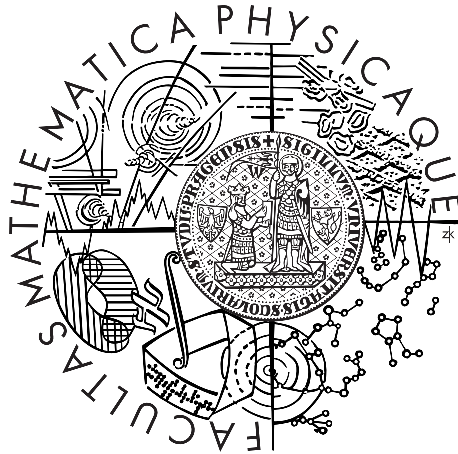
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Ondřej Majerech

PNS for the game Arimaa

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: RNDr. Jan Hric

Study programme: Computer Science

Specialization: General Computer Science

Prague 2012

I would like to thank my thesis supervisor for his numerous remarks and invaluable insight into the topic of Arimaa and artificial intelligence.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Libiš date August 1, 2012 Ondřej Majerech

Title: PNS for the game Arimaa

Author: Ondřej Majerech

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jan Hric, Department of Theoretical Computer Science and Mathematical Logic

Abstract: The game of Arimaa is a strategic board game that has proved to be a challenge to computers. Not only because of its huge branching factor, but also thanks to the difficulty in creating a good evaluation function to be used with the Alpha-Beta algorithm. Proof-Number Search is an algorithm that does not depend on a heuristic evaluation function and it has been successfully applied to solving endgames of various other games. In this work, we adapt and implement the Proof-Number Search method for the game of Arimaa.

Keywords: Proof-Number Search, Arimaa, endgames, artificial intelligence

Název práce: PNS for the game Arimaa

Autor: Ondřej Majerech

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Jan Hric, Katedra teoretické informatiky a matematické logiky

Abstrakt: Arimaa je strategická desková hra, kterou se stále počítačům nepovedlo pokořit. Problematický je zejména velký větvící faktor stromu hry a celková její charakteristika, díky které je obtížné sestrojit vhodnou ohodnocovací heuristiku pro alfa-beta prohledávání. Proof-Number Search je algoritmus nezávislý na dobré ohodnocovací heuristice, který byl již úspěšně použit na řešení koncovek jiných deskových her. V této práci jsme adaptovali a implementovali algoritmus Proof-Number Search pro hru Arimaa.

Klíčová slova: Proof-Number Search, Arimaa, koncovky her, umělá inteligence

Contents

1 Introduction **3**
1.1 Research Objective . . . 3

2 The Game of Arimaa **4**
2.1 Rules of Arimaa . . . 4
2.2 Difficulty for Computers . . . 5

3 Search Algorithm **7**
3.1 Proof-Number Search . . . 7
3.1.1 Representing a Game with an AND-OR Tree . . . 7
3.1.2 Proof and Disproof Numbers . . . 8
3.1.3 Expansion of the Tree . . . 8
3.2 Depth-First PNS . . . 9
3.3 Adaptation of PNS for the game of Arimaa . . . 10
3.3.1 Steps versus Moves . . . 10
3.3.2 Double Steps . . . 11
3.3.3 Lambda Vertices . . . 11
3.3.4 New Vertices . . . 12

4 Enhancements of the Algorithm **13**
4.1 Zobrist Hashing . . . 13
4.2 Transposition and Proof Tables . . . 14
4.2.1 Transposition Table . . . 15
4.2.2 Proof Table . . . 16
4.3 Step Ordering . . . 17
4.3.1 Static Step Ordering . . . 17
4.3.2 Killer-move Heuristics . . . 18
4.4 Garbage Collector . . . 18

5 Implementation **20**
5.1 Program Structure . . . 20
5.2 Board Representation . . . 20

6 Experiments **22**
6.1 Results . . . 22

7 Conclusion **26**
7.1 Future Work . . . 26

Bibliography **28**

Glossary **30**

A User's Manual **31**
A.1 Supported Systems 31
A.2 Running the Program 31
A.2.1 Graphical Interface 31
A.2.2 Batch Interface 32
A.3 Building the Program 33

Attachments **34**

1. Introduction

Ever since the dawn of the field of artificial intelligence in the 1950's, board games such as Chess have been attracting the interest of computer scientists and engineers. Due to the lack of processing power in the 1950's, computers struggled to reach even the level of amateur human players. [15, 6]. However, as time progressed and the raw computing capacity of computers increased, game-playing programs also became stronger. Finally, in 1997, a computer beat Garry Kasparov, the reigning Chess champion at that time. After Kasparov's defeat, computers have gradually become capable of regularly beating human players in Chess. [17]

With Chess beaten, the interest shifted to more challenging games, one of which is the game of Go. The Alpha-Beta algorithm – commonly used for computer Chess – is of little success in computer Go, because of the greater branching factor and difficulty in constructing a fast and accurate positional evaluation function. So far, the *Monte-Carlo Tree Search* (MCTS) method has had more success in computer Go than Alpha-Beta. [17, 9]

In June 2003, Omar Syed introduced a new challenging game: *Arimaa*. [14] It is a game specifically designed to be hard for computers yet easy for human players. It is a game similar to Chess but with much larger branching factor.

Because of the similarity with Chess, the Alpha-Beta method has been applied to Arimaa – however, its application to Arimaa suffers similar problems as the application to Go – the branching factor of Arimaa is even larger than that of Go, and constructing a good and fast positional evaluation function is also challenging. [7] In 2009, Tomáš Kozelek ([9]) brought the MCTS technique from Go to Arimaa. While this technique does not depend on a good evaluation function as heavily as Alpha-Beta, it is a statistical method that is unsuitable for determining the game-theoretic value of a position, and it is weak for tactical positions.

In our thesis we present an implementation of another method: An adaptation of the *Proof-Number Search* algorithm. Like the Alpha-Beta algorithm, it can determine the exact minimax value of a game tree. However, instead of relying on a positional evaluation function, it creates non-symmetric trees and uses the shape of the tree to guide the search in a best-first manner. Proof-Number Search can be used for post-mortem analysis of a game position, solving whole games, or tournament play. [2]

1.1 Research Objective

In this work we implement the Proof-Number Search method for solving Arimaa endgame positions: Positions which can presumably be won by one of the players in a small number of turns. We also adapt and implement some known techniques for improving the performance of Proof-Number Search and observe their benefits when used in application for the game of Arimaa. As a part of this work, we develop a program that allows the user to execute the presented algorithm on a given position and observe its behaviour, including the game tree created during the course of the algorithm.

2. The Game of Arimaa

Arimaa is a two-player game with perfect information. [14] It was designed to be playable on a standard Chess board with Chess pieces. At the same time, it was designed to be difficult for computers to solve using the tree-search approach by having a very large branching factor – the number of valid moves from a given position is in the range of thousands and may even reach as high as 50,000. [14] Furthermore, the game is designed to be more strategic than Chess, with more emphasis on long-term goals and less emphasis on tactical situations.

Omar Syed announced a US $ 10,000 prize for the first person to develop a computer program that can beat the best human player by 2020. With current techniques supposedly inefficient against Arimaa, Syed believes this prize will motivate the development of new approaches for computer game playing.

2.1 Rules of Arimaa

Arimaa can be played on a standard Chess board, however four special squares need to be marked – these are the squares c3, f3, c6, f6 (using standard Chess notation), called *trap squares* or just traps. The standard set of Chess pieces can also be used with Arimaa, but the names are different from those in Chess. Each player starts the game with the following pieces: One *elephant*, one *camel*, two *horses*, two *dogs*, two *cats*, and eight *rabbits*. Instead of white and black, the colours of pieces are *gold* and *silver*.

The starting position is not fixed; instead, at the start of the game, the gold player places all of their pieces on the first two ranks of the board. Once the gold player is done placing their pieces, the silver player then chooses the initial placement of their pieces, placing them all on the last two ranks of the board (the two ranks closest to the player).

The players then take turns moving their pieces, with gold player the first to move. Except for rabbits, all pieces may move one square up, down, left or right; rabbits may only move up or to the sides but not backwards. Pieces are allowed to move only to empty squares. A player may make one to four *steps* per turn, where each step is a movement of one piece. Each player's turn must result in a net change in the game position (other than letting the opponent play).

Each piece has an associated *strength*. Elephants are the strongest pieces, followed by camels, horses, dogs, cats and finally rabbits who are the weakest.

There are three kinds of steps in Arimaa:

Single steps The movement of a piece to a neighbouring square.

Pushes If a current player's piece is standing on a square next to a position (horizontally or vertically) that is occupied by a weaker opponent's piece, this weaker piece may be *pushed*. First the weaker opponent's piece is moved to an empty adjacent square, and then the player's stronger piece is moved to the now-empty square which the weaker piece originally occupied.

Pulls Similar to push steps, these are only possible with pieces standing next to weaker opponent's pieces. First the stronger piece is moved to an unoc-

cupied neighbouring square, then the weaker piece is moved to the square where the first piece originally stood.

Push and pull steps are called *double steps* in this work as they move two pieces. They also count as two steps, so if a player makes a push or pull step at the beginning of their turn, they only have two more steps available to them within that turn. Both piece movements in a double step have to follow each other and both have to be done within the same turn (which means that if a player has already taken three steps, they may not make a push or pull step within their current turn). Furthermore, rabbits may be pushed backwards even though they may not move that way on their own. It is not allowed to push or pull one's own pieces, and neither is it allowed to simultaneously push and pull multiple pieces, or to simultaneously pull multiple pieces.

If a piece is standing next to a position occupied by a stronger opponent's piece and there is no piece of the player's colour on any of the remaining surrounding squares, the piece is said to be *frozen*. (If there is such a piece of the same colour, it is called a *supporting piece* or *supporter*.) A player is not allowed to move their frozen pieces; however they may push or pull an opponent's frozen piece. The strength of the supporting piece plays no role here, and neither does the fact whether it is frozen as well or not – merely its presence on a neighbouring square is important. Pieces of the same strength but of opposite colours do not affect each other.

If a piece happens to be standing on a trap square (c3, f3, c6, or f6) and there is no supporter on any of the surrounding squares, the piece is *captured*: it is removed from the board and never returns to the game. Just as with frozen pieces, it does not matter whether the supporting piece is stronger or weaker, or whether it is frozen – the only condition is that it belongs to the same player as the piece standing on the trap and that it is positioned next to the trap.

The goal of the game is to get one's rabbit to the last row of the board, called the *goal row*. If a player's rabbit reaches the goal row, the player wins.

The game also ends when a player's last rabbit is captured in which case this player is said to be *eliminated* and loses the game. Finally, if a player is on turn but there is no allowed move they could make, that player is *immobilized* and loses the game.

To guarantee that each game is finite, there is the so-called *third-time repetition rule*: If the same position occurs for the third time within a game, the player who caused the third repetition loses the game.

Further details about the rules can be found in [3].

2.2 Difficulty for Computers

As we stated earlier, the main difficulty of Arimaa lies in its huge branching factor, which means that brute-force methods (such as Alpha-Beta or Proof-Number Search) have to explore many positions in order to fully explore the game tree to some depth (and thus determine its minimax value). Another reason is Arimaa's emphasis on strategy rather than tactics. [14] Long-term goals are important in Arimaa and captures can be postponed for longer than in Chess, leading to less cut-offs in Alpha-Beta searches.

Furthermore, static position evaluation it is not easy. Material balance alone is not sufficient to compare positions. Aspects such as trap control, mobility of pieces or balance of forces need to be taken into account. The value of individual pieces is not fixed, but rather changes as the game progresses. For instance, the value of an individual rabbit initially low, but it increases as the number of remaining rabbit lowers as the player needs rabbits to defend their own goal row, make goals threats and even not to lose the game due to elimination.

Attacks in Arimaa are slower than in Chess in the sense that they span more turns: A piece may move only at most 4 squares away from its starting square during a player's turn. For a program to compete with strong human players, it needs to look ahead at least 5 moves. [7]

As the starting position of Arimaa is not fixed, opening databases like in Chess are of little use. There are almost 65 million possible initial set-ups. [6] Likewise, endgame databases are not effective because most games end with the board still relatively crowded in comparison to Chess.

3. Search Algorithm

In this chapter we describe how a game can be represented by an AND-OR tree, and the algorithm used to search the AND-OR tree to find a solution. We also describe one of the variants of the algorithm, and our adaptation for the game of Arimaa.

3.1 Proof-Number Search

The Proof-Number Search (PN Search or PNS) algorithm as described by Allis ([1]) is a best-first algorithm that works by maintaining and expanding an *AND-OR tree* – a tree with two *types* of nodes: *AND* and *OR*. The algorithm is then guided by the shape of the tree.

3.1.1 Representing a Game with an AND-OR Tree

In an AND-OR tree, each vertex represents a possible state of the game. [15] OR vertices represent steps (possibly) taken by the *attacker* – the player to make the first turn. Conversely, AND vertices represent the *defender*'s possible steps. The root vertex is always of the OR type and represents a given *initial state*. All other vertices represent states obtainable from their parent's state by performing a valid movement of the pieces as described by the Arimaa rules.

Each node of the tree can be evaluated to *true*, *false* or *undecided*. A leaf node with value of true represents a forced win and is said to be *proved*. Similarly, leaf nodes evaluated to false represent a forced loss and are *disproved*. Whether a position is a forced win or forced loss is again determined by the Arimaa game rules. If the state associated with a vertex is neither a forced win or a forced loss, the vertex is undecided. To obtain a proof or disproof of an undecided vertex, its successors have to be generated and then proved or disproved.

An interior OR node is proved if at least one of its children is proved, and it is disproved if all its children are disproved; to prove an interior AND node, it requires that all of its children be proved, and it takes one disproved child for the AND node to be disproved as well. This corresponds to the fact that for an attacker's position (OR-type vertex), it suffices to find a single move that guarantees a win for the attacker, while for defender's positions (AND-type vertices), *all* of the defender's possible replies have to be analysed and determined to guarantee a win for the attacker.

The goal of the Proof-Number Search algorithm is to prove the root vertex of the tree, which is always of the OR type, and thus discover a winning strategy for the attacker: If the root is proved, it is necessarily followed by another proved vertex as its successor thanks to the way the tree is built. The attacking player then can perform the step corresponding to the proved successor. This successor either directly forces a win for the attacker or, if it is an OR vertex, describes another step the attacker can choose in order to win. Proved AND vertices that do not directly correspond to a forced win for the attacker are followed only by proved vertices, which describe the possible ways for the attacker to force a win after each of the defender's possible replies.

A major disadvantage of the original Proof-Number Search algorithm is that all nodes have to be kept in memory, necessitating a termination of the algorithm once no more memory is available for its execution. [8] We have experimented with two approaches to mitigate the memory requirements of PNS: A depth first variant described in section 3.2, and an ad-hoc way of discarding parts of the tree described in section 4.4.

3.1.2 Proof and Disproof Numbers

Two numbers are associated with each vertex in the tree: *proof number* (PN), representing the minimum number of child nodes that have to be proved in order to prove the node, and *disproof number* (DN), analogously representing the minimum number of vertices to be disproved in order to disprove this node. [8] Undecided leaf nodes receive initial PN and DN values both equal to 1 in our current implementation. Decided nodes (leaf or interior) require no further expansion, proved nodes are therefore assigned proof number 0 and disproof number ∞, disproved nodes receive disproof number of 0 and proof number of ∞.

If n is an interior vertex of type OR, its PN and DN values are given by the following formulae:

$$pn_n := \min \{pn_c \ : \ c \in \text{children}(n)\} \tag{3.1}$$

$$dn_n := \sum_{c \in \text{children}(n)} dn_c, \tag{3.2}$$

in correspondence to the minimal work necessary to prove or disprove such a vertex. If n is of type AND, the equations are analogous:

$$pn_n := \sum_{c \in \text{children}(n)} pn_c \tag{3.3}$$

$$dn_n := \min \{dn_c \ : \ c \in \text{children}(c)\} . \tag{3.4}$$

Figure 3.1 shows an example part of an AND-OR tree with proof and disproof numbers.

3.1.3 Expansion of the Tree

In each iteration of the algorithm, a *most proving node* is selected by a recursive process: In an interior OR node the child with minimal PN value is selected, and in an interior AND node the child with minimal DN value is selected. This process starts at the root of the tree, and continues until a leaf vertex is finally selected. [8] This leaf node is then *expanded* by generating its descendants and evaluating them. Allis called this method *immediate evaluation*, and also presented an alternative one called *delayed evaluation*, where vertices are evaluated only when selected for expansion. [1] In this work, we chose to implement only the immediate evaluation method. After a node has been expanded, the tree is traversed from this node back to root, updating the proof and disproof numbers for each vertex on the path according to the formulae shown in the previous section.

Algorithm 1 shows a high-level overview of PNS.

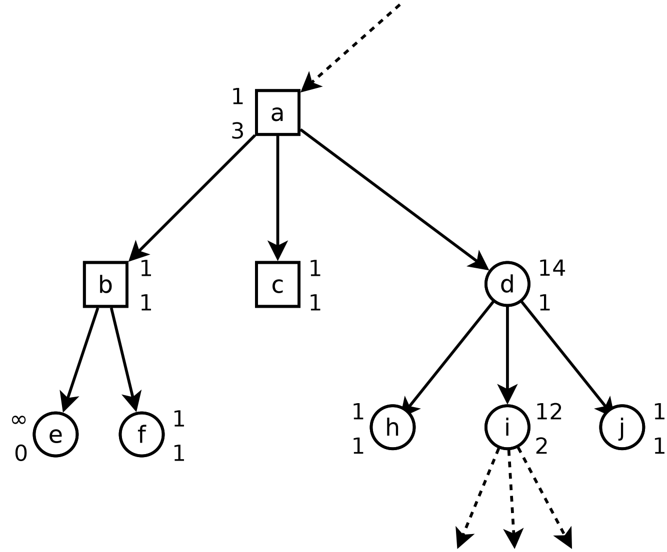


Figure 3.1: Possible part of a search tree. Squares represent OR nodes, circles AND nodes. The two numbers next to a vertex are its proof and disproof numbers, PN the top one, DN the bottom one.

Algorithm 1 PNS in Pseudocode

while $root.pn > 0 \wedge root.dn > 0$ **do**
    $v \leftarrow root$
    while v is not a leaf **do**
        $v \leftarrow$ `choose_successor`(v)
    end while
    `expand`(v)
    while v is not `null` **do**
        `update`(v)
        $v \leftarrow v.parent$
    end while
end while

3.2 Depth-First PNS

Nagai ([11, 12]) introduced a variant of PNS called Df-PNS (Depth-First PNS). In addition to proof and disproof numbers, Df-PNS maintains two other values for each vertex: *proof number threshold* (pnt) and *disproof number threshold*(dnt). The algorithm performs iterative deepening in each interior node, excluding the root. The subtree rooted in a node is stopped to be searched once its proof number exceeds the pnt threshold, or its disproof number exceeds the dnt threshold. [8] Once the numbers of a vertex exceed its threshold, the search of the subtree rooted therein is stopped, and the subtree can be removed from memory.

Root vertex is assigned the thresholds $pnt = dnt = \infty$. For all other interior nodes, the thresholds are assigned as follows (assuming p is an OR node): Choose node n, a successor of p, with the lowest proof number value. If there exists another node, s, that is a successor of p and its proof number value is the second-

lowest of all of p's successors, then let the thresholds for n be given as

$$pnt_n := \min\{pnt_p, \lceil pn_s(1+\epsilon)\rceil\} \tag{3.5}$$

$$dnt_n := dnt_p - dn_p + dn_n, \tag{3.6}$$

[8, 13] where $\epsilon > 0$ is some real constant. If n is the only successor of p, we let $s = n$, and assign the thresholds as given by the formulae above.

We are using the so-called $1 + \epsilon$ *Trick* developed by Pawlewicz and Lew to reduce the amount of multiple subtree re-creations. [13] In our implementation, we settled on the value $\epsilon = 0.9$. (See chapter 6.)

An overview of the depth-first variant is shown in algorithm 2.

Algorithm 2 Df-PNS in Pseudocode

```
current ← root
while root.pn > 0 ∧ root.dn > 0 do
    while current is not a leaf do
        n ← best_successor(current)
        s ← second_best_successor(current)
        assign_thresholds(n, s, current)
        current ← n
    end while
    expand(current)
    v ← current
    while v is not null do
        update(v)
        v ← v.parent
    end while
    while current is not root ∧ current exceeds thresholds do
        current ← current.parent
        Potentially remove children of current from memory
    end while
end while
```

3.3 Adaptation of PNS for the game of Arimaa

3.3.1 Steps versus Moves

As Arimaa allows a player to use one to four steps in their turn, there are two possible choices for representing its state space:

1. The direct approach: Each vertex represents a player's turn. This would lead to a rather classical AND-OR tree with each OR vertex succeeded exclusively by AND vertices and vice-versa. Such a representation would make it straightforward to attach only unique positions during vertex expansion; however, the move generator itself would be non-trivial, possibly creating a separate step-based tree. Furthermore, this approach would result in a tree with large branching factor – one position may lead to thousands of different moves. [14]

2. Step-based approach: Each vertex represents a single step that a player can take. This reduces the branching factor to tens of successors at the expense of making the tree deeper. Step generation is relatively straightforward and does not require an exploration of any separate state space. However, there is now the possibility of creating vertices in the tree that ultimately represent the same move. As an upside, not all moves have to be fully generated using this method, allowing the algorithm to explore only promising ones.

In this work we chose the second approach because of the lesser branching factor and possibility of not having to generate all applicable moves. Hash tables (see subsection 4.2) are used to reduce the effect of duplicate moves represented by different step sequences.

3.3.2 Double Steps

Another thing to consider was the representation of double steps: pushes and pulls. Once again, two choices are obvious:

1. Each vertex represents exactly one step, whether it be a part of a push/pull or not. Choosing this approach would allow for easy counting of the number of steps taken by a player simply by counting the number of vertices on a path. It would, however, introduce a dependency between certain pairs of vertices: Both steps comprising a push or pull have to be made in the same turn and they have to immediately follow each other. This would pose an additional complication during the expansion of a vertex, as well as complicate move ordering.

2. Vertices represent either single step that is not a part of a push or pull, or they represent a pair of steps comprising push/pull. Counting the number of steps taken (and, by extension, number of steps that a player is still allowed to take) becomes more difficult with this approach: While traversing a subtree, a counter needs to be maintained for keeping track of the number of steps taken so far. An advantage to this approach is that move ordering and hash tables are now enabled to treat pushes and pulls as separate from ordinary steps without having to consider more than one vertex.

We chose the latter approach to better differentiate between single and double steps. Double steps move an opponent's piece, and thus are potentially more offensive than single steps – a fact that can be exploited during step ordering.

3.3.3 Lambda Vertices

Thanks to the game rules allowing players to give up some of their four available steps, directly representing merely the steps taken would mean that each player's turn could start at an arbitrary depth in the tree, and opponent's replies to one-, two-, or three-step turns could be analysed before analysing the replies to full four-step turns.

Since it is often advantageous to use as many steps as possible, we wanted to delay the exploration of replies to moves not using all four available steps. For this reason, we introduced *lambda vertices* into the tree: vertices representing

that a step is left unused in a player's turn. These are always taken to use up one of the player's available steps in their turn. During vertex expansion, one lambda vertex is always generated among the successors.

3.3.4 New Vertices

The type of a generated successor of a vertex is decided as follows: If, after playing the step represented by the successor (or giving up one step, in the case of a lambda successor), the player is still allowed to make at least one more step within their turn, the successor is assigned its parent's type. Otherwise, it is assigned the opposite type.

Within one move it is possible to move a piece in one step, and then move it to its original position in a later step. This two-step combination leads to no change on the board other than the player giving up two of their available steps. To reduce the number of vertices generated, we check for such repetitions within a move during step generation. If a generated step repeats a position already encountered within the same move, it is not added to the list of successors.

4. Enhancements of the Algorithm

To keep the search effective, we implemented several heuristics. Firstly in section 4.1 we introduce a well-known hashing method for game positions. In section 4.2 we describe methods to re-use information from previously-explored parts of the tree while processing yet-unexplored parts. In section 4.3 we show two methods to improve the selection of a good successor when there is no single one with the lowest proof or disproof number. Finally, in section 4.4, a method to reduce the memory requirements of the algorithm is described.

4.1 Zobrist Hashing

During the course of the algorithm, it is often necessary to determine whether two vertices correspond to the same position. Another common need is to remember information about explored subtrees, and later retrieve this information given the root of a subtree. Both of these needs may be addressed by a *hashing function* that assigns an integer to each position; this integral value then can be used in comparisons and for indexing purposes.

An efficient method for position hashing was introduced by Zobrist. [20] During initialization, for each combination of colour, piece type and position a random number is generated and stored in an array `pieces[type][colour][position]`. Furthermore, six more random numbers are generated: `gold`, `silver`, and an array of four elements called `remaining[n]`. The first two values are used to encode the player on turn; the array `remaining` encodes how many steps the player is still allowed to make within their turn.

Hash value for a position from which the given player is about to start their turn is then computed as shown in algorithm 3; the symbol $\oplus$ stands for the *bitwise XOR operation*.

Algorithm 3 Generating an initial Zobrist hash

```
function ZOBRIST-INITIAL(board, player_to_move)
    H ← 0
    for all (type, colour, position) ∈ board do
        H ← H ⊕ pieces[type][colour][position]
    end for
    H ← H ⊕ player_to_move ⊕ remaining[4]
    return H
end function
```

Whenever a step is made, the hash value can easily be updated without having to scan the whole board again. Assuming that a piece of a certain `type` and `colour` has been been moved from `old_position` into `new_position` by `current_player` and the player to move after this step is `next_player`, the new Zobrist hash value can be calculated from the old value H using algorithm 4. In the algorithm, an *elementary step* is either a displacement of a single piece or its

capture; a step is composed of one or more elementary steps. (For instance, the step *rg3w rf3x Dg4s* consists of the elementary steps *rg3w*, *rf3x*, and *Dg4s*. [1])

Algorithm 4 Update of the Zobrist hash

```
function ZOBRIST-UPDATE(H, step, current_player, next_player, remaining_steps)
    H ← H ⊕ remaining[remaining_steps]
    for all elementary step ∈ step do
        if elementary step is a displacement then
            H ← H ⊕ pieces[type][colour][old_position]
            H ← H ⊕ pieces[type][colour][new_position]
            remaining_steps ← remaining_steps - 1
        else                                    ▷ elementary step is a capture
            H ← H ⊕ pieces[type][colour][captured_from]
        end if
    end for
    H ← H ⊕ current_player ⊕ next_player
    if current_player = next_player then
        H ← H ⊕ remaining[remaining_steps]
    else
        H ← H ⊕ remaining[4]
    end if
    return H
end function
```

When two different board configurations receive the same Zobrist hash, a *type-1* error occurs. If the Zobrist hash consists of N bits and there are M distinct positions, the probability of at least one type-1 error can be approximated as follows: [6]

$$P(\text{at least one error}) \approx 1 - \exp\left(-\frac{M^2}{2 \cdot 2^N}\right). \tag{4.1}$$

We use 64-bit Zobrist hashes in our implementation. If five million distinct positions are generated, the probability of a type-1 error according to formula 4.1 is approximately 6.776×10^{-7}.

Since Zobrist hash values are used throughout the program to efficiently decide whether two positions are the same, and to index transposition tables, a type-1 error could potentially result in an invalid solution of a subtree or even of the root. However, since the probability of such collisions is low, we decided to ignore that possibility.

4.2 Transposition and Proof Tables

During the expansion of a tree, a vertex may be created that ultimately represents a position that has already been encountered and perhaps even explored to a certain depth in the tree. To avoid re-expansions of such vertices, transposition

[1] See `http://arimaa.com/arimaa/` for an explanation of the move notation used here.

tables are used in Chess programs, [10] and this technique may be easily adapted for the game of Arimaa.

4.2.1 Transposition Table

Transposition table is a hash table of a fixed size that stores information about positions encountered earlier in the search. [10] As a key into the transposition table we use $Z \bmod N$, where Z is the Zobrist hash of a position, and N is a user-settable value which determines the size of the table.

With each key, the following information is stored:

- The proof and disproof number values of a vertex corresponding to the position
- The Zobrist hash Z used for detecting key collisions.
- Size of the subtree rooted in the corresponding vertex at the time the record was inserted into the transposition table. This value is used to implement a *replacement scheme* (see below).

To fill the table with values, whenever a vertex's PN or DN value is updated, its new proof and disproof number values are inserted into the transposition table with the corresponding key. Whenever a new vertex is created, the transposition table is consulted whether it contains a record with the new vertex's hash. If so, the new vertex receives PN and DN values equal to those found in the transposition table. This way, information gathered from positions encountered earlier in the search can be reused when the same positions are encountered again via different path.

Since the size of the transposition table is fixed, generally to a value much less than the number of all possible hash values, collisions in the key (called *type-2* errors by Zobrist [20]) are to be expected. A *replacement scheme* is used to determine when a new entry is to overwrite an older one. Because a retrieval of a "good" value from the transposition table can prevent the expansion of a potentially large subtree (and thus improve the overall performance of the algorithm), it is advantageous to prefer "better" entries while deciding which values to keep. Lazar [10] has identified a few replacement schemes usable with transposition tables:

1. Prefer *deeper* keys. That is, prefer the record that has been searched to a greater depth.
2. Prefer *newer* keys. This is a very simple replacement scheme where an entry always overwrites any older one. This may be further combined with ageing, where each entry is also associated with a *time stamp* identifying the number of the iteration during which the entry was added. During collision, the entry is then replaced only if it is "old" – that is, if it was inserted during iteration number preceding $c - A$, with c being the current iteration number, A some constant controlling the ageing scheme.

3. Prefer *older* keys. Complimentary to the previous scheme: always preserve the pre-existing record in the transposition table, if any.

4. Prefer *greater subtrees*. Similarly to the “prefer deeper” scheme, except using the real count of vertices in the explored subtree, instead of just the depth.

5. Two-level replacement scheme. Each entry in the transposition table contains two records, called first and second. Upon collision, if the new entry has been searched to a greater depth than the first record, the second record is replaced by the current first, and the new record becomes a new first. Otherwise, the new entry is inserted as the second record. (In this case, the deeper scheme is used for the first record; other schemes can be used, too.)

We have experimented with the “newer” scheme combined with ageing, “deeper” and “greater subtrees”. We finally settled for the “greater subtrees” scheme as it proved to give the best results in our implementation.

Usage of transposition tables introduces the *graph-history interaction problem* (GHI). [4] Consider, for instance, a proved OR-type vertex v, that was expanded to contain a proved vertex u as one of its successors, which constitutes v's proof. v is then stored into the transposition table with values $pn = 0$, $dn = \infty$. Suppose that at some later time, a vertex v' is created, representing the same position as v, but assume that the path from root to v' already contains the position of u twice.

If the algorithm were to naively use the stored values, it would potentially return an incorrect result: The step leading from v' into the position associated with u may not be made thanks to the third-time repetition rule. Unless another step can be made from v' that guarantees a win for the attacker, v' may not be marked as proved.

For this reason, we do not store proved or disproved vertices' values in the transposition table. Retrieved values may thus be imprecise, but at least they will not lead to incorrect results.

4.2.2 Proof Table

Our not storing of proved and disproved vertices' values in the transposition table means that some nodes have to be re-explored even though they could be easily proved by a simple hash table lookup if their values were stored. To address this problem, we implemented a second hash table called *proof table*. It is used in a fashion similar to that of the transposition table, with the notable difference that *only* the values of proved or disproved vertices are stored.

Proof table is again indexed by $Z \bmod M$, with Z being the Zobrist hash of a position, M a user-settable value which determines the size of the proof table. The proof table may be of a different size than the transposition table.

The following information is stored for each key:

- The Zobrist hash Z
- The player for whom the corresponding position is a forced win

- History of the source position at the time of insertion into the table. This is a dynamically sized array of the Zobrist hashes of the positions on the path from root to the corresponding vertex. Since the repetition rules only apply to positions resulting from complete moves, only hashes of those positions which represent the start of a player's turn need to be stored.

The proof table uses the same replacement scheme as the transposition table.

Whenever a vertex is marked as proved, its values are inserted into the proof table. Upon creation of a node, the proof table is also consulted for a proof/disproof value. If a value is successfully retrieved, the stored history is compared with the history of the new vertex – if no third-time repetition is detected, the entry is used; otherwise it is ignored.

However, it should be noted that we only check the paths leading to the vertices, and not the subtrees rooted in them. As such, it is still possible to wrongly mark a node as proved when it should be disproved or vice-versa. For instance, consider a situation where a step made from one position leads to a third-time repetition, and all other steps from the same position become disproved – this vertex is then inserted into the proof table as disproved. Later in search, the same position is encountered again via different path, the stored value is used and the position is again marked as disproved – even though it is possible that the step that led to a third-time repetition from the original node is now good to use and potentially even leads to a proof of the node.

We decided to ignore the problem scenario described in the previous paragraph for performance reasons and because we never observed any invalid results as a result of this effect during our testing. The described problem may be avoided in our implementation at the cost of performance by disabling the proof table for a selected run of the program.

4.3 Step Ordering

During the successor selection process, it can happen that there are multiple successors with the same minimal PN or DN value, and no single best one. In these cases it is still desirable to choose a successor that might easily prove or disprove a subtree. To determine which successor to choose in such cases, we introduced two heuristics into our implementation.

Once the successors of a vertex are generated, they are sorted according to the heuristics. Later in search, whenever the choice of a successor is not unique, the first successor with a minimal PN or DN value is selected.

4.3.1 Static Step Ordering

The first step-ordering heuristics uses game-dependent knowledge. Since our focus is on endgames, we implemented a heuristic that prefers advancements of rabbits, and captures and dislocations of opponent's strong pieces.

Each type is assigned a cost value as shown in table 4.1. The cost of rabbits depends on their distance to the goal rank after the step has been taken. If the rabbit is in the player's first half of the board (for gold player that means ranks 1

to 4, for silver ranks 8 to 5), the rabbit's cost is 1. Otherwise, it is $2 \cdot (\text{rank} - 1)$ for gold rabbit, and $2 \cdot (8 - \text{rank})$ for silver rabbit.

During the generation of possible steps from a position, each step is assigned a score. Initially, this score is 0. For each piece that is captured as a result of this step, $4 \cdot \text{cost(piece)}$ is added to the score if the captured piece belongs to the opponent, and subtracted if it is the moving player's piece. For push and pull steps, $2 \cdot \text{cost(pushed or pulled piece)}$ is added. Finally, if the step moves the current player's rabbit, the rabbit's cost is added; if it moves an opponent's rabbit, the cost of the rabbit is subtracted. This prefers advancements of one's own rabbits, and penalizes advancements of opponent's rabbits.

Type	Cost
Elephant	10
Camel	7
Horse	5
Dog	3
Cat	2
Rabbit	varies

Table 4.1: Heuristic costs of Arimaa pieces

4.3.2 Killer-move Heuristics

The killer-move heuristics is a domain-independent technique that has been used for dynamic move ordering in various games. [6] It is based on the assumption that if a step has proved effective earlier in the search, there is a chance it will be effective in other positions as well.

The killer-move heuristics maintains a list of at most k steps for each level of the tree. If a vertex causes a cutoff[2] in the tree, its step is inserted into the list of killer steps for its parent's level.

At some later time in the search we use this information to order new vertices: If the step of a vertex is listed as a killer for its parent's level, it is moved to the front of the successors list. If this step is examined first and also causes a cutoff, the other successors will not even have to be analysed.

Since the number of vertices on the path from the vertex to a node does not necessarily have to correspond to the number of steps taken on that path, the level of a vertex v is computed as $4 \cdot m + 4 - r$, where m is the number of completed moves on the path from root to v, r is the number of steps the respective player is still allowed to make from v. This way each vertex is assigned a computed level as if the tree contained two separate vertices for each push or pull step.

4.4 Garbage Collector

Allis [2] described two ways to reduce the memory requirements of the algorithm.

[2]A *cutoff* occurs when a single vertex proves or disproves its parent. After a cutoff has occurred, no further successors of that same parent have to be explored.

Delete Solved Subtrees This way is based on the observation that the subtrees rooted in proved or disproved vertices are no longer needed, and may be discarded.

Delete Least Proving Called a "last resort" technique by Allis, this method removes subtrees which are least likely to be needed. The technique originally presented by Allis sets both the proof and disproof numbers to ∞ for non-proved vertices whose children have been removed. When the PN and DN values of root both equal ∞, the search is terminated. This is so that subtrees are not regenerated once removed.

Our *garbage collector* method is based on these two techniques. In our implementation, however, we decided not to change the proof and disproof numbers of non-proved vertices while removing their successors, allowing repeated re-creation of subtrees.

The garbage collector is controlled by two parameters, called `gc_low` and `gc_high`. Once the number of vertices in the tree exceeds `gc_high`, the GC algorithm (see algorithm 5) is run on the root vertex of the whole tree until the size of the tree drops below `gc_low` or no further nodes can be removed.

Algorithm 5 The garbage collection algorithm

```
function GC(v)
    for all p ∈ {c ∈ children(v) : pn_c = 0 ∨ dn_c = 0} do
        remove_children_of(p)
    end for
    while v is not a leaf ∧ tree size > gc_low do
        w ← worst non-leaf successor of v
        if w ≠ null then
            GC(v)
        else
            remove_children_of(v)
        end if
    end while
end function
```

As we stated in section 3.2, for the depth-first variant of the algorithm when the numbers of a vertex exceed its thresholds, the subtree rooted therein may be removed from memory. However, when GC is enabled we do not remove those subtrees immediately but rather let the garbage collector do it at some later time.

5. Implementation

During the implementation of our program, we came across a few interesting design decisions to make. In this chapter, we describe these decisions.

5.1 Program Structure

Our implementation consists of two parts: A C++ module which implements the search algorithm and the data structures necessary for its operation, and a user interface implemented in the Python programming language.

We wanted to provide a graphical interface that could visually represent the created tree. However, we believed that implementing the whole program in the C++ language could make the graphical user interface code unnecessarily complicated. This is why we chose Python as the tool to create the user interface part.

Therefore, we chose to use C++ only for the algorithm module because of greater efficiency of native code, and the possibility to manage program's memory in a more fine-grained manner. On the other hand, user interfaces are not generally computationally intensive and therefore do not necessitate the raw power of native code. The Python platform ships with various useful libraries that facilitate the creation of a user interface.

Another option would be to implement several independent programs: One to analyse the positions, and one or more others to visualise the analysed output. That way, no inter-language interface would be necessary, however at least two implementations of some of the data structures (such as the search tree itself) would have to be maintained – each for one used language. Furthermore, unless some way of inter-process communication were implemented, there would be no easy way for the graphical interface to display continuous information about the running computation.

5.2 Board Representation

One of the basic data structures to implement is one that can efficiently hold the state of the playing board and answer queries about it.

At the most basic level, such a structure could merely provide the *put piece*, *get piece*, and *remove piece* operations. However, to efficiently implement the step generator and goal-checker, it is advantageous to provide further operations, such as *get all rabbits* or *get all opponent's pieces.*

The direct option would be to represent a board by an 8×8 matrix of integers, each integer encoding the type and colour of the piece on a given position, or a special value indicating that the position is vacant. While this representation is straightforward and allows for a fast implementation of the *get piece*, *put piece* and *remove piece* operations, operations such as *get all rabbits* require iterating over the whole structure.

Another option is using *bitboards*. [5] For each type we maintain a 64-bit integer that encodes on which positions this type appears: If the n-th bit is set,

a piece of the respective type occupies the rank $\lfloor \frac{n}{8} \rfloor + 1$ and file 'a' $+ (n \bmod 8)$. Similarly, we maintain a 64-bit integer for the gold player indicating the positions occupied by gold pieces, and an integer encoding the positions occupied by any piece at all.

While the bitboard representation makes the *get piece*, *put piece*, *remove piece* operations more complicated, it can be efficiently used to retrieve lists of pieces of given type or colour. These lists may be combined using bitwise operations to produce other lists of pieces. (For instance, a list of all gold rabbits can be computed as `all_rabbits & gold_pieces`, where & represents the *bitwise and* operation.) Since efficient computation of such lists is more important during goal-checking and step-generation, we implemented the bitboard approach.

6. Experiments

To test the behaviour of our implementation, we used the collection of "Arimaa 'Win in 2' Puzzles" available from [3]. It is a collection of positions that occurred in real games in the Arimaa gameroom. These positions can be won by the attacker in two steps.

However, early into our testing we discovered that many of these positions were too difficult for our implementation to solve within the 60-second limit we wanted to use for the parameter tuning. In order to gather more data from successfully solved positions we randomly picked 100 positions containing 15 pieces or less from the puzzles collection and ran our parameter tuning on these.

The set of test positions is included on the attached CD.

6.1 Results

Our first goal was to test the various settable parameters of our implementation and determine which ones give the most promising results. For each test we ran our program on all 100 reduced endgame positions for a maximum of 60 seconds. We observed the number of successfully solved positions and the average time to solve them. Since for each set of parameters, some configurations were able to solve less positions than others, we measured the average time over those positions successfully solved by all runs within a parameter set.

All tests were carried on machines each equipped with eight-core Intel ® Core ™ i7 CPUs clocked at 2.67 GHz. Each machine had 6 GB of physical memory installed, of which at least 3 GB was available when our tests started.

ϵ Value for Df-PNS First we wanted to find an optimal value of ϵ used in the Df-PNS algorithm. The Df-PNS algorithm was run on all 100 positions for a maximum of 60 seconds, it was given proof and transposition tables of 256 MB. Garbage collector was enabled with settings `gc_high` of 5 million, `gc_low` 3 million. The values of ϵ ranged from 0.1 to 1.1 with 0.2 stepping.

The plot 6.1 shows the average time necessary to solve each position that could be solved, table 6.2 shows how many positions each configuration could solve.

The most promising values appear to be 0.5, 0.7, and 0.9. While $\epsilon = 0.7$ is the fastest, curiously it solved the fewest positions. We decided to continue our testing with the value $\epsilon = 0.9$ as that appears to be a reasonable compromise between speed and proving capability.

Transposition Table Size To test the effects of the transposition table, we ran both the PNS and Df-PNS algorithms with transposition table size varying from 32 MB to 512 MB. Proof table size was fixed at 32 MB, garbage collector again ranged from 5 million to 3 million. The depth-first variant used $\epsilon = 0.9$ as determined by the previous test.

The plot 6.3 shows the average times needed to solve positions by each algorithm. Table 6.4 contains the number of solved positions by each configuration.

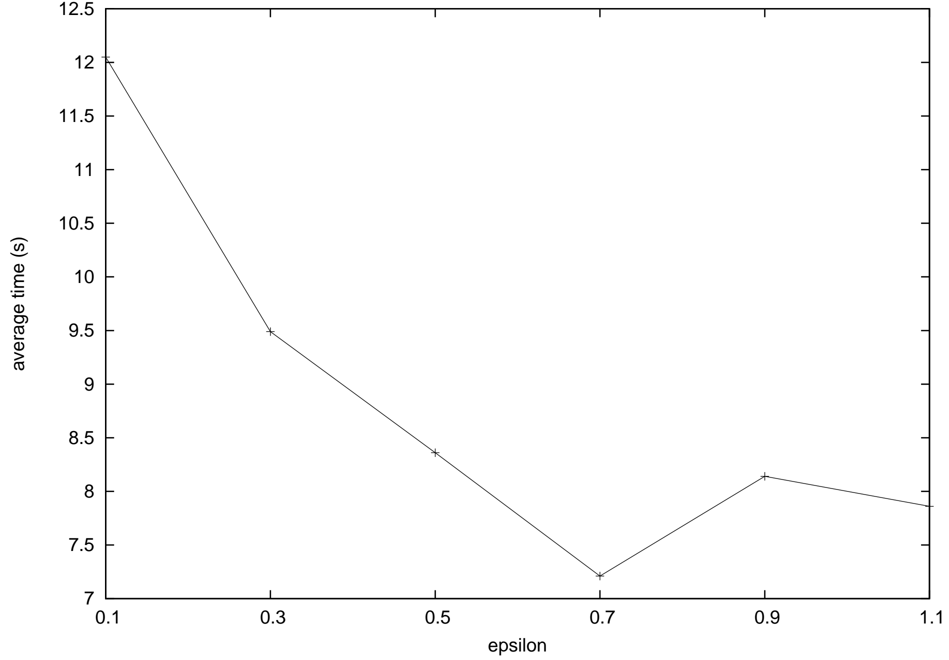


Figure 6.1: Df-PNS Searching Time Depending on ϵ

ϵ	Solved Positions
0.1	16
0.3	18
0.5	16
0.7	14
0.9	17
1.1	15

Figure 6.2: Number of Solved Positions Depending on ϵ

Based on these results, we have settled for the size of 256 MB for further testing.

Proof Table Size Next we wanted to see how the algorithms perform with various proof table sizes. We used the same configuration as for our previous test, except that the transposition table size was fixed at 256 MB, and the size of the proof table varied from 32 MB to 512 MB.

The running times are shown in plot 6.5. In all configurations, the PNS algorithm was able to solve 18 positions. For the configuration with proof table size of 64 MB, Df-PNS could solve 15 positions; in all other configurations, Df-PNS could solve 16 positions.

Garbage Collector Next we measured the effect of the garbage collector on the efficiency. Again we ran our program on all 100 positions and observed the number of positions our implementation could successfully solve within 60 seconds and how fast it could find the solutions. We ran the PNS algorithm with transposition table size of 256 MB, proof table size of 64 MB; the Df-PNS algorithm was run with both tables sized at 256 MB. Furthermore, for the test

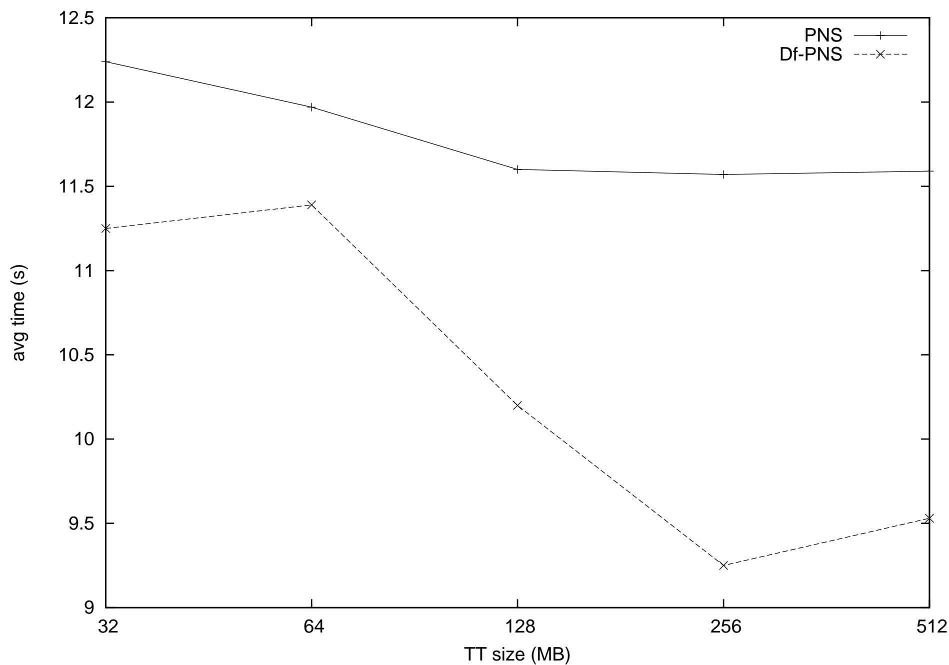


Figure 6.3: Effects of the Transposition Table

TT Size (MB)	Solved Positions	
	PNS	Df-PNS
32	17	14
64	18	16
128	18	16
256	18	16
512	18	15

Figure 6.4: Positions Solved for Each TT Size

with garbage collector disabled, the PNS variant was limited to use at most 1 GB of memory for its search tree. Table 6.6 contains the results.

It appears that higher GC thresholds do not yield a better performance. As an explanation, we propose that some amount of subtree re-creation is actually beneficial: During such re-creations, vertices are initialized with values stored in the hash tables – these values could come from searches performed to a depth greater than the depth of the original subtree. Thus, searching a tree with GC enabled allows for a greater information sharing between subtrees, and to an extent behaves like a search through a *graph*.

Long Runs Finally we ran the program with a 500-second time limit using GC setting of `gc_high` = 2 million, `gc_low` = 1.6 million. The PNS algorithm used 64-megabyte proof table, while the depth-first variant had 256 MB of memory allocated for the proof table; both variants used transposition table sized at 256 MB. Table 6.7 summarizes the results on the reduced positions. We also ran the implementation on a collection of 100 randomly chosen positions from the endgames database, this time without any limitation on the piece count of the positions ("Big Positions"). The results from that run are in table 6.8.

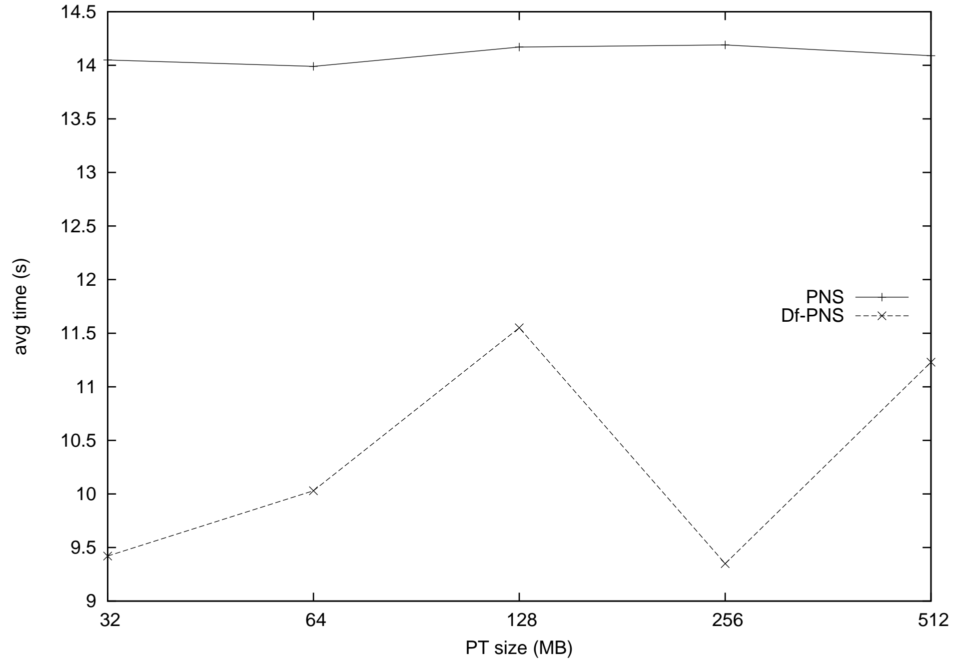


Figure 6.5: Effects of the Proof Table

`gc_high` (million)	`gc_low` (million)	Solved positions		Average time (s)	
		PNS	Df-PNS	PNS	Df-PNS
disabled	*disabled*	14	15	8.03	4.31
1	0.2	18	17	5.63	4.15
1	0.5	19	18	5.17	4.19
1	0.8	18	18	6.22	4.18
2	0.4	18	17	6.48	5.02
2	1	17	16	6.45	4.69
2	1.6	20	17	7.59	4.26
5	1	17	15	5.88	4.69
5	2.5	16	15	5.76	5.02
5	4	17	15	6.61	4.90
10	2	15	16	7.54	4.67
10	5	15	15	7.89	5.73
10	8	15	15	5.94	4.96

Figure 6.6: Effects of GC

Algorithm	Solved Positions	Average Time (s)
PNS	36	105.35
Df-PNS	18	21.81

Figure 6.7: Results of a 500-second Run on Positions with At Most 15 Pieces

Algorithm	Solved Positions	Average Time (s)
PNS	22	52.24
Df-PNS	11	87.26

Figure 6.8: Results of a 500-second Run on “Big” Positions

7. Conclusion

We have successfully implemented the Proof-Number Search algorithm for the game of Arimaa. We have implemented both the original algorithm as described by Allis, and a depth-first variant of the method.

During our testing on the reduced-size test positions, the depth-first variant consistently proved less positions than the PNS algorithm, although when it did manage to find a proof, it did so faster on average than PNS. Curiously, Df-PNS appeared to be slower than PNS when applied to the set of "big" positions.

Even more surprising is the fact that increasing the garbage collector limits seemed to impact performance in a negative way. As we stated earlier, we think this is due to a positive interaction with the hash tables: During a subtree re-creation, more accurate PN and DN values might be retrieved from the hash tables than those originally associated with the original subtree. It might be advantageous to implement a directed acyclic graph version of the PNS method instead of relying on such garbage collector-hash tables interaction.

We have tried to apply the PNS method to Arimaa without relying on domain-dependent heuristics. While such a "pure" application of PNS is capable of solving some positions, we believe that some domain-dependent heuristics might improve the performance of the method significantly.

PNS uses the number of opponent's possible replies to a player's turn to explore those parts of the tree which appear to require the least amount of effort to be proved or disproved. However, it appears that in Arimaa the number of possible replies may not be the most suitable criterion. For example, a strong attacking move may unfreeze one or more opponent's pieces and thus leave the opponent with more possible replies. The PNS algorithm could then choose to first explore a weaker move that limits the opponent's mobility more than the hypothetical strong move.

7.1 Future Work

Our presented implementation can still be improved in many ways, including:

- Building a directed acyclic graph instead of a tree.
- Including a way to statically check whether a player can reach the goal from a position, rather than having to search through a subtree until reaching a goal position.
- Our way of solving the graph-history interaction problem is rather crude. A better way using Zobrist hashes of paths could be implemented. We would also like to include a check for repetitions while using entries from the transposition table (rather than just the proof table), and test whether such checking helps guide the search better.

Further research could focus on the following:

- Heuristic initialization of the PN and DN values of newly-created vertices. Allis [1] proposed to use the minimal number of *node evaluations* necessary to prove or disprove a vertex rather than the number of successors.

Winands [16] proposed a way to use heuristic evaluation functions from alpha-beta-based solvers.

- Not considering all possible replies and instead focusing only on the first few "best" ones. Yoshizoe [19] introduced a way called Dynamic Widening to do so without sacrificing the completeness of the algorithm.

- Wu [18] elaborated on building a move-ordering heuristic for Arimaa. Including such a heuristic could be especially advantageous in combination with Dynamic Widening or similar techniques.

- Application of the PNS method for achieving tactical goals during mid-game rather than just forcing a win.

Bibliography

[1] L.Victor Allis: *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, Rijksuniversiteit Limburg, Maastrich, The Netherlands, 1994.

[2] L.Victor Allis, Maarten van der Meulen, H. Jaap van den Herik: *Proof-number search.* Artificial Intelligence, Volume 66, Issue 1, March 1994, Pages 91-124, ISSN 0004-3702, DOI: 10.1016/0004-3702(94)90004-3.

[3] Arimaa Homepage. `http://arimaa.com/`

[4] Murray Campbell: *The Graph-History Interaction: On Ignoring Position History.* Proceedings of the 1985 ACM Annual Conference on The Range of Computing, ACM NewYork, NY, USA, 1985, pp. 278-280, ISBN: 0-89791-170-9

[5] Stefano Carlini: *Arimaa: From Rules to Bitboard Analysys.* Knowledge Representation Thesis, University of Modena and Reggio Emilia, 2008

[6] Christ-Jan Cox: *Analysis and Implementation of the Game Arimaa.* Master's thesis, Universiteit Maastrich, Institute for Knowledge and Agent Technology, 2006

[7] David Footland: *Building a World-Champion Arimaa Program.* Proceedings of the 4th International Conference on Computers and Games (CG 04), Springer-Verlag, Berlin, Heidelberg, 2006 LNCS, pp. 175-186. ISBN: 978-3-540-32488-1.

[8] H. Jaap van den Herik, Mark Winands: *Proof-Number Search and Its Variants.* Oppositional Concepts in Computational Intelligence, Studies in Computational Intelligence, 2008, Vol. 155/2008, Springer, pp. 91-118. DOI: 10.1007/978-3-540-70829-2_6

[9] Tomáš Kozelek: *MCTS Methods in the Game of Arimaa.* Master's Thesis, Charles University in Prague, 2009.

[10] Sashi Lazar: *Analysis of Transposition Tables and Replacement Schemes.* University of Maryland Baltimore County, 1995

[11] Ayumi Nagai, Hiroshi Imai: *Application of df-pn+ to Othello endgames.* Proceedings of Game Programming Workshop in Japan 1999, Hakone, Japan, 1999, pp. 16-23

[12] Ayumi Nagai: *Df-pn Algorithm for Searching AND/OR Trees and its Applications.* PhD thesis, The University of Tokyo, Tokyo, Japan, 2002

[13] Jakub Pawlewicz, Łukasz Lew: *Improving depth-first pn-search:* $1 + \epsilon$ *trick.* Proceedings of the 5th International Conference on Computers and Games (CG 06), Springer-Verlag, Berlin, Heidelberg, 2007 LNCS, pp. 160-171. ISBN: 978-3-540-75537-1.

[14] Omar Syed: *Arimaa: A New Game Designed to be Difficult for Computers.* Journal of the International Computer Games Association 2003, Vol. 26, No. 2, pp. 138-139

[15] Stuart Russell, Peter Norvig: *Artificial Intelligence: A Modern Approach.* 3rd ed. Upper Saddle River: Prentice Hall, 2009. ISBN 978-0-13-604259-4.

[16] Mark H.M. Winands, Maarten P.D. Shadd: *Evaluation-function Based Proof-Number Search.* Proceedings of the 7th International Conference on Computers and Games (CG 10), Springer-Verlag, Berlin, Heidelberg, 2010. pp. 23-35, ISBN: 978-3-642-17927-3

[17] Mark Winands, Yngvi Björnsson, Jahn-Takeshi Saito: *Monte-Carlo Tree Search Solver.* Proceedings of the 6th International Conference on Computers and Games (CG 08), Springer-Verlag, Berlin, Heidelberg, 2008, LNCS, pp. 25-36. ISBN: 978-3-540-87607-6

[18] David Jian Wu: *Move Ranking and Evaluation in the Game of Arimaa.* Thesis on Harvard College, Cambridge, Massachusetts, USA, May 2011.

[19] Kazuki Yoshizoe: *A New Proof-Number Calculation Technique for Proof-Number Search.* Proceedings of the 6th International Conference on Computers and Games (CG 08), Springer-Verlag, Berlin, Heidelberg, 2008. LNCS, pp. 135-145. ISBN: 978-3-540-87607-6

[20] Albert L. Zobrist: *A New Hashing Method with Applications for Game Playing.* Technical Report #88, Computer Science Department, The University of Wisconsin, Madison, WI, USA. Reprinted (1990) in Journal of the International Computer Games Association, 1970, Vol. 13, No. 2, pp. 69-73

Glossary

Branching Factor The average number of successors in a game tree; or the exact number of successors of a given vertex.

Df-PNS Depth-First Proof-Number Search

DN Disproof number. Associated with each vertex in the tree; it is the lower bound of the number of vertices that have to be disproved in order to disprove the vertex this number is associated with.

Game Tree Tree whose vertices represent positions of a game, and edges represent valid moves that lead from one position to another.

GC Garbage Collector or Garbage Collection. Technique in which subtrees not likely to be selected for expansion are removed from memory.

GHI Graph-History Interaction. A situation where naively reusing cached information about a position encountered earlier in the search can lead to imprecise or incorrect results.

Minimax Tree Game tree where a value is associated with each of its vertices. For leaves this value is given by an evaluation function and corresponds to the outcome of the game for a given player; for interior nodes, this value is computed either as the minimum or maximum (depending on the player to move from the respective position) of the values of the children of the node.

MCTS Monte-Carlo Tree Search

PN Proof number. Associated with each vertex in the tree; it is the lower bound of the number of vertices that have to be proved in order to prove the vertex this number is associated with.

PNS Proof-Number Search

Position In the context of *Arimaa game position*, this refers to the configuration of pieces on the playing board and the player to move.

PT Proof Table. Hash table that stores the winning player for positions proved or disproved at some time during the search.

TT Transposition Table. Hash table that stores the PN and DN values of positions explored at some time during the search.

A. User's Manual

Our implementation can be downloaded from `https://github.com/Oxyd/APNS/` in its source code form; precompiled binaries for Windows are available in the Downloads section on the same address. Both the source code and Windows binaries can also be found on the attached CD.

A.1 Supported Systems

Our program has been tested to build and work on GNU/Linux and Windows operating systems. Precompiled binaries are only made for the Windows platform; GNU/Linux users have to build the program from source (see section A.3).

On both platforms, an implementation of the Python 2 programming language[1] is required to run the program. It has been tested to work with versions 2.6 and 2.7. Furthermore, the Tkinter library is necessary for the GUI version (although not required if one wishes to run the batch version only.)

Both 32-bit and 64-bit systems are supported. Windows users who want to use the precompiled binary version need to download the appropriate distribution archive.

A.2 Running the Program

The program can be run in two modes: Graphical (GUI) mode, and batch (command line) mode. The former is recommended for users who wish to experiment with the program.

Having extracted the binary distribution archive or having built the program from source, the GUI version can be launched by double-clicking the `gui.pyw` file or executing `./gui.pyw` from a shell in the program's directory. The program's main window should appear on screen thereafter.

The batch version can be run by using the command `python batch.py` or just `./batch.py` in the program's top-level directory. When run without any arguments, the program will print a summary of the options available. The program can be run as `./batch.py --help` to print a more detailed description of the available arguments.

A.2.1 Graphical Interface

The GUI of the program is divided into three panes: The topmost one contains a toolbar with buttons used to execute the program's functions; the left pane displays the built search tree (once one has been created); the right pane then displays the position corresponding to the vertex selected in the tree pane.

Initially, only two buttons are enabled: *New Initial Position* which allows the user to load or create a position that is to be analysed, and *Load Search* that can be used to load a previously saved search tree.

[1] Visit `http://python.org/`.

Clicking the *New Initial Position* button will bring up a dialog window with a simple Arimaa position editor that can be used to specify one manually. One may also use the *Load* button to use a pre-existing position saved in a file.

Once a starting position is specified, the algorithm can be launched by using the *Run Search* button. It will display a dialog window that allows various parameters of the algorithm to be set, and that allows to start the search. If the search is started, a dialog window with information about its progress and an option to interrupt the algorithm will be shown.

Once the algorithm has stopped (whether by proving or disproving the root vertex or otherwise), the left pane of the main window will be updated to contain the current tree created so far. Each vertex is displayed on its own line, which contains the following information:

Step The step that leads from the vertex's parent to the given one, or the word *lambda* for lambda vertices.

Type The word *OR* or *AND* denoting the type of the vertex.

PN, DN Proof and disproof number values.

*** or **** The vertex that is currently considered to be the best one among its siblings is marked with an asterisk. Two asterisks mark those vertices that lie on the path from the root to the most-proving node.

If the root vertex is neither proved nor disproved, the *Run Search* button can be used again to continue running the search, possibly with different parameters. The generated tree can be saved to a file by using the *Save Search* button. If one wishes to start a search of the same initial position again from the beginning, the *Reset Search* button can be used to discard everything but the initial position and search preferences. Finally, when a vertex of the tree is selected, the position it represents can be saved to a file using the *Save Position* button.

A.2.2 Batch Interface

The batch interface offers the same options of running the program as the GUI version. It can be used to run the program to see how fast it can solve a position. The tree created from a run of the batch version can also be saved to a file and at some later time loaded and examined in the graphical interface.

Most important options controlling the program's behaviour are the following:

`-p` ***position*** Specifies a file containing the position to be analysed.

`-s` ***search file*** Specifies a file containing the tree created during a previous search. If specified, the search will start with the given tree and continue its analysis. This option is mutually exclusive with the `-p` option.

`-d` ***tree file*** Gives the file name where the tree is to be saved once the algorithm stops.

`-a` ***algorithm*** Can be either `pns` or `dfpns`. Specifies which algorithm to use. Default value is `pns`.

`-t` ***limit*** Limits the search to the given number of seconds.

`-r` ***size*** Specifies the size, in megabytes, of the transposition table.

`-o` ***size*** Similar to `-r`, controls the size of the proof table.

`-k` ***count*** Specifies the number of killer steps to be remembered for each level.

`-G` ***gc high*** Sets the `gc_high` threshold.

`-g` ***gc low*** Sets the `gc_low` threshold.

`-Q` When given, the program will not print any information about its progress, only a summary at the end.

`-q` With this option, the program will not print anything except possible errors.

`-w` If specified and if the root is proved when the program terminates, the best move will be printed at the end of the program's execution.

For example, the program can be run with the following command to analyse a position stored in the file `pos.txt` for a maximum of 60 seconds, using 128 MB proof and transposition tables, `gc_high` of 2 million and `gc_low` of 1 million:

```
./batch.py -t 60 -r 128 -o 128 -G 2000000 -g 1000000 -p pos.txt
```

A.3 Building the Program

To build the program the following prerequisites have to be installed:

GCC Compiler Collection Version 4.6 or later.

Boost Library Collection Version 1.48 or later, including development libraries.

Python Version 2.6 or 2.7, also including development libraries.

SConstruct

If all prerequisites are met, the program can be built using the command `scons`. If some of the libraries are in an unusual location or the user wishes to use some special build configuration, the file `config/linux.py` will have to be modified accordingly. Further information about building the program can be found on the attached CD in the Programmer's Manual.

Attachments

The attached CD contains the following:

`thesis.pdf` This text in PDF format.

`programmer-manual.pdf` Programmer's manual for the program.

`APNS` The program in its *source code* form.

`apns-windows-32bit-v1.5.4-bc.zip` Precompiled binary distribution for 32-bit Windows systems.

`apns-windows-64bit-v1.5.4-bc.zip` Precompiled binary distribution for 64-bit Windows systems.

`python-2.7.3.msi` Python installer for 32-bit Windows systems.

`python-2.7.3.amd64.msi` Python installer for 64-bit Windows systems.

`reduced-results` Files generated during the testing on reduced positions.

`big-results` Files generated during the testing on "big" positions.

Both the source code and binary distributions contain a subdirectory named `example-positions` containing the positions used in our testing.