

University of Alberta

Library Release Form

Name of Author: Haizhi Zhong

Title of Thesis: Building a Strong Arimaa-playing Program

Degree: Master of Science

Year this Degree Granted: 2005

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Signature

Haizhi Zhong
221 Athabasca Hall,
University of Alberta
Edmonton, AB
Canada, T6G 2E8

University of Alberta

Building a Strong Arimaa-playing Program

by

Haizhi Zhong

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment
of the requirements for the degree of **Master of Science**

Department of Computing Science

Edmonton, Alberta
Fall 2005

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Building a Strong Arimaa-playing Program** submitted by Haizhi Zhong in partial fulfillment of the requirement for the degree of **Master of Science**.

Jonathan Schaeffer
Supervisor

Martin Mueller

Bob Hayes

Date: _____

ABSTRACT

Arimaa is a two-player game that was intentionally designed to be difficult for computers to play well. In this game, each move is composed of up to 4 steps, which gives Arimaa a large branching factor preventing computer programs from searching deep.

We have created an Arimaa-playing program, which contains a multi-functional move generator, an evaluation function, and a search engine with many Alpha-Beta search algorithm enhancements implemented. The large branching factor in Arimaa is challenging, but through some innovative techniques, it can be partially overcome. This has enabled us to create a program strong enough to challenge the current computer champion.

ACKNOWLEDGMENTS

I would like to thank Dr. Jonathan Schaeffer for his supervision and help through this research. Without his timely guidance, inspiration and encouragement, this thesis could not have been done.

I would like to thank the members of my committee for their efforts on my behalf. Their comments regarding this work were an important part of its completion.

I would like to thank David Fotland, who gave me a lot of knowledge and inspiration. Thanks also go to Jeff Bacher, Karl Juhnke, Toby Hudson, Don Dailey, and other friends in the Arimaa player and Arimaa-playing program developer society. They were always warmhearted and selfless in helping me.

I would like to thank Omar Syed, who created this great game that is fun for people to play and be a challenge for computers.

My final thanks go to my families. They provided me endless support and they were always the source of my power and courage.

Table of Contents

1. INTRODUCTION	1
1.1 Introduction to Arimaa.....	1
1.2 Rules of Arimaa	2
1.3 Challenge	5
1.4 Thesis Outline	6
2. GAME-TREE SEARCH	7
2.1 Minimax Search	8
2.2 Alpha-Beta Search	10
2.3 Transposition Table	12
2.4 Move Ordering.....	14
2.5 Null Move	15
2.6 Iterative Deepening.....	15
2.7 Conclusion	16
3. EVALUATION FUNCTION.....	17
3.1 Material Value	18
3.2 Piece-Position and Frozen-Position Tables	19
3.3 Elephant Blockade	21
3.4 Trap and Goal Evaluation	22
3.5 Forks	26
3.6 Pins.....	27
3.7 Hostages.....	28
3.8 Example	29
3.9 Temporal Difference Learning	30
3.10 Conclusion	322
4. SELECTIVE MOVE GENERATION	33
4.1 Step-based VS. Turn-based.....	34
4.2 Remove Repetitions	35
4.3 Forward Pruning in Move Generation	38
4.3.1 Step Combo.....	39
4.3.2 Step Combo in Arimaa.....	41
4.3.3 Move Generation with Step Combo	43
4.4 Remove Reversible Moves	46
4.5 Avoid the Third Time Repetition.....	48
4.6 Summary	48
5. SEARCH ENHANCEMENTS	49
5.1 Step-based Alpha-Beta Search.....	49

5.2 Move Order Heuristic	50
5.2.1 Iterative Deepening	51
5.2.2 Killer Move	52
5.2.3 History Heuristic	52
5.3 Null Move	54
5.4 Transposition Table	55
5.5 Razoring	56
5.6 Summary	58
6. FUTURE WORK	60
6.1 Evaluation	60
6.2 Move Generation	61
6.3 Game-Tree Pruning.....	62
6.4 Comparison with Fortland's Work	63
6.5 Result	64
BIBLIOGRAPHY	65
APPENDIX: EVALUATION VALUE TABLES.....	67

List of Figures

Figure 1.1: Arimaa pieces.	2
Figure 1.2: Setting up. Note the initial arrangements of the two players are different.	3
Figure 1.3: A typical situation.	3
Figure 2.1: A sample game tree for the game tic-tac-toe.	7
Figure 2.2: A search tree with node values. The square nodes indicate that it is the turn of the maximizing player, and the circles indicate that it is the turn of the minimizing player.	8
Figure 2.3: The pseudo-code for the Minimax search function.	9
Figure 2.4: Alpha-Beta cutoffs.	10
Figure 2.5: Pseudo-code for the Alpha-Beta search function.	11
Figure 2.6: Pseudo-code of the Alpha-Beta search function with a transposition table.	13
Figure 2.7: Pseudo-code for iterative deepening.	16
Figure 3.1: Elephant blockade.	21
Figure 3.2: All trappable squares. The number indicates the distance from a square to a trap. Some squares have 2 numbers because they are close to 2 traps.	22
Figure 3.3: Goal paths for a gold Rabbit (up to 4 steps).	23
Figure 3.4: A small part of the trap evaluation decision tree. To make it simple, in this figure we assume the step limit is always 4. There is only 1 trappable case covered in this figure.	24
Figure 3.5: A fork situation. The silver Dog at d3 can be captured in both the c3 and f3 traps.	26
Figure 3.6: A pin situation. The gold Horse at c6 is framed. The gold Elephant at d6 cannot move without losing the Horse.	27
Figure 3.7: A hostage situation. The gold Elephant at b2 takes the silver Camel at a3 as a hostage. The silver Elephant cannot leave the squares adjacent to the trap at c3.	28
Figure 3.8: An example for evaluation.	29
Figure 4.1: Comparison of turn-based search tree and step-based search tree. ...	34
Figure 4.2: The mini-tree generated for one node.	34
Figure 4.3: Repetitions.	36
Figure 4.4: Step combos.	40
Figure 4.5: All possible step combo combinations in Arimaa.	41
Figure 4.6: Step combos in the move generation. The thin lines will be pruned.	43
Figure 4.7: After generate step 2.	43
Figure 4.8: After generate step 3.	44
Figure 4.9: Reversible moves.	46
Figure 5.1: Pseudo-code for the step-based Alpha-Beta search.	50
Figure 5.2: The Alpha-Beta tree for razoring.	56

List of Tables

Table 3.1: Material values.....	18
Table 3.2: Number of trap evaluation and goal evaluation cases.	25
Table 3.3: Result of trap evaluation and goal evaluation.....	25
Table 4.1: Repetition types. Theses 3 moves are taken from the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.	37
Table 4.2: Convert the step combos that have less than 4 steps.....	41
Table 4.3: The moves played in real games. It is calculated based on all the positions in the 3 games in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.	42
Table 4.4: Proportion of step combo combinations. It is the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann, 3 moves are taken from each side at different stages of the game.....	42
Table 4.5: Result of the pruning (4-step search). It is the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.	45
Table 5.1: Result of using the Iterative Deepening. It is based on a 9-step Alpha-Beta search, without any other enhancements. The moves are taken from the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.	51
Table 5.2: Result of using the Killer Move. It is based on a 9-step Alpha-Beta search with Iterative Deepening. The moves are taken from the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.	52
Table 5.3: Result of using the History Heuristic. It is based on a 9-step Alpha-Beta search with Iterative Deepening and Killer Move. The moves are taken from the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.	54
Table 5.4: Null Move cutoffs at different levels. It is the 22g move of the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.....	54
Table 5.5: Result of using the Null Move. It is based on a 9-step Alpha-Beta search with Iterative Deepening, Killer Move and History Heuristic. The moves are taken from the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.	55
Table 5.6: Result of Transposition Table. It is based on a 9-step Alpha-Beta search with all the enhancements mentioned above. The moves are taken from the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.	56

Table 6.1: Play against the top 4 programs of the 2005 Computer Arimaa Championship. The games were played in 15 seconds per move, 2 minutes in the starting reserve.64

Chapter 1

INTRODUCTION

1.1 Introduction to Arimaa

Computers have successfully challenged and defeated humans in many strategy games, like Othello [3], Checkers [15], and Chess [9]. They did this using a combination of deep brute-force search and minimal human knowledge of how to play the game well.

Omar Syed, a former NASA computer engineer, believed that even though brute-force searching computers had made such impressive achievements, they were still even not close to matching the kind of real intelligence used by humans in playing strategy games. To prove his point, he and his son designed a 2-player game called Arimaa in 1997, which was intentionally made “easy” for the human players, but “hard” for the computers [17].

When playing strategy games such as Chess, the computer is actually exploring all the move combinations to look ahead as far as possible, so that it can pick out the move that leads to the most favorable position. This brute-force approach of examining each move as deep as possible is quite different than the way used by humans. Humans typically do little search, but use lots of knowledge. This was taken to the extreme in the 1997 Deep Blue versus Kasparov chess match: man 2 positions per sec; machine 200,000,000 positions per second [9].

The game of Arimaa exploits this difference by having a very large set of legal moves in a position (the so-called branching factor). Since the size of the search tree is based on b^d (b for branching factor and d for search depth), the large branching factor of Arimaa results

in the search tree size growing rapidly, making a deep search impractical even on the most advanced computer. By contrast, for the human players, the branching factor does not influence the difficulty of the game as much. The challenge is to build a computer program capable of playing a strong game of Arimaa [17].

1.2 Rules of Arimaa

Arimaa is a game for two players, Gold and Silver, played on an 8x8 board with a standard chess set [18]. Pieces are given different names. Each player has 1 Elephant, 1 Camel, 2 Horses, 2 Dogs, 2 Cats and 8 Rabbits, in order from the strongest to the weakest. See Figure 1.1







Name	Picture	Gold Abbr.	Silver Abbr.	Number	Strength
Elephant		E	e	1	Strongest
Camel		M	m	1	2nd
Horse		H	h	2	3 rd
Dog		D	d	2	4th
Cat		C	c	2	5th
Rabbit		R	r	8	Weakest

Figure 1.1: Arimaa pieces.

The game starts with the players setting up the pieces on their nearest two rows. There is no fixed initial position, so the pieces may be placed in any arrangement (see Figure 1.2). There are 64 million different possible initial arrangements, so it is almost impossible for a computer to use pre-computed databases of opening analysis [5]. So far there is no conclusion as to which arrangement is the best; different arrangement may lead to a different strategy.

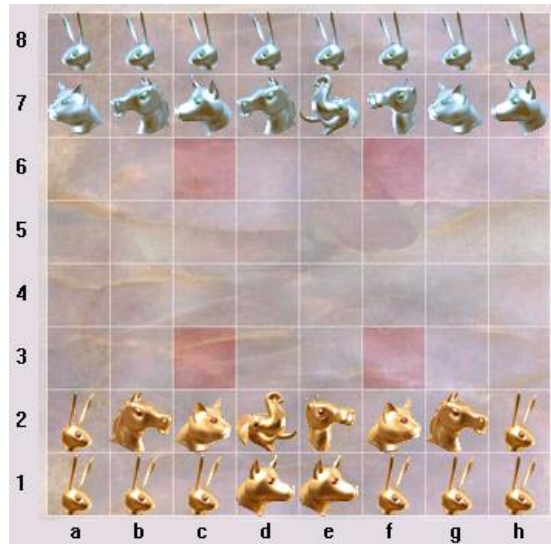


Figure 1.2: Setting up. Note the initial arrangements of the two players are different.

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

A single move consists of up to 4 steps. Moving one piece to an adjacent square vertically or horizontally counts as one step.

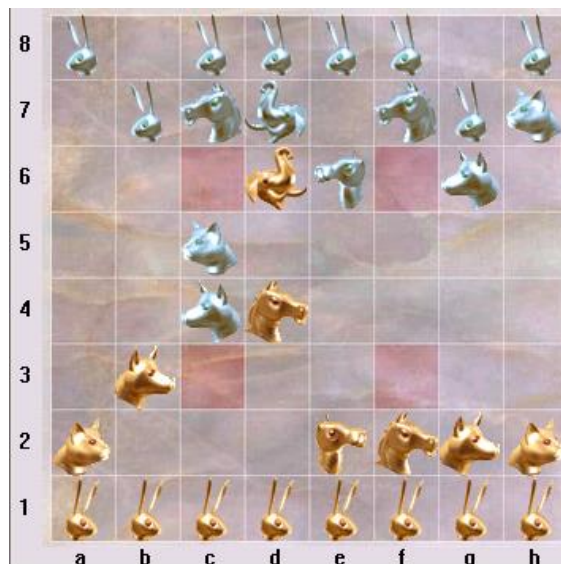


Figure 1.3: A typical situation.

For example, in Figure 1.3, the legal moves of the gold player include:

- **Ch2n Ch3n Ch4n Ch5n¹**: The Cat at h2 moves 4 steps north.
- **Me2n Me3n Me4e Mf4e**: The Camel at e2 moves 2 steps north and 2 steps east.
- **Me2n Me3w**: The Camel at e2 moves 1 step north and 1 step west. This move only contains 2 steps.
- **Ca2e Hd4e Dg2n Rb1n**: The Cat at a2 and the Horse at d4 move 1 step east; the Rabbit at b1 and the Dog at g2 move 1 step north.

Some moves with a different order of steps lead to the same result. For example, “**Mc5e Md5n Hb8e Ra8e**” and “**Hb8e Mc5e Md5n Ra8e**” are identical moves. There are over 10,000 unique legal moves in this position to choose from.

The stronger pieces can push weaker enemy pieces out of the way and move into their square, or pull weaker enemy pieces along with them. For example, in Figure 1.3, the gold Elephant at d6 can move to e6 and push the silver e6 Camel to e5 (... **me6s Ed6e**...). The same Elephant moving to d5 and pulling the silver Camel to d6 is also legal (... **Ed6s me6w**...). Pushing or pulling takes up 2 steps, and can be done together in the same turn. However a piece cannot pull while completing a push.

When a weaker piece is beside a stronger enemy piece, it becomes frozen and cannot move unless there is a friendly piece beside it. For example, in Figure 1.3 the silver Camel at e6 is frozen (by the gold Elephant at d6). The gold Horse at d4 threatens the silver Dog at c4, but the latter is free to move because there is a friendly Cat at c5 providing support.

The board has four distinctly marked squares at c3, c6, f3 and f6, which will be referred to as “traps”. Any piece that is on a trap square and does not have a friendly piece beside

¹ The notation of Arimaa is as follows. Every step is recorded by 4 letters. The first letter indicates the piece color and piece type, upper case for the gold pieces and lower cases for the silver pieces. The second and third letters indicate the column and row of the piece’s position. The fourth letter indicates the direction which the piece moves, “n” for north, “e” for east, etc. For example, “**Ca2e**” means the gold cat at column “a” and row “2” moves 1 step east. Removing a trapped piece from the game is recorded by marking an “x”. For example “**Hf2n Hf3x**” means the gold horse at “f2” moves north and is trapped. Notice it still counts as one step, not two [18].

that trap square will be removed from the game. For example, in Figure 1.3, the gold Horse at d4 can capture the silver c4 Dog by pushing it into the trap at c3 (... **dc4s dc3x Hd4w**...this still counts as 2 steps). If the silver e6 Camel is pushed into the trap at f6 by the gold Elephant (**me6e Ed6e**), it won't be removed for there is a friendly Horse at f7 and a friendly Dog at g6.

All the rules are quite simple and intuitive for the human players, and make this a fun game to play.

1.3 Challenge

The main reason why this game is difficult for a computer is the large branching factor. Compared to an average of about 35 moves in a typical Chess position, or roughly 200 in a Go position, a player has to choose between 2,000 to 3,000 moves for their first move in Arimaa, and about 5,000 to 40,000 moves in the mid-game. If we assume an average of 20,000 possible moves at each turn, looking forward just 2 moves (each player taking 2 turns) means exploring about 160 million billion positions. Even if a computer was 5 times faster than Deep Blue and could evaluate a billion positions per second it would still take it more than 5 years to explore all those positions [17].

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

Compared to the capture rule in Chess, pushing, pulling, freezing and trapping in Arimaa is more difficult for the computer to handle. Some of the successful heuristics used in games like Chess do not work well in this game [17].

The match that computers challenge humans has happened twice so far. Both times the world's best Arimaa-playing program, "bot_bomb" built by David Fotland, was beaten

by humans (8-0 by Omar Syed in 2004 and 7-1 by Frank Heinemann in 2005). Another fact indicates the huge gap between humans and computers in Arimaa playing more clearly: strong human players can beat the best Arimaa program with a handicap of 1 Camel, 2 Dogs and 1 Cat, and they are still competing to win with a larger handicap.

1.4 Thesis Outline

In this thesis, our goal is to build a program capable of playing a strong game of Arimaa. We will investigate some search techniques which are often used in other strategy games and assess how effective they are in Arimaa. We will also investigate building an evaluation function for this game and assess how easy/difficult this is.

The outline of the thesis is as follows. Chapter 2 presents background information of heuristic search, and takes a look at some popular search enhancements.

Chapter 3 gives a description of the evaluation function used in our Arimaa program. We try to put every piece of knowledge that is proven to be essential into the evaluation function, and avoid calculating too much uncertain or trivial stuff.

Chapter 4 introduces the process of move generation. We remove all repetitions and reversible moves. We introduce the new idea of the step combination to reduce the branching factor and make some forward pruning.

In Chapter 5 we present the search enhancements used in our program, and give out the experimental results and their analysis. These enhancements include Transposition Table, Null Move, Move Ordering, History Heuristic, Iterative Deepening and Razoring.

In Chapter 6, some interesting possible future research directions are discussed.

Chapter 2

GAME-TREE SEARCH

Like other 2-player games, Arimaa can be represented by a large game tree, with the computer doing a depth-first depth-limited search over it. The idea is to start at the current position, generate the set of all possible successor positions, the evaluation function will be applied to those positions, the one with best value will be picked out, and its value will be backed up to the starting position. This procedure could be used recursively until the limiting depth (or time) to get a more accurate result [6, 11, 13].

Figure 2.1 illustrates this idea with a search tree for the game tic-tac-toe.

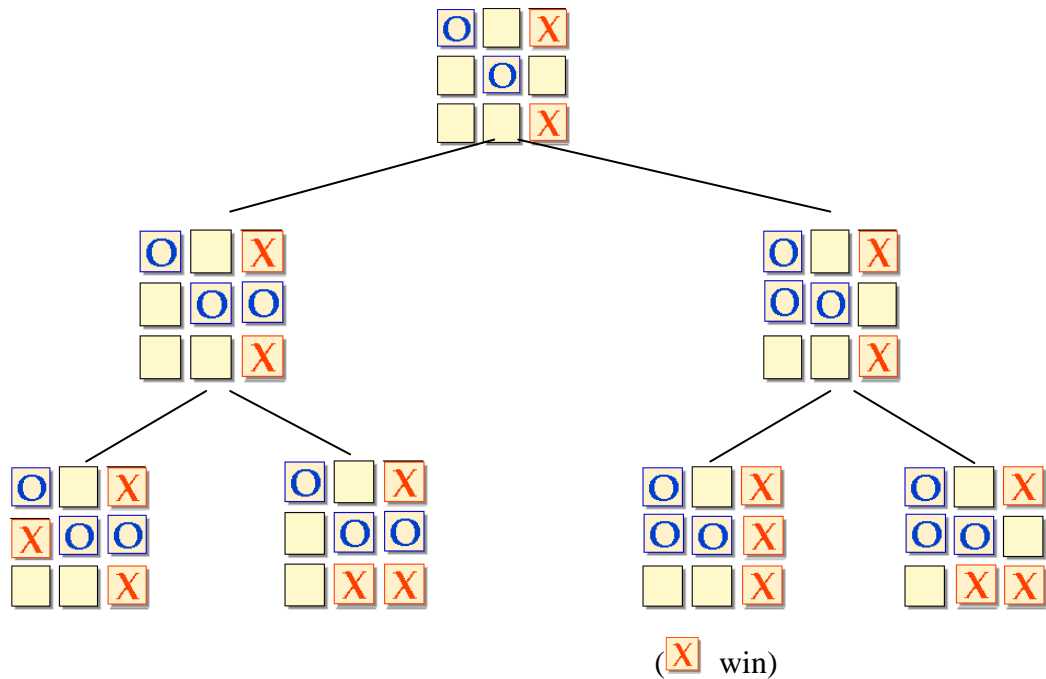


Figure 2.1: A sample game tree for the game tic-tac-toe.

It is well-known that searching deeper generally improves the quality of the decision made by the search [19].

To play a game, the program needs:

- 1) A move generator and the ability to make and undo a move.
- 2) An algorithm for efficiently traversing the tree.
- 3) Knowledge of when the game is over, and what the result is (win, lose or draw).
- 4) An evaluation function to assess how good the position is.

2.1 Minimax Search

In the game tree, all the moves represent by the nodes at odd levels belong to the current player (represented by squares in Figure 2.2), and all the moves represent by the nodes at even levels belong to the opponent player (represented by circles in Figure 2.2). The computer assumes both sides play their best. Certainly, the current player's best move is the move with the highest evaluation. But the opponent player's best move is the current player's worst situation, which is the one with the lowest evaluation value. Thus, the search procedure must select the move with the maximum value at odd levels, and the move with the minimum value at even levels, as shown in Figure 2.2. This searching procedure is called Minimax [6, 11, 13].

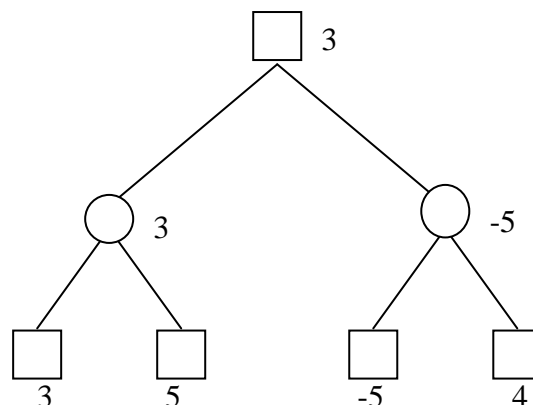


Figure 2.2: A search tree with node values. The square nodes indicate that it is the turn of the maximizing player, and the circles indicate that it is the turn of the minimizing player.

```

int miniMax( state s, int depth, int type )
{
    if( isTerminalNode(s) || depth==0 )
        return evaluate(s);

    Vector succ = generateSuccessors(s);

    if( type==MAX )
    {
        score = -INFINITE;
        for( child=succ[0]; child!=succ.last(); child=succ.next() )
        {
            value = miniMax( child, depth-1, MIN );
            if( value>score )
                score = value;
        }
    }
    else // MIN
    {
        score = INFINITE;
        for( child=succ[0]; child!=succ.last(); child=succ.next() )
        {
            value = miniMax( child, depth-1, MAX );
            if( value<score )
                score = value;
        }
    }
    return score;
}

```

Figure 2.3: The pseudo-code for the Minimax search function.

Figure 2.3 shows the pseudo-code for the Minimax search function. If a node is terminal (which means the result of this node is a certain win, lose or draw), or the depth limit is reached, we stop searching deeper and return the result of the evaluation function.

Otherwise we generate all the legal successors of this node, iterate through all of them, recursively call the Minimax function, get the maximum (for the maximizing player) or minimum (for the minimizing player) value as the result to return.

It is easy to prove that if the average branching factor of the tree is b , and search depth is d , then Minimax search will examine $O(b^d)$ nodes.

2.2 Alpha-Beta Search

A Minimax search is very time consuming. The efficiency can be greatly improved by recognizing that some positions in the search tree are provably irrelevant, and therefore can be eliminated (or “pruned”) from the search [13].

The idea is to maintain a search “window” composed of a pair of bounds for every node: the lower bound (*alpha*) is a lower bound on the best that the player to move can achieve and the upper bound (*beta*) is an upper bound on the best that the opponent can achieve. During the search, if we get an *alpha* value bigger than or equal to the *beta* value, then further search at this node is irrelevant.

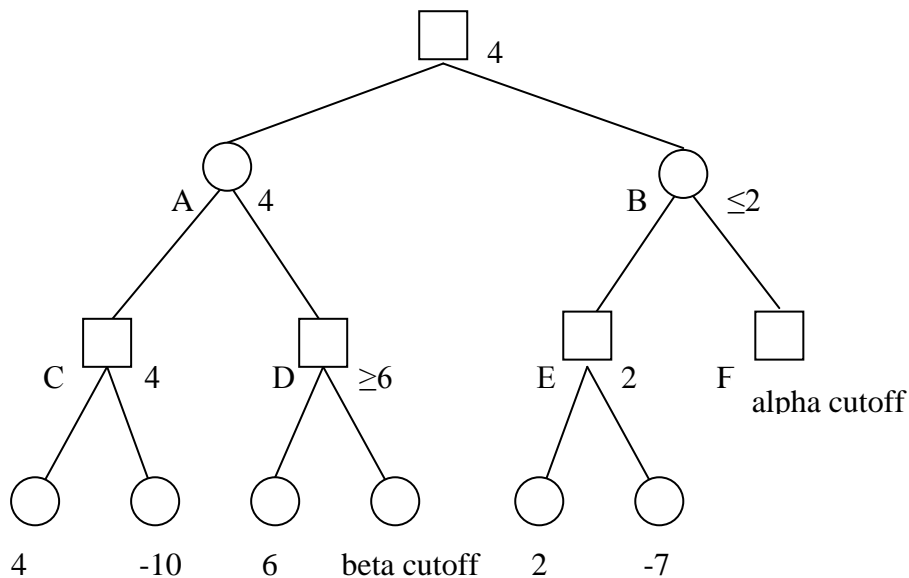


Figure 2.4: Alpha-Beta cutoffs.

Consider the example tree in Figure 2.4. After visiting node C and its sub-trees, we know that the value of node A must be smaller than 4 (upper bound). After first child of node D is examined, we see that node D is guaranteed a min value of 6 (lower bound). 6 is bigger than 4, the upper bound of node A. That means node D cannot be the best child

of node A, no matter what final value it gets. Therefore the search of its other children can be skipped. This is called a beta cutoff.

Similarly, node E sets node B's upper bound to 2. Comparing node B to node A (whose value is 4), node B cannot have a higher value, so searching node F and its sub-tree can be saved. This is called an alpha cutoff.

```
int alphaBeta( state s, int depth, int alpha, int beta )
{
    if( isTerminalNode(s) || depth==0 )
        return evaluate(s);

    Vector succ = generateSuccessors(s);

    score = -INFINITE;
    for( child=succ[0]; child!=succ.last(); child=succ.next() )
    {
        value = -alphaBeta( child, depth-1, -beta, -alpha );
        if( value>score )
            score = value;
        if( score>alpha )
            alpha = score;
        if( alpha>=beta )
            break;
    }
    return score;
}
```

Figure 2.5: Pseudo-code for the Alpha-Beta search function.

Figure 2.5 shows the pseudo-code for the Alpha-Beta search function. It is given in the so-called NegaMax formulation. The idea is that if we swap *alpha* and *beta* and negate them as well as the return value, we can treat the Min nodes identically to the Max nodes instead of alternating Max nodes and Min nodes.

The effect of Alpha-Beta search is that in the best case, the number of nodes examined can be reduced to roughly $O(b^{d/2})$ without changing the search result. This so-called best case of Alpha-Beta search happens when every first child of a node causes a cutoff

(if one is possible). Hence it is essential that the search program invest resources to order node successors from most to least likely to make the cutoffs happen as soon as possible.

In the worst case the Alpha-Beta search could equal the Minimax search, with no cutoffs at all.

2.3 Transposition Table

In a huge game tree, the same game position may be reached by different paths, and each path has a different node for it. Searching nodes that represent the same situation multiple times is redundant.

Keeping a record of the previous search results can eliminate this kind of overhead.

Having this data allows us to do a lookup before exploring a node, to see if it has been visited before, and decide if further search is necessary. The most popular way of saving the data is using a hash table, because it is fast to access and easy to implement [12]. This is often called a transposition table.

Even if the transposition table information is insufficient to cause a cutoff, it can still be beneficial. Useful information can be saved to narrow down the search window or improve the move ordering. Usually, the information includes search bounds, the best successor, the search depth and the search result.

Zobrist's method is a well-used encoding method to generate the index hash key for the transposition table [22]. It is very fast, only taking several bit operations. The basic idea is to give every possible piece and square combination a unique random code. Producing a hash key for a position is easily done by doing exclusive-or (xor) operations on those codes of the combinations that occur in that position.

```

int alphaBeta( state s, int alpha, int beta, int depth )
{
    if( isTerminal(s) || depth==0 )
        return evaluate(s);

    ptr = lookupTT(s);
    if( ptr!=NULL && ptr->depth>=depth )
    {
        if( ptr->bound==LOWER)
            alpha = MAX( alpha, ptr->value );
        if( ptr->bound==UPPER)
            beta = MIN( beta, ptr->value );
        if( ptr->bound==ACCURATE )
            alpha = beta = ptr->value;
        if( alpha>=beta )
            return ptr->value;
    }

    ... ..
    // generate and traverse the successors of the node and get a value

    if( value<=alpha )
        bound = UPPER;
    else if ( value>=beta )
        bound = LOWER;
    else
        bound = ACCURATE;
    saveTT ( s, value, bound, depth );
    return value;
}

```

Figure 2.6: Pseudo-code of the Alpha-Beta search function with a transposition table.

Figure 2.6 shows pseudo-code of the Alpha-Beta search function with a transposition table. Note that the transposition table only helps when the search result stored in it has a search depth that is no less than the current search depth. The search window might be narrowed, even if no cutoff occurs.

Since collisions are unavoidable with a hash table, the hash key should be at least 64-bits. Replacement of table entries should also be taken into consideration [2].

2.4 Move Ordering

The earlier an Alpha-Beta cutoff happens at a node, the smaller the tree that is built.

Ordering the nodes, making plausible ones be searched sooner, increases the chance that a good move will be found early and cause a cutoff.

If the node being explored has an entry in the transposition table and the transposition table information does not cause a cutoff, the best successor from the previous time this position was searched is a good candidate for being the best, and thus should be search first. If this move is not applicable or fails, some popular methods can provide other plausible candidates.

The Killer Move is a popular method that is effective in many games. The idea of this method is that a good move in one branch of the tree is also good at another branch at the same depth. At each ply a number (usually 2) of moves that cause cutoffs are stored and tried first before other moves are searched. These moves are called killer moves. There should be a scheme to replace the killer moves with the non-killer moves that newly cause a cutoff. Using Killer Move does not need knowledge of the specific domain

The History Heuristic [14] is another popular application-independent move ordering method. It is based on an assumption: If a move has a history of being best (highest score or cause a cutoff) in a state, it is likely this move is also good for a similar state. History Heuristic is implemented by maintaining a history score table of all possible steps. Every time after searching an internal state, add a value (this value should reflect the search depth, a recommended equation is 2^{depth}) to the history score of its best successor step. This history score is used to re-sort moves.

Although the Killer Move and History Heuristic are very powerful, sometimes building an application-dependent sorting function will be a better choice in improving move ordering.

Reordering all the successors at a node is not always worthwhile, because sorting is time-consuming. It is highly application-dependent, useful in Chess (about 40 moves in a position), but not in Arimaa (many thousand moves in a position).

2.5 Null Move

The idea of the Null Move is assume there is always something to do than nothing. If a position before a move is found already good enough for the player to move, a move by the same player after this position will only make its result better. Therefore further search over this position is unnecessary and can be skipped.

Whether a position is “good enough” is tested by doing a null move (doing no move at all) and then searching with reduced (1 or even more) depth. If the result of this shallow search is higher than the *beta* value, which means this position is so good that the player to move can afford passing a move and still be all right, then further search is skipped. By replacing a normal search with a shallower search, some search time cost is saved. If the result of the reduced-depth search is lower than *beta* value, we do a normal search.

Null moves can be recursive, but 2 null moves in a row are forbidden.

2.6 Iterative Deepening

We can do a shallow search first and use the result to do a deeper search, and iterate this procedure until time runs out or a limiting depth is reached. This method is called Iterative Deepening, see Figure 2.7. At first this idea was introduced for providing a time control mechanism for real-time matches, but researchers found that it is also a way to save searching time as well.

The time spent on shallow searches looks like a waste, but in fact it is not. The shallow search could be used to re-sort all possible moves of the root. Furthermore, it also fills

the transposition table with valuable data, and thus helps to increase cutoffs and improve move ordering.

```
for( depth=INIT_DEPTH; ; depth+=DEPTH_STEP )
{
    val = alphaBeta( depth, -INFINITY, INFINITY );
    if( timeout() || depth==MAX_DEPTH)
        break;
}
```

Figure 2.7: Pseudo-code for iterative deepening.

2.7 Conclusion

Many game-playing programs use Alpha-Beta search to efficiently traverse the game-tree. Extensive research into Alpha-Beta has been done for over 40 years. Programs can search very efficiently in games like Chess, Checkers and Othello, coming close to the Alpha-Beta best-case search tree [12].

Chapter 3

EVALUATION FUNCTION

An evaluation function is used to measure how good a position is. Without an estimation of the desirability of positions reached we cannot select out the best move. A successful evaluation function should be strongly correlated with winning or losing, and be fast to compute [4].

The evaluation function is important in every game-playing program. In the game of Arimaa, the role it plays is even more significant. Because of the large branching factor, all Arimaa programs cannot search very deep (normally they cannot finish a 3-move search in a reasonable amount of time). To search 1 ply deeper requires the program to be several hundreds of times faster, which is very difficult to achieve. Therefore, putting more knowledge into the evaluation function and building a better evaluation function is a sound way to improve the program.

We try to put every piece of knowledge that is proven to be essential into the evaluation function, and avoid calculating too much uncertain or trivial stuff. From the view of implementation, we mainly adopt two ways to make the function fast: calculating the score incrementally and using pre-computed tables as much as possible. Our approach is to keep the old score of the board and only incrementally adjust the score by updating it at each step. This is much faster than calculating the score from scratch for every move.

The evaluation function consists of several terms:

- 1) Material Value

- 2) Piece-Position Value
- 3) Frozen-Position Value
- 4) Elephant Blockade Evaluation
- 5) Trap Evaluation
- 6) Goal Evaluation
- 7) Fork Evaluation
- 8) Pin Evaluation
- 9) Hostage Evaluation

Each of them is discussed in turn.

3.1 Material Value

The material balance is almost always the most important part for many games. Table 3.1 shows the material values of all the pieces in Arimaa. These values are adopted from David Fotland program [8]. The value of a Rabbit changes based on how many Rabbits have been lost. The last one is 12 times as valuable as the first one, since if it is lost then the player cannot win the game.

Rabbits	100, 150, 200, 250, 300, 400, 500, 1200
Cat	250
Dog	300
Horse	600
Camel	1100
Elephant	1800

Table 3.1: Material values.

Theoretically, all material values in Arimaa should be relative. If a player has an advantage in material, then exchanging pieces of equivalent strength is beneficial to this

player, because it makes the situation simpler and the advantage more difficult to be reversed. Therefore, slightly increasing the values of all the pieces that belong to a player every time he loses a piece seems a reasonable way to make the evaluation function work better. But in practice, dynamic material values make the system much more complicated. It is very difficult to assess whether this difference is worthwhile.

At the beginning of a game, most players will be happy to trade two Cats and a Dog for a valuable Camel. But this trade may not be a good deal at the end of the game, when there are not many pieces left. After many pieces have been exchanged, the number of pieces becomes more and more important, even though the pieces might be over-ranked. To reflect this tendency, some people suggest beside piece values, we should also assign another value corresponding to the number of pieces belongs to the same side left on the board. It is an interesting and reasonable thought, but we didn't try it in our program.

3.2 Piece-Position and Frozen-Position Tables

In our program, every piece rank and position combination is assigned a value to encourage pieces to move to favorable positions. For each piece rank, the position values are horizontally symmetrical. Vertically reversing the position values for a gold piece gives the values for the equivalent silver piece, since their goals are vertically reversed. Compared to the material values, the position values are very small. In other words, material dominates the evaluation. Two positions that are materially equivalent can be differentiated by their position values.

Elephants are invulnerable in the game of Arimaa, so letting them control the center region is rewarding. For them, the value of a square is mainly dependent on the distance to the center of the board.

Cats and Dogs are weak pieces. Moving them into the enemy's territory is unlikely to be a good choice. In a real game, most of the time their role is to guard the traps on their

side of the board. Therefore the squares adjacent to their traps have encouraging values, and advanced squares are set to negative values.

The Camels and Horses are strong pieces, but not invulnerable. They should go to the enemy zone to attack, but at the same time avoid being trapped or framed. The position values set for them are a combination of the Elephants and weak pieces.

Every square has two values for a Rabbit. The Rabbits are the weakest pieces and cannot move backwards; moving them forward is dangerous at the beginning of the game. Thus the “normal” value of a square at the advanced ranks is set to negative to discourage the Rabbits from advancing. We also assigned a “motivate” value for every square, which will be used to encourage the Rabbit to strive for a win, whenever we find that the enemy is too weak to guard their territory. In both cases the values for the side files are assigned a little higher score than the center files, because the latter is more dangerous for being trapped and more difficult to reach a goal.

For all pieces, the four trap squares are given negative values. Even if a trapped piece has an adjacent friendly piece, occupying a trap square is still a potentially dangerous situation.

Similarly, we assign a value for every piece rank and position combination for the frozen pieces. Being frozen is always bad, the possible moves are restricted and potential paths are blocked. A strong player is always attempting to freeze as many opponent’s pieces as possible. Having a single piece freeze two or more opponent’s pieces can be very advantageous.

How bad a freezing will be depends on the piece rank and its position. Being frozen, a strong piece is worse than a weak piece, and staying at the back ranks are not as bad as being close to an enemy trap.

The piece-position and frozen-position tables are given in Appendix. Note these values have been manually set. There are learning methods than can help determine good sets

of values (e.g. temporal different learning [16]), but attempts to do this have been unsuccessful (see later discussion).

3.3 Elephant Blockade

The motility of Elephants is very important in Arimaa. In Figure 3.1, the silver player has just blockaded the gold Elephant so that it has no legal moves. There is no empty square for the gold Elephant to step into, and no empty square into which it can push the silver pieces. This is a very bad situation for the gold player [10].

A novice Arimaa player will think that since nine silver pieces are required to maintain the blockade, including the silver Elephant and Camel, this blockade is more costly to the silver player than to gold. But that is not true. In making a blockade, a weak piece serves as well as a strong one; therefore silver player can easily free his strong pieces by replacing them with weak pieces, and get the upper hand.



Figure 3.1: Elephant blockade.

In our program, to measure the Elephant blockade situation, we check every adjacent square of the current positions of an Elephant to get the following information:

- 1) Does this direction face the center of the board or an edge?
- 2) Is this square empty?
- 3) How many squares adjacent of this square are occupied?

The information for both sides results in a pair of 6-scale estimate values, which will be added into the evaluation score. The table of these values is given in Appendix.

3.4 Trap and Goal Evaluation

Like many other games, the loss and gain of material in Arimaa is the dominant indicator of who is losing and who is winning. Therefore, investing some time in calculating the future material balance is worthwhile. For every position at level $4n+m$ ($0 \leq m < 4$), we calculate the possibility that any enemy piece can be trapped in the next $4-m$ steps, and subtract the material value of the most valuable piece that can be trapped from the evaluation score. Therefore, search step $4n+m$ ($0 \leq m < 4$) will give us a roughly $4n+4$ step result.

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

Figure 3.2: All trappable squares. The number indicates the distance from a square to a trap. Some squares have 2 numbers because they are close to 2 traps.

To trap an enemy's piece, the player needs to pull or push it onto a trap square, and force its supporting piece to leave the squares adjacent to the trap. For the pieces that belong to the opponent of the player to move, the possibility of being trapped exists only if the distance from their position to any trap square is equal to or less than two steps. See Figure 3.2. Among all 64 squares on the board, 44 of them are at most two steps from a trap. A piece at c4, c5, f4, f5, d3, e3, d6 or f6 can be captured into two different traps. For these squares, the threats from both sides need to be calculated.

During the process of trapping an enemy's piece, sacrificing a friendly piece may be unavoidable. In this case, the evaluation depends on the comparison of the values of the lost piece and the gained piece.

For every position at level $4n+m$ ($0 \leq m < 4$), we check the possibility that any Rabbit of the player to move can achieve the goal in the next $4-m$ steps, and if so score the situation as infinite (win) or minus infinite (lost). It can discover an unavoidable win or loss situation up to 4 steps earlier, saving further search and greatly improving the performance of the endgame play.

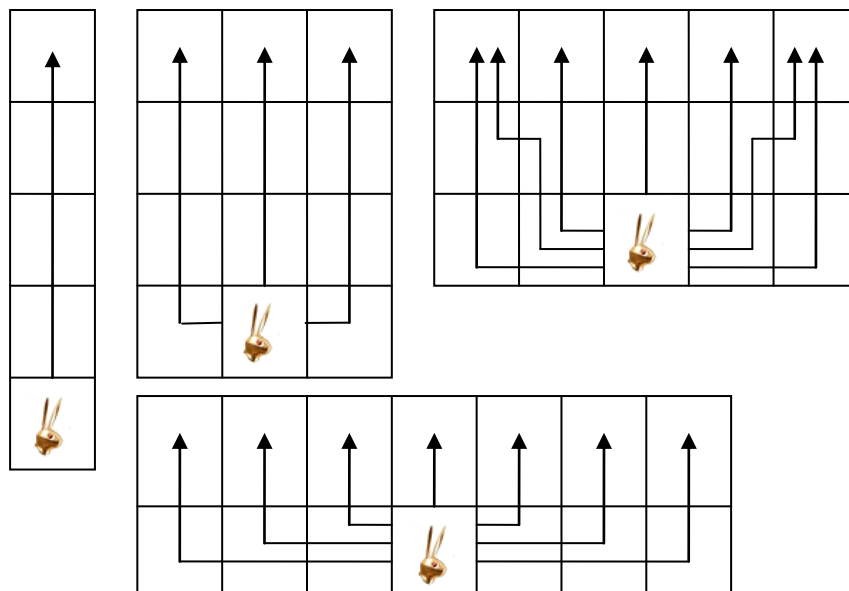


Figure 3.3: Goal paths for a gold Rabbit (up to 4 steps).

Only the Rabbits that belong to the player to move need to be checked. We classify the goal situation of a Rabbit by how far it is from the goal rank. Figure 3.3 shows all possible goal paths for a gold Rabbit.

The difficulty in finding out the situation of trapping or reaching the goal in 4 steps is obvious. There are many different paths. Pulling, pushing, blocking, freezing and supporting: all these rules make the situation very hard to handle.

The easiest way to solve the problem is to generate all the legal moves of the next turn. This method is simple to implement, but not practical due to the time-consuming cost of move generation (see Chapter 4). To make the program fast, we have to use decision trees to handle all the trapping and goal situations case by case (see Figure 3.4).

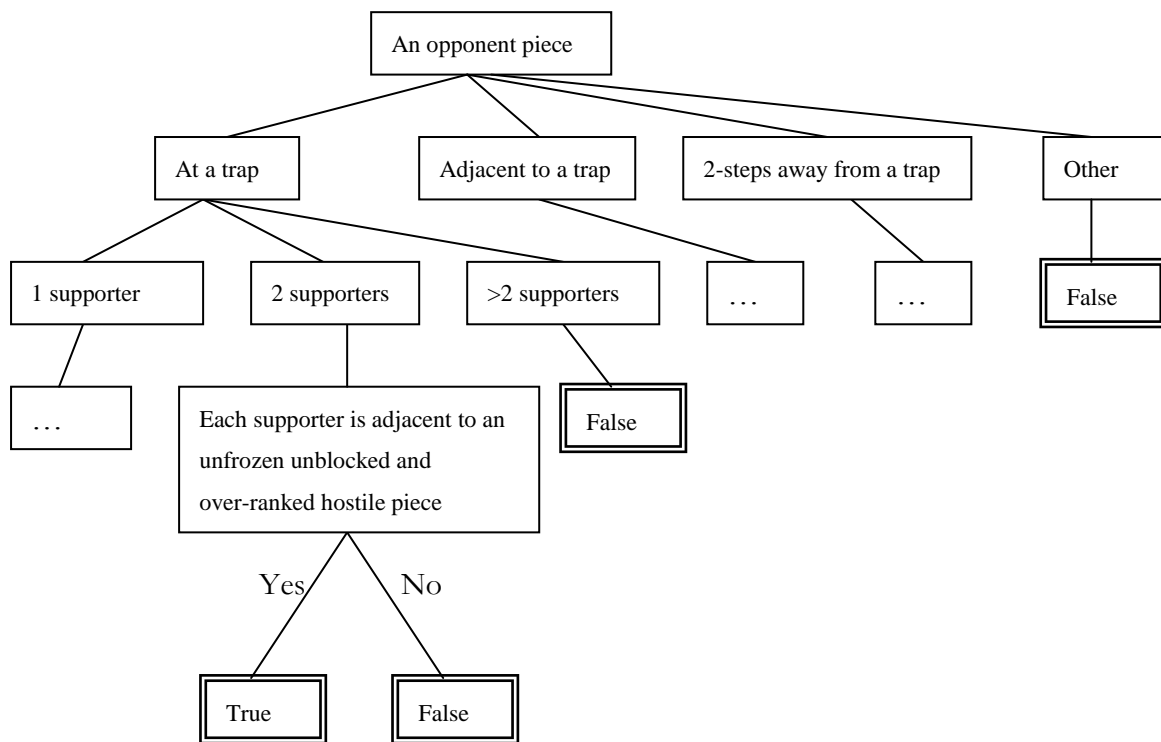


Figure 3.4: A small part of the trap evaluation decision tree. To make it simple, in this figure we assume the step limit is always 4. There is only 1 trappable case covered in this figure.

The difficulty of implementing such a decision tree is based on the number of cases that have to be considered. Table 3.2 gives out the numbers of cases. As the step number increases, the number of cases grows dramatically. All cases need to be manually carefully handled, and each of them has many details that need to be taken care of, that make the building of decision tree a huge and hard task. However, the execution time cost of the decision tree is very low. It doesn't make the search distinctly slower.

So far our decision tree can handle all the cases that take up to 3 steps to trap or goal, but the 4 steps cases are not accomplished yet.

Step	Trap case	Goal case
≤ 1	1	1
≤ 2	3	4
≤ 3	10	29
≤ 4	>40	>100

Table 3.2: Number of trap evaluation and goal evaluation cases.

Table 3.3 shows the result of trap and goal evaluation. It can make a 5-step search roughly equivalent to an 8-step search, which is a great improvement. It also helps to give Iterative Deepening a better move ordering. Implementing the cases for all 4 steps will make it better, but considering the difficulty involved, it is not critical.

Search	Roughly equivalent search	
	step ≤ 3	step ≤ 4
5-7 step	8 step	8 step
8 step	11 step	12 step
9-11 step	12 step	12 step
12 step	15 step	16 Step

Table 3.3: Result of trap evaluation and goal evaluation.

From Table 3.3, to make the best use of the searching time, apparently $4n+1$ (such as 5, 9 or 13) is the preferred search depth.

3.5 Forks

A “fork” means that one piece is threatened to be capture in two different opponent’s traps. It is a common way to get an immediate material gain. In Figure 3.5, the silver Dog at d3 can be captured in both the c3 and f3 traps. It is very difficult for the silver player to provide protection for both opponent’s traps in one move. Even if the silver player was fortunate enough to do so, normally there will be flaws in this constrained defensive position for the opponent to exploit [10].

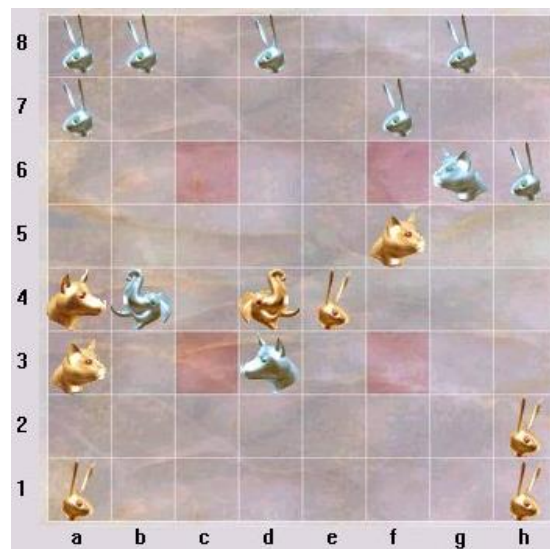


Figure 3.5: A fork situation. The silver Dog at d3 can be captured in both the c3 and f3 traps.

In our program, we evaluate the fork situation caused by a central Elephant. If one side has such a situation, we add or reduce a certain value from the evaluation score based on the rank of the threatened piece. The appendix gives the table of values.

3.6 Pins

In Figure 3.6, the gold Horse at trap c6 is surrounded on three sides by opposing pieces which prevent it from pushing its way off the trap square. The gold Elephant providing support is “pinned”, it cannot move without letting the Horse be captured immediately. The silver player can make his position even better by using his Horse at a5 to free his Elephant at c5. This is called a “pin” situation. One piece is in great danger and the Elephant loses its mobility. Obviously it is a very bad situation for the gold player [10].

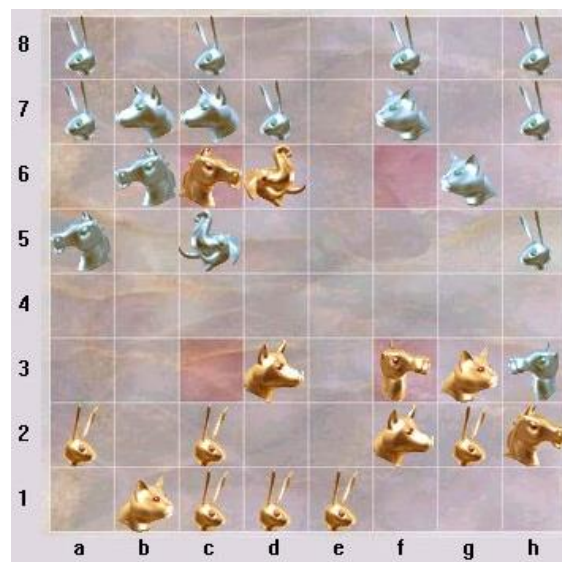


Figure 3.6: A pin situation. The gold Horse at c6 is framed. The gold Elephant at d6 cannot move without losing the Horse.

Pin situations are not difficult to detect. In our program we evaluate them based on the rank of the framed piece, and modify the evaluation score accordingly. The appendix gives the table of values.

3.7 Hostages

Figure 3.7 shows a hostage situation. The silver Camel at a3 is threatened by the gold Elephant at b2 and has no way to go. The silver Elephant has to stay adjacent to the trap at c3 to prevent the silver Camel from being captured. In the current position, it is very difficult for the silver player to free his Camel, and the silver Elephant has limited mobility. The chance of a weak hostage piece being captured or pinned is high [10].

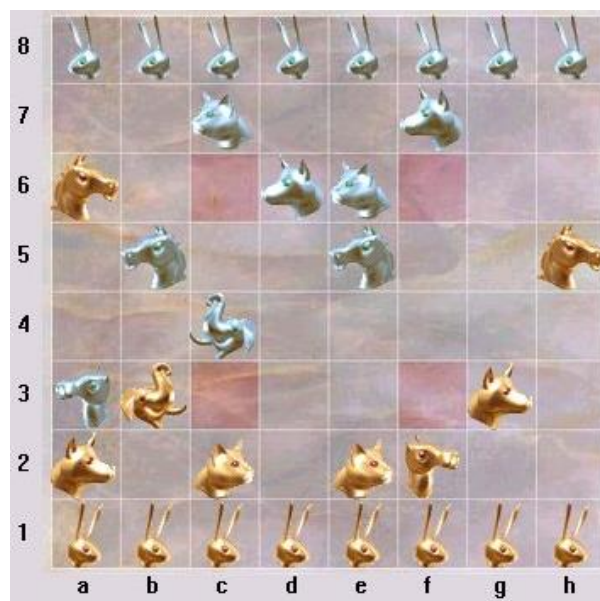


Figure 3.7: A hostage situation. The gold Elephant at b2 takes the silver Camel at a3 as a hostage. The silver Elephant cannot leave the squares adjacent to the trap at c3.

Oddly enough, taking a Camel or a weak piece hostage is always considered a big gain, but taking a Horse hostage is considered a loss. A Horse being frozen is not a serious problem, and the Horse hostage situation hurts the mobility of the “kidnapping” Elephant much more than the “supporting” Elephant. In playing Arimaa, the human players voluntarily offer one of their Horses to the opponents as a hostage so often that this tactic has been dubbed the “E-H” attack.

In our program, we evaluate hostage situation based on the rank of the hostage piece and modify the evaluation score accordingly. The appendix gives the table of values.

3.8 Example

We will use the situation shown in Figure 3.10 as an example to present our evaluation function. The evaluation score is always from the gold player's point of view, which means a positive value means gold is the favored side.

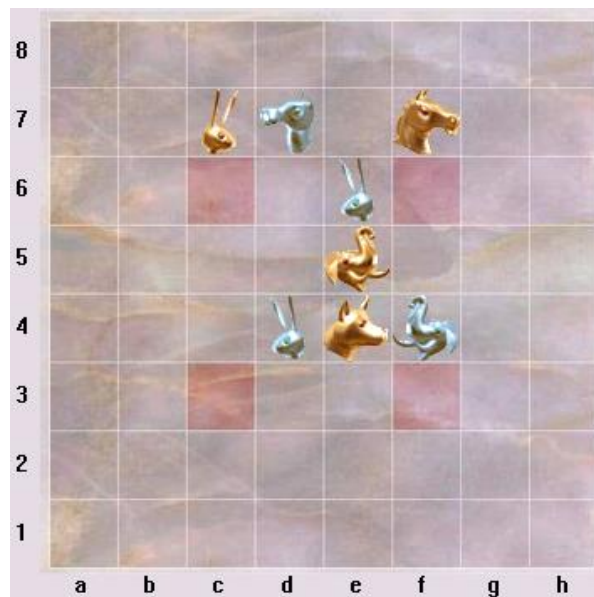


Figure 3.8: An example for evaluation.

In the board shown in Figure 3.8, the gold side has 1 Elephant, 1 Horse, 1 Dog and 1 Rabbit left. The material score is 3700 (1800+600+300+1200). The silver side has 1 Elephant, 1 Camel and 2 Rabbits. The material score is 4600 (1800+1100+1200+500). So the material balance is -900 (3700-4600).

The position scores of the gold pieces are: 12 (Elephant), 1 (Horse), -6 (Dog), 25 (Rabbit), for a total 32. The position scores for the silver pieces are: 10 (Elephant), 3 (Camel), 2 (Rabbit at e6), 14(Rabbit at d4), for a total of 29. So the position balance is 3 (32-29).

The gold Rabbit is frozen, the score of that is -1. Two silver Rabbits are also frozen, the scores are 0 (Rabbit at e6) and -1 (Rabbit at d4). So the frozen balance is 0 (-1-(-1)).

There is no Fork, Frame, or Pin situation for either side in the current position.

The trap evaluation is based on which side is to play. If the gold side plays, the most valuable trappable silver piece is a Rabbit (**re6e rf6x E5en**), and the value is 500. If the silver side plays, it can capture the gold Rabbit (**Rc7s Rc6x md7w**), the gold Dog (**De4e ef4e Df4s Df3x eg4w**), and gold Horse (**ef4n ef5n ef6s Hf7s Hf6x**). The values are 600, 300 and 1200 (notice the last Rabbit is more valuable than a Horse or a Camel). 1200 is the biggest value.

If it is the gold side's move, the gold Rabbit can get a victory in the next turn (**Ee5w Ed5w Ec5n Rc7n Ec6x**, assume the 4-step goal evaluation is implemented). The silver side cannot win the game in a turn.

Now we have all scores for the evaluation. If the next turn belongs to the gold player, the final score is: -900 (material balance) + 3 (position balance) - 0 (frozen balance) + 500 (trap evaluation) + infinite (goal evaluation). The result is infinite, a certain win. If the next turn belongs to the silver player, the final score is: -900 (material balance) + 3 (position balance) + 0 (frozen balance) - 1200 (trap evaluation) + 0 (goal evaluation), the result is -2097, which means the silver player has a big advantage.

3.9 Temporal Difference Learning

There are hundreds of constants present in our evaluation function. If we consider that each one is linked to a feature of the evaluation function, then the constant value is the weight of that feature. So far, all of these weights have been manually set based on experience.

Obviously, letting the computer set these values by itself should be a better way, which is what Reinforcement Learning tries to do. The most popular approach in Reinforcement Learning is using temporal difference learning ($TD(\lambda)$) [16]. The idea is to let the computer play thousands of games against itself. At every move for every feature, if the

number measuring that feature is changed, the computer will adjust the weight value of this feature based on the difference of the evaluation of the board situation. For each feature, the formula for the weight change is as follows:

$$w_{t+1} - w_t = \alpha(Y_{t+1} - Y_t) \sum_{k=1}^t \lambda^{t-k} \nabla_w Y_k$$

In this formula, $w_{t+1} - w_t$ is the change of the weight of the feature between move $t+1$ and move t , and $Y_{t+1} - Y_t$ is the difference of the evaluation values between move $t+1$ and move t . α is a constant between 0 and 1 called the “learning rate”, and λ is another constant between 0 and 1 called the “feedback coefficient”. $\nabla_w Y_k$ is the gradient of the evaluation value with respect to weights.

Theoretically, there is no reason that TD(λ) cannot work with Arimaa. We tried it but unfortunately failed to get a promising result. The program was set to search 5-step every move, and we let it play over 2000 games against itself. The result of updated weights is not reasonable. The reason probably is one or more of follows:

- 1) There are some bugs we have not found.
- 2) There are too many features to learn (several hundreds) and we did not run the program long enough.
- 3) The search is too shallow. Experience shows that the TD learning should be done using search depths similar to the game search depths, which is at least 9-step in our case. But currently TD training using 9-step search is not practical, because it takes too long [16].

Though we cannot figure out what is wrong, we still believe that TD(λ) is suitable in Arimaa. More research is needed in this area.

3.10 Conclusion

The large branching factor of Arimaa prevents us from searching deep. So far no computer program can break the wall of a 3-move fixed depth search. Therefore, our program, as many other Arimaa-playing programs, puts more effort into making a better evaluation function.

As more knowledge is added to the evaluation function, the search becomes distinctly slower. Research shows that this is not caused by the evaluation function itself becoming slower; it is because the new knowledge changes the principle variation more dramatically between different levels, and may causes the moves to be considered in a worse order.

Because of the combination of a step-based search and the forward trap/goal evaluation, we found an 11-step search is not distinctly better than a 9-step one. Since we are far from reaching searching 13-step, 9-step is the favorite practical search depth. Experience shows in a 3 minute per move game, having more knowledge in the evaluation function and a slower search normally is better than having a slim evaluation function and searching 1 or even more step deeper.

There are still so many things unknown in building a good evaluation function for Arimaa. More research is needed.

Chapter 4

SELECTIVE MOVE GENERATION

Generating moves correctly and fast is very important for any game program. It is even more important for the game of Arimaa. In our program, on average it costs about 30% to 40% of the execution time.

For any game, normally the computer generates all legal moves available for the player to move for searching at each turn. We call it turn-based approach. Arimaa is a game in which every move contains multiple steps. Therefore moves belonging to the same turn but in different steps can be generated and searched separately. We call it step-based approach. We will compare these two approaches, and illustrate why the latter is a much better choice.

For Arimaa, many legal moves with different step sequences lead to an identical state. These repetitions cost space and time, and don't help in improving the search result. They should be removed as soon as possible. This is a very important task that should be part of the move generation process.

Removing the moves that unlike is a good candidate before search is called forward pruning. It reduces the branching factor and makes the search tree smaller. We try to do some forward pruning in move generation to make the search faster without sacrificing much in precision.

4.1 Step-based VS. Turn-based

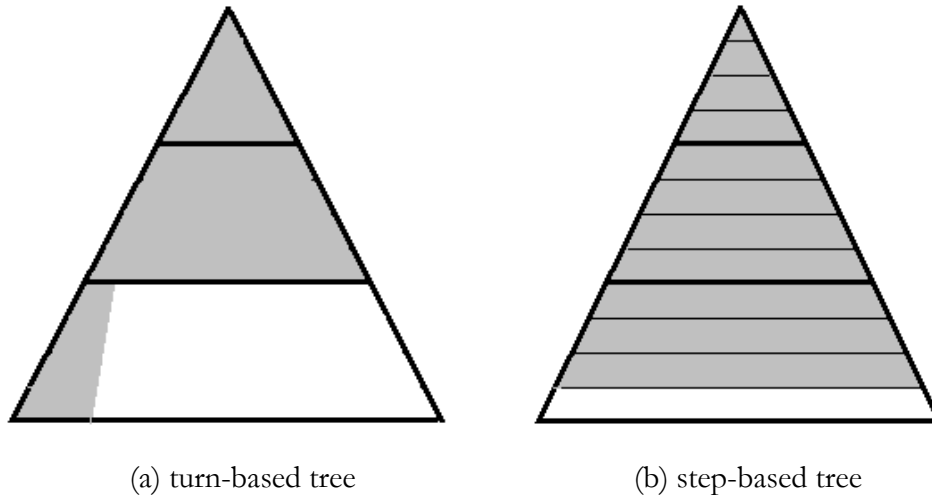


Figure 4.1: Comparison of turn-based search tree and step-based search tree.

In searching the game tree, normally for each position the computer generates all legal moves available. This process can be called turn-based move generation, see Figure 4.1 (a). In Arimaa, the generation and search of a node in the same turn can be broken down to 4 steps. Each time the program only generates and searches one step of a move; see Figure 4.1 (b). This idea was first introduced in David Fotland's Arimaa program [8]. It changes Arimaa into an unusual game with a branching factor less than 50, but the side to move changes every four ply.

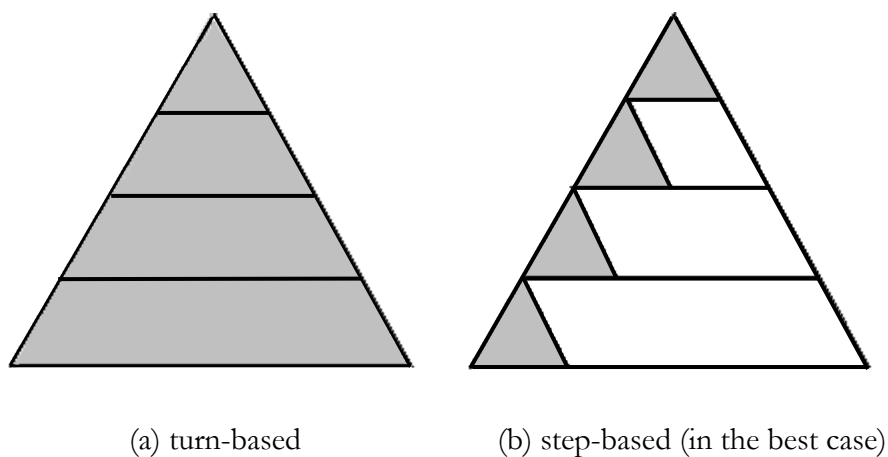


Figure 4.2: The mini-tree generated for one node.

The benefits of step-based move generation are:

- 1) In turn-based search, the program generates all successors of a node, even though Alpha-Beta cutoffs will prune most of them. In step-based search, the move generation is broken down to each step. Thus only a small part of the mini-tree of a node is generated due to Alpha-Beta pruning, see Figure 4.2.
- 2) Search of the last turn is depth-first in the turn-based search tree and breadth-first in the step-based search tree. The grey regions in Figure 4.1 show this difference. In Figure 4.1(a), the turn-based approach completed a 2-turn (8-step) search and begin search turn 3; its result is close to an 8-step search. In Figure 4.1(b), the step-based search completed an 11-step search and goes to step 12. Even though the number of searched nodes is the same, the result can be an 8-step search compare to an 11-step search. Certainly the latter is much more effective in improving the result.
- 3) Step-based search provides more iteration for the iterative deepening and helps create better move ordering for the deepest search.

The step-based search is literally hundreds of times faster than the turn-based search. In our test, using the Alpha-Beta search without any enhancements, the turn-based search can never complete a 2-turn (8-step) search in 10 minutes, but the step-based search can complete it in several seconds.

4.2 Remove Repetitions

A simple example of a repetition is having a piece randomly move one step and then return to its original position in the next step. It is legal, but the board doesn't change at all.

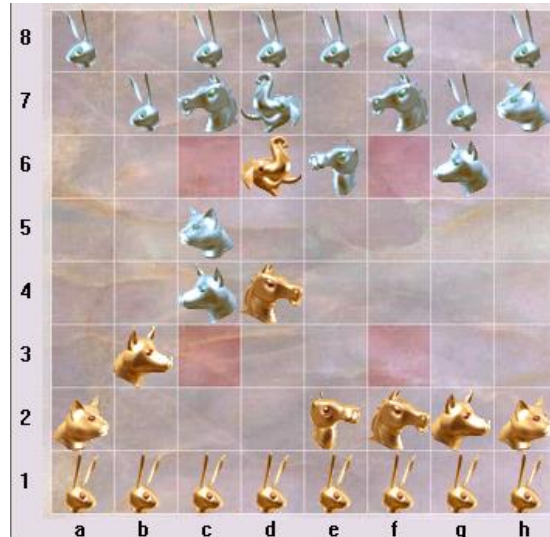


Figure 4.3: Repetitions.

In Figure 4.3, we can find more examples of repetitions:

- **Ch2n Ch3n Ch4n Ch5n, Ch3n Ch2n Ch4n Ch5n, Ch3n Ch4n Ch2n Ch5n ...**
- **Me2n Me3w, Me2w Md2n, Me2w Md2w Mc2n Mc3e...**

The portion of unique moves out of all legal moves is about 7%; over 90% of the legal moves are repetitious and should be removed.

In removing repetitions, certain rules must be followed to get a correct result:

- 1) Among a group of identical moves, the one that covers all successor moves that this state may lead to must be the move with the smallest number of steps. Therefore this move must be kept, and other moves can be removed. For example, if a 2-step move and a 4-step move are repetitions, we keep the 2-step move and remove the 4-step move. If all the moves have the same number of steps, keep one of them and remove all others.
- 2) The pulling moves need to be handled carefully. For example, in Figure 4.3, “**Ed6s Ed5n**” leads to a pulling move “**Ed6s Ed5n cc5e**”, and therefore should not be simply treated as identical to a null move and be removed.

To remove the repetitions, we use a hash table to store all the visited board states. Each entry of the hash table contains a number of board states, and can be accessed by the Zobrist' hash value of the board state [22]. Every time the program explores a new move which leads to a certain state, it checks all the visited states stored in the corresponding entry of the hash table. If this state is found, which means it has already been visited in a previous move generation, the new move will be thrown away; otherwise, it will be kept and saved in the hash table.

The move generation hash table is like a transportation table without a replacement scheme. Each entry of the hash table is set to contain a certain number of board states. This number should make sure that there won't be any overflows (or at least the overflows are very few) and at the same time the size of the table is not too big. After some testing, we set this number to 8 and the total number of entries to 2 millions.

Move#	2:0	2:2	3:1	3:3	4:0	4:1	4:2	4:3	4:4
12g	8	216	227	3608	6	2	3609	1	36610
17s	23	374	601	6980	12	0	8156	1	76635
22g	14	169	261	2437	5	3	2846	3	22371

Table 4.1: Repetition types. Theses 3 moves are taken from the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.

Table 4.1 shows all the types of the repetitions and the frequency with which they occur in a typical Arimaa board. A move is marked by the move number and which side to play, for example "12g" means the move is #12 and it is the gold to play. The two numbers used to mark a type are step numbers. For example, 4:2 means a 4-step move is identical to a 2-step move. The bigger number is always put before the smaller one.

If we generate all the successors of a node at the same time, as in a turn-based program, the method introduced above can detect and remove all the repetitions easily. But our step-based program generates the moves according to the number of their steps. Thus repetitions with different step numbers cannot be found.

Fortunately, Table 4.1 shows us that the majority of the repetitions occur with the same step number (2:2, 3:3 and 4:4), and work fine with our step-based move generation. The cases of 4:0, 4:1 and 4:3 are so rare that ignoring them won't create much overhead. The cases of 2:0 and 4:0 are easy to handle; you only need to put the original node into the hash table beforehand. Thus we conclude that 3:1 and 4:2 are the only cases that need to be taken care of.

In our program, we directly pick out all 3:1 and 4:2 repetitions based on their step information. A move that contains a pair of counteracted steps may be a repetition, for example, “**Me2n Rd1n Rd2w Me3s**” in Figure 4.3 contains a pair of counteracted steps “**Me2n Me3s**” and is a repetition. But we also need to consider moves like “**Me2n Hf2n Hf3n Me3s**”, which contains the same pair of counteracted steps but is distinct. There are also moves like “**Me2w Md2w Mc2n Mc3e**” which are not distinct (identical to “**Me2w Md2n**”) but don't contain a pair of counteracted steps. Based on the above research, a carefully designed subroutine in our program can accurately remove all the 3:1 and 4:2 repetitions. The hash table will handle other cases.

Currently, we generate all the legal steps first, detect and remove the repetitions afterward. Due to the big portion of the repetitions, it will be several time faster if we can build a move generator in such a way that it doesn't generate the repetitious moves at all, and therefore we wouldn't need a hash table to store and detect visited status. In Arimaa, because there are many complicated cases that caused by the pulling, pushing, trapping and freezing, to build such a move generator is not an easy task. More research is needed in this direction.

4.3 Forward Pruning in Move Generation

If the move generator can pick out the moves which are likely to be good candidates to consider, and ignore those moves with slim chances (so-called forward pruning or selective search), it will improve the program's performance. It is a trade off between speed and veracity, and this needs be carefully balanced.

In Arimaa, a move is composed of up to 4 steps. About 10% of all the moves generated each turn contain less than 4 steps. Since it seems that the computer can always find something better than doing nothing with those passed steps to improve the situation, it looks like we can remove all those moves to reduce the branching factor. But after some research, we found this idea is not as reasonable as it seems. If a player takes 3 steps to do a task, his other pieces may all stay in safe positions, and the extra 1 step can do nothing but harm the situation. Therefore skipping the step is the best option. Strong human players often make a 3-step, 2-step or even 1-step move in a real game, especially in a defensive situation. We have to keep all the moves with passed steps.

Another obvious selection strategy is that since only the moves with a high evaluation should be considered valuable, the move generator can eliminate all those low-valued moves in every step. The problem with this approach is that often a move loses some value in one step and gains more value in the next one. The steps cannot be simply considered isolated from each other; there might be some relationship between them.

4.3.1 Step Combo

To analyze and overcome this problem, we introduce a concept: Step Combo. A step combo is a sequence of steps in a move such that the order of the steps in this sequence cannot be changed. That is, in a step combo, step i depends on step $i-1$. If any 2 steps are swapped, we get a different board position or an illegal move. A combo may contain 1 to 4 steps.

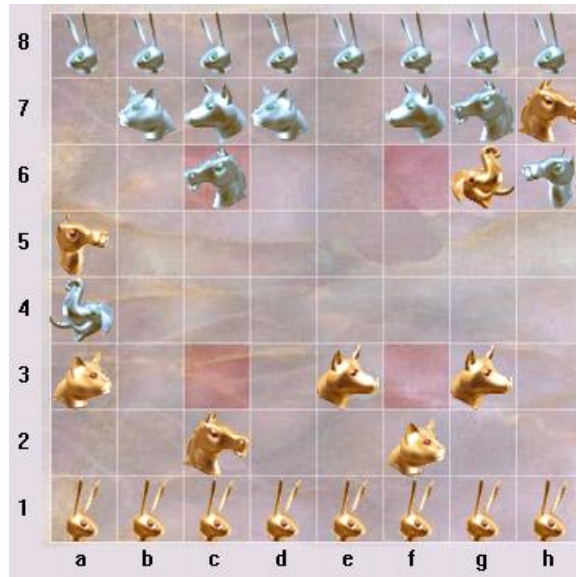


Figure 4.4: Step combos.

In Figure 4.4 we can find many kinds of step combo cases, include:

- Pulling (like “**ea4e Ma5s**”) and pushing (like “**Ma5n ea4n**”).
- A piece moving several steps in a row (like “**Rd1n Rd2n Rd3n Rd4e**”).
- A piece go to support a frozen piece and let the latter escape (like “**Ra1n Ca3e**”).
- A piece move away from a square to make room for another piece to move into (like “**Hc2e Rc1n**”).
- A piece move adjacent to a trap to let another piece safely move into the trap without being taken off (like “**De3w Hc2n**”).
- A single step does not have any relation with the next step or previous step (like every step in “**Ra1n Rb1n Rd1n Re1n**”).

In contrast, a move such as “**Hc2e Ra1n**” is not a step combo, because altering the order of the two steps does not change the result.

From the definition, it is easy to prove that to get the same result, the step order in a step combo cannot be changed; however, the order of the step combos can be changed arbitrarily. For example: the move “**Eg6e mh6s Ra1n Rb1n**” is built up by 3 step combos, “**Eg6e mh6s**”, “**Ra1n**” and “**Rb1n**”. The order of “**Eg6e mh6s**” cannot be reversed, but change the order of combos, and the move “**Ra1n Rb1n Eg6e mh6s**”, “**Rb1n Ra1n Eg6e mh6s**” and “**Ra1n Eg6e mh6s Rb1n**” will lead to the same result.

The score gained (or lost) by every move is the sum of score gains (or losses) that every step combo makes. Every combo has a certain score gain or loss, and does not influence other step-combo score contributions. Within a step combo, a step does not have a separate value. All the steps contained in a step combo must be considered as a whole.

4.3.2 Step Combo in Arimaa

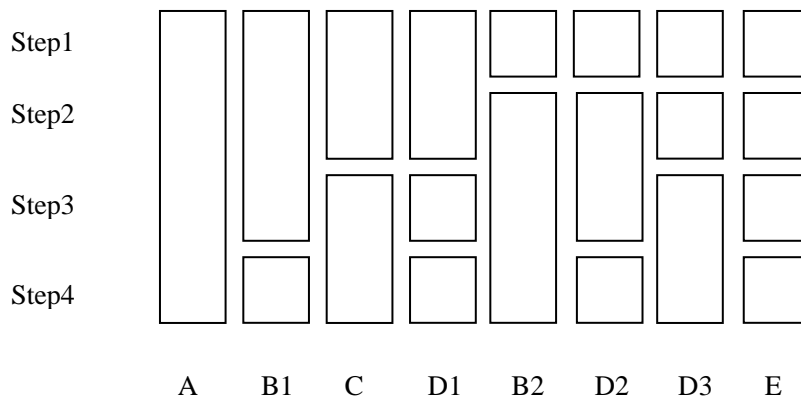


Figure 4.5: All possible step combo combinations in Arimaa.

Figure 4.5 shows all possible step combo combinations in the 4 steps of a move in Arimaa. This figure will be used throughout this section. Since the order of step combos doesn't matter, case $B2$ is identical to case $B1$ (because every move in $B2$ has an equivalent move in $B1$), and case $D2$ and case $D3$ is identical to case $D1$. Therefore, we have 5 distinct cases: $A(4)$, $B(3+1)$, $C(2+2)$, $D(2+1+1)$ and $E(1+1+1+1)$. The marks in the brackets indicate the numbers of steps in all step combos that build up this move, and the order of combos doesn't matter.

#step<4	#step=4
(1)	A(4)
(2)	A(4)
(1+1)	B(3+1)
(3)	A(4)
(2+1)	C(2+2)
(1+1+1)	D(2+1+1)

Table 4.2: Convert the step combos that have less than 4 steps.

The step number in a move can be less than 4. To make it simple, we will convert them into 4-step combinations based on the table in Table 4.2. It does not have any impact in following discussion.

	A(4)	B(3+1)	C(2+2)	D(2+1+1)	E(1+1+1+1)
Game1	34%	33%	10%	21%	1%
Game2	33%	29%	15%	20%	2%
Game3	36%	24%	15%	23%	1%
Total	34%	29%	14%	21%	2%

Table 4.3: The moves played in real games. It is calculated based on all the positions in the 3 games in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.

Table 4.3 shows the step combo combinations as a proportion of the moves that played in the real games. Table 4.4 shows the step combo combinations as a proportion of all the legal moves and distinct moves. The difference is dramatic. Less than 2% of the moves played in the real games belong to case $E(1+1+1+1)$, but this case is 42% of all the legal moves and 31% of all the distinct moves.

Apparently, exploiting this difference, treating every generated move based on the case it belongs to, is a promising way to improve the efficiency of move generation and searching.

Move#	Legal Moves						Distinct Moves					
	Total	A	B	C	D	E	Total	A	B	C	D	E
12g	274038	21%	20%	2%	25%	32%	20589	18%	12%	7%	42%	21%
17s	631944	16%	17%	1%	23%	43%	23305	16%	10%	4%	40%	30%
22g	172921	23%	21%	2%	27%	27%	13808	23%	16%	7%	39%	15%
27s	551326	15%	16%	1%	29%	48%	30680	17%	10%	2%	34%	37%
32g	170734	22%	20%	2%	26%	30%	12761	21%	16%	6%	39%	18%
37s	926523	13%	14%	1%	20%	52%	48813	14%	8%	2%	35%	42%
Sum	2727486	16%	17%	1%	24%	42%	149956	17%	11%	4%	37%	31%

Table 4.4: Proportion of step combo combinations. It is the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann, 3 moves are taken from each side at different stages of the game.

4.3.3 Move Generation with Step Combo

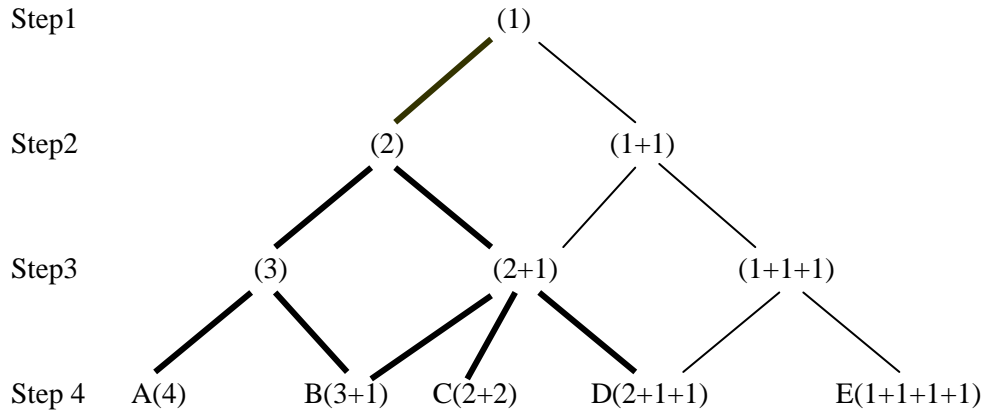


Figure 4.6: Step combos in the move generation. The thin lines will be pruned.

Figure 4.6 shows all the step combo combination possibilities in each step of the move generation process. The move nodes are generated step by step, and the situation in each step is different. At step 1, the move generator does not have enough information to tell whether a step combo has been accomplished or not and therefore cannot eliminate any moves.

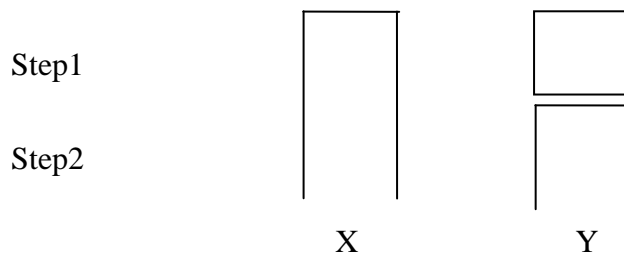


Figure 4.7: After generate step 2.

Figure 4.7 shows the situation after two steps have been generated. There are two kinds of possibilities. In case *X*, the first 2 steps belong to a step combo and the moves must belong to $A(4)$, $B(3+1)$, $C(2+1)$ or $D(2+1+1)$ in Figure 4.5, and should be kept for further generation.

In case Y , step 1 and step 2 are not in a same step combo. The moves built up by this kind of two steps must belong to cases $B2(1+3)$, $D2(1+2+1)$, $D3(1+1+2)$ or $E(1+1+1+1)$ in Figure 4.5. From the discussion above, we know that any moves belonging to cases $B2(1+3)$, $D2(1+2+1)$ and $D3(1+1+2)$ must have some equivalent moves with the form of cases $B1(3+1)$ and $D1(2+1+1)$. Since moves in cases $B1$ and $D1$ are kept, the moves in cases $B2$, $D2$ and $D3$ can be removed without any risk.

Case $E(1+1+1+1)$ is different for it does not have an equivalent expression covered by any other case. Consider the characteristic of this case, in which 4 steps are totally separated in a move. The chance of a critical situation in which a move in case E is the only savior is extremely rare. Ignoring this case is a wise choice. The data in the Table 4.3 proves this point as well.

Thus at a very early stage we eliminate all nodes in case Y in Figure 4.7, which is the majority of cases in 2-step nodes. The only possibility of eliminating a good move exists is case $E(1+1+1+1)$. The loss of a few moves, comparing to the huge saving, is an acceptable risk.

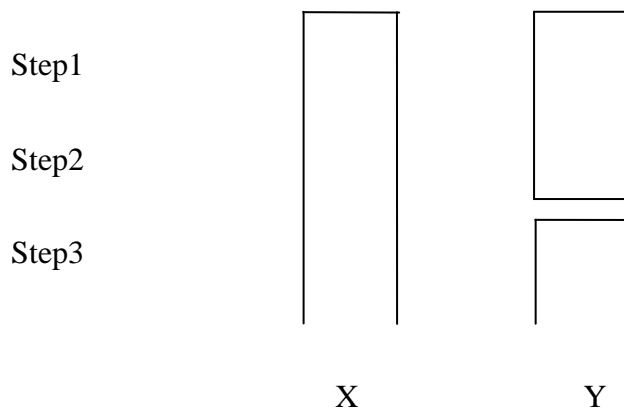


Figure 4.8: After generate step 3.

Figure 4.8 shows the situation after three steps have been generated. All nodes in case X cover cases $A(4)$ and $B(3+1)$ in Figure 4.5, and undoubtedly should be kept for further generation.

Case Y leads to case $C(2+2)$ and $D(2+1+1)$ in Figure 4.5. Although a more aggressive pruning approach is a possible option, based on the data in Table 4.3 and 4.4 (the proportions of C and D in the moves of real games are 14% and 21%, their proportions in the distinct moves are 4% and 37%) and the result of some testing, we conclude that keeping all moves is worthwhile.

At step 4, all the promising moves are generated.

Move#	Legal move			Distinct move		
	Before	After	Save	Before	After	Save
12g	274038	44887	84%	20589	12635	39%
17s	631944	79090	87%	23305	19158	18%
22g	172921	32108	81%	13808	9240	33%
27s	551326	58486	89%	30680	13858	55%
32s	170734	30077	82%	12761	8025	37%
37g	926523	89117	90%	48813	16869	65%
Total	2727486	333765	88%	149956	79785	47%

Table 4.5: Result of the pruning (4-step search). It is the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.

Table 4.5 compares the number of moves generated with and without selective move generation. In a typical situation of Arimaa, the selective move generator prunes about 30-50% of the distinct moves. Furthermore, it reduces over 85% of the redundant legal move, saves the majority of looking up and comparing time, and makes the move generation itself much faster.

About 2% of the moves played in the real games were overlooked by the selective move generation, all of them belong to case $E(1+1+1+1)$ in Figure 4.5. The mismatch rate alone does not reflect the result very well; to evaluating the result accurately we should assess the possible consequences as well (although both will be counted as a mismatch, missing a move that can trap an enemy piece is obviously not equal to missing an idle move), and review the difference between the overlooked move and the best generated move.

Theoretically, overlooking a move in case E should not create a big problem. We carefully studied all 5 overlooked E moves that played in the real games, and concluded that none of them was critical. According to our evaluation function, the substituted moves provided by our move generator are as good or even better.

4.4 Remove Reversible Moves

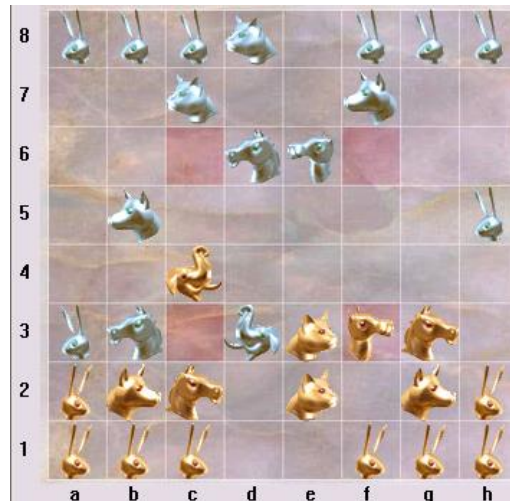


Figure 4.9: Reversible moves.

Figure 4.9 is the first game in the 2005 Arimaa Challenge, computer champion bot_bomb versus Frank Heinemann. The game proceeded as follows:

```

14g  Ec4s Ec3n hb3e Db2n   14s  Db3s hc3w me6e mf6e
15g  Ec4s Ec3n hb3e Db2n   15s  Db3s hc3w hb3e hc6d
16g  Ec4s Ec3n hb3e Db2n   16s  Db3s hc3w hb6w db5n
17g  Ec4s Ec3n hb3e Db2n   .....

```

At each turn, the computer (gold side) repeated a 4-step move, and the opponent moved back to the original position in 2 steps and got 2 extra steps to improve his situation. This is a serious pitfall in all Arimaa programs which is frequently exploited by human players [17].

If the search ends with the opponent to move, it will discover that there is no forward progress and probably not make a reversible move. The problem is that currently most Arimaa programs can only search 12 steps or shallower, so they must evaluate the position without giving the opponent a chance to move away.

We implemented a rule in the move generation to remove all the reversible moves. Assume in a situation that the current player can get state $A1$ in at least $x1$ steps and get state $A2$ in at least $x2$ steps. If the opponent can convert $A2$ into $A1$ in y steps, and $x2-x1 > y$, then the moves that lead to state $A2$ are considered reversible and will be removed.

The most important reversible moves are $3:1$ (a 3-step move can be reversed in 1 step, the player to move may have an extra step, and the opponent may get 3) and $4:2$ (a 4-step move can be reversed in 2 steps; the opponent may get 2 extra steps). We remove all of them in move generation. Unfortunately, in some very rare situations, a $3:1$ reverse move may be a reasonable choice and should be kept in the search tree, and that will cause an oversight by our program.

The $4:4$ and $3:3$ moves are also reversible. In these cases the opponent won't get any advantage in reversing the board. But they may lead to an endless loop (especially when the program plays another program), so we simply remove these moves. Sometimes this means sacrificing a bit of value in position. A better design might be letting the program decide whether remove $4:4$ and $3:3$ reversible moves based on current situation. If the program is losing, keeping the reversible moves can be used as a defensive method, for an endless loop (draw) is better than a loss. Thus the responsibility of breaking the loop (and the sacrifice if there is any) will go to the opponent, since he is the winning side.

Despite the risk that is involved, we believe that pruning all the reversible moves is a good trade off to make the computer player stronger. The frequency of the reversible moves is below 1%, so we don't gain much in reducing the branching factor by removing them.

We also investigated so-called “quiescence search” as another solution for this problem: whenever the program detects a reversible move, it could extend the search until the opponent finished a move. The problem is that in Arimaa each turn has 4 steps. Therefore this extension must be 4 steps deep. It will cost too much and not be practical.

4.5 Avoid the Third Time Repetition

The rule of Arimaa prescribes that if after a turn the same board position and side to move is repeated three times, then the player causing this to occur the 3rd time loses the game. To avoid that, we keep the track of position history in our program, and remove the moves that lead to the 3rd time position repetition.

4.6 Summary

We successfully built a powerful move generator. It generates moves in a step-based way, removes all the repetitions and reversible moves, and makes some safe forward pruning.

There is still huge potential in improving the move generator to make it faster and more powerful. The step combo is a new idea and needs further investigation. We believe it can provide more useful information to the search engine and make it run faster.

Chapter 5

SEARCH ENHANCEMENTS

All search programs, when optimized for a specific search task, will use a number of enhancements to the normal Alpha-Beta search algorithm. These enhancements could include Transposition Table, Move Ordering, Iterative Deepening, and many other general as well as problem-specific ideas. In Arimaa, all of these enhancements need to be modified to suit the step-based search.

All these enhancements result in a reduced search tree. Most of them can guarantee the result will be identical to the original unimproved search. For the large branching factor in Arimaa, which makes the game-tree grow too fast, we also investigated some “unsafe” methods. They enable more pruning, but their result might be different from the original search, because they will lose some good moves. For these methods, the trade off of making more cutoffs and keeping veracity needs to be carefully balanced [21].

5.1 Step-based Alpha-Beta Search

In the game of Arimaa, we use a step-based Alpha-Beta search for the reasons given in Chapter 4. Therefore, we need to modify the algorithm of Alpha-Beta search as in Figure 5.1.

Comparing to the standard Alpha-Beta code in Figure 2.5, the main difference is that the player to move changes every 4 steps, so we accordingly exchange the alpha and beta values every 4 steps. There won't be any cutoff in the first 3 steps. The pruning begins with the 4th step.

```

int alphaBeta( state s, int depth, int alpha, int beta )
{
    if( isTerminalNode(s) || depth==0 )
        return evaluate(s);

    Vector succ = generateSuccessors(s);

    score = -INFINITE;
    for( child=succ[0]; child!=succ.last(); child=succ.next() )
    {
        if( (SET_DEPTH-depth)%4==3 ) // complete moves, SET_DEPTH is
            // current searching depth
            value = -alphaBeta( child, depth-1, -beta, -alpha );
        else // incomplete moves
            value = alphaBeta( child, depth-1, alpha, beta );

        if( value>score )
            score = value;
        if( score>alpha )
            alpha = score;
        if( alpha>=beta )
            break;
    }
    return score;
}

```

Figure 5.1: Pseudo-code for the step-based Alpha-Beta search.

5.2 Move Order Heuristic

In the best case of Alpha-Beta search, the number of nodes to search might be reduced to roughly $b^{d/2}$. A good move ordering heuristic is very important to make the Alpha-Beta search close to its best case [6, 11, 13].

We use Iterative Deepening and sort the root moves (the 4th step) based on the result of the previous search. For the search of other steps, we will first try the best successor that was stored in the transposition table if it is available, and then 2 killer moves, and other moves ordered by the history heuristic function.

5.2.1 Iterative Deepening

In Iterative Deepening, a shallower search can help in improving the move ordering for the next deeper one in two ways:

- 1) Before searching to depth $d+1$, order the moves at the root based on the score returned from depth d . It is based on the following assumption: The best move in the shallow search is very likely still best in the deeper search. This is true in Arimaa and many other games.
- 2) A shallow search visits many nodes in the game-tree which will be visited again in the deep search. The search results of these nodes are kept in the transposition table. Not only the result values will help to create cutoffs and narrow down the search window, more importantly, it also provides the best successor of the shallow search which is very likely a good candidate for the deeper search.

We start the iteration from the 4th step. The search goes one step deeper each time.

Move#	Node			Time		
	-ID	+ID	Save	-ID	+ID	Save
12g	38472802	13270154	65.5%	236s	118s	50%
17s	104899599	40836632	61.1%	883s	368s	58.3%
22g	48433140	52550287	-8.5%	351s	365s	-4%

Table 5.1: Result of using the Iterative Deepening. It is based on a 9-step Alpha-Beta search, without any other enhancements. The moves are taken from the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.

Table 5.1 shows the effect of Iterative Deepening. Typically it saves about 50% of the searching time and nodes. In an infrequent case, like the move “22g” where the best move is already at the top of the list in the original order, Iterative Deepening won’t give any saving.

5.2.2 Killer Move

If the best successor of the transposition table is not applicable or fails, we will try using the killer moves. Two killer moves are maintained at each ply, and a record of the numbers of cutoffs that caused by each of them is also kept. A successful cutoff by a non-killer move overwrites one of the killer moves which created fewer cutoffs for that ply.

Like using the best successor of the transposition table, using the killer moves doesn't require the program to generate all the successors of a node. The program tests if a killer move is a valid successor. If it is, then the move is generated and tried. We only generate all the successors when the best successor and killer moves fail. It makes the search faster.

Move#	Node			Time		
	-KM	+KM	Save	-KM	+KM	Save
12g	13270154	2268722	82.9%	118s	23s	80.6%
17s	40836632	3613778	91.2%	368s	34s	90.8%
22g	52550287	6578833	87.5%	365s	46s	87.4%

Table 5.2: Result of using the Killer Move. It is based on a 9-step Alpha-Beta search with Iterative Deepening. The moves are taken from the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.

Table 5.2 shows the difference that Killer Move makes. It saves about 80% to 90% of the search effort, which is a very improvement.

5.2.3 History Heuristic

We still need some way to improve the order of the other moves in the queue, in case the best successor of transposition table and the killer moves are not applicable or fail to cause a cutoff.

In Arimaa, some moves obviously have a better chance to be a good candidate than other moves. For example:

- A move traps an enemy piece.
- A move contains pulling or pushing.
- A move in step combo type $A(4)$ or $B(3+1)$.
- A move freezes enemy pieces, or unfreezes friend pieces.
- A move in which the Elephant changes its position.
-

The knowledge listed above enables us to build a game-specific move-sorting function, but it is complicated to implement and tune.

In our program, we adopt a more popular method to address the move ordering problem: the History Heuristic [14]. This is implemented by maintaining a table for all possible steps (which is $2*6*64*4$ in total). If a step becomes the best one in the search, the program will add 2^{depth} to the history score of this step. We sort the moves based on this history value.

How to sort the nodes is a problem that needs to be addressed. One way is using some fast sorting method to sort all of them, but if a cutoff happens early on, then the time spend on the sorting is wasted. A partial sorting (only order the best n moves) is another approach, but how to decide this n value and handle the situation if the best n moves fail is still annoying.

In our program, we avoid sorting all the nodes at the same time. Instead, every time our sorting function only provides one best successor to the search engine, and remove it from the successor list. If this node fails, the sorting function will be called again to get the second best successor, and so on. Thus if the cutoff occurs soon, which is very likely, we won't spend a lot of time in sorting. Although the sorting function itself is slow, the overall sorting time is reduced.

Move#	Node			Time		
	-HH	+HH	Save	-HH	+HH	Save
12g	2268722	1853626	18.3%	23s	19s	17.4%
17s	3613778	2740167	24.2%	34s	27s	20.6%
22g	6578833	3189776	51.5%	46s	28s	39.1%

Table 5.3: Result of using the History Heuristic. It is based on a 9-step Alpha-Beta search with Iterative Deepening and Killer Move. The moves are taken from the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.

Table 5.3 shows the result of History Heuristic. Normally it saves about 20% to 40% searching nodes and time.

5.3 Null Move

The Null Move heuristic in our program was originally designed in this way: If the program was searching a complete move (on level $4n$, like 4, 8, 12...), it would skip a whole move (4 steps) in the null move testing; if the search was at another level ($4n+m$, $0 < m < 4$, like 5, 6, 7, 9, 10...), it would skip the rest of the steps ($4-m$) in the same move.

We observed that null move cut-offs could happen at any level, but it only made a big improvement at the $4n$ levels. At the other levels ($4n+m$, $0 < m < 4$), the pruning nodes are so few that it is not worth the time spent on null move testing (see Table 5.4). Based on this result, in our program we only test null-moves on the $4n$ levels.

	NM tried	Cutoff	Rate
Level4	10934	6269	57.3%
Level5	4570	77	1.7%
Level6	8054	263	3.2%
Level7	63522	54	0.1%
Level8	660311	536303	81.2%

Table 5.4: Null Move cutoffs at different levels. It is the 22g move of the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.

Move#	Node			Time		
	-NM	+NM	Save	-NM	+NM	Save
12g	1853626	1828702	0%	19s	19s	0%
17s	2740167	2334584	14.8%	27s	22s	18.5%
22g	3189776	2425144	24%	28s	20s	28.6%

Table 5.5: Result of using the Null Move. It is based on a 9-step Alpha-Beta search with Iterative Deepening, Killer Move and History Heuristic. The moves are taken from the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.

Table 5.5 shows the result of using the Null Move. Normally it saves about 20% to 30% in nodes and time. In some cases, such as the move “12g”, there is no any saving at all.

We cannot explain what makes the move “12g” special.

5.4 Transposition Table

The transposition table in our program is a hash table that contains 2^{20} entries. Any game state in our program is represented by a 96-bit number, which is created using Zobrist’s hashing method [22]. The last 20 bits of the Zobrist’ hash value of a state are used to map the index of the table entry of this state [12].

In each entry of the hash table, we save the search bounds, the best successor, the search depth, the complete Zobrist’ hash value and the search result. If a collision happens, we always keep the state with the deeper search depth.

In game-playing programs, normally a transportation table can reduce the search tree size dramatically. Unfortunately it doesn’t help much in the game of Arimaa. The reason for this is that a transportation table only works well if the program searches 3 moves or deeper. Not many repetition states with different paths will show up if the program searches shallower than that. Three moves are 12 steps in Arimaa, which is too deep to finish in a reasonable time. Since our program normally only searches 10-11 steps in 3 minutes, the transportation table cannot provide many cutoffs.

Move#	Node			Time		
	-TT	+TT	Save	-TT	+TT	Save
12g	1828702	2045876	-11.9%	19s	20s	-5.2%
17s	2334584	2262193	3.2%	23s	23s	0%
22g	2425144	2396387	1.2%	20s	20s	0%

Table 5.6: Result of Transposition Table. It is based on a 9-step Alpha-Beta search with all the enhancements mentioned above. The moves are taken from the first game in the 2005 world Arimaa championship final, Karl Juhnke versus Frank Heinemann.

Table 5.6 shows the result of Transposition Table. Sometime it makes a small improvement; sometimes it makes the search a little slower. As long as we cannot search over 12-step deep, removing it from the program won't show much difference.

5.5 Razoring

Razoring is a method that uses the static evaluation values to create cutoffs [1]. The basic idea of this method is based on an assumption: For any state, the player to move must have at least one move that can improve his position. Hence, after taking his best move the state should be better than before for him. This assumption is not always true for Chess because of the existence of zugzwang.² This problem may exist in Arimaa as well, and needs more research.

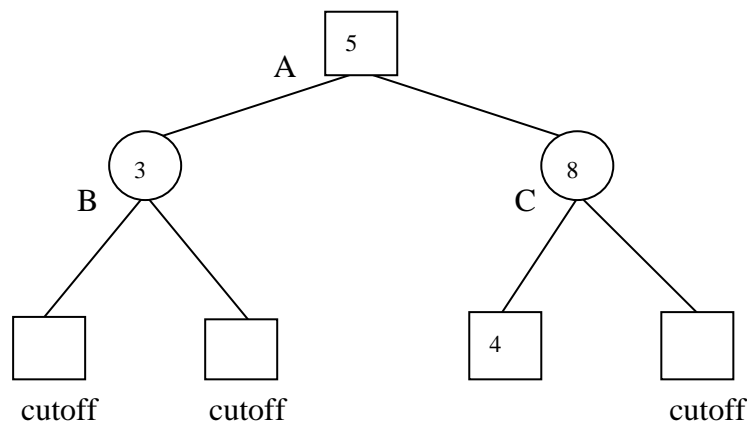


Figure 5.2: The Alpha-Beta tree for razoring.

² Zugzwang is a state in which every move available to the players to move creates a position worse than the present board position for them. In other words, "pass" is the best choice if it is applicable.

Consider the tree in Figure 5.2. The root node A has a static evaluation value of 5. Node B's static evaluation value is 3, which is less than 5. From the assumption introduced above, the best successor of node A cannot be worse than node A (which means the value of the best successor cannot be lower than 5, since it is Max's turn), therefore it cannot be node B. We can eliminate B without examining its sub-tree. For node C, which has a static evaluation value of 8 and is better than node A, we can use 5 as its lower bound to narrow down the search window. Node C's first child returns 4, which is less than 5, so all its other successors can be eliminated.

In the game of Arimaa, Razoring only works when a full move is completed, in other words, every 4 steps ($4n$). Since the program cannot complete a 3-move (12-step) search in a reasonable time, Razoring doesn't enable much pruning this way.

The concept of step combo that we introduced in Chapter 4 may help us do more. When a step combo is finished, since the rest of the steps of a move have nothing to do with the steps that already been taken, we can apply Razoring based on the static values. That enables us to razor nodes in the $4n+2$ and $4n+3$ levels (since during the move generation we already removed all those moves in which the first 2 steps are not contiguous; we cannot razor in the $4n+1$ levels.).

The problem in this method is that we have to make sure that in the pruning of the duplicated moves, we kept the moves with the best step-combo order. For example, "**Ea2n Ea3n Ea4n Rb1n**" is a $B(3+1)$ move, which has an identical move "**Ea2n Ea3n Rb1n Ea4n**" in $D(2+1+1)$ -- we have to keep the first one and remove the second. The order of the step-combo still makes a difference, even among the identical moves with the same type.

We haven't successfully implemented this method because it is not easy to solve the step-combo order problem. But we think it is an interesting thought and worth further investigation.

Unlike Alpha-Beta, Razoring cannot guarantee finding the best decision. This algorithm assumes that a Max node's static evaluation value (evaluation without search) is a lower bound of its dynamic evaluation value (evaluation after search its sub tree); and a Min node's static evaluation value is an upper bound of its dynamic evaluation value. It is not a safe assumption. If the best move gets a short-term lower score but in the long-term a higher score, this move could be overlooked.

It is easy to see that the severity of this "short-sight" problem in Razoring is closely related to the search depth. This idea was introduced in 1970's Chess programs, when 6-ply was considered a deep search, and has been abandoned in today's Chess programs, where Deep Blue can easily search over 13-ply deep. We believe that it is a suitable technique for Arimaa, since in this game searching 4-ply is still a target beyond reach currently.

5.6 Summary

We implemented many enhancements into the Alpha-Beta search algorithm. But because the branching factor is too large in Arimaa, the program still cannot search very deep.

So far we can only finish searching 10 or 11 steps in 3 minutes. Other top Arimaa programs, like the world champion bot_bomb, are at the same level. To make the program distinctly stronger, we need to reach searching 13-step, which is very difficult to do. There is a bottleneck at 12-step that prevents the Transposition Table from being effective.

We believe the "Enhanced Razoring" method introduced above has potential and is worth some investigation. A deeper forward pruning enhancement is another interesting way to put effort.

There is a serious horizon effect problem in our program, where a bad move hides an even worse threat because the threat is pushed beyond the search horizon. Normally

this problem is solved by Quiescence Search, which makes sure that evaluations are done at stable positions, i.e. positions where there are no direct threats. But in Arimaa, where every move is composed of up to 4 steps, Quiescence Search is not practical for it is too costly. More research is needed to address this problem.

Chapter 6

FUTURE WORK

Omar Syed, the designer of Arimaa was so confident in the computer's disadvantage compared to humans in this game, that he offered a prize to any computer program that could defeat the human champion in the next 20 years [17]. From the result of the first two official Man vs. Machine Arimaa matches (the best Arimaa program was defeated 8-0 and 7-1 by the human champions), it seems that there is still a long way to go before a computer can beat the best humans in this game.

6.1 Evaluation

As a deeper understanding of this new game is gained, it becomes easier for us to improve the evaluation function by updating the features and refining the weight values. The official website of Arimaa (www.arimaa.com) released the Arimaa Games Archive in September 2004, which means that all the games played in the Arimaa game room since the game was first released are available for research. It gives us the chance to run the program over thousands of quality matches, to analyze the difference in selecting moves between superior human players and our program, and improve the latter [4].

In trap/goal evaluation, our program can only find the trap/goal situations 3 steps ahead so far. Making it 4 steps is doable, and it will improve the program's play.

There are many interesting ideas for creating an Arimaa evaluation function that were discussed in the Arimaa website forum [17], but were ignored in our program for various reasons. These ideas include Dynamic Piece Value, Basic Piece Value, Trap Control and

Rabbit Formation. We believe some of them may make a performance difference, and all of them are worth more investigation.

We also need to do more in letting the computer study and discover the knowledge of this game automatically. Though we failed in making TD(λ) method tuning the weights of features in Arimaa, we still believe it is feasible. The result of TD(λ) is search-depth sensitive, which means the result obtained from shallow search games may not serve the deeper searching program very well [16]. Therefore, if we can make TD(λ) works with shallow search (less than 5-step) games, the next step is to do TD(λ) tuning by playing games that search deeper (over 9-step). There is a lot to do in this area.

The endgame and opening database technique was successfully used in the Checkers program Chinook [5, 20]. In the game of Checkers, there are only two piece types and only a few pieces left in an endgame, which makes it easy to use an endgame database. Arimaa has different piece types and usually there are many pieces left in the endgame, which means it is not suitable for an endgame database. The opening setup in Arimaa is arbitrary, but only some of them are worth playing. We believe after some research, an opening book database for Arimaa can be created.

6.2 Move Generation

We put a lot of effort to make our program generating moves fast, but there is still potential for improving it.

One task in move generation of Arimaa is removing all the repetition moves, which takes a big portion of the move generating time. If we can find a way that is able to automatically skip generating all or a big portion of repetition moves, instead of generating them first and testing/removing them later, it will save a lot of computing time. We don't know whether this idea is doable. Even if it is, it might be very complicated to implement.

We invented the concept of step combo, and successfully pruned over 1/3 of the moves before search by using it. We believe with more research this idea can create a bigger difference. Our forward pruning is rather conservative. A more aggressive approach might be a feasible option. Furthermore, the step combo information can be fed into the search engine, helping building a better move ordering and causing more cutoffs.

6.3 Game-Tree Pruning

There is also potential for improving the game-tree search. So far we can only search about 10-step in 3 minutes. Breaking the bottleneck of searching 12-step, which prevents the Transposition Table from being effective, is very important. Searching 13-step will make the program distinctly stronger.

Since there is a lot of knowledge available in Arimaa for distinguishing the good candidate moves from the slim-chanced moves, a well-designed application-specified move ordering method might work better than the History Heuristic.

Razoring is a powerful method which can create many cutoffs. It is suitable for a shallow searching situation, as in an Arimaa-playing program. The problem that prevents it from being fully functional in Arimaa is that a move must be treated as a whole, which means Razoring can only be used every 4 steps. Step combo enables Razoring pruning at different steps, instead of different moves, which will make a big difference. It needs the moves generated in the best step combo order, which is not easy to implement.

Consider the large branching factor in Arimaa, even getting close to the Alpha-Beta best case search tree is still challenging. A deep forward pruning is worth more research, to balance the speed and the chance of losing good moves. An Arimaa-playing program called `bot_cruelless`, created by Jeff Bacher, uses deep forward pruning; it achieved second place at the 2005 world computer Arimaa tournament.

6.4 Comparison with Fortland's Work

David Fortland's Arimaa program won the computer world championship in 2004 and 2005. He wrote an article in 2004 to give a brief introduction to his work. It gave us an opportunity to study his methods and compare it to what we have in our program [8].

It seems that the move generator in Fortland program doesn't remove repetition moves. Instead, the redundant nodes caused by repetitions are removed by the transposition table in searching. This approach is easy and simple, but we believe it costs more than our way.

In move generation, Fortland program doesn't make any forward pruning. It doesn't prune any reversible moves either, although Fortland noticed this problem.

The evaluation function in Fortland's program has different terms, such as Rabbit Formation. Fortland did not use temporal difference or any other machine learning methods; he adjusts all the evaluation weights manually.

In his evaluation function, there is also a trap/goal evaluating decision tree (in fact we got this idea from him). But he doesn't evaluate trap/goal by completing unfinished steps left in the current move as we did. Instead, he statically evaluates how many steps it will take to trap each piece on the board (1 to 6 steps) or have each Rabbit reach the goal (1 to 8 steps), assuming no intervening moves by the opponent. This implementation is much difficult than our way, and the idea behind it is not very clear. We don't fully understand the benefits of this approach, and we believe that our way is more reasonable.

The search enhancements used in Fortland's program includes Transposition Table, Null Move, Iterative Deepening, Killer Heuristic, History Heuristic and Search Extension, pretty much everything we have in our search engine. However the implementation seems to be different.

The Search Extension in Fortland’s program enables a highly pruned search to find defenses against goal threats, which is not a part of our program. According to Fortland, his program usually finds 20 step goal sequences within the 3 minute time limit, when it only completes 10 or 11 full steps of search.

6.5 Result

Rank	Name	Author	Play Gold		Play Silver	
			Win	Lose	Win	Lose
1	Bomb	David Fortland	1	4	1	4
2	Clueless	Jeff Bacher	5	0	4	1
3	GnoBot	Toby Hudson	5	0	5	0
4	Loc	Gerhard Trippen	4	1	4	1

Table 6.1: Play against the top 4 programs of the 2005 Computer Arimaa Championship. The games were played in 15 seconds per move, 2 minutes in the starting reserve.

Our program didn’t attend 2005 Computer Arimaa Championship, so it does not have an official rank.

Table 6.1 is the result of our program playing against the top 4 Arimaa-playing programs (15 seconds per move, 2 minutes in the starting reserve). It shows that our program still cannot beat David Fortland’s champion program, but is stronger than other 3 programs.

Our program searches as fast as David Fortland’s program. We believe the reason of its being outplayed is mainly in the evaluation function. To beat Fortland’s program, we need to improve our evaluation function by carefully checking and updating the terms, and making $TD(\lambda)$ working to tune the weights.

BIBLIOGRAPHY

- [1] Birmingham, J.A. and Kent, P. (1977). Tree-searching and Tree-pruning Techniques, *Advances in Computer Chess* 1, M.R.B. Clarke (ed.), pages 89-97.
- [2] Breuker, D., Uiterwijk, J. and van den Herik, J. (1996). Replacement Schemes and Two-level Tables, *ICCA Journal*, vol.19, no.3, pages 175-180.
- [3] Buro, M. (1997). *Logistello: A Strong Learning Othello Program*, NEC Research Institute. Princeton NJ.
- [4] Buro, M. (1998) *From Simple Features to Sophisticated Evaluation Functions*, *Computers and Games*, Springer Verlag, LNCS 1558, pages 126-145.
- [5] Buro, M. (1999). Toward Opening Book Learning, *ICGA Journal*, vol. 22, no. 2, pages 98-102.
- [6] Campbell, M and Marsland, T.A. (1983). A Comparison of Minimax Tree Search Algorithms, *Artificial Intelligence*, vol. 20, no. 4, pages 318-334.
- [7] Culberson, J. and Schaeffer, J. (1998). Pattern Databases, *Computational Intelligence*, vol. 14, no. 4, pages 318-334.
- [8] Fotland, D. (2004) *Building a World Champion Arimaa Program*, *Computers and Games 2004*, unpublished.
- [9] Hsu, F. (2002). *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*, Princeton University Press.
- [10] Juhnke, K. (2004). *Arimaa: Strategy and Tactics*, en.wikipedia.org/wiki/Arimaa.
- [11] Knuth, D. and Moore, R. (1975). An Analysis of Alpha-Beta Pruning, *Artificial Intelligence*, vol. 6, no.4, pages 293-326.

- [12] Plaat, A., Schaeffer, J., de Bruin A. and Pijls W. (1996) Exploiting Graph Properties of Game Trees, AAAI, pages 234-239.
- [13] Reinefeld, A. and Marsland, T. A. (1987). A Quantitative Analysis of Minimax Window Search, 1987 IJCAI, pages 951-954.
- [14] Schaeffer, J. (1989). The History Heuristic and the Performance of Alpha-Beta Enhancements, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol.11, no.11, pages 1203-1212.
- [15] Schaeffer, J. (1997). One Jump Ahead - Challenging Human Supremacy in Checkers, Springer-Verlag, New York.
- [16] Schaeffer, J., Hlynka, M and Jussila, V. (2001) Temporal Difference Learning Applied to a High-Performance Game-Playing Program, IJCAI, pages 529-534.
- [17] Syed, O. (2001). Arimaa -The Game of Real Intelligence, www.arimaa.com.
- [18] Syed, O. (2001). Official rules and notation, www.arimaa.com.
- [19] Thompson, K. (1982). Computer Chess Strength, Advances in Computer Chess 1, M.R.B. Clarke (ed.), vol.3, pages 55-56.
- [20] Thompson, K. (1986). Retrograde Analysis of Certain Endgames, ICCA Journal, vol. 9, no. 3, pages 131-193.
- [21] Ye, C. and Marsland, T.A. (1992). Experiments in Forward Pruning with Limited Extensions, ICCA Journal, vol. 15, no. 2, pages 55-66.
- [22] Zobrist, A. (1970). A New Hashing Method with Applications for Game Playing, ICCA Journal, vol.13, no.2, pages 69-73.

APPENDIX: EVALUATION VALUE TABLES

8	0	1	1	2	2	1	1	0
7	1	2	2	3	3	2	2	1
6	2	4	-4	10	10	-4	4	2
5	3	8	12	12	12	12	8	3
4	3	8	10	10	10	10	8	3
3	2	4	-4	10	10	-4	4	2
2	1	2	2	3	3	2	2	1
1	0	1	1	2	2	1	1	0
	A	B	C	D	E	F	G	H

(a) Elephant position value.

8	-2	-1	0	1	1	0	-1	-2
7	-1	0	1	2	2	1	0	-1
6	0	1	2	3	3	2	1	0
5	1	2	3	4	4	3	2	1
4	2	3	4	5	5	4	3	2
3	1	2	3	4	4	3	2	1
2	0	1	2	3	3	2	1	0
1	-1	0	1	2	2	1	0	-1
	A	B	C	D	E	F	G	H

(b) Camel position value.

8	-2	-1	0	1	1	0	-1	-2
7	-1	0	1	2	2	1	0	-1
6	0	1	-5	3	3	-5	1	0
5	1	2	3	4	4	3	2	1
4	2	3	4	5	5	4	3	2
3	1	4	2	3	3	2	4	1
2	0	1	2	3	3	2	1	0
1	-1	0	1	2	2	1	0	-1
	A	B	C	D	E	F	G	H

(c) Horse position value.

8	-4	-6	-6	-6	-6	-6	-6	-4
7	-7	-8	-10	-9	-9	-10	-8	-7
6	-9	-10	-12	-11	-11	-12	-10	-9
5	-6	-8	-9	-9	-9	-9	-8	-6
4	-3	-5	-6	-6	-6	-6	-5	-3
3	-1	-1	-2	-3	-3	-2	-1	-1
2	1	2	4	2	2	4	2	1
1	0	1	1	1	1	1	1	0
	A	B	C	D	E	F	G	H

(d) Dog position value.

8	-4	-6	-6	-6	-6	-6	-6	-4
7	-7	-8	-10	-9	-9	-10	-8	-7
6	-9	-10	-12	-11	-11	-12	-10	-9
5	-6	-8	-9	-9	-9	-9	-8	-6
4	-3	-5	-6	-6	-6	-6	-5	-3
3	-1	-2	-2	-3	-3	-2	-2	-1
2	1	2	4	2	2	4	2	1
1	0	1	1	1	1	1	1	0
	A	B	C	D	E	F	G	H

(e) Cat position value.

8	INF	INF	INF	INF	INF	INF	INF	INF
7	-14	-16	-8	-16	-16	-8	-16	-14
6	-16	-10	-16	-10	-10	-16	-10	-16
5	-8	-12	-12	-12	-12	-12	-12	-8
4	-5	-8	-8	-8	-8	-8	-8	-5
3	-3	-5	-6	-6	-6	-6	-5	-3
2	-1	-2	-1	-4	-4	-1	-2	-1
1	0	0	0	0	0	0	0	0
	A	B	C	D	E	F	G	H

(f) Rabbit position value (normal).

8	INF	INF	INF	INF	INF	INF	INF	INF
7	25	25	25	25	25	25	25	25
6	20	20	20	20	20	20	20	20
5	16	15	14	14	14	14	15	16
4	8	7	6	6	6	6	7	8
3	4	3	2	2	2	2	3	4
2	2	1	1	1	1	1	1	2
1	0	0	0	0	0	0	0	0
	A	B	C	D	E	F	G	H

(g) Rabbit position value (motivate).

Table 1: Position value table for the gold pieces. The values for the silver pieces are vertically reversed.

8	-20	-24	-28	-26	-26	-28	-24	-20
7	-24	-30	-20	-32	-32	-20	-30	-24
6	-26	-20	0	-20	-20	0	-20	-26
5	-16	-12	-15	-10	-10	-15	-12	-16
4	-10	-7	-7	-7	-7	-7	-7	-10
3	-6	-2	0	-2	-2	0	-2	-6
2	-2	-2	-2	-2	-2	-2	-2	-2
1	-2	-2	-2	-2	-2	-2	-2	-2
	A	B	C	D	E	F	G	H

(a) Camel frozen position value

8	-20	-24	-28	-26	-26	-28	-24	-20
7	-24	-30	-20	-32	-32	-20	-30	-24
6	-26	-20	0	-20	-20	0	-20	-26
5	-16	-12	-15	-10	-10	-15	-12	-16
4	-10	-7	-7	-7	-7	-7	-7	-10
3	-6	-2	0	-2	-2	0	-2	-6
2	-2	-2	-2	-2	-2	-2	-2	-2
1	-2	-2	-2	-2	-2	-2	-2	-2
	A	B	C	D	E	F	G	H

(b) Horse frozen position value

8	-2	-2	-3	-2	-2	-3	-2	-2
7	-3	-10	-5	-8	-8	-5	-10	-3
6	-10	-5	0	-3	-3	0	-5	-10
5	-3	-3	-3	-3	-3	-3	-3	-3
4	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	0	-1	-1	0	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1	-1	-1	-1
	A	B	C	D	E	F	G	H

(c) Dog frozen position value

8	-1	-1	-2	-1	-1	-2	-1	-1
7	-2	-9	-5	-7	-7	-5	-9	-2
6	-9	-5	0	-2	-2	0	-5	-9
5	-2	-2	-2	-2	-2	-2	-2	-2
4	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	0	-1	-1	0	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1	-1	-1	-1
	A	B	C	D	E	F	G	H

(d) Cat frozen position value

8	0	0	0	0	0	0	0	0
7	-1	-1	-1	-1	-1	-1	-1	-1
6	-1	-1	0	-1	-1	0	-1	-1
5	-1	-1	-1	-1	-1	-1	-1	-1
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	A	B	C	D	E	F	G	H

(e) Rabbit frozen position value

Table 2: Frozen position value table for the gold pieces. The values for the silver pieces are vertically reversed.

Scale	0	1	2	3	4	5
Score	0	5	10	30	50	80

Table 3: Elephant blockade table. The scale indicates the blockade level, 0 for free and 5 for a full blockade.

Camel	50
Horse	40
Dog	30
Cat	25
Rabbit	15

Table 4: Fork table.

Camel	0
Horse	25
Dog	20
Cat	10
Rabbit	15

Table 5: Pin table.

Camel	80
Horse	0
Dog	30
Cat	20
Rabbit	10

Table 6: Hostage table.