A2$, the first going to $NORT$ and second going to $EAST$, and both reach the position $B2$. The most efficient code we created for handling the single moves and those situations is presented in appendix B.

When we wanted to speed up the move generation, we tried a new approach: isolating the pieces before generating the move. The resulting move generator code for the rabbit piece is as follows:

```
// The not frozen Gold Rabbits.
bitboard from = NOT_FROZEN(GR);

// Until 'from' is not empty, do the
// function and, after, remove the
// Last Significant 1 Bit (LSB)
for (; from; from &= (from - 1))
{
   // tempLSB contains only the LSB.
   bitboard tempLSB = LSB(from);

   // Check 1Step moves.
   one_step = (NORT(tempLSB) | EAST(tempLSB) |
               WEST(tempLSB)) & group[EMPTY];

   // Insert the moves if not empty.
```

```
   if(one_step)
      insert(tempLSB, RABBIT, one_step, 0, l);
}
```

The code for inserting the moves is now more clean and fast; the following extract is limited to ONE_STEP movements:

```
// Set the initial piece position.
move.np = from;

// Until 'from' is not empty, do the
// function and, after, remove the
// Last Significant 1 Bit (LSB)
for (; one_step; one_step &= (one_step - 1))
{
   // tempLSB contains only the LSB.
   bitboard tempLSB = LSB(one_step);
   // Add the current final position.
   move.np ^= tempLSB;
   // Add the move to the move array.
   l.addMove(move);
   // Remove the current final position.
   move.np ^= tempLSB;
}
```

This kind of approach gave us also another opportunity: pre-generate all the possible $NEAR(n)$ results from a given n, useful to generate faster the piece movements:

```
for (int n = 0; n < 64; n++)
{
   // Create a bitboard with only one '1' in position n.
   bitboard from = BIT_MSK(n);
   // Create the NEAR_A array.
   NEAR_A[n] = NEAR(from);
   // Create the NEAR_N array (for Gold Rabbits).
   NEAR_N[n] = NORT(from) | WEST(from) | EAST(from);
   // Create the NEAR_S array (for Silver Rabbits.
   NEAR_S[n] = SOUT(from) | WEST(from) | EAST(from);
}
```

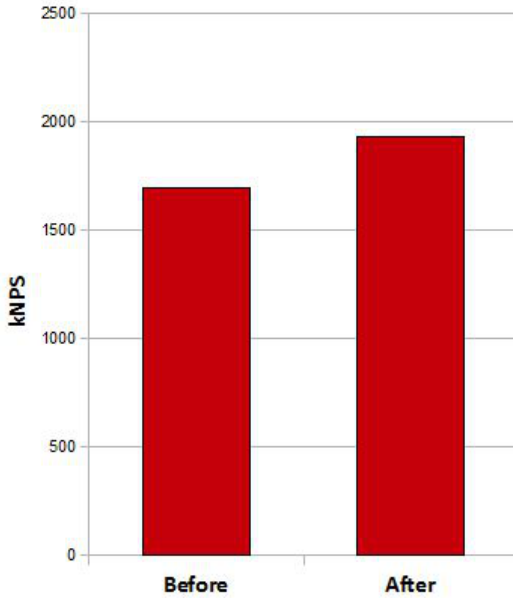

Figure 5.5: Before and After Generate Improvements

Those approaches combined gave a total speedup of 250 kNPS, that was enough to reach a speed of 1'948 kNPS(see Figure 5.5), that makes the bitboard based engine faster than the array based one(see Figure 5.8).

5.3.4 64-bit Platform

This platform was not a real solution, but a mistake corrected. In fact, we noticed that we forgot to run the program with a 64-bit platform. This means that all the previous test were running with a 32-bit platform. As consequences, our bitboard based solution is faster that an array based solution even when running at his worst condition.

Activating the 64-bit platform, we gained a further improvement of 30%(see Figure 5.6), reaching a final speed of 2'525 kNPS.

5.3.5 Optimization Considerations

To sum it up, we were able to develop a bitboard based engine faster than an array based one. In Figure 5.8, we see the speed increase given by each

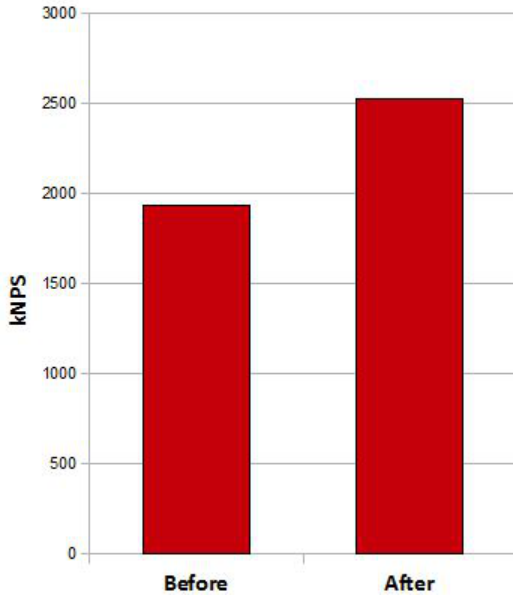

Figure 5.6: Before and After Activating the 64-bit

analyzed optimizationand the compared speed of the bitboard and the array engines.

5.4 Further Improvements

Even if we were able to develop a bitboard engine faster than an array based one, we are not completely satisfied with our program: there are still some parts of the code that do not look good enough, and could perform better. For this reason, we are continuing even now the operations of code polishing and optimizing, implementing new parts and/or modifying the existing ones, whenever we have new ideas.

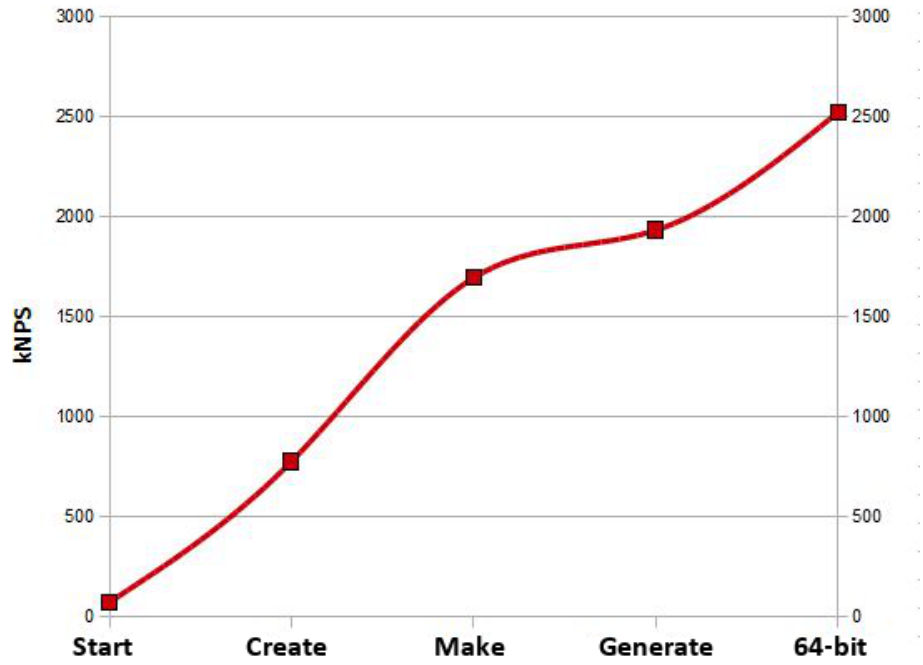

Figure 5.7: Bitboard Engine Compared Improvements

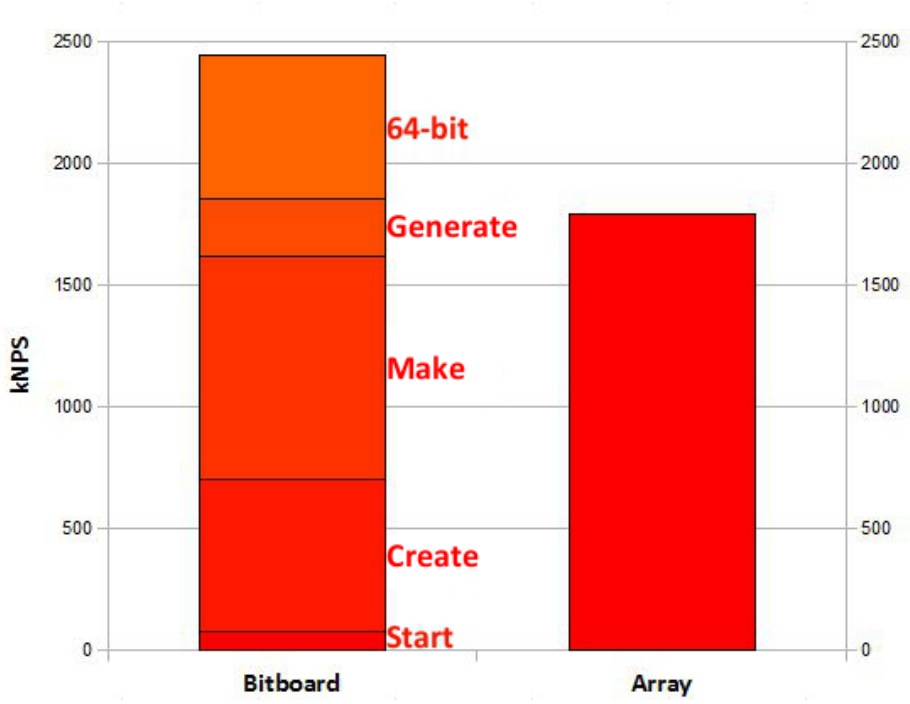

Figure 5.8: Bitboard and Array Engines Compared Speed

Chapter 6

Conclusions

Douglas Richard Hofstadter coined a self-referencing law [6], that represents perfectly the number of goals reached:

> 'It always takes longer than you expect, even when you take into account Hofstadter's Law.'

In this thesis, we have seen the main Problem Solving techniques (used also in *Deep Blue*) and an efficient Knowledge Representation system, called Bitboard; we also illustrated a possible way to represent the Arimaa rules trough bitboards.

We examine the Arimaa engine we developed in relation with the goals we took. Additionally, we propose a possible solution to reach the *second goal*.

6.1 Goals Achieved

We studied most of the literature of traditional abstract game playing techniques and also of AI for video games (Game AI), reaching the *first goal*. As previously remarked, some of the traditional AI game playing techniques are presented in chapters 3 and 4.

The Game AI has developed to be enjoyable for a player, not for being stronger than them. In particular, in First Person Shooters and Role Playing Games, where the player has to defeat hundreds (or thousand) of enemies,

the AI is generally scripted (or pre-determined) and not adaptive. The AI of Strategy Games and Driving Games, in the other hand, is so weak that usually cheats to be challenging for the player. The only example of a learning AI is in pet games like 'Tamagotchi' and 'Black and White' [24], but those are useless for our goal (develop a program capable to beat a professional human player in Arimaa).

The *tangible goal* was to create an Arimaa engine, capable to generate the possible moves, as much fast as possible. As shown in section 5.3, we were able to develop a fast bitboard based Arimaa engine, capable to visit 2.5 million positions per second. Unfortunately, the work on the engine is not concluded, as it needs some enhancements.

6.2 Future Work

As future work for the Arimaa engine, we have to implement a technique to avoid to analyze more than once a move. In fact, during an Arimaa game, is easy to make distinct steps that will produce the same move; for example, if we look at Figure 6.1, the gold Camel in $e2$ could reach with two steps $d3$, going through $e3$ or $d2$. Again, the same camel could return to his original position after two or four steps.

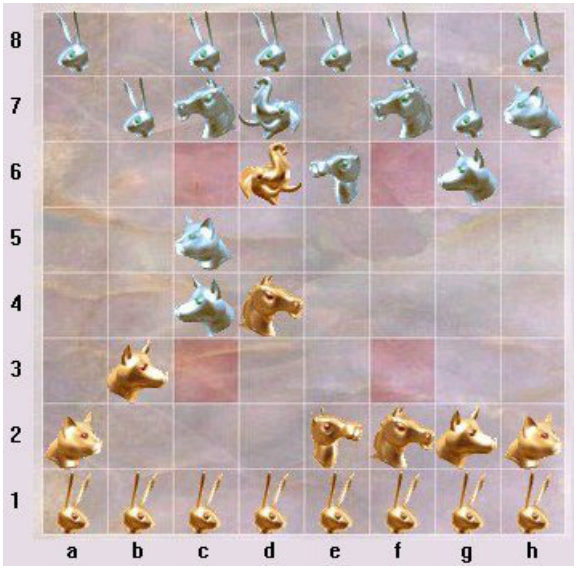

Figure 6.1: Typical Situation

Our engine does not see those moves as the same, but as distinct ones, augmenting exponentially the number of states expanded. One way to avoid this behavior, is to save all the visited states, check if the current one is present in memory, and, if it is, do not expand it. To obtain this, we started to implement an hash function called 'Zobrist Hashing' [30], an hash coding method developed especially for board games. Unfortunately, we could not finish the code implementation in time for the master thesis.

Starting from the work of this thesis it is possible to develop new techniques to allow our program to compete with professional human player.

Our idea to reach the *second goal*, is related to the division between strategy and tactic presented in section 3.6.1. As we stated, traditional AI techniques are great where tactic is prevalent, but for a game like Arimaa we have to rely mainly on strategy. There are some wikibooks about strategy in Arimaa, but they are written for humans (that makes them difficult to be coded for a computer AI) and incomplete (in an AI point of view); for this reason, we would like to use another approach to synthesize the strategies: analyze the past Arimaa matches (taken by humans against humans) using Object Recognition techniques, another AI branch.

When we will have defined the strategies, we will be able to start to study how they are used in games, linking them with a cause-and-effect approach. Then, we will be able to use the strategy map as an evaluation function, but calculated only for the current player move, reducing enormously the search tree.

This approach will be used in conjunction with our Arimaa engine; in fact, the approaches that try to obtain a better result in strategical games like Go (i.e. the pseudo-random 'Monte Carlo method' [25]), suffer in tactical situation. In our program, traditional problem solving techniques are used only to search for captures or goal reaching, but only for one move ahead.

Additionally, using our engine, we can test if a specific move presents weaknesses; this can be done visiting the possible successive moves of the opponent, and nullify the move if it leads to an opponent's rabbit reaching the goal, or an allied piece being captured. The moves to be tested will be independent one from another, and, therefore, they will be able to be examined in parallel; that will give the possibility to use different threads for each move, resulting in a full usage of multi-core architectures.

In fact, in our tests, the CPU usage of the Arimaa engine is only 12-13% of the maximum avalaible; our machine has four cores, executes two threads

per core, and, since our engine uses only $\frac{1}{8}$ of the maximum speed, it uses only one thread. A way to fully use a multi-core architecture is to introduce the parallel move examination above mentioned.

Appendix A

Testing Board

The following board situation is the one we used to test the speed of the engines. 65 indicates the number of the moves, *b* indicates it is black (silver) turn to move.

This board was taken from the testing position of the program 'botFairy' developed by Ola Hannson; this board is named 't001' in 'Fairy_full' folder [4].

```
65b
 +-----------------+
8| r r r   r   r r |
7| r d d     c     |
6| m               |
5|   E             |
4| H           D   |
3| h D c e     C   |
2| R   M     C   R |
1| R R R       R R |
 +-----------------+
   a b c d e f g h
```

Appendix B

Old Insert Code

The following code is the old version of the insert code. The purpose of this code was to isolate the moves(see section 5.3.3) generated with a bitboard (described in section 4.6.2).

This code was the last version of the move isolation code we developed, and even if it is a little scary, too much long and fairly ugly, it was proven to be relatively efficient.

```
// Check for 1step moves.
if(one_step)
{
   move.enemy = ELEPHANT;
   // Check for every direction.
   if(GET_N(one_step, n))
   {
      move.oneStep = N;
      l.addMove(move);
   }
   if(GET_E(one_step, n))
   {
      move.oneStep = E;
      l.addMove(move);
   }
   if(GET_W(one_step, n))
   {
      move.oneStep = W;
```

```
      l.addMove(move);
   }
   if(GET_S(one_step, n))
   {
      move.oneStep = S;
      l.addMove(move);
   }
}

// Check for 2step moves: PULL
if(one_step && two_step)
{
   move.type = PULL_STEP;
   // Check for every direction.
   if(GET_N(two_step, n))
   {
      move.twoStep = N;
      // Find Enemy piece.
      for (int shape = RABBIT; shape < piece; shape++)
      {
         if (BIT_GET(state.piece[OFF_MOVE][shape], n+8))
         {
            move.enemy = shape;
            break;
         }
      }
      // Check for the 3 remaining directions.
      if(GET_E(one_step, n))
      {
         move.oneStep = E;
         l.addMove(move);
      }
      if(GET_W(one_step, n))
      {
         move.oneStep = W;
         l.addMove(move);
      }
      if(GET_S(one_step, n))
      {
         move.oneStep = S;
         l.addMove(move);
```

```
   }
}
if(GET_E(two_step, n))
{
   move.twoStep = E;
   //Find Enemy piece.
   for (int shape = RABBIT; shape < piece; shape++)
   {
      if (BIT_GET(state.piece[OFF_MOVE][shape], n+1))
      {
         move.enemy = shape;
         break;
      }
   }
   // Check for the 3 remaining directions.
   if(GET_N(one_step, n))
   {
      move.oneStep = N;
      l.addMove(move);
   }
   if(GET_W(one_step, n))
   {
      move.oneStep = W;
      l.addMove(move);
   }
   if(GET_S(one_step, n))
   {
      move.oneStep = S;
      l.addMove(move);
   }
}
if(GET_W(two_step, n))
{
   move.twoStep = W;
   // Find Enemy piece.
   for (int shape = RABBIT; shape < piece; shape++)
   {
      if (BIT_GET(state.piece[OFF_MOVE][shape], n-1))
      {
         move.enemy = shape;
         break;
```

```
      }
   }
   // Check for the 3 remaining directions.
   if(GET_N(one_step, n))
   {
      move.oneStep = N;
      l.addMove(move);
   }
   if(GET_E(one_step, n))
   {
      move.oneStep = E;
      l.addMove(move);
   }
   if(GET_S(one_step, n))
   {
      move.oneStep = S;
      l.addMove(move);
   }
}
if(GET_S(two_step, n))
{
   move.twoStep = S;
   // Find Enemy piece.
   for (int shape = RABBIT; shape < piece; shape++)
   {
      if (BIT_GET(state.piece[OFF_MOVE][shape], n-8))
      {
         move.enemy = shape;
         break;
      }
   }
   // Check for the 3 remaining directions.
   if(GET_N(one_step, n))
   {
      move.oneStep = N;
      l.addMove(move);
   }
   if(GET_E(one_step, n))
   {
      move.oneStep = E;
      l.addMove(move);
```

```
      }
      if(GET_W(one_step, n))
      {
         move.oneStep = W;
         l.addMove(move);
      }
   }
}

// Check 2Step moves: PUSH.
if(pushFull && two_step)
{
   move.type = PUSH_STEP;
   // Check for every direction.
   if(GET_N(two_step, n))
   {
      move.oneStep = N;
      // Find Enemy piece.
      for (int shape = RABBIT; shape < piece; shape++)
      {
         if (BIT_GET(state.piece[OFF_MOVE][shape], n+8))
         {
            move.enemy = shape;
            break;
         }
      }
      // Check for the 3 remaining directions.
      if(GET_N(pushFull, n+8))
      {
         move.twoStep = N;
         l.addMove(move);
      }
      if(GET_E(pushFull, n+8))
      {
         move.twoStep = E;
         l.addMove(move);
      }
      if(GET_W(pushFull, n+8))
      {
         move.twoStep = W;
         l.addMove(move);
```

```
   }
}
if(GET_E(two_step, n))
{
   move.oneStep = E;
   // Find Enemy piece.
   for (int shape = RABBIT; shape < piece; shape++)
   {
      if (BIT_GET(state.piece[OFF_MOVE][shape], n+1))
      {
         move.enemy = shape;
         break;
      }
   }
   // Check for the 3 remaining directions.
   if(GET_N(pushFull, n+1))
   {
      move.twoStep = N;
      l.addMove(move);
   }
   if(GET_E(pushFull, n+1))
   {
      move.twoStep = E;
      l.addMove(move);
   }
   if(GET_S(pushFull, n+1))
   {
      move.twoStep = S;
      l.addMove(move);
   }
}
if(GET_W(two_step, n))
{
   move.oneStep = W;
   // Find Enemy piece.
   for (int shape = RABBIT; shape < piece; shape++)
   {
      if (BIT_GET(state.piece[OFF_MOVE][shape], n-1))
      {
         move.enemy = shape;
         break;
```

```
      }
   }
   // Check for the 3 remaining directions.
   if(GET_N(pushFull, n-1))
   {
      move.twoStep = N;
      l.addMove(move);
   }
   if(GET_W(pushFull, n-1))
   {
      move.twoStep = W;
      l.addMove(move);
   }
   if(GET_S(pushFull, n-1))
   {
      move.twoStep = S;
      l.addMove(move);
   }
}
if(GET_S(two_step, n))
{
   move.oneStep = S;
   // Find Enemy piece.
   for (int shape = RABBIT; shape < piece; shape++)
   {
      if (BIT_GET(state.piece[OFF_MOVE][shape], n-8))
      {
         move.enemy = shape;
         break;
      }
   }
   // Check for the 3 remaining directions.
   if(GET_E(pushFull, n-8))
   {
      move.twoStep = E;
      l.addMove(move);
   }
   if(GET_W(pushFull, n-8))
   {
      move.twoStep = W;
      l.addMove(move);
```

```
        }
        if(GET_S(pushFull, n-8))
        {
            move.twoStep = S;
            l.addMove(move);
        }
    }
}
```

Bibliography

[1] Gray matter. http://code.google.com/p/gray-matter/.

[2] Samuel Bowles. *Microeconomics: Behavior, Institutions, and Evolution*. Princeton University Press, 2004.

[3] Z-Man Games. Arimaa rules. http://www.zmangames.com/boardgames/files/arimaa/Arimaa_Rules.pdf.

[4] Ola Hansson. botfairy. http://www.arimaa.com/arimaa/download/botFairy/botFairy.zip.

[5] Brian "Janzert" Haskin. A look at the arimaa branching factor). http://arimaa.janzert.com/bf_study/, 2006.

[6] Douglas Richard Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1979.

[7] Charles E. Leiserson, Harald Prokop, and Keith H. Randall. Using de bruijn sequences to index a 1 in a computer world. Technical report, MIT Laboratory for Computer Science, 1998.

[8] DOD Dictionary of Military and Associated Terms. Tactical level of war. http://www.dtic.mil/doctrine/dod_dictionary/data/t/7465.html.

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

[10] Fritz M. H. Reul. *New Architectures in Computer Chess*. PhD thesis, Tilburg University, June 2009.

[11] Stuart J. Russel and Peter Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall, second edition, 2003.

[12] Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(314), 1950.

[13] Omar Syed. Arimaa tutorial(flash version). http://www.arimaa.com/arimaa/learn/flash/jc/new/.

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

[15] L. C. Thomas. *Games, Theory and Applications*. Dover Publication, 2003.

[16] Chess Programming Wiki. Bitscan. http://chessprogramming.wikispaces.com/BitScan.

[17] Chess Programming Wiki. De bruijn multiplication. http://chessprogramming.wikispaces.com/BitScan#Bitscanforward-DeBruijnMultiplication.

[18] Chess Programming Wiki. Least significant one. http://chessprogramming.wikispaces.com/General+Setwise+Operations#TheLeastSignificantOneBitLS1B.

[19] Wikipedia. Abstract strategy game. http://en.wikipedia.org/wiki/Abstract_strategy_game.

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

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

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

[23] Wikipedia. Deep blue. http://en.wikipedia.org/wiki/Deep_Blue_(chess_computer).

[24] Wikipedia. Game artificial intelligence. http://en.wikipedia.org/wiki/Game_artificial_intelligence.

[25] Wikipedia. Monte carlo method. http://en.wikipedia.org/wiki/Monte_Carlo_method.

[26] Wikipedia. Opening book. http://en.wikipedia.org/wiki/Opening_book.

[27] Wikipedia. Tactic. http://en.wikipedia.org/wiki/Tactic_(method).

[28] Wikipedia. Taxicab geometry. http://en.wikipedia.org/wiki/Taxicab_geometry.

[29] Wikipedia. Zero-sum. http://en.wikipedia.org/wiki/Zero-sum.

[30] Albert L. Zobrist. A new hashing method with application for game playing. Technical report, University of Wisconsin, April 1970.