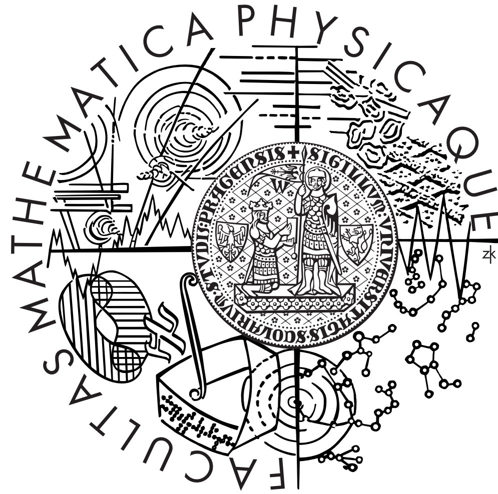
Charles University in Prague
Faculty of Mathematics and Physics



Master's thesis

Tomáš Kozelek

Methods of MCTS and the game Arimaa

Department of Theoretical Computer Science
and Mathematical Logic

Supervisor: RNDr. Jan Hric

Study program: Theoretical Computer Science

2009

On this place I would like to thank to the supervisor of my thesis for all the time spent at consultations, for relevant comments and for many corrections he did in the course of the work on the program and in the thesis itself. Also I would like to thank to all of my friends who gave me inspiring ideas and to my family for their support.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 5. srpna 2009 Tomáš Kozelek

Contents

1 Introduction **5**

1.1 Thesis Preview 5
1.2 Arimaa - The Game of Real Intelligence 5
1.3 The Rules of Arimaa 6
1.4 MCTS motivation 8
1.5 Research guideline 8
1.5.1 Objectives 8
1.5.2 Research Questions 9
1.5.3 Hypotheses 9

2 AI in Arimaa **10**

2.1 Why is Arimaa difficult for computers 10
2.2 Algorithms 11
2.3 Peculiarities of Arimaa 11
2.4 Existing programs 12
2.5 Limitations 13

3 MCTS **14**

3.1 Origin 14
3.2 Overview 15
3.3 Enhancements 17
3.3.1 The notion of learning in UCT 18
3.3.2 Core algorithm improvements 18
3.3.3 Domain knowledge application 19
3.3.4 Transient learning 20
3.3.5 Parallelization 22
3.3.6 Optimization 23
3.4 Performance 23
3.5 MCTS in Arimaa 24

4 Akimot Approach **26**

4.1 Overview 26
4.2 Board representation 26
4.3 UCT tree 28
4.3.1 Steps vs. Moves 28

4.3.2 Easy way effect 29
4.3.3 UCT entities 30
4.3.4 Transpositions 32
4.4 Playouts 34
4.4.1 Playouts organization 34
4.4.2 Step generation and selection 34
4.5 Evaluation 35
4.5.1 Evaluation Scheme 35
4.5.2 Evaluation Elements 37
4.6 Domain Knowledge in Steps 38
4.7 Information sharing 38
4.7.1 History Heuristic 38
4.7.2 UCT-RAVE 39
4.7.3 Move Advisor 39
4.8 Speedup 41
4.8.1 Parallelization 41
4.8.2 Optimization 42

5 Performance and Experiments 44
5.1 Methodology 44
5.2 Experiments 45

6 Conclusion 50
6.1 Achievements 50
6.2 Research Guideline Revisited 51
6.2.1 Objectives 51
6.2.2 Research Questions 51
6.2.3 Hypotheses 52
6.3 Future Work 52

A User manual 56
A.1 About 56
A.2 Background 56
A.3 Installation 57
A.4 Options and Configuration 58
A.5 Session 59
A.5.1 Position formats 61
A.6 Arimaa Engine Interface 62
A.7 Arimaa Test Suite 64
A.8 Gameroom 66
A.9 Match environment 67
A.10 Simple Arimaa Development GUI 68

B Glossary 69

Title : MCTS methods in the game of Arimaa
Author: Tomáš Kozelek
Department: Department of theoretical informatics and mathematical logic
Supervisor: RNDr. Jan Hric
Supervisor's e-mail address: jan.hric@mff.cuni.cz

Abstract:
Game of Arimaa is an artificially created strategic board game with the purpose to be difficult for computers. A vast majority of introduced computer engines for Arimaa are based on successful approaches from chess, namely the minimax algorithm with $\alpha\beta$ pruning and further extensions. In this thesis we have analyzed the applicability of the so called MCTS methods in the game of Arimaa. MCTS methods are a state-of-the-art approach to the computer Go with bright prospects in other strategic games as well. We have implemented a MCTS based Arimaa engine called Akimot and adapted the MCTS techniques for the Arimaa environment. We have experimented with various MCTS enhancements known from computer Go and identified which are prospective in our setup. Moreover, we have proposed several new enhancements on ourselves. Performance experiments show that our MCTS approach is comparable to an average $\alpha\beta$ engine.
Keywords: Arimaa, MCTS, Monte Carlo, UCT, Go

Název práce : MCTS techniky ve hře Arimaa
Autor: Tomáš Kozelek
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce: RNDr. Jan Hric
e-mail vedoucího: jan.hric@mff.cuni.cz

Abstrakt:
Arimaa je strategická hra vytvořená za účelem být obzvláště těžká pro počítače. Většina existujících programů hrajících hru Arimaa je založena na ověřených postupech z problematiky počítačových šachů obzvláště pak na $\alpha\beta$ prořezávání s rozšířeními. V této práci jsme se zaměřili na prostudování použitelnosti MCTS technik ve hře Arimaa. MCTS techniky jsou momentálně nejlepší známé algoritmy pro počítačové Go s dobrými vyhlídkami i v dalších strategických hrách. Naprogramovali jsme počítačového hráče založeného na MCTS, kterého jsme pojmenovali Akimot. V naší implementaci jsme přizpůsobili známé MCTS postupy pro prostředí hry Arimaa. Provedli jsme experimenty s různými vylepšeními známými z počítačového Go a určili jsme, které z nich jsou použitelné v naší implementaci. Navíc jsme navrhli a otestovali několik vlastních rozšíření. Experimenty ukázali, že náš MCTS program je srovnatelný s průměrným $\alpha\beta$ programem.
Klíčová slova: Arimaa, MCTS, Monte Carlo, UCT, Go

Chapter 1

Introduction

1.1 Thesis Preview

In this thesis, our goal is to apply MCTS[1] techniques in the game of Arimaa. These algorithms proved to be very successful in computer Go and we would like to check what potential they have in a different field.

Chapter 1 provides an introduction to the game of Arimaa and MCTS techniques in general and presents the research guideline of the thesis.

Chapter 2 outlines why Arimaa is difficult for computers and introduces existing approaches to the game, their (dis)advantages, their limitations and their success.

Chapter 3 lays out basic principles of MCTS methods, mentions some known enhancements that have been proposed and tested mostly in the domain of computer Go and elaborates on their applicability in Arimaa.

Chapter 4 explains how we have built up our MCTS engine for Arimaa and what enhancements we have used.

Chapter 5 shows results of experiments we have performed.

Chapter 6 discusses the achievements and future work.

Appendix A gives the user documentation for the project

Appendix B provides the glossary

1.2 Arimaa - The Game of Real Intelligence

The game of Arimaa was created in 1997 by Omar Syed and his son Aamir (Arimaa = A + reversed(Aamir)). The main impulse for Arimaa creation was the famous Kasparov - Deep blue match (see [1]). This was a triumph for computers in the field of chess

[1] Monte Carlo Tree Search

programs, a huge milestone in the field of Artificial Intelligence. However, many computer scientists believe that brute-force approach combined with very capable hardware is far from what Artificial Intelligence should be about. Omar is one of them and he decided to prove his point by creating a game that might be played with a standard chess set and is easy to learn and play well for humans but which is far more difficult for computers than chess is (see [2]).

To boost the bot development Omar offered a financial prize of 10,000 USD for the first computer program to beat a human champion in an annual match. Even though this challenge has been held for 6 years now, the prize still hasn't been claimed. Time has proven that the game of Arimaa is not only deep, interesting game for humans but also a challenging problem for computers.

1.3 The Rules of Arimaa

Arimaa is a two player zero sum game with perfect information played on the 8x8 board. Players are called *Gold* and *Silver* and each of them possesses 16 pieces in the beginning of the game. These are (ordered from the strongest to the weakest): 1 x elephant (E), 1 x camel (M), 2 x horse (H) , 2 x dog (D), 2 x cat (C), 8 x rabbit (R). One letter shortcut expresses both piece and the color of the player - uppercase for the gold player and lowercase for the silver player.

The game starts by setting up the pieces in the player's two closest rows. The initial set up of the pieces is not prescribed. See Figure 1.1 for one possible initial setup. The goal of the game is to transport one of the 8 weakest pieces (rabbits) to the most distant row. The most elementary movement is called *the step*. All pieces are allowed to make the same steps: 1 square to the left, right, front or backwards. Only rabbits are not allowed to step backwards.

A player's *turn* (also called a move) in the game of Arimaa consists of up to 4 steps. These might be distributed among different pieces. There are three kinds of steps:

single step Move one piece to a neighboring square.

push Push a weaker piece out of the way and move on its place. The player who is performing a push selects (an empty) intersection to which is the opponent's piece pushed.

pull Move to a neighboring square and pull a weaker piece along.

Pushes and pulls count as 2 steps (both the opponent's and the friendly piece are moved). Pushing and pulling simultaneously is forbidden. When a piece is adjacent to a stronger opponent piece then it is *frozen* and it cannot move, unless it is also adjacent to a friendly piece (it is *supported*).

There are 4 special squares on the board. These are called *traps* and are positioned at `c3, c6, f3, f6`. If there is a piece on the trap square and it has no supporter (adjacent friendly piece), then this piece is *trapped* and it is removed from the board.

Figure 1.1: One possible initial setup.

Figure 1.2: Example position.

Notation for writing down the moves in Arimaa is as follows. All the steps in a move are written in a row separated by whitespace. Every step is written in the form (one letter for piece and player)(row coordinate in alphabet)(column coordinate as a number)(letter "x" if the piece is trapped, or a first letter of step direction - North, East, South, West). For example, move `Rf2n Rf3x ee2e rh8s` can be interpreted as follows: the silver elephant at e2 pushes the rabbit at f2 to the north where the rabbit dies in the trap, afterwards the silver rabbit moves from h8 to the south. First the result of the push is written down (the elephant cannot actually go to the rabbit's location because it is not empty). `Rf2n` says rabbit (it is a gold rabbit because R is written in uppercase) from f2 moves north. `Rf3x` means the gold rabbit at f3 dies. `ee2e` expresses that the silver elephant from e2 moves to the east (that caused the push). The move ends with the silver rabbit moving from h8 to the south (`rh8s`). The silver player can still make one single step (the total count of his steps so far is 3), but he used his right to *pass* and let the opponent play.

In the example position displayed in Figure 1.2 the following holds:

- Possible moves for Gold include: `Mg1n Mg2n df4s df3x Hg4w`, `Db1n Db2s Rc2w`, `Ec4e rc5s rc4s rc3x Ed4w`.
- Pieces `rc5, df4` are frozen.
- Rabbits cannot move backwards, therefore `Rh3s` is an illegal step, but rabbits can be pushed backwards by opponent's pieces – for instance `rc5n Ec4n` is a legal step.
- Pieces `rc5, df4, cb6` might be trapped when it is *Gold's* turn to move. For instance the move `rc5e Ec4n cb6w cc6x Hb5n` traps the cat `cc6`.
- Piece `Rf2` might be trapped when it is Silver's turn to move.

Instead of using all 4 steps in a move, player might play *pass* and discard the rest of his move. Position repetitions in Arimaa are handled by the following rules:

- pass: Position after the move must be different from the position before the move. Therefore playing pass as a first move counts as a loss.
- 3 times repetition: Repeating position for a third time means a loss for the player who performed the move.

There are three ways how to win the game:

- goal: If one of the 8 rabbits is on the last row at the end of the move, then its owner wins the game.
- elimination: If all of a player's rabbits are eliminated then this player loses the game.
- immobilization: When a player has no legal move to play, he loses the game.

Detailed rules description can be found in [3].

1.4 MCTS motivation

Computer Go has always been considered one of the grand challenges to the Artificial Intelligence. While computers have been conquering the world of chess, top Go playing programs have been struggling around the level of lower intermediate human player. Detailed elaboration on why Go is much more difficult for computers is given in [4].

In 2006 new and interesting approaches to the domain of computer Go emerged and situation has evolved notably since then. These algorithms are referred to as MCTS (Monte Carlo Tree Search) and are based mainly on statistical sampling of the position and selective iterative search. Since 2006 there has been a boom of MCTS based Go programs pushing the bar of programs' strength higher and higher resulting into some surprising achievements against human players. In principle MCTS methods are applicable to a wide variety of problems, however, so far relatively little effort has been put into trying out these algorithms outside of their flagship – the computer Go. For consistency we mention the successful applications of MCTS methods in the following games: Lines Of Action, Amazons, Hex.

1.5 Research guideline

Guideline according to [5].

1.5.1 Objectives

- To propose and implement MCTS integration in a bot playing the game of Arimaa.
- To identify ways of improving MCTS algorithms which work in Arimaa.
- To check whether MCTS might be successful in a game dissimilar to the game of Go.

1.5.2 Research Questions

- How the Monte Carlo playouts must be rebuilt to be applicable in the game of Arimaa?
- Which of the proposed improvements to the MCTS algorithms are domain independent?
- Is there a potential for MCTS algorithms in Arimaa to start the kind of revolution they did in computer Go?

1.5.3 Hypotheses

- Monte Carlo playouts used "as is" from the game of Go will provide a weak Arimaa player.
- Standard improvements from computer Go will improve the MCTS Arimaa player as well.
- MCTS Arimaa player might be competitive to the existing $\alpha\beta$ searchers.

Chapter 2

AI in Arimaa

2.1 Why is Arimaa difficult for computers

There are several reasons why Arimaa is difficult for computers:

Huge branching factor

Arimaa was intentionally created as a game with an enormous branching factor. While in chess the number of moves in an average position is around 35, in Go it is around 200, in Arimaa this number rockets to 17,000 unique moves on average and in complicated middle game situations up to 50,000 unique moves (see [6]). One might argue that such a huge branching factor is a very limiting issue for humans as well. However, humans approach to the game is not as sensitive to the branching factor as the brute force approach, which proved successful in chess.

Stability

Tactical combinations are much less common than in chess. There are mainly two reasons for this: the fact that a move consists of 4 steps and the absence of “long shooting” pieces. The game of Arimaa might be played in such a way that the winner is decided based on the ability of longterm planning rather than snatching a tactical combination. This longterm planning is a huge weapon in the hands of the humans who can beat computers with systematic cumulation of little strategic advantages.

Position evaluation

Static evaluation of position in Arimaa consists of many aspects and a good evaluation function is not as straightforward as it is in chess. Not only material balance but also trap control, local piece domination, mobility, rabbits’ advances, hostage situations, etc. must be taken into account. Arimaa tactics and strategy are closely related to building a good evaluation function and are described in [7]. Precise implementation of evaluation function is often the result of a long hand-tuning process. Attempts to use automatic parameters tuning techniques to increase the accuracy of the evaluation function unfortunately have failed so far (see [8]). On the other hand, use of the static evaluation function is still way more successfull in Arimaa than in Go.

Opening

Initial positioning of pieces is not prescribed which virtually makes opening books obsolete. It has not been agreed on yet which opening setup is the best (or if there exists such a concept at all).

No endgame

While in chess or Go the position usually gets simpler when approaching the end of the game, this might well not be the case in Arimaa where most of the games are decided in the late middle game.

2.2 Algorithms

Variety of approaches have already been tried in Arimaa. Sadly from the AI point of view, those which succeeded are basically "good old" brute force algorithms from chess adapted to a new environment. Specifically this is a well-known model of full-width tree search done by the Minimax algorithm with $\alpha\beta$ pruning and static evaluation in the leaves. Naturally, many enhancements (also known from the domain of chess programs) have been implemented on various levels of abstraction in this model. For instance:

- *Transposition tables* with *Zobrist hashing* for avoiding re-evaluation of previously encountered positions.
- *Killer moves* and *history heuristic* for effective information sharing in the tree.
- Search extensions for extending the search into "silent positions".

2.3 Peculiarities of Arimaa

Compared to chess there are several (sometimes intentional) further peculiarities that must be dealt with in the computer approach to Arimaa.

Multiple steps in a move

First programs playing Arimaa have taken the so called *move-based-approach* - i.e. generating all possible unique moves from the position and evaluating them. As it was mentioned, due to the large branching factor this option has proved to be infeasible. Therefore, the *step-based-approach* has been widely adopted. In this scenario, programs iteratively generate all legal steps from the position. In combination with $\alpha\beta$ pruning this approach is much more efficient than the one mentioned previously.

Frequent repetitions

Different moves (step sequences) might lead to the same positions. Actually, the ratio $\frac{\text{unique positions}}{\text{all positions}}$ after a single move is relatively small. This issue is traditionally handled by Transposition tables. However, in the *step-based-approach* it is desirable to refine repetitions handling in a more effective manner – many step

sequences can be pruned quite early. A nice solution to this called "step combo" was provided in [8].

Reversible moves

From the nature of the game the following scenario is possible: in position P1, player 1 makes a move A resulting in position P2, player 2 makes a sequence of steps (less than 4) resulting in position P1 (he has reversed the position) and then he can make some extra step(s). The effect is the same as if player 1 discarded his move (A is called a *reversible move*) and player 2 played 1 or more steps for free. Clearly this often leads to an advantage for player 2. Existing programs (being deterministic searchers) often do have issues with this phenomena and may stubbornly play reversible moves several times in a row while the other player is cumulating advantage.

2.4 Existing programs

A spot where virtually all Arimaa games are played is the online gameroom[1]. Both bots and humans compete in here. A variant of ELO rating system is used for player's evaluation. There are currently a few programs with rating over 1800. All of these are principally very similar (all based on $\alpha\beta$ model) and any of them might be considered as a reasonably strong player. Moreover, in blitz games some of these have rating around 2000. This corresponds to a strong human player. For comparison, strongest human players are rated around 2450.

Top programs include:

- Bomb by David Fotland
- Clueless by Jeff Bacher
- Gnobot by Toby Hudson
- OpFor by Brian Haskin

So far only two programs have earned the right to represent computers in the annual Human - Computer Arimaa challenge. These are bot Bomb (computer champion in 2004-2008 see [9]) and bot Clueless (computer champion in 2009). Hardware for Arimaa challenge is selected by Omar Syed, inventor of the game, and usually it is a piece of hardware you can buy for around 1,000 $. Accumulated score for 6 years of the challenge is 47:5 for humans including games with handicap (also not always top humans played against challenging bot). This clearly shows current human dominance in the game.

[1] http://arimaa.com/arimaa/gameroom

2.5 Limitations

While top Arimaa programs have already achieved relatively high level of play, still top human players are able to defeat them even with big handicaps (e.g. giving the handicap of a dog, a horse, or even a camel). The reason for this is that human players have learned how to steer the game into a rather peaceful flow avoiding complicated tactical fights and slowly cumulating little advantages. Unless computers make a significant progress in longer planning issues (or are given much stronger hardware), their computing capabilities will be defeated by human reasoning abilities and Arimaa branching factor.

Chapter 3

MCTS

3.1 Origin

One of the greatest obstacles for building a decent computer program playing the game of Go has always been an evaluation function. An interesting alternative to this was proposed in 1993 in [10]. The described method (called *Monte Carlo*) works as follows:

1. Play a pseudo-random game (also called a *playout* – see Appendix B) from the given position – there is only one restriction on move selection in playouts: filling one's own eyes[1] is prohibited (otherwise the game could be infinite).
2. In the end of the game (no move is possible) compute the result (per player: number of stones played plus surrounded intersections).
3. Save the result of the game and repeat.
4. Use the mean from accumulated simulations as a position evaluation.

This way, an estimate on winning probability for both players is obtained. Surprisingly, this estimate proved to bear some promising accuracy potential, especially in positions where strategy thinking was of higher importance than tactical thinking (i.e. opening). In the same paper, the method was naturally extended into a complete Go playing engine:

1. Keep performing Monte Carlo playouts from starting position till time is up. For every simulation update the result bound to the first move in the simulation.
2. In the end, return the move with the best estimate on winning probability.

Tests on 9x9 board performed in [10] showed that such a Go playing engine was better than a human novice (around 25kyu) even with a very few simulations per move. An important fact to note is that this approach uses virtually no domain knowledge.

[1] see Appendix B

3.2 Overview

Monte Carlo approach to computer Go was revisited around 2005. While Monte Carlo playouts proved to be an interesting idea on evaluation of Go position, a program based solely on this principle was giving quite poor results and didn't scale well (after around 5000 simulations, the search reached a plateau and didn't progress any further). An elegant idea was proposed: moves that have better results from simulations than others should be allocated more "attention" (more simulations) than others. This approach can be applied iteratively, resulting in the following algorithm outline:

1. Start from the root node (representing the current position).
2. Traverse the tree in the best-first manner (according to a specific *exploration formula*) down to a leaf.
3. If in a leaf node and few simulations have gone through this node then *expand* the node.
4. Perform the Monte Carlo simulation from the leaf node and backpropagate the result of the simulation.

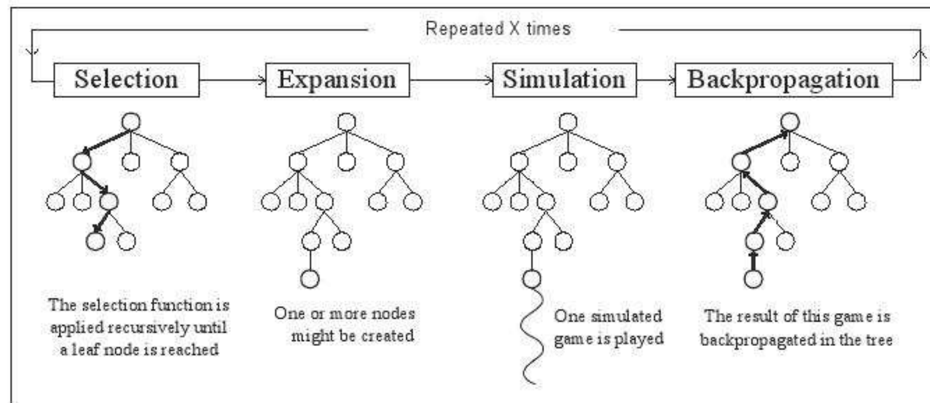


Figure 3.1: MCTS scheme.

This way, an asymmetric tree (in the sense that some branches are deeper and more explored than others) is built up and kept in the memory. A scheme of this algorithm frame is depicted in Figure 3.1 taken from [11].

The core of the presented pseudo-algorithm lies in the *exploration formula*. The initial phase of research in the field of MCTS in computer Go was boosted by the utilization of results from the mathematical games theory, particularly *multi-armed bandit problem*. The motivation behind this is that the node N in the search tree together with its children N_i where $i = 1, \dots, n$ represents a singe multi-armed bandit B. The children of the node N are the arms of the bandit B. This way the whole search tree is made up of independent bandits.

A deterministic algorithm *UCB1* for playing a multi-armed bandit was presented in [12]. Its principle is the following:

1. Play each arm once.
2. Play the arm maximizing the formula $\overline{X}_i + \sqrt{\frac{2 \ log \ n}{n_i}}$
   where $\overline{X}_i$ is an average value of the arm i.
   n is the total number of games with the bandit
   n_i is the number of games when the i-th arm was played

This strategy is believed to balance the exploration and exploitation of bandit's arms well. Moreover, UCB1 is proved to provide a logarithmic *regret* (regret is an expected loss after n plays caused by the fact that the algorithm does not play the optimal arm consistently). This is a very desirable property considering that previously proposed strategies usually operated with a linear regret.

In [13] Kocsis et al. have refined the multi-armed bandit problem applied to the minimax-like trees. They have shown that under certain assumptions (that values of nodes in the tree are independent) applying *UCB1* formula to tree-like organized multi-armed bandits provides an asymptotically optimal exploration-exploitation strategy (it converges to minimax algorithm). They named the algorithm *UCT*[2].

However, theoretical fundings for the application of the theory developed by Kocsis et al. to the game of Go might be inaccurate. The arms of the bandits (the moves) in Go are NOT independent (as demanded by Kocsis). There are dependencies both in the single node (for instance the locality factor in the given position) and also across multiple nodes (the same move might be good in different positions). However, the first experimental results in Go given by UCT combined with Monte Carlo playouts (see [14]) have proved to be excellent and laid out a foundation for further intensive research in the area.

UCT is quite different from standard tree search methods (i.e. $\alpha\beta$). In comparison to these, both pros and cons can be observed.

Pros:

- It is an anytime algorithm.

  This is a very desirable property especially in playing a board game with limited time. $\alpha\beta$ searchers usually perform iterative deepening to achieve similar effect.

- It handles uncertainty in a natural manner.

  The algorithm reacts very smoothly to the fluctuations of estimates of the node values. In the beginning of the node's "life-cycle", exploration is dominant and all its children (arms of the bandit) get played several times. However, with the rising number of node visits it is quite likely that several (sometimes only one)

[2]Upper Confidence bound to Trees

children with superior performance are found and they are granted majority of attention. Given a sufficient amount of time, values estimated by this process should converge to the same results as found by the Minimax algorithm.

- It automatically generates an asymmetric tree.

  Promising lines of play are allocated more attention and thus they are getting explored deeper. On the other hand, subtrees with uninteresting moves in the root remain quite shallow. A lot of time is saved for the exploitation of good moves, on the other hand, good variations starting by unpromising-looking moves might remain unexplored.

Cons:

- It is weak in tactical combinations.

  This is quite a serious disadvantage. $\alpha\beta$ searchers are from the nature of the Minimax algorithm extremely good at revealing tactical shots. On the other hand, UCT tree search might simply "overlook" a deeper tactical combination because it doesn't allocate enough resources to a seemingly unyielding move near the root.

- It is "slower" than $\alpha\beta$.

  In contrast to evaluate-one-by-one-node strategy of $\alpha\beta$ searchers, UCT tree search has to descend deep down to the tree to perform the evaluation. Descending through the tree is a relatively costly operation - exploration formulas for children of all nodes along the way to the leaf must be computed and compared. On the other hand, UCT descent is a much smaller computational burden than Monte Carlo playouts.

- The whole tree must be kept in memory.

  This is somehow similar to the human approach to the board game playing. Yet it imposes further requirements on the computer memory and indirectly also on the processing power.

3.3 Enhancements

While UCT algorithm with Monte Carlo playouts has shown promising performance it is just a starting point for numerous extensions. A collection of these extensions and variants of UCT algorithm is often referred to as MCTS[3] methods. Many of current top Go playing engines are commercial and source-closed. Thus it is perfectly possible that enhancements mentioned below have already been further developed or even replaced by more sophisticated solutions. For instance it is well known that *MoGo*, one of the top Go programs with very stable performance have stopped using UCB-like exploration formulas (formerly believed to be one of the pillars of the MCTS techniques) quite some time ago.

[3] Monte Carlo Tree Search

3.3.1 The notion of learning in UCT

The act of choosing the best move in a given position can be looked at as a process of learning the values of moves and selecting the move with the best value. There are three core elements in this learning process:

Online learning is the MCTS algorithm itself – approximating the distribution of the values of the moves by Monte Carlo simulations in the asymmetrically growing tree.

Offline learning represents inputing the domain knowledge to bias the playouts or node initializations in the tree.

Transient learning [4] corresponds to applying values learned in a particular position on similar positions as well. More on this topic is given in §3.3.4.

3.3.2 Core algorithm improvements

While *UCB1* is proved to have logarithmic regret and is very desirable from this point of view, improved strategies with better actual performance have been introduced. One of these is *UCB-tuned* which works as follows:

1. Play each arm once.
2. Play arm maximizing the formula $\overline{X}_j + \sqrt{\frac{log\ n}{n_i} min\{1/4, V_i(n_i)\}}$
   where $V_i(s) = \left(\frac{1}{s}\sum_{k=1}^{s} X_{j,k}^2\right) - \overline{X}_{j,s} + \sqrt{\frac{2log\ n}{n_i}}$
   $V_i(s)$ is an estimate on upper bound of variance of arm i.

The demand of UCB-like strategies to play each arm once in the beginning is quite limiting. This might be an issue, especially if a good arm is discovered quickly and could be given more attention. Therefore the notion of *FPU* (First Play Urgency) was introduced. FPU is a global constant whose value was empirically set around 1.1. Value of the arm is decided as follows:

$$val_i = \begin{cases} FPU & \text{if } n_i = 0 \\ \overline{X}_i & \text{if } n_i > 0 \end{cases}$$

This way a good arm might be played several times in a row even though other arms haven't been played at all yet. While this technique improved performance of early UCT engines it became rather obsolete with the introduction of more sophisticated methods for initialization of a node's value (e.g. *progressive bias* introduced in [15]).

[4] we refer to this one as information sharing as well

3.3.3 Domain knowledge application

UCT algorithm contains practically no domain knowledge at all (beside the moves generation and “don’t fill the eye” rule in playouts). However, very soon it was discovered that adding the domain knowledge might significantly improve the strength of a MCTS player. Two questions appear:

Where to apply the knowledge?

There are two obvious slots where applying the knowledge could be appropriate. In the tree or in the playouts (these are referred as heavy playouts then). There is a reasonable argumentation for using the knowledge both in the tree and in the playouts. Knowledge in the tree works relatively close to the root and should help to quickly filter out bad moves. On the other hand, if there was very heavy knowledge in the playouts and the playouts were close to the perfect play then it would be enough to perform a single playout to estimate the given position.

An investigation on this issue is provided in [16]. The result of this survey suggests that the highest efficiency was reached with *best-of-N*[5] heuristics in playouts.

The system of simple patterns in the playouts proved to be very successful as well. This improvement is suggested in [17] and it is believed it was one of the reasons of initial *MoGo* domination in the field of computer Go. There are several small patterns (size 3 x 3) where every intersection has one of the values : empty, black stone, white stone, don’t care. A move which fits the pattern is likely to be selected in the playout.

In the case of applying the knowledge in the tree it is desirable to use the offline learned value in the beginning of the node’s existence but when the number of simulations through this node gets higher it is desirable to shift to using the online learned value. Interesting methods on how to apply the knowledge in the tree were proposed in [15]. These are called *progressive bias* and *progressive unpruning* and they provide a smooth transition from the offline learned value of the move to the online estimate given by the UCT algorithm.

What kind of Go knowledge to use?

This is tightly connected to the previous question. In the tree, relatively slow methods for pattern matching and shape recognition might be used without a significant slowdown. On the other hand, current mainstream puts pressure on keeping the Monte Carlo playouts very fast thus the knowledge used in them is relatively simple.

A natural attempt was made in [18] to train a simulation policy by the means of reinforcement learning of linear combinations of binary features. Computer Go player based on this trained policy outperformed by far both random policy and

[5] heuristically best move from N random moves is selected

handcrafted policy in the tournament of standard game of Go. However, results from using the learned policy in playouts were rather poor compared to the handcrafted policy. This revealed a strange paradox when a correlation between the success of policy in playouts and the strength of the policy as a standalone player is rather unclear.

Many researchers found using handcrafted domain knowledge superior to generating the knowledge automatically. On the other hand, a successful method of automatic knowledge extraction in the context of MCTS methods was presented in [19]. A set of fixed binary features is introduced (e.g. is a given move extension, is it an atari[6], is it close to the last move, ...). Every pattern is given an ELO-like rating expressing the chance of a move being played if it satisfies the feature. These values were automatically harvested from the games of strong players. A move can then be thought of as a team of these patterns and its estimated value is a joint value of its patterns. Experiments with using this approach for generating the knowledge in both the tree and the playouts were made and they proved to raise the level of the Go playing engine significantly.

3.3.4 Transient learning

Motivation

Information sharing mechanisms have proved to be significant performance boosters in tree search algorithms. In $\alpha\beta$ approach probably best-known are *killer moves* and *history heuristic* techniques (see [20]). Both of these are used successfully in the state-of-the-art Arimaa engines. On the other hand, transient learning solutions have been proposed and implemented for MCTS methods as well. While in $\alpha\beta$ searchers all sharing happens on the tree level, in MCTS the sharing process might take place on up to 4 different levels - depending on where the information is gained and where it is used. In this manner, the *tree-playout level* means that information is gained in the tree (from the cumulated statistics of the move/position) and is used in the playout (for selecting the next move to be played). So far the transient learning mechanisms in MCTS have proved to be useful on following levels (for every level an example follows):

tree-tree level The *grandfather heuristic* introduced in early MoGo. A node was initialized with the value of its grandfather (following assumption that the move which is good will be on average good even after another move is played).

tree-playout level History heuristic as described in [21]. A table with the number of "UCT wins" (how many times a move was selected in a UCT descent) is kept in first order and second order (reaction to another move) manner. These statistics are then used in the playout for selecting a move (first order) or making a reply to opponent's move (second order). A similar approach to history heuristic is mentioned in [22].

[6] see Appendix B

playout-tree level UCT-RAVE mechanism discussed below.

UCT-RAVE

The drawback of the UCT algorithm is that it must sample every child of a particular node many times before it is able to produce a low-variance estimate on the value of the action leading to that child. This might result in a slow online learning process. An elegant solution to this was proposed in [18] and is called UCT-RAVE[7]. Normally, when UCT performs a simulation from a particular node (here simulation means both the Monte Carlo Simulation and continuing UCT descent in the tree) only the child representing a first move of the simulation is updated. However, already in [10] an *AMAF*[8] heuristic was mentioned. The trick is to consider all subsequent moves in the simulation starting from a certain node as if they were played first. The basis of such a heuristic is that in the game of Go many sequences might be transposed without losing the meaning behind the moves.

UCT-RAVE is an extension of AMAF in the UCT framework. The idea is the following:

- Keep RAVE statistics in every node (*RAVE-value*, *RAVE-visits*) besides standard UCT statistics.
- After every simulation from the node update the RAVE statistics for all children which had their move played in the simulation. RAVE statistics are updated in the very same way as UCT statistics.
- When selecting a move for descent, combine both UCT and RAVE statistics of the move. This is done in such a way, that when a number of visits to the node is small more weight is given to the RAVE estimate and as the number of the visits to the node is increasing the weight is continuously shifting to the UCT estimate.

UCT and RAVE estimates are combined as follows:

$$\begin{aligned} Q_{UCT-RAVE}(s,a) &= \beta(Q_{RAVE}(s,a)) + (1-\beta)(Q_{UCT}(s,a)) \\ \beta &= \sqrt{\frac{k}{3n(s)+k}} \end{aligned}$$

where β is a balancing element with k being an *equivalence parameter* guiding the shift of weight from RAVE estimate to UCT estimate.
Q_{UCT} is the UCT exploration formula as given above. $Q_{RAVE}(s,a) = \overline{X}_{RAVE,a} + c\sqrt{\frac{log\ m(s)}{m(s,a)}}$ is RAVE exploration formula (more than resembling the UCT one)
$\overline{X}_{RAVE,a}$ is a mean RAVE value of action a in the state s
$m(s)$ is number of RAVE visits to the state s
$m(s,a)$ is number of RAVE visits to the state resulting from s by making an action a.

[7] Rapid Action Value Estimation
[8] All Moves As First

RAVE provides a biased estimate of the node's value, however it is more accurate in the beginning of the node's existence than UCT estimate. In its first application in *MoGo* it was documented to significantly improve the performance against the benchmark engine *GnuGo* (66% of wins with RAVE vs. 24% of wins without). Nowadays RAVE is considered a "must-have" extension of a strong MCTS Go engine.

3.3.5 Parallelization

It was independently shown in several sources (for instance [14], [11]) that increasing the number of simulations significantly improves the MCTS player. In [11] an outline was made that doubling the number of simulations adds 50 ELO points of strength to the engine. Such a scalability poses a strong motivation for parallelization of the algorithm. Several models for parallelization have been proposed (see [11] for further descriptions and benchmarks).

Leaf model
: Control thread descends through the UCT tree to the leaf from where work threads perform Monte Carlo simulations. This is a very straightforward and easy to implement model. The drawback of this model is the inefficiency of resources allocation - too much processing power might be concentrated to unpromising moves.

Root model
: Every thread works separately in its own tree. When the time is over all the created trees are merged and results for particular nodes are summed up. This model is also very simple to implement yet the results it yields are satisfactory. Surprisingly when the number of threads is relatively small (<= 16) this model performs better than hypothetical *N times speedup model*. A possible explanation for this phenomenon is that spawning several independent trees and merging them in the end helps the algorithm to escape from a local optimum.

Tree model
: Multiple threads work on the single tree. There are two variants of locking: *global mutexes* and *local mutexes*. The performance of the first one stagnates already with a few threads, as most of the threads are waiting for the lock. In the second variant particular nodes are locked which provides a decent performance.

Simulation Servers model
: This is a variation of the *leaf model* designed for a cluster of machines connected via LAN. A control thread descends the UCT tree to the leaf and launches the work threads. However, opposed to the *leaf model*, the control thread performs a new update and a UCT descent when any of the workers finishes.

3.3.6 Optimization

Many methods for optimizing the speed of simulations or tree navigation have been proposed. These include efficient representation handling (for instance in computer Go the term of *pseudo-liberty* was introduced to quickly handle the capture threats in Monte Carlo playouts), pre-computations of functions (*log*, *sqrt* functions in the UCT tree) or children caching. Children caching is a simple yet effective heuristic used during the UCT descent in the tree. A few best children are "cached" in every node and during the descent the best one is selected from these. After some particular amount of descents goes through the node the cache is "flushed" and new nodes are selected to populate the cache. This simple technique when tuned well, provides a reasonable speedup of descents in the UCT tree without losing the accuracy of node selection. On the contrary this principle must be further tuned for use in a parallelized engine - number of threads should influence amount of descends before the cache flushes (the more threads the shorter the time before the cache flushes).

3.4 Performance

Whereas Go is usually played on the board of the size 19x19, in principle it can be played on the board of an arbitrary size. Especially 9x9 and 13x13 have become very popular in the Go community. Even though playing Go at 9x9 board is believed to be several orders of magnitude simpler than playing Go at 19x19, computer Go programs have never reached reasonable level at 9x9 either. Until MCTS programs appeared. Taking into account computational demands of MCTS, it was only natural that the first MCTS programs have focused on the domain of 9x9 computer Go. Very soon 9x9 MCTS engines surpassed the traditional Go programs. In 2006 at the computer Go Olympiad in Turin, CrazyStone engine started an uninterrupted golden medal streak of MCTS programs at 9x9 Go. Traditional approach to 19x19 was defeated one year later by MoGo taking a gold medal at the Olympiad in Amsterdam.

The strength of MCTS programs is nowadays generally considered superior to the strength of other approaches. Lately, questions have started to arise whether MCTS is a long-searched-for "holy grail" of computer Go. The one that allows computers to match (and overcome) the skills of humans (see [23]). Several prestigious matches have already been organized. The results show that MCTS programs are more or less equal to the very strong players in the 9x9 domain but need at least 7 handicap stones in the 19x19 domain. The list of all matches between humans and computers is available at [24]. Programs playing in these challenges are usually massively parallelized – for instance MoGo traditionally runs on Dutch computer cluster called Huygens (having allocated around 600 - 1000 cores) in serious matches.

Top MCTS programs include:

- CrazyStone by Remi Coulom First MCTS winner of 9x9 computer Go olympiad (2006). One of the strongest engines.
- Leela by Gian-Carlo Pascutto First commercial MCTS program.

- Many Faces of Go by David Fotland Remake of famous and successful traditional go program. Winner of 2008 computer Go Olympiad (in both 9x9 and 19x19 category).
- MoGo by Sylvain Gelly et al. Probably most “famous” MCTS engine. The team behind MoGo is generally considered being responsible for many breakthroughs in the development of MCTS theory.
- SteenVreter by Eric Van den Werf Winner of 2007 9x9 computer Go Olympiad.

3.5 MCTS in Arimaa

While MCTS methods proved to be very successful in the domain of Go their performance in the game of Arimaa is questionable. Here we list potential prospects as well as pitfalls for MCTS algorithms in Arimaa.

Prospects:

- Arimaa is supposed to be more of a strategic game.
  Tactical combinations are supposed to be rather rare. This could be a good news for MCTS searchers which are known to be weak in tactics.
- A promising profit might be gained from transient learning in the UCT tree.
  A concept of transposition is very common in Arimaa. If approached properly a lot of information might be shared across the UCT tree.
- The game is played on a relatively small board.
  If we look at the Arimaa search space from the point of view of the steps then the size of the search space approximately corresponds to the 9x9 Go, where MCTS engines have reached the level of strongest amateur players.

Pitfalls:

- Position is much less stable than in the game of Go.
  One of possible explanations of the success of Monte Carlo playouts in Go is that it is not “easy” to ruin a good position. In other words let there be a position where player A has (sufficient) advantage over player B. Then there is a good chance that the game will end by a win for player A no matter of his strength as long as B is of the same strength. This idea is further researched in [25] where promising results are shown by learning “balanced” playout strategy rather than strong playout strategy.

  There might be an issue with this property in Arimaa. A good position in Arimaa might be characterised for instance by achieving successful elephant blockade even at a cost of material sacrifice. However, it is quite likely that a blockade wouldn’t be preserved by a weak player (an analogy of random playout).

- Position often doesn't get simplified towards the end of the game.
  MCTS Go engines are known for their close-to-perfect play in the endgame. From the very nature of the game of Arimaa, there is no good analogy to the endgame in Go. In equal game the position might be quite complicated when the goal is scored.

- Knowledge encoding might be difficult.
  Computer Go is quite convenient from the point of position encoding (and connected to this the knowledge encoding). Patterns might be stored quite easily and efficiently with bitsets, maintaining a set of potential moves is not an issue as well (it more or less corresponds to the set of empty intersections), etc. This doesn't hold in Arimaa though. For instance checking whether a move is applicable in a particular position is not trivial (trapping and freezing must be taken into account). Step generation must be performed iteratively during the playout – which might lead to very low ratio of playouts per second.

Chapter 4

Akimot Approach

4.1 Overview

This chapter presents our approach to building a MCTS engine for Arimaa. We have decided to describe the concepts and motivation for our solutions and skip the implementation details. Our MCTS implementation addresses (among others) following issues:

Board representation - What representation is suitable for our purpose?

UCT tree - How to organize UCT tree (step based or move based)? How to define basic UCT elements (e.g. UCT descend technique, exploration formula, condition for node expansions, etc.)? How to handle transpositions in the tree?

Playouts - How long should the playout be? How to generate steps/moves in the playout? How to bias playouts?

Evaluation - When to evaluate the position? What should the evaluation consist of? How to transform the evaluation result for later back-propagation?

Domain knowledge in steps - What knowledge to use and where to apply it?

Information sharing - How to share information across the tree to increase performance?

Speedup - What parallelization model is suitable for our setup? What optimization techniques to use?

4.2 Board representation

Good board representation is essential for any board game playing engine. Natural demands on a good board representation are:

- quick step generation

- small size of the board representation - This way, often used paradigm *play-unplay* step might be replaced by much simpler *copy board-play-drop board*.

In Akimot we have experimented with two approaches to the board representation in Arimaa.

integer board

This is a straightforward representation of Arimaa board. The board is maintained as an array of integers where every cell corresponds to a particular coordinate of real board. It is a good practice to use one dimensional array and recalculate the indexes if necessary instead of using a two dimensional array. We have extended this array to contain 100 elements - 64 for board cells and 36 for edges. This way out of board check is quite elegant. Every cell of the array holds a single integer value determining

- whether it is an out of board cell
- whether it is an empty cell
- piece type and color of present piece

While this representation is quite easy to implement, its performance in above criteria is rather inferior.

bitboards

An efficient representation inspired by chess programming. Board is represented as 14 64-bit integers (bitboards) - one bitboard for every (piece type, color) pair plus a bitboard for (all pieces, color) pair. If there is 1 at position i in bitboard for golden cats, then there is a golden cat standing at corresponding coordinates (typically $(i/8, imod8)$) on the actual board. Having an extra bitboard for all pieces of particular color is a nice speedup trick since often it is sufficient to know what color is the cell occupied by or whether it is empty. A nice property of this approach is the size of the board model. Its core takes $14 \times 8B = 112B$ which is quite a difference compared to $100 \times 4B = 400B$ in the case of integer board. Detailed description of this representation is provided in [26].

Initially we have used the integer board approach because of its simplicity. However later we decided to switch to the more efficient bitboards representation giving us more than two times speedup in performing playouts.

Necessary part of an Arimaa engine is management of position signatures and ability to recognize already visited position. This is also a premise for correct implementation of 3-times repetition rule (see §1.3). We followed a standard practice in handling position signatures by using a Zobrist hash keys framework (see [27]).

4.3 UCT tree

4.3.1 Steps vs. Moves

One of key early decisions we encountered was whether the search should be *step based* or *move based* - i.e. whether the elementary unit in the tree represents a step or a move. Both approaches have its specifics. Authors of traditional $\alpha\beta$ programs generally agree that *step based* method is superior. The main reason is finer granularity of the search enabling better cutoffs and more effective iterative deepening search. In the case of UCT, different aspects might play role as well.

move based

+ Is quite straightforward and easy to implement.
+ Transpositions can be handled already during the moves generation.
+ Might be a solution to the *easy way effect* (see §4.3.2).
- The shape of the UCT tree is believed to be wide and shallow with nodes having an enormous number of children (see [6]). This is a serious obstacle because a lot of time is inefficiently spent in UCT descents. The bottleneck lies in UCT formula recalculations for every child of the node.
- Information sharing (see §4.7) is very difficult.

step based

- Implementation brings quite a number of obstacles (it is not *MIN-MAX* tree anymore, rather *MIN-..-MIN-MAX..MAX*, the moves might have different number of steps leading to a very unbalanced tree, the after-search selection of a move to be played is not as straightforward as in the *move based* approach).
- Transpositions must be handled explicitly in the tree, which brings further complications (see §4.3.4).
- UCT tree must be rather deep in certain variations in order for the algorithm to be competitive (good players are known to look ahead at least two moves or up to 16 steps). As known from computer Go to play moves deep in the tree with good confidence is hard to achieve.
- There is an *easy way effect* (see §4.3.2).
+ The shape of the UCT tree is narrow and deeper with average number of node's children in range of 15 - 30. Most of the bad variants should be recognized quite early and given little attention. This naturally is a two edge sword bringing danger that good variants starting with bad looking moves (like tactical combinations) will be overlooked.
+ It is meaningful from the point of the humans' view - they also think rather in terms of steps and their combination to final moves. Algorithm should benefit from information sharing on steps across the tree(see §4.7).

We considered all the mentioned pros and cons. Even though the *step based* approach poses many implementation challenges we believe that it is the right choice to make. In the end the decisive factors for us were the favorable shape of the tree, promising early cutoffs of bad variations and the possibility of benefit from information sharing.

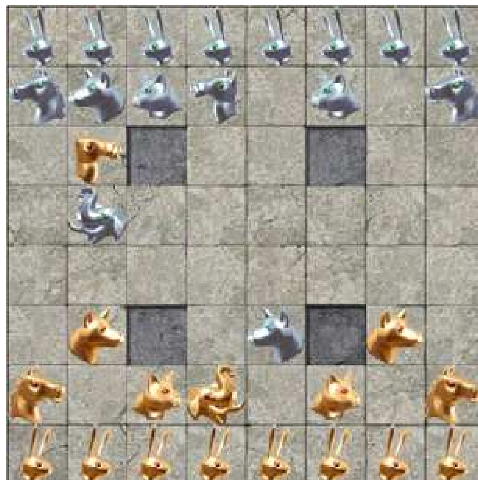


Figure 4.1: Easy way effect

4.3.2 Easy way effect

This phenomena refers to situation when a local (tactical) obstacle might be solved by multiple ways (let's say 2). One of these is easier to spot(typically needs less steps), but leads to inferior solution. Suppose the following example: in Figure 4.1 a gold camel is in danger and appropriate action must be taken (it is gold's move). The gold player should send one of his pieces to guard the `b6` trap. While this trap is deep in the sphere of influence of the silver player the figure dispatched on this mission should be an elephant - therefore a proper way to play is probably `Ed2n Ed3n Ed4n Ed5n` pictured in Figure 4.2. This way an elephant is protecting the dangerous trap for camel while silver dog is still in the gold's sphere of influence. An inferior but simpler-to-spot way to play is depicted in Figure 4.3 and came from a move `Db3e Db3n Db4n Ed2n`. While now silver dog is in immediate danger gold is about to experience serious problems around `c6` trap. We performed many tests with this position and when given shorter time limits the algorithm quite often selects the wrong way to play. The behaviour is similar (yet not so frequent) even with absence of silver dog at `e3`. We believe that the reason for such a failure is the fact that step sequence `Db3e Db3n Db4n` is one step shorter than sequence `Ed2n Ed3n Ed4n Ed5n` and thus easier to spot for UCT. Once a way to save the camel is found it becomes "superior" to other solutions from the point of view of UCT algorithm and leads to reinforcing the wrong move. The fact that horizon proving the move wrong lies deeper in the tree makes the mistake difficult to spot (in this sense is the situation similar to the *horizon effect* as known from chess). The phenomena is the more apparent the "easier" the easy way is.

Figure 4.2: Success

Figure 4.3: Failure

If we would implement the *move based* approach we believe that the fact that step sequences saving the camel are of different length would not play the role anymore and the advantages of positioning a strong elephant to guard the `c6` trap would dominate.

With introduction of further enhancements and under reasonable time conditions the negative impact of this effect becomes more subtle. Naturally this effect is specific to the UCT approach and doesn't happen to $\alpha\beta$ engines.

4.3.3 UCT entities

We have used a variation of standard trick to limit the size of the tree called *maturity threshold*. Original idea is following: a node is expanded if at least maturity threshold simulations have already gone through it. It is a good practice to set maturity threshold approximately to the expected number of children of the node (this would be around 15 in our case). The variation we used is slightly different: we use lower maturity threshold value (around 5), but node n must have at least $maturity\ threshold + depth(n)$ visits to be expanded. The motivation is to have a better dynamics of a tree growth. It's beneficial not to waste time and expand nodes quickly near the root and gradually postpone the expansion with rising distance from the root. In our implementation performance of this solution is superior to that of the traditional one.

The exploration formula in our program is as following:

$$\overline{X}_i + c\sqrt{\frac{log\ n}{n_i}} + \frac{h_i}{n_i} + \frac{hh_i}{\sqrt{n_i}}$$

where $\overline{X}_i + c\sqrt{\frac{log\ n}{n_i}}$ is a standard UCB1 formula,
h_i is a knowledge heuristic value of the step in the node, implementing the progressive bias transition as described in [15],

hh_i is a history heuristic value of the step in the node (more on history heuristic is in §4.7.1).

Even though UCB1 is said to be obsolete, in our approach it has worked better than other formulas we have tried. We have experimented with UCB-tuned for a while. However not only it was more computationally demanding (giving less playouts per second) but also it hasn't shown any promising results in the benchmark - even with an attempt to hand-tune the constants.

The exploration constant c in UCB1 formula is fixed at value 0.2. We made some experiments with dynamically tuning this value to achieve better balance in exploration and exploitation. Namely we inspected two scenarios:

decay with time

In this model we started with higher value in the beginning to support the exploration and then as the number of simulations have increased we would lower the c coefficient to give way to the exploitation. The exact formula was as following :

$$c = max(0.01, min(0.25, \frac{k}{\sqrt{n}}))$$

for constant k and
n number of visits.

variance based

Here we started with the idea from UCB-tuned to incorporate the variance of playouts' results as an indicator of node's stability. The idea is as following: the smaller the variance, the more stable the node and the smaller the exploration constant should be. Formula we used was:

$$c = max(0.01, min(0.25, k\left(\frac{1}{n}\sum_{l=1}^{n} X_l^2\right) - \overline{X}^2))$$

for constant k
n number of visits
$\overline{X}$ average playouts' results and
X_i result of the i-th playout.

In both models with some constant hand-tuning we managed to get to the 60% winning rate against algorithm with fixed c. However the performance was very unstable regarding the different time settings. For longer time per move settings the performance of these enhancements were even inferior to fixed value of c.

Initially we have been using FPU as well. We observed, that best performance is given for FPU around 0.9. However since we are using various initialization methods (progressive bias, history heuristic) we decided to drop the FPU. Instead we initialize every node in the UCT tree with v virtual visits. This corresponds to playing v games with result 0 (following meta-heuristic that a lot of positions in the tree are expected

to be equal). The motivation behind using virtual visits is following: if a node representing good move has a "bad luck" and its first couple of simulations give rather poor results then it might take some time before this move is revisited again. This leads to overall inferior performance. Virtual visits diminish this effect by smoothing the differences between sibling nodes shortly after their initialisation. As more simulations are invested to the nodes the effect of virtual visits fades away. This significantly increased the performance of the program and proved the FPU to be obsolete (the winning rate of algorithm with $v = 1$ against variant with FPU was around 64% in the time when this change was introduced). We performed large scale testing to identify the best value for virtual visits parameter in final stage of program development. For details see §5.

We played around with an idea regarding the update phase after the playout was made. The motivation is that simulation from a particular node should be the more accurate the more visits the node have. The more visits the node have, the more time is in principle spent in the tree instead of in the playouts for simulations going through the node. Promoting the accurate playouts is achieved by giving the bigger the weight to the result the more the simulations have gone through the node. We named this variant *relative update* and used the following formula for propagating results from the simulation.

$$\begin{aligned} w &= max(f(n), 1) \\ Q(s,a) &= Q(s,a) + w \cdot (sample - Q(s,a))/(n+w) \\ n &= n + 1 \end{aligned}$$

where w is a weight of the playout, *sample* is the result of the playout, f is a slowly growing function - typically a variant of *log* or *sqrt*. The only difference from traditional update formula is the weight entity - in traditional formula $w = 1$.

However the results of this approach were rather disappointing and even after hand-tuning the weight generation we were unable to get results at least matching the pure version.

4.3.4 Transpositions

Effective handling of transpositions is a must, considering the large redundancy in positions generation in Arimaa. As mentioned above, implementation of such a mechanism in UCT is way more challenging than in $\alpha\beta$ approach. A nice survey on this topic is given in [28]. We decided to implement the model called *UCT2* described in the paper. In this approach the performance and difficulty of implementation is well balanced. The model applied to our program works as follows:

There are transposition tables T. T contains pairs $< k, l >$, where k is the key and l is the list of nodes. All nodes in the tree having the same key k are listed in l. Key k for the node is computed as combination of Zobrist signature of corresponding position and depth of the node in the tree (we take into account only transpositions happening

on the same depth). Moreover every node has a pointer to the corresponding list of nodes l. First node in l is called r (representative). Now the following is assured:

- All the nodes in l share the same value $Q_s(t)$. The update mechanism is simple: when a node is about to be updated (a result of playout is ready) instead the central value in transposition tables gets updated and this value is then propagated to all the nodes involved (including the node that initiated the update).
- Nodes in l have different visit values $N_s(t)$ (sharing the visits as well could lead to a misleading bias, for more info see [28]).
- The children of the nodes are "shared". In practice when one of the nodes creates the children (usually it is r), the others point to them.

The scheme of the model is depicted in Figure 4.4. The winning rate of such a model is approximately 60% against the trivial implementation ignoring the transpositions at all.

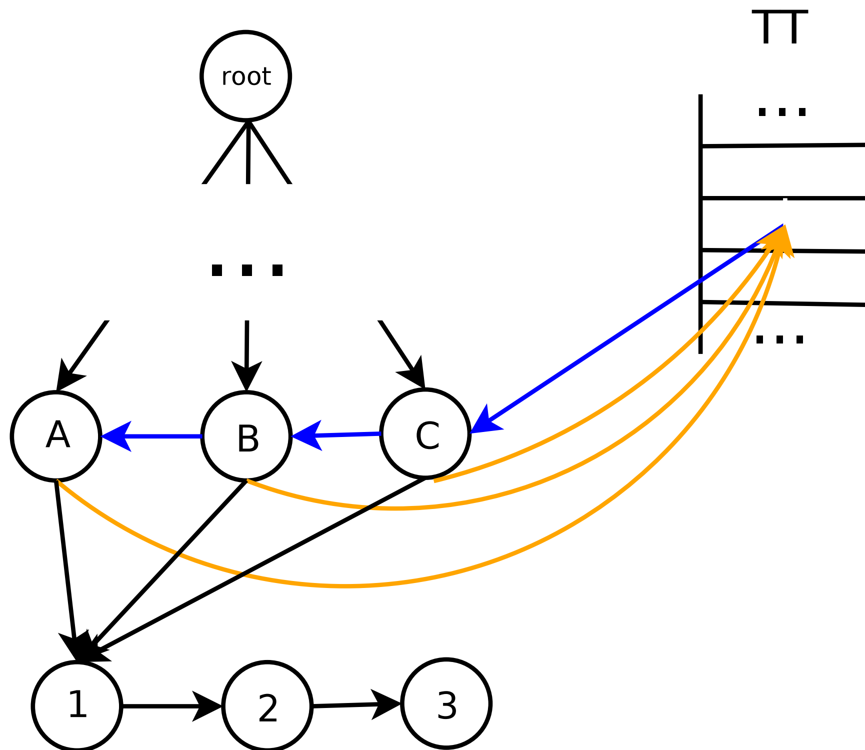


Figure 4.4: transpositions handling scheme.

4.4 Playouts

4.4.1 Playouts organization

Quite early we verified that Monte Carlo playouts to the end of the game (as known from computer Go) are not a way to pursue in Arimaa. This issue was discussed at Arimaa forum as well. The reason is simple. In Arimaa random attack is stronger than random defense. In an experiment we conducted, player who had one more rabbit instead of an elephant was on average favored by the playouts. Moreover in Go the position generally gets simplified towards the end of the game and the value of the moves is decreasing. This is not the case in Arimaa.

We decided to merge the approaches from the world of MCTS and the world of traditional $\alpha\beta$ searchers. A Monte Carlo playout is performed to a certain depth and then a simple evaluation function is applied to the position. This is a key concept behind the playouts in our program. It showed up that the most promising results are given by performing a playouts to random depth in interval of $[1, 4]$ moves.

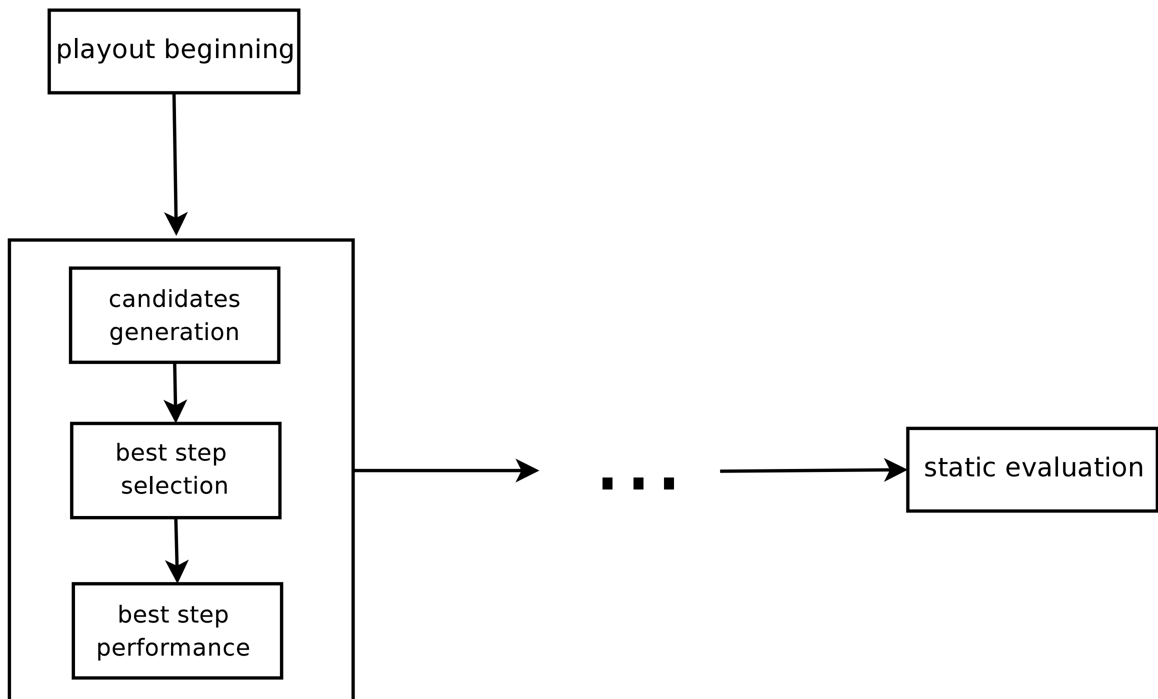


Figure 4.5: playout scheme.

Playouts' organization is depicted in Figure 4.5.

4.4.2 Step generation and selection

In Go maintaining set of potential moves is relatively easy and can be naturally updated when a move is played. In Arimaa situation is way more complicated and often it is computationally more feasible to recalculate set of potential steps after a step was made

than to try to keep this set updated. As shown in Figure 4.5 step generation is one of key elements of the playouts playing mechanism. We have experimented with two approaches:

step oriented generation
: This is a straightforward implementation. All possible steps from given position (including pass if applicable) are generated as candidates and handed to the selection mechanism.

move oriented generation
: This approach operates with idea that only a few pieces participate in a single move. In the beginning of the move a few random pieces are selected (typically 2 or 3). For every step to be made in this move candidate steps are generated only from preselected pieces. If no candidate is generated pass move is played. This leads to a significant speedup of playouts while preserving the quality of step generator. In the end performance of this method proved superior to the previous one (see §5). We also call this technique *optimized step generation in playouts.*

Best candidate (step) selection is currently based on *best-of-N* method (inspired by [16]). N random candidates are selected from all generated candidates (or at most half of them), evaluated and the candidate with best evaluation is then selected. We observed the best results for $N = 3$. Evaluation of the single step is provided by the domain knowledge unit.

We have experimented with roulette-like approach as well. N candidates (higher than in previous case) are evaluated and the best is selected randomly according to probabilities proportional to these evaluations. However this approach wasn't as successful as the previous one.

4.5 Evaluation

4.5.1 Evaluation Scheme

Once playout reaches its end, the position p is evaluated. The core of the evaluation process is the static evaluation function for a single player. This is basically a linear combination of binary features ϕ with weights θ. We map the relative evaluation of the position (difference in evaluation for gold and silver) to $[0, 1]$ interval representing estimated winning probability for gold. This is done by application of the sigmoid function σ.

$$eval(p) = \sigma\left(\phi_{gold}(p) \cdot \theta - \phi_{silver}(p) \cdot \theta\right)$$

where

$$\sigma(x) = \frac{1}{1 + e^{-\lambda x}}$$

for constant λ dependent on the evaluation elements

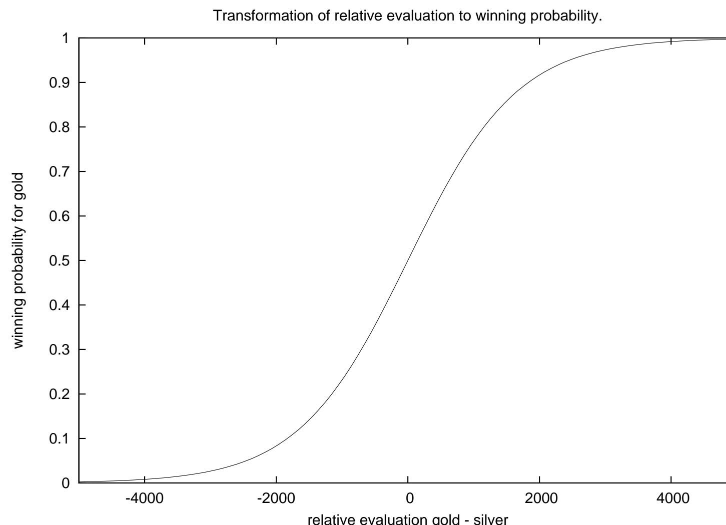


Figure 4.6: Evaluation transformation.

The sigmoid function used in the program is depicted in Figure 4.6. Its constants are tuned in such a way that if we consider only difference in material then following winning probabilities and material imbalances correspond:

52.9% rabbit

57.4% cat

72.8% horse

78.9% camel

99.8% elephant, camel, 2 horses, dog and cat

UCT algorithm operates with values from $< -1, 1 >$ interval, on the contrary the evaluation unit outputs values from much larger interval. Therefore the sigmoid transformation is necessary and quite natural part of evaluation process. Moreover it follows a commonsense that the better the position is for one player the less significant (from the winning probability point of view) are further improvements of such a position. This however leads to an interesting phenomena. If the position is very favorable for our program it is not "pressured" enough to play well and quite often it plays inferior moves. This is analogical to the well described situation in computer Go when in winning situations towards the end of the game UCT players often played unattractive moves into their own territory thus minimizing the probability of upset.

We are aware that there is a good potential for further improvements in evaluation and in transformation of evaluation into winning probability. We consider absolute weights of the features (e.g. cat's value is always 150, frozen camel always loses 20% percent of its value), however some features should have different weights depending on the context - the actual position. For instance having a cat advantage (with all

other aspects of evaluation cancelled out) should lead to different winning probabilities depending on the board situation. In the beginning of the game the cat advantage is much less significant than in the situation when only couple of pieces are left on the board.

4.5.2 Evaluation Elements

Our evaluation function consists of following independent blocks:

material
: Every piece is assigned a static value (relative strength of pieces in not taken into account). Moreover we have penalization connected to number of player's rabbits (less rabbits means higher penalization). Frozen pieces are identified and they receive a penalty relative to their value.

traps
: Every trap for every player is categorized according to: number of friendly guards, number of enemy guards, owner of the dominant guard. In some cases following attributes might be taken into account: presence of weak enemy pieces in the vicinity of the trap, whether the trap field itself is empty or not.

pins and frames [1]
: Frames are detected and penalized by the relative value of the framed piece. The same goes for pins. Moreover if the pinned piece is not locally strongest piece further penalty is given - because of possibility of pinned piece being pushed/pulled away and framed piece dying in the trap.

hostages
: Frozen camel 2-squares away from the empty traps are detected as camel hostage situation and penalized. This is a common pattern in Arimaa strategy.

blockades
: Check for complete elephant blockade is made and highly rewarded if found.

This evaluation function is principally inspired by [8]. However we strived to keep it rather simple. Moreover we found out that pieces positional evaluation (small values assigned to (piece, position) combinations) plays minor role in our program. While in the $\alpha\beta$ approach even very small difference in evaluation plays role, this is not the case in UCT. Since the value of the position in the UCT tree is determined as a mean of values of all the simulated games from this position, the small differences arising from evaluation are irrelevant.

The evaluation is enhanced by a goal-search extension. The reason for having the goal-search extension is pretty obvious - scoring a goal ends the game right-away therefore it is desirable to detect such a situation even couple of steps ahead. This is implemented by a very narrowed 4-steps look ahead whether the player to move can score

[1] see Appendix B

the goal. If this is the case no static evaluation is performed and position is treated as a win for player with a goal score.

4.6 Domain Knowledge in Steps

Step knowledge module is a function $eval_{step}$ taking position p, step s and outputting a value v of s in p. Similarly to the case of static evaluation, $eval_{step}$ is a linear combination of weighted binary features. Function $eval_{step}$ is used in two locations:

- As a heuristic value of the step in progressive bias (see §4.3.3).
- As a heuristic value of the step in *best-of-N* step selection in the playout (see §4.4.2).

Binary features included in $eval_{step}$ are based on empirical analysis of the game and appropriate weights are hand-tuned. The rules these features imply contain following:

- Moving an elephant is slightly promoted.
- Making an inverse step[2] is demoted.
- Pushing and pulling moves are slightly promoted.
- Killing opponent's piece is very promoted.
- Suicides are very demoted except for the few special cases (e.g. potential sacrifice for making a path for rabbit to the goal)
- Moving a rabbit is demoted in the beginning phase of the game, but promoted in the late phase (the actual phase is determined by the number of pieces still in the game).
- Steps close to the previous step are promoted.

4.7 Information sharing

4.7.1 History Heuristic

Well-known and widely-used technique in $\alpha\beta$ tree search. The idea is to collect statistics (number of cutoffs) on the moves throughout the tree (this is usually done for a particular level in the tree) and use these values for moves ordering in a new node (for more information see [20]).

History Heuristic was already introduced in MCTS (see §3.3.4). However it always operated on *tree-playout level* and worked with number of successes of the move in the UCT descents. We decided to try out a different model. We collect statistics (average value, number of visits) on every step in the tree, regardless of the level (depth in

[2]leading to the same position as before previous step

which the node lies). These statistics are used in the exploration formula as described in §4.3.3. Such a model is rather different from those introduced in MCTS literature earlier. It might be categorized as *tree-tree level* (regarding the naming convention presented in §3.3.4) and collects the same statistics as every node in the tree.

Enabling this mechanism had a very positive impact on the performance (see §5).

4.7.2 UCT-RAVE

UCT-RAVE pushed the performance of computer Go programs to a different level. Therefore it was quite natural to try it out in our Arimaa implementation as well. However we had certain doubts regarding its applicability in this case. There are two main reasons:

- Our playouts are relatively short compared to those in computer Go.
- Locality is stressed in the playouts.

As a consequence of these two we believed there might be a lack of information from RAVE simulations yielding no particular advantage. When implemented this issue proved to be a real obstacle. Moreover due to a non-trivial overhead less simulations were performed than with pure UCT. As a result UCT-RAVE engine demonstrated rather inferior performance without good future prospects.

4.7.3 Move Advisor

One of major ineptitudes of UCT approach is a weakness in tactical situations. This is well documented in computer Go and it was confirmed with small experiments we did as well. We believe that one of the way to fight this lies in incorporating tactical information into the playouts. In Arimaa this tactical information might be for instance a sequence achieving:

- scoring a goal
- trapping opponent's piece
- protecting dangerous trap
- taking opponent's piece hostage

It is perfectly possible to write search extensions (let's say 4 step look-ahead) for these tasks in Arimaa. However, the speed is an issue. Performing (even limited) look-ahead in the playouts is expensive. We proposed a solution we named a *Move Advisor*. The core idea is simple:

1. We perform the tactical look-ahead (this typically happens in the UCT tree in a node which has just been expanded). The trick is that this look-ahead is not performed often, thus not wasting the time.

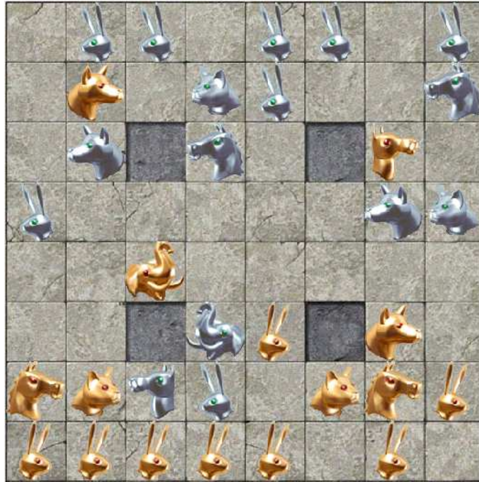


Figure 4.7: Move advisor scenario.

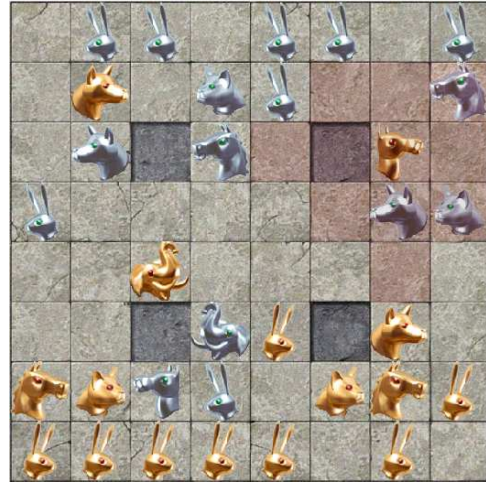


Figure 4.8: `Mg6e dg5n dg6w Mh6w` mask

2. We store the tactical sequence (move) together with its *context* (a definition of local situation necessary for a sequence to be legal and have a desired effect).

3. In the playout with certain probability the control is given to Move Advisor. A suitable sequence is selected (based on previous performance of the sequence) and played.

4. After the playout the performance statistics of the sequences that were played are updated.

We implemented the described mechanism and tested its performance. For tactical look-ahead we used a search extension, gathering sequences resulting in trapping opponent's piece (narrowed 4-steps look-ahead - a variant of extension for recognizing goals described in §4.5.2). This look-ahead is ran for both players every time a new node is expanded in the UCT tree. The biggest obstacle is how to encode a local situation : it comes down to generality vs. verification speed (during the move selection in the playout) trade-off. We store a move together with its "local" context in the same form as the bitboards are stored. "Local" context means that the neighboring position of every step in the move and every trap used in the move are stored. The places belonging to the context are determined by a simple bit mask. The move is considered legal in a given position if its masked context matches the masked position bitboard.

The example scenario is displayed in the Figure 4.7 Gold to move. This position was encountered in the UCT tree during the node expansion. Gold can trap a silver dog with a move `Mg6e dg5n dg6w Mh6w`. This tactical shot was found by a trap check search extension and the move was passed to the Move Advisor. The context (mask) generated for the move is displayed in the Figure 4.8.

Move Advisor holds all the moves retrieved from the search extensions in the tree together with their contexts and urgency (an analogy to the pure UCB1 exploration formula as described in §3.2). In the playout, the control is given to Move Advisor with certain probability $p_{advisor}$ (this is dependent on configuration, we experimented

with values around $p_{advisor} = 0.2$). If the Move Advisor is in control it goes through all the moves it has gathered so far and selects the applicable move (legal in current position) with the best score. After the playout, its result is backpropagated to the Move Advisor and all the moves used in the playout get their scores updated analogically to the update in the UCT tree. This technique operates on the *tree-playout level* according to taxonomy in §3.3.4.

The program version equipped with Move Advisor performs less simulations (around 95% of pure version). However, it was able to consistently beat the pure version in 52.5% of the games[3]. And even though it is still quite naive (especially the context creation and verification) it has proved to bear potential for future research.

4.8 Speedup

4.8.1 Parallelization

Since we had an opportunity to run the program on up-to 4-core machines, we decided to parallelize it as well. From the models discussed in §3.3.6 we chose the *root model*. Mainly for its simplicity to implement and proclaimed efficiency for low number of cores. The number of simulations was almost linear ($0.95 \times single_core_simulations \times cores_num$). However the performance was truly disappointing - under various time conditions the win ratio of 4-core mode against single core mode was only around 50% (the same held for experiments on 2 cores). This is especially surprising because of the excellent performance of this model in computer Go.

We believe that using the *tree model* with local mutexes as described in [11] would provide much better results. However we have decided not to incorporate it into the engine mostly because of a lack of time and potential implementation uneasiness. Instead we proposed and implemented a new model called *island* parallelization model. The basic idea is that threads should work separately (islands) with time-to-time synchronization ("expedition" from one island to another). The implementation is pretty straightforward:

- There is one master tree and one tree for every thread. Every node has a link to a corresponding node in the master tree.
- During the node creation its link to a corresponding node in the master tree is created. Moreover if there is not such a node in the master tree it is created first (this operation is locally mutexed).
- Before UCT formula is calculated a node is synchronized with its master with certain probability (synchronization is locally mutexed). Synchronization represents both updating statistics in the master node and local node.

The observations we made regarding the model are:

[3]the winning rate used to be higher, however it decayed with adding other enhancements

- In principle this model lies between the *root model* and the *tree model*. For this reason we haven't expected a significant boost in performance (the *root model* totally failed), still the parallelized version proved to be stronger than a single core version (see §5). For this very same reason we believe that this model is worth trying in the domain of computer Go where the *root model* proved to be successful.

- We have kept the history heuristic tables and transposition tables local in particular trees. This was done mostly for implementation convenience. Sharing the transposition tables could bring further speedup because the overhead necessary for maintaining the tables would be performed only once in the master tree. On the other hand, there would be a trade-off in (locally) mutexing the shared transpositions tables on update.

- The synchronisation frequency (probability of synchronizing the node) might be set according to the number of threads in order to minimize the potential time spent waiting for the lock. One strategy is to have a high probability of syncing if there are little number of threads and vice versa.

- A nice property of this implementation is that it could be reprogrammed back to the *root model* implementation simply by dropping the continuous synchronization and performing a full-tree synchronization in the end of the search (actually we followed this path in the reverse direction).

4.8.2 Optimization

We have focused on both the higher and the lower level optimization. An example of higher level optimization we used is the children caching technique as described in §3.3.6. The actual implementation is as follows:

- When a node has over $threshold_{ccache}$ visits ($threshold_{ccache} = 50$) its caching mechanism is activated.

- Node creates a children cache[4] for $size_{ccache}$ children ($size_{ccache} = 5$). Postponing the children cache creation like this - until it is truly needed - yields extra performance compared to approach when a cache is statically created in the process of the particular node creation.

- The cashed is flushed every time when $\lfloor \sqrt{visits} \rfloor$ equals to a yet unvisited natural number.

The result is approximately 20% speedup in the number of playouts per second. This variation beats the "not-optimized" variation in 57% of the games.

Towards the end of the development process we have performed lower level optimizations as well. We have done some profiling to identify performance bottlenecks. We

[4] dynamically allocated array

aimed for a Pareto effect (80-20 rule) making little optimizations that would produce good effects. One of the bottlenecks was in the step generation. Especially finding an index of a first bit in bitboard (so called *lix* function). Originally we have been using trivial implementation of cycling the bit vector and stopping on the first bit that is on. Some processors even have instructions for *lix* operation. We haven't gone that far though. We used a version of binary search with precomputed values. We created a precomputed logarithm table (int to int mapping) for computing the first bit position in 8 bit vectors. To narrow down the potential position of first bit from 64 bit vectors to 8 bit vectors we perform the binary search (hard-coded if statements for performance). This optimization raised the performance by about 10%.

Further optimizations include:

- pre-computing *sqrt* and *log* functions used in the exploration formulas calculation.
- minimizing the size of nodes in the UCT tree in order to cut down amount of memory access
- creating own object allocation mechanism (object pool) for *board* objects. Motivation for this is that instead of *play-unplay* realm we use the *copy board-play-destroy board* realm.

Chapter 5

Performance and Experiments

5.1 Methodology

We have been measuring the performance of the engine on three levels via:

Arimaa Test Suite [1]

We maintained a set of approximately 20 hand created tests (representing mostly tactical shots). These were quite unreliable as a measurement whether the engine improved. On the other hand, they did a good job signalling if the performance of the program went down rapidly. As a consequence we used the *ATS* as a kind of preliminary sieve.

Offline matches

This was our main tool for performance study. We have let the engine play thousands of games against itself and anchor engines. As anchor engines we have successively used: *bot_sample* (4 steps lookahead), *bot_sample* (8 steps lookahead), *bot_fairy* (3s/move). Each of these beats the previous one in about 95% games. While the *bot_sample* is relatively weak (in both configurations), *bot_fairy* is a reasonable opponent. Some of its modifications showed good performance in the past computer Arimaa challenges. As the time went on our engine was able to match all of these (with a little time handicap).

Performance in the online gameroom

During the final stage of the program development process we let the engine to play in the online gameroom[2] (with *bot_akimot* nickname). While this is not as good measure of performance as offline matches it gave us a glimpse of engine's strength against human players under specific time conditions (15s/move). The program was able to achieve rating around 1600 with peak in history at 1733. It also showed some interesting victories against strong Arimaa human players.

[1] see Appendix A

[2] http://arimaa.com/arimaa/gameroom

5.2 Experiments

The experiments are depicted in following graphs. If not stated otherwise, then the opponent is an anchor bot *bot_fairy* 3s/move. Every point in every graph was computed based on up to 400 games played under specified conditions. Result r was computed as a mean of outcomes of these games (1 for a win, 0 for a loss). With this setup we are 95% confident that the true result would lie in the interval $r \pm 0.05$. This is based on the interval estimate for alternative distribution:

$$\overline{X} \pm \mu_{1-\frac{\alpha}{2}} \sqrt{\frac{\overline{X}(1-\overline{X})}{n}}$$

where μ is quantile of normal distribution, $\alpha = 2$ and n is the number of trials

Conducting all these experiments was quite time expensive. It took us more than a month to gather all the data. This is one of the reasons why we have chosen *bot_fairy* 3s/move as a standard anchor bot - using bot with longer thinking times would require even longer time to simulate all the games.

scalability test

Simple scenario to show that the performance scales well with additional time. Engine's best configuration (see Appendix A) plays against variations of anchor bot (with predefined number of seconds per move $(1, 2, 4, 8, 16)$. Same time conditions were used in other tests as well to show that presented improvements scale well with time. Results are given in Figure 5.1. The time scale is logarithmic. As expected against opponent with fixed time (*bot_fairy* 3s/move) the winning ratio is rising quite quickly. The good news is that the winning ratio is rising (slowly though) also against opponent with equal time conditions (*bot_fairy* equal). For 16s/move the winning percentage is nearly 50%. Moreover, the shape of the curve looks promising for our engine under even longer time settings.

improvements test

Here we have selected a fixed set of improvements (add-ons) to experiment with (according to our beliefs we have chosen the most significant ones). These were:

- knowledge in playout (plk) - see §4.6
- knowledge in tree (tk) - see §4.3.3
- history heuristic (hh) - see §4.7.1
- optimized step generation in playouts (opl) - see §4.4.2

First we have tested the setup with all of these improvements disabled (plain version). As expected results were so poor, that disabling another modules would probably have very little effect. Experiments with using only single add-on at a time (plain version + add-on) are depicted in Figure 5.2. It was quite surprising to observe that using knowledge in playout add-on has a very little benefit over the plain version. Experiments with using the add-ons cumulatively are given

in Figure 5.3. The order was chosen according to results of add-ons in previous experiment. The effect of using the playout knowledge is quite reasonable here and it surpassed expectations based on its performance as a single add-on (this shows dependency among add-ons). Based on these scenarios we have identified history heuristic and optimized step generation in playouts to be the most significant improvements.

parameters tuning test

Games against anchor bot proved to be the most valuable tool in program's parameters' tuning. As an example we present an analysis showing impact of number of virtual visits on the performance. Engine is run under various time conditions with different virtual visits value. Results are displayed in Figure 5.4. For low values of virtual visits ($\{1, 2, 3\}$) the performance is quite poor. The peak is for $vv = 4$ and $vv = 5$ - this value is used in the strongest engine configuration. With higher values the performance starts dropping again. Very poor results of FPU ($vv = 0$, $fpu = 1.1$) variant was initially quite a surprise to us. However, we realised we dropped the FPU concept quite a long time ago and we have introduced some significant changes in node's initialization since then (progressive bias, history heuristic). These changes don't cooperate well with FPU, leading to inferior tree exploration and thus poor performance.

parallelization test

Opponent of the engine in the parallelization test was the Akimot engine running on single thread. Engines are using no transposition tables since the multi threaded version hasn't yet been optimized for cooperation with these (it works, however with poor performance). Results of the experiment are given in the Figure 5.5. While the results of parallelized version are clearly superior for lower times, the difference is gradually dropping as the time per move increases. This is expected behaviour - additional time for thinking is the more worth the less the time engine has.

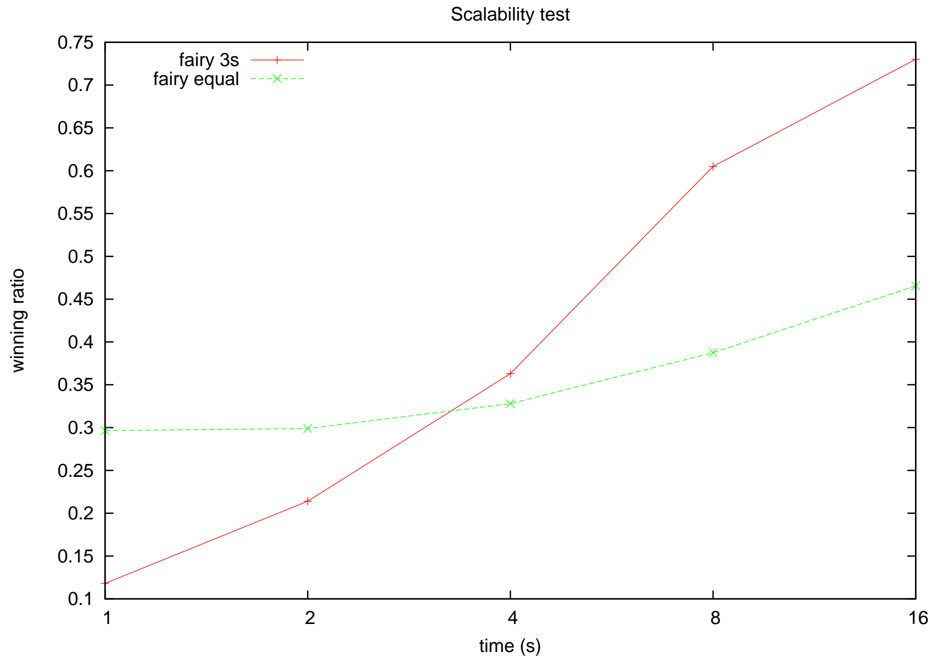


Figure 5.1: Scalability

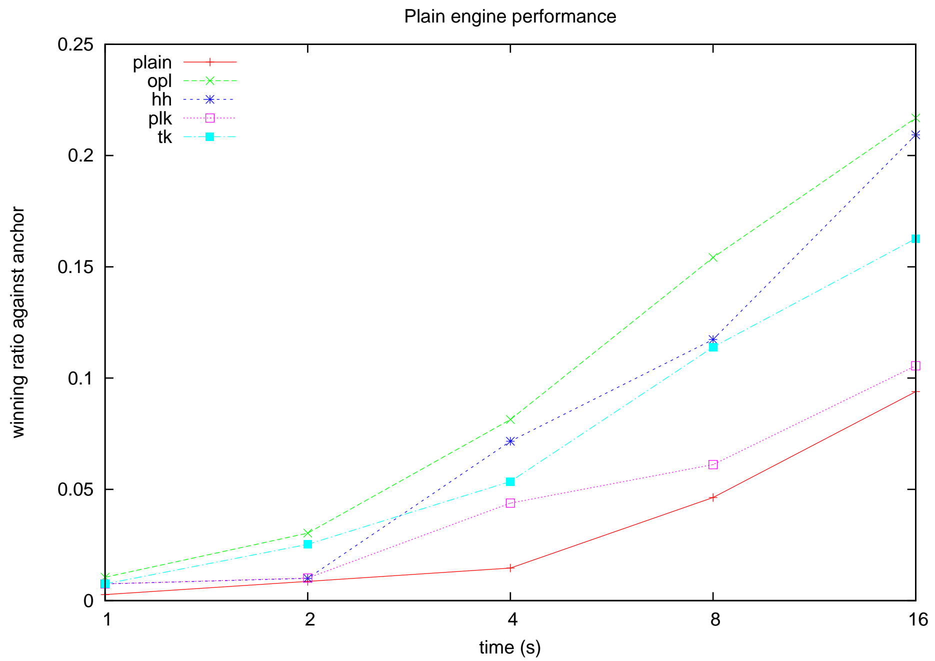


Figure 5.2: Improvements single

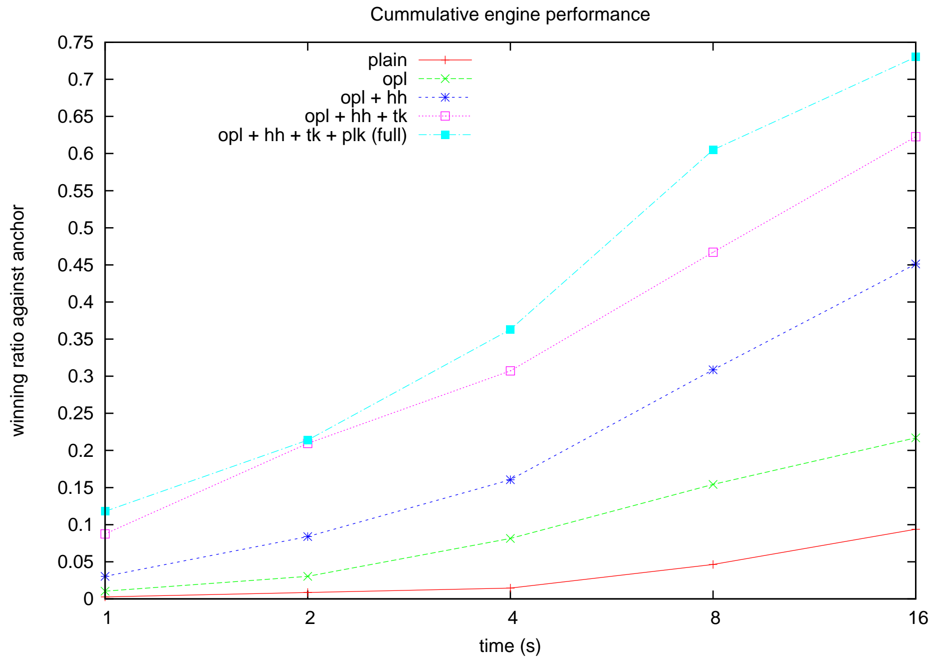


Figure 5.3: Improvements combination

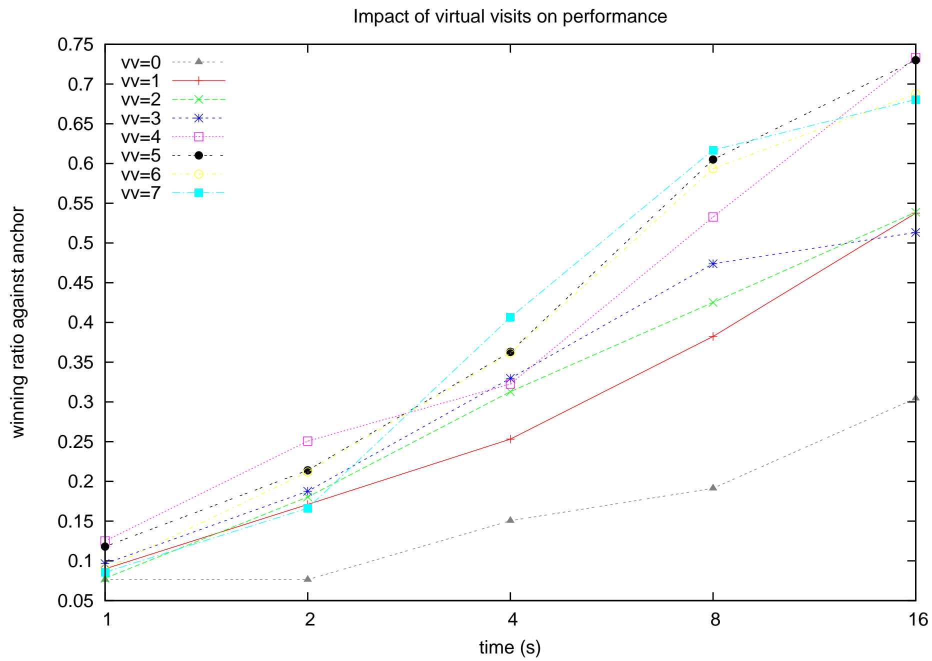


Figure 5.4: Virtual visits impact

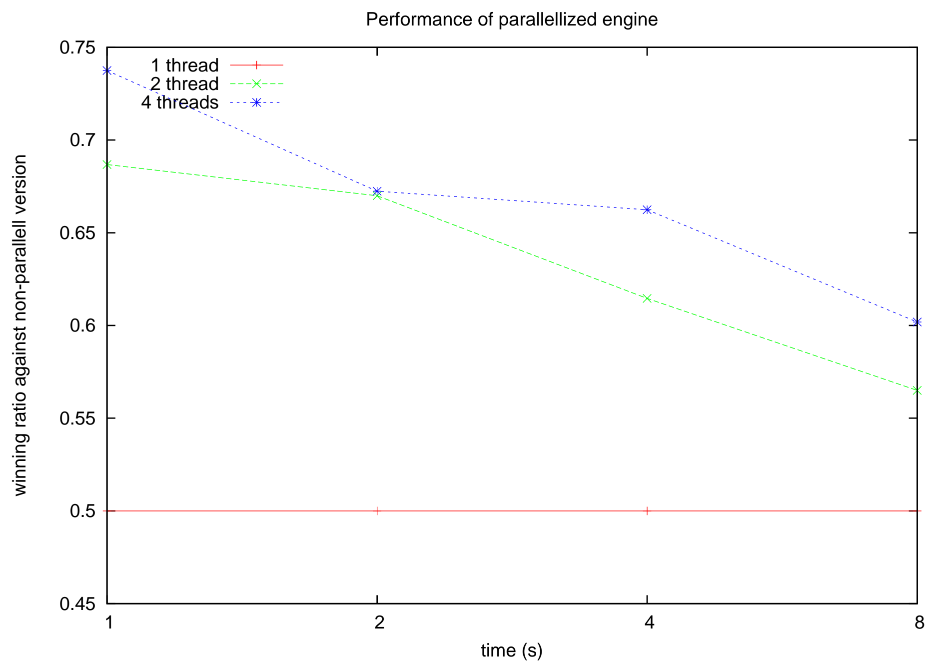


Figure 5.5: Parallelization

Chapter 6

Conclusion

6.1 Achievements

In this thesis we have analyzed applicability of MCTS methods in the game of Arimaa. We have adapted existing MCTS algorithms for Arimaa in our MCTS bot called Akimot. We have shown that as a *proof-of-concept* our MCTS implementation was successful.

A naive implementation of UCT engine (with playouts to the end of the game) gave very poor performance. Therefore we have devised a hybrid playouts concept consisting of shortened playout followed by a static evaluation. Moreover, we have optimized playouts to tackle non-trivial issue of step generation in Arimaa.

Naturally we have implemented many known enhancements from computer Go and tested which work in our setup. It came out that domain knowledge incorporation (both in playouts and the UCT tree), *transposition tables*, *virtual visits* or *children caching* provide measurable improvement. On the other hand, UCB-tuned formula or UCT-RAVE framework haven't shown a desired effect, in spite of being very successful in computer Go.

We have proposed several new enhancements for the generic MCTS algorithm as well. For instance the *tree-tree* level *history heuristic* contributed to information sharing across the UCT tree and raised the performance of the engine significantly. The *move advisor* framework strengthened the tactical profile of the playouts. We have also proposed and implemented a new parallelization method called the *island* model.

During the development we have naturally encountered many obstacles. For instance the described *Easy way effect* caused by the UCT algorithm's nature, peculiarity of Arimaa having 4 steps in a move and chosen search granularity (step-based search).

We have conducted numerous performance tests to prove good scalability of our engine. Akimot is competitive with an average $\alpha\beta$ engine *bot_fairy* under short time settings, having better prospects for longer time settings. We have also tested several particular parts of the engine and identified their contribution to the performance.

As a part of the development process we have created several supportive applications for our engine: Arimaa Test Suite, Simple Development Gui, Rabbit Goal Tester, etc. The whole project will be provided to the Arimaa programming community.

6.2 Research Guideline Revisited

6.2.1 Objectives

- To propose and implement MCTS integration in a bot playing the game of Arimaa.
  Done We have created a MCTS Arimaa engine called Akimot.

- To identify ways of improving MCTS algorithms which work in Arimaa.
  Done We have devised and tested numerous extensions to the standard UCT algorithm.

- To check whether MCTS might be successful in a game dissimilar to the game of Go.
  Done We have conducted various experiments against existing bots and human players.

6.2.2 Research Questions

- How the Monte Carlo playouts must be rebuilt to be applicable in the game of Arimaa?
  We have verified that using Monte Carlo playouts "as is" (playouts to the end of the game) results in a poor performance. We have proposed and implemented a hybrid evaluation consisting of shortened MC playout, followed by a static evaluation.

- Which of the proposed improvements to the MCTS algorithms are domain independent?
  History heuristic on the *tree-tree* level posed a big improvement in our approach. This enhancement is not documented in the literature and on its own it could be a good method to improve the UCT algorithm in other domains as well. For us the main motivation for using the history heuristic was a substitution of inept UCT-RAVE. While we had little time to improve the *move advisor* concept, it might carry a good potential for incorporating the search knowledge into the playouts. We also believe that proposed *island* parallelization model would be promising in computer Go.

- Is there a potential for MCTS algorithms in Arimaa to start the kind of revolution they did in computer Go?
  Rather not. Even though our program proved to be able to achieve reasonable results against average engines, the top engines are on a completely different level. We see the main reason in the fact that static evaluation function provides a base for a relatively strong player in Arimaa as opposed to the situation in Go (where static evaluation functions performance was truly poor). This allows the $\alpha\beta$ Arimaa bots enhanced with search extensions and hand coded knowledge to compete with moderately strong humans.

6.2.3 Hypotheses

- Monte Carlo playouts used "as is" from the game of Go will provide a weak Arimaa player.
  True

- Standard improvements from computer Go will improve the MCTS Arimaa player as well.
  Partially True Knowledge in playouts, progressive bias, virtual visits, children caching, etc. improved the strength of the engine. On the other hand, using standard techniques as *UCB-tuned*, *UCT-RAVE*, etc. yielded no significant improvement.

- MCTS Arimaa player might be competitive to the existing $\alpha\beta$ searchers.
  Partially True As mentioned above Akimot is competitive to the average $\alpha\beta$ searchers. However, the top bots are on a different level.

6.3 Future Work

We have decided to keep up the work on the engine and maybe participate in some future Arimaa Computer Challenge. Especially following issues deserve further attention (they are ordered according to our beliefs on their importance):

parameter tuning
: We strived rather for introducing more extensions than to spend extra time to tune the parameters for maximal performance. There are tens of constants marginally influencing the behavior of the algorithm. We believe that automatic tuning of these, in for instance evolutionary manner, might bring significant strength improvement. Moreover, for most of the experiments we have used rather short thinking times (at most 16s/move), optimal parameters setting might be different for longer thinking time.

search extensions
: We have programmed the search extensions (trap check and goal check) as reduced 4 step lookaheads. Other Arimaa engines are known to use the static check in the form of a decision tree instead. This significantly speeds up these extensions and allows them to be used for instance in the static evaluation.

time management
: Currently the time management during the game is overly simple. Engine performs search for a fixed time per move and then outputs the best move so far. From the nature of UCT way more efficient time management can be implemented. With information on move confidence "obvious" moves might be played quickly thus saving time for potentially complicated positions.

parallel model
: Even though the results of parallelization with the *island model* were rather

mediocre, still using multiple cores is worth it. We would like to incorporate parallelization with transposition tables and use the parallel functionality in the engine. It would also be interesting to check its performance in the domain of computer Go.

move advisor

We believe that the framework we introduced as a *move advisor* bears further contribution to the engine's strength. Especially more precise specification of the move context is worth examining.

playouts

Monte Carlo playouts are a natural performance bottleneck of the MCTS engine. The playouts in Akimot are relatively light only with the basic knowledge incorporated. Applicability of additional knowledge or maybe balancing methods as introduced in [25] deserves further study.

initial setup

Right now single fixed initial setup is used (for both players). It would be desirable to use a small "library" of good initial setups (this might be fetched from the Arimaa Games Archive).

Bibliography

[1] Kasparov deep blue match 1997. http://www.research.ibm.com/deepblue/home/html/b.shtml.

[2] Omar Syed. Arimaa: A new game designed to be difficult for computers. *Journal of the Internation Computer Games Association*, 26(2):138–139, 2003.

[3] Arimaa homepage. www.arimaa.com/arimaa.

[4] Martin Mueller. Computer go. *Artificial Intelligence*, 134, 2002.

[5] Martijn C. Schut. *Scientific Handbook for Simulation of Collective Intelligence*. 2007.

[6] Arimaa branching factor study. http://arimaa.janzert.com/bf_study/.

[7] Introduction to arimaa strategy. http://en.wikibooks.org/wiki/Arimaa/Introduction_to_Strategy.

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

[9] David Fotland. *Building a World-Champion Arimaa Program*. Smart Games, 2004.

[10] Bernd Brugmann. Monte carlo go. Technical report, 1993.

[11] Guillaume M. Chaslot, Mark H. Winands, and H. Jaap Herik. Parallel monte-carlo tree search. In *CG '08: Proceedings of the 6th international conference on Computers and Games*, pages 60–71, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[13] Levente Kocsis and Csaba Szepesvari. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[14] Sylvain Gelly and Yizao Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go, December 2006.

[15] Guillaume Chaslot, Mark Winands, Jaap H. van den Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for Monte-Carlo tree search. In *Joint Conference on Information Sciences, Salt Lake City 2007, Heuristic Search and Computer Game Playing Session*, 2007.

[16] Peter Drake and Steve Uurtamo. Move ordering vs heavy playouts: Where should heuristics be applied in Monte Carlo Go? In *Proceedings of the 3rd North American Game-On Conference*, 2007.

[17] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, France, November 2006.

[18] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 273–280, New York, NY, USA, 2007. ACM.

[19] Rémi Coulom. Computing Elo ratings of move patterns in the game of Go. *ICGA Journal*, 30(4):198–208, December 2007.

[20] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11(11):1203–1212, 1989.

[21] Peter Drake. Heuristics in monte carlo go. In *In Proceedings of the 2007 International Conference on Artificial Intelligence, CSREA*. Press, 2007.

[22] Bruno Bouzy. History and territory heuristics for monte-carlo go. *New Mathematics and Natural Computation*, 2(2):1–8, 2006.

[23] Humans no match for go bot overlords. http://www.wired.com/wiredscience/2009/03/gobrain.

[24] Humans computers go challenge. http://www.computer-go.info/h-c/index.html.

[25] David Silver and Gerald Tesauro. Monte-carlo simulation balancing. 2009.

[26] Stefano Carlini. Arimaa: From rules to bitboard analysis. Technical report, December 2008.

[27] Albert L. Zobrist. A new hashing method with applications for game playing. *ICGA Journal*, 13(2):69–73, 1990.

[28] James H. Brodeur Benjamin E. Childs and Levente Kocsis. Transpositions and move groups in monte carlo tree search. 2008.

Appendix A

User manual

A.1 About

Akimot is an engine for the game of Arimaa. The very core function of the program is to take a valid Arimaa position and produce a suggested move. The program is command line oriented, highly configurable and supporting communication in both traditional *getMove interface* and relatively new *Arimaa Engine Interface*. There are several support applications delivered together with the program. The whole project is distributed under the GNU GPL license.

A.2 Background

The program is written in C++ programming language. The target platform are the *Linux* systems. Building the program at *Windows* machines hasn't been tested. For development we used the *Vim* integrated development environment together with *Scons* software build system. As a source version control tool we used the *Git* software. As a unit testing module we used the *cpptest* library. We used *gdb* for debugging and *gprof* for profiling. Moreover we programmed several external testing tools. For instance to verify that our search extension for finding goal works properly we harvested all the positions where rabbit can score a goal from all the games ever played online. And through *AEI* we tested the ability of our program to find the goal moves in these positions (> 50000).

A.3 Installation

Project is distributed as a snapshot of the development version. This snapshot is included on the attached CD. The rough organization of the project is following:

akimot

- source files - **.h*, **.cpp*
- *a* - shortcut for `akimot -e -a init`
- *AUTHORS* - authors information
- *COPYING* - license information
- *default.cfg* - an example akimot's configuration file
- **doc** - Doxygen generated reference documentation
- *Doxy* - configuration file for documentation generator
- *init* - file with initial commands for the AEI session with the program
- *INSTALL* - installation and compilation instructions
- **other** - Support software
  - **aei** - Arimaa Engine Interface source codes
  - **match** - Match environment with example bots
  - **ats** - Arimaa Test Suite code and tests
  - **tagui** - Development GUI
  - **rabbits** - Large scale goal check unit
  - **aga** - Tools for downloading and filtering the games from online archive
- *paths.py* - small support file with paths definition
- *s* - shortcut for `scons opt=1`
- *SConstruct* - configuration file for the build process
- *TODO* - programming issues TODO list

There are no precompiled binaries therefore the program itself must be built from the source codes. Recommended way for building the binary is to use a *Scons* tool. There is a prepared *SConstruct* configuration file for this job. Following commands might be used :

`scons` development build

`scons opt=1` optimized build (**recommended for standard use**)

`scons prof=1` build including profiling information

`scons dbg=1` build for debugging purpose

`scons -c` clean the build (removes object files and binary)

`scons cfg=1` copies configuration file *default.cfg* to predefined places (e.g. *match* dir, *ats* dir, etc.)

The built program should get automatically installed to the proper destinations in subsequent directories (e.g. **match/bot_akimot**, **ats/bot_akimot**, etc.). You can force the installation as well - for instance to force installation of optimized source to **match/bot_akimot** just issue `scons opt=1 match`.

A.4 Options and Configuration

Command line options are used to customize the general behavior (mode the program is in, communication protocol to use, etc.). The syntax for running the program is: `akimot [options] [position_file [game_record_file]]`. The files listed after options are part of prescribed communication via the *getMove interface*. The options are following:

`-h` prints small help

`-b` runs benchmarks

`-e` uses extended AEI command set

`-a file` uses given file to init the AEI session

`-g` runs in getMove mode (`position_file` and/or `game_record_file` must be supplied then)

`-c file` uses given file as a configuration file

Configuration file is used to modify the properties of the search engine. The default configuration file named `default.cfg` is present in the project's root. This configuration file is used by the program if not specified otherwise in command line options. Moreover, it contains the "best" configuration used for scalability tests and games in the gameroom. Every item in the file is documented and should be easy to understand. Various issues might be influenced in the configuration file, for instance:

- details of the algorithm (time settings, exploration coefficient in UCT, mature level, length of the playouts, etc.)
- various (on/off) extensions (transposition tables, knowledge bias in playouts, search extensions, etc.)
- parallelism degree (number of threads to use)
- weights of the evaluation method (e.g. what are particular pieces worth, how are traps evaluated, what is camel hostage penalty etc.)
- weights of the step evaluation

Little effort was invested to make the program's input processing "dummy proof".

A.5 Session

Program supports two distinct ways of interaction:

getMove mode

So far, this communication protocol has been recognized as an official protocol for computer challenge Arimaa championships. Program takes three files with *game_position*, *game_record* and *game_state* and outputs a move to be made. For a long time this was an only way how to connect a bot to the online gameroom. In Akimot this mode must be explicitly toggled with `-g` option, moreover not all three files must be present (actually information from *game_state* file are not used by Akimot at all). Example sessions:

load from the position

```
tomik@linda ~/src/akimot $./akimot -g data/captures/02.ari
Ed2n Ed3n Ed4n Ed5n
```

load from the record

```
tomik@linda ~/src/akimot $./akimot -g data/captures/02.are
Ed2n Ed3n Ed4n Ed5n
```

load from the record (preferred over the position)

```
tomik@linda ~/src/akimot $
./akimot -g data/captures/02.ari data/captures/02.are
Ed2n Ed3n Ed4n Ed5n
```

AEI mode

Akimot implements the textual AEI protocol as described below. This is a preferred way of communication with the program. We also used this model for a connection to the gameroom. Example session[1] :

```
tomik@linda ~/src/akimot $ ./akimot -e
#start the initial opening phase (a handshake)
<aei
>id name akimot
>id author Tomas Kozelek
>id version 0.1
>aeiok
#handskake was performed successfully now ping the engine
<isready
>readyok
#start the new game
<newgame
#set the time for move per sec to 5
<setoption name tcmove value 5
<setpositionfile data/captures/02.ari
```

[1]symbol <and >are used only here to emphasize the direction of communication, # marks a comment

```
<go
>log Debug: Search finished. Suggested move: Ed2n Ed3n Ed4n Ed5n
>info stat UCT:
>info stat   355069 playouts
>info stat   4.90003 seconds
>info stat   72462 playouts per second
>info stat   449253 nodes in the tree
>info stat   23326 nodes expanded
>info stat   77962 nodes pruned
>info stat   6.76431 average descends in playout
>info stat   best move: Ed2n Ed3n Ed4n Ed5n
>info stat   best move visits: 113535
>info stat   win condidence: 0.436083
>info time 4.90004
>info winratio 0.436083
>bestmove Ed2n Ed3n Ed4n Ed5n
>log Info: over
<quit
>log Info: bye
```

Akimot allows to init the *AEI* session with user defined commands. Commands are written to the init file and given to the program on the command line. Only the beginning of the session (e.g. `aeiinit`, `newgame` command, etc.) or the whole session as well might be issued. In the case of whole session being performed from the init file some commands from AEI extended set must be used(i.e. `gonothread` instead of `go` otherwise following commands are sent to the engine too early). Example init file called *init* with comments (`akimot -e -a init` runs the engine with this init file in extended AEI mode):

```
aei
newgame
setoption name tcmove value 6
setpositionfile data/captures/06.ari
gonothread
#view the board and the resulting tree
boarddump
treedump
#make the move proposed by the previous (recent) search
makemoverec
#start new search(now from the other player's view)
gonothread
quit
```

A.5.1 Position formats

Current position might be communicated to program in several ways. All formats are accompanied with representation of example position (see Figure A.1) in given format.

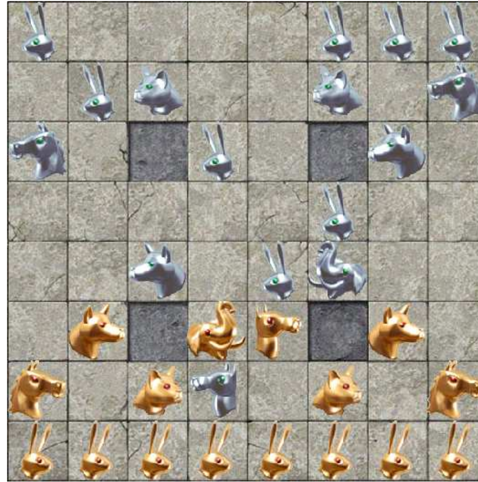


Figure A.1: Example position.

standard position format

This format is used for *game_position* file in getMove mode as well as in combination with `setpositionfile` command in AEI mode. Token 8b means this is position in the 8th move with black(silver) to play.

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

compact position format

Position is given by the side to move (w/b) followed by a string embraced in '[' ']' and consisting of the piece letter or a space for each of the 64 squares. This format is used only in AEI mode (in combination with `setposition` command). Information on move number is not included.

```
b[r    rrr rc  c hh  r  d      r    d re   D EM D H Cm C HRRRRRRRR]
```

game record format

This is an official system of recording the single game of Arimaa. Game recorded in this format might be passed to Akimot in getMove mode as a *game_record* file.

```
1w Ra1 Rb1 Rc1 Rd1 Re1 Rf1 Rg1 Rh1 Ha2 Db2 Cc2 Md2 Ee2 Cf2 Dg2 Hh2
1b ra8 rb8 rc8 rd8 re8 rf8 rg8 rh8 ha7 db7 cc7 ed7 me7 cf7 dg7 hh7
2w Ee2n Ee3n Ee4n Ee5n
2b ed7s ed6s ed5s rd8s
3w Ee6w me7s Db2n Dg2n
3b rc8e re8s dg7s ed4e
4w me6s Ed6e Ee6w re7s
4b db7s ee4e rb8s rd7e
5w re6e Ed6e me5w Ee6s
5b rd8s rd7s re7s rf6s
6w md5s Ee5w md4s Ed5s
6b db6e dc6s re6s dc5s
7w Ed4e dc4e Ee4s dd4e
7b ha7s de4w re5s dd4w
8w Md2e md3s Ee3w Me2n
8b
```

A.6 Arimaa Engine Interface

Arimaa Engine Interface is an interface allowing engine to connect to gameroom or other applications. *AEI* is written in Python and was contributed to the Arimaa community by Brian Haskin. AEI defines a textual protocol based on *UCI*[2] for communication with the Arimaa engine. AEI is meant to replace older *bot interface* for connecting to the online gameroom. While AEI is not yet a widely adopted communication protocol in the Arimaa programming community, it gains popularity pretty quickly. We used AEI as a primary communication protocol with Akimot and as the only mean of connecting the engine to the gameroom. Moreover we use AEI as a module in related Python applications we wrote during the development process, namely : Arimaa Test Suite, Arimaa Development GUI, Rabbit Goal Tester.

We have implemented most of the commands defined by AEI and even added some more (we call these an *extended AEI set* - for these to be recognized, engine must be started with `-e` option). The full specification of AEI protocol is given in *aei-protocol.txt* in **other/aei** directory. The following list is based on this specification and represents a list of commands supported in Akimot. Commands from extended AEI set are marked with *.

[2]Universal Chess Interface

Controller to Engine Messages

`aei` First message sent to begin the opening phase. Waits for `id` messages and an `aeiok` message back from the engine to end the opening phase.

`isready` Pings engine. Engine responds with `readyok`.

`newgame` Signals the start of a new game.

`setposition <position>` Sets the current position from the string in compact position format (see §A.5.1).

`setpositionfile`* `<filename>` Gives the path to a file with a position in standard position format.

`setoption name <id> [value <x>]` Set further options. Supported options are:
`tcmove` - The per move time for the game.
Other options (see full specification) are parsed and recognized, however they currently have no effect on the program behavior.

`makemove <move>` Makes a move. Stops any current search in progress.

`makemoverec*` Makes a move that was found as a bestmove in the last search.

`go [infinite]` Performs search according to its time management and responds with the best move found. Further option `infinite` specifies to search until the `stop` command is received.

`gonothread*` Analogical to `go` command. An engine is supposed to search in the current thread (usually current thread is used for performing the AEI communication, while search runs separately). This is useful in batch AEI scripts where commands are given sequentially in advance.

`boarddump*` Prints the current board.

`treedump*` Prints the search tree from the last search.

`eval*` Evaluates current position.

`goalcheck*` Performs goalcheck on current position.

`trapcheck*` Performs a check whether some pieces can be trapped in current position.

`stop` Stops the current search. The engine responds with the bestmove found.

`quit` Exits the session.

Engine to Controller Messages

`id <type> <value>` Sends identification during the opening phase of the session. Sends information on following identifiers `name`, `author`, `version`

`aeiok` Ends opening phase and starts the main phase of the session.

`readyok` Answer to `isready` message after all previous messages from the controller have been processed.

`bestmove <move>` Best move found in search

`info <type> <value>` Information about the current search. Akimot issues info messages of following types (all bound to previous search):

- `time` How long it took to perform the search.
- `winratio` Expected winratio of the suggested bestmove.
- `stat` Various statistics from the previous search. These include information on number of playouts, playouts per second, number of nodes in the tree (all, expanded, pruned), average descend depth, etc.
- `goalcheck` Information on performed goalcheck.
- `trapcheck` Information on performed trapcheck.

`log <string>` Logging information. Log messages start with *Error:*, *Warning:* or *Debug:* to indicate special handling by the controller.

A.7 Arimaa Test Suite

Arimaa Test Suite is a small Python application for testing the strength of an Arimaa engine on predefined Arimaa positions representing particular tactical or strategic maneuvers. This tool can be viewed as a sort of unit test for algorithmic side of the program. The main motivation for creating this framework was a need for quick testing whether algorithmic changes hurt or improve the performance of the program. Even though its accuracy is questionable it proved to be a useful tool to quickly identify clearly bad extensions.

The framework has a given format for defining tests. Every test carries : single Arimaa position, a comment on position, multiple tags and test how well are the criteria fulfilled. Currently implemented criteria are: score goal, prevent opponent's goal and piece position criteria which is basically an *AND-OR* description of position changes with weights. This description (*after_piece_position* field) is a collection of *blocks* separated by "|" character. Every block consists of *atoms* separated by whitespace and optional weight indication in the end of the block definition separated by ":" character. Atoms express certain assumptions about position of pieces after the move. If the assumption is correct after the particular move we say that atom is *satisfied*. Atom can be of one of the following types:

- An indication of position - e.g. `Eb3`, this atom is satisfied if after the move there will be gold elephant at `b3`
- a negation of position indication - e.g. `!ce2`, this atom is satisfied if after the move there is not a silver cat at `e2`.
- An indication of trapping - e.g. `Hf6x`, this atom is satisfied if during the move the gold horse is trapped at `f6`.

A block is *satisfied* if all of its atoms are satisfied. If a block is satisfied it is *evaluated* with its weight (or 1.0 if there is no explicit weight given) otherwise it is evaluated with 0. Weights are meant to provide broader evaluation for possible moves than just passes/fail. The result of the test is the maximum from evaluations of its blocks. Example ATS test file (corresponding to position depicted in Figure A.2):

```
[settings]

comment =
  Elephant blockade at f7.

position =
  12b
   +-----------------+
  8| r r r r . c r . |
  7| h . c m r E h r |
  6| . d X . . X d . |
  5| . . . . . r . . |
  4| . . . . . . . . |
  3| . H C e . X D . |
  2| . D . M . C . H |
  1| R R R R R R R R |
   +-----------------+
     a b c d e f g h

tags = elephant, blockade

[criteria]

condition = piece_position

after_piece_position = re8 re7 cd7 me6 | re8 re7 md7 ce6 : 0.8 |
                       re8 re7 md7 de6 : 0.85 | re8 cd7 me7 re6 : 0.7
```

When started (`python ats.py`), *ATS* reads from its configuration file (default is *ats.cfg*) information on: time per test, tests to use, how often repeat every test, engine

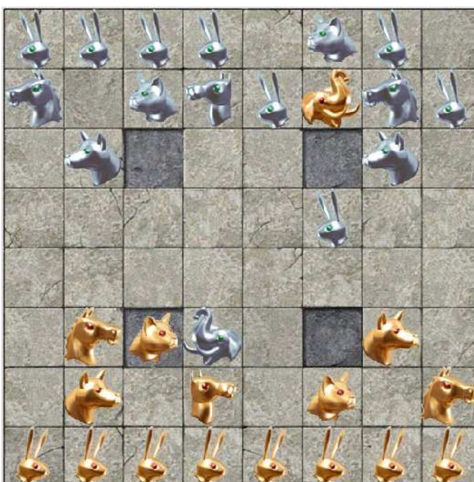


Figure A.2: ATS example position.

to connect, where to log. The engine is connected through AEI and selected tests are sequentially presented to it. This is accompanied with logging information. In the end, statistics on performance are given to the user. For more examples please see tests in **akimot/other/ats/tests** directory.
Example configuration file:

```
[global]
engine = akimot/akimot -c akimot/akimot.cfg
tests = 15
time_per_test = 5
cycles = 1
```

A.8 Gameroom

There is an online gameroom (created by Omar Syed) accessible to both human players and bots at *http://arimaa.com/arimaa/gameroom/*. Akimot engine uses AEI to connect to the gameroom. A bot's account must be created online in the gameroom first. Such a bot is then identified by its name and password for connection to the gameroom. There is a script called *gameroom.py* in **other/aei** directory for connecting the bot to the gameroom. Attributes of a session (bot's name, password for connection, engine to use, number of games to play, time settings, etc.) are specified in the configuration file called *gameroom.cfg* in the same directory.

A.9 Match environment

Besides connecting the engine to the online gameroom it is naturally possible to let it play against other engines offline. There are several freely downloadable engines at Arimaa homepage for this purpose - for instance *bot_sample* by Don Dailey or *bot_fairy* by Mika Ole Hansson. There is an AEI extension for offline match as well, however the mentioned engines don't support the AEI. For this reasons we sticked with using the *match environment* based on getMove interface also created by Omar Syed. The match environment is positioned in **other/match** directory. Every bot is supposed to have its own directory (e.g. **other/match/bot_akimot**). The alias names for bots (with possibly various command line arguments) are listed in configuration file *bots*. The script conducting the game is called *match*. A game between two bots is performed by issuing `match bot_1_alias bot_2_alias` where aliases must be defined in *bots* file.

For our purposes we have created a *tour.py* script to perform a tournament between two engines consisting of specified number of games. The *tour.py* script iteratively performs the match between specified bots by calling the *match* script itself. It makes the process of organizing the results more convenient. Configuration files for bots (if available), setup notice and record of every match played in the tournament are stored in the tournament's directory. Moreover result of the tournament is written to file *list.txt* on the same level as tournament's directory.
The syntax is following: `python tour.py bot_1_alias bot_2_alias [options]`
Where options are:

`--game_dir dirname` Tournament's directory (default **matches/some_number**).

`--games_num number` Number of games in the tournament (default 100).

`--comment comment` Comment on a tournament.

`--silent` Ignore the bots log output (otherwise written to appropriate log file in the final directory).

`--mode` Running mode - default is *standalone*, other options are *master, slave, finish_only*. This option served a very specific purpose during the development and is not recommended to use. For more details see below.

For effectively running the large number of games we have devised the *tour.py* script with ability to run simultaneously on (potentially) multiple machines (aka cluster). This was quite specific demand in the development process and is meant for interested and experienced users (won't work out-of-the-box). The principle is following. The code and configuration files that the games should be run with, must be present on the cluster machines. Moreover it is advisable if the cluster machines have a shared filesystem (i.e. AFS). On the controlling machine the *tour.py* script is invoked in *master* mode. The script connects to predefined computers (via ssh using the *psshlib.py* script based on the *pexpect* library) and starts its instances in *slave* mode, these run only a single game and finish. After which the controlling script starts the next instance. In the end final log entry to the *list.txt* is made by a single instance in *finish_only* mode. The *psshlib.py*

script is included in the package. Users interested in this feature should study this script and define their login credentials (if any) in there.

A.10 Simple Arimaa Development GUI

During the process of development we strongly missed a simple-to-use Arimaa game viewer. While it is possible to view the game records online through a Java applet this was quite slow and sometimes unreliable. For this reason we have developed a very simple GUI for Arimaa developers. The GUI is written in *Python* using *Qt4*. It's capable of:

- Loading a game record and replaying the game with possibility to jump back and forth in the record.
- Loading a position from the standard position format (see §A.5.1).
- Dumping a position to the standard position format.

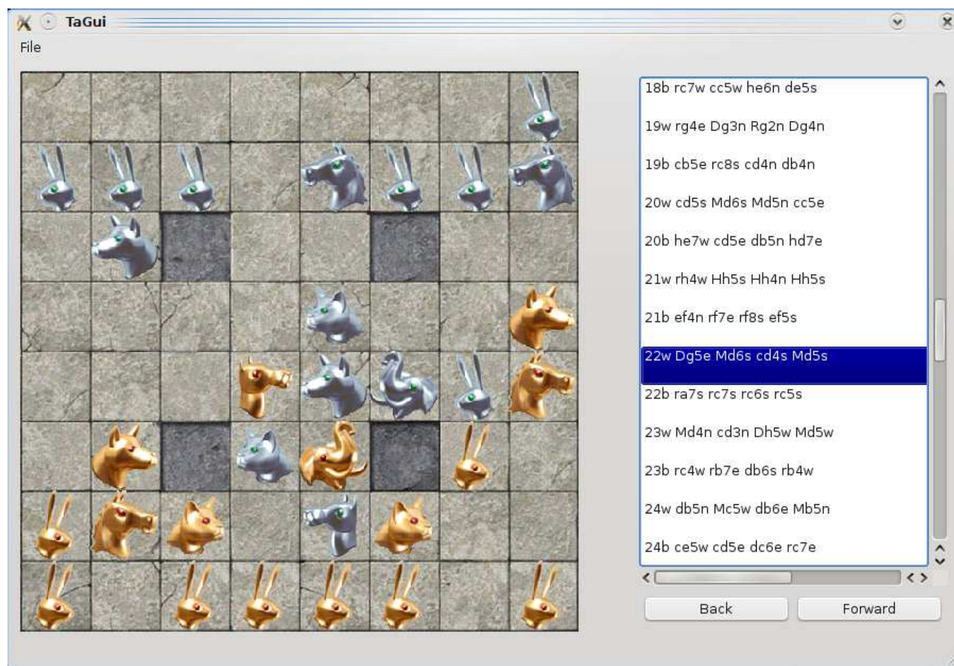


Figure A.3: The development GUI.

The picture of the GUI is given in the Figure A.3.

Appendix B

Glossary

For further information on terms regarding Arimaa strategy and tactis with example diagrams see [7].

atari
: Term from the game of Go. It is a move which threatens to capture some opponent's block of stones in the next move. This block thus has the last liberty after atari move.

blockade
: Situation when the piece cannot move at all (or can move very locally and is denied access to the other parts of the board) is called a blockade. Example situation is an elephant blockade. Even though many pieces must be dispatched to block the elephant it might be well worth it when blocking player manages to free its elephant from the blockade giving him the strongest mobile piece on the board.

eye
: Term from the game of Go. It is an empty intersection surrounded by the group of stones and securing life for this group if there is at least one more eye point in the group (opponent cannot play into the both eye points simultaneously).

frame
: A piece standing on the trap square unable to move away is said to be framed. Typical situation is that framed piece is surrounded from three sides by opponents pieces and from one side by friendly supporting piece.

goal
: A move that gets one of the player's rabbit on the last row of the board (from his point of view).

hostage
: Situation when a piece is frozen and threatened to be trapped. This typically happens in the sphere of influence of the owner of a freezing piece. The point is to force the opponent to dispatch some of his forces to protect the trap in question

and use the material advantage in the different part of the board. Typical hostage situations include: elephant taking camel hostage or camel taking horse hostage.

MCTS
: Monte Carlo Tree Search. General name for algorithms based on Monte Carlo simulations.

maturity threshold
: Number of simulations necessary for node expansion.

node expansion
: Expansion of a given node N (representing position P) means adding a child node N_i to N for every possible position P_i reachable from position P by some legal move. Node can be expanded during the MCTS simulation if it has been already visited given number of times (*maturity threshold*) and is not expanded yet.

pin
: The piece guarding the framed piece is pinned (it cannot be moved without losing the framed piece).

playout
: A (pseudo) random game from given position simulated by program (usually to the end of the game or to the point of meeting some criteria). The result of the playout is then backpropagated to the tree.

simulation
: One iteration of the MCTS algorithm. Consists of the descent through the tree followed by a playout and finished with a backpropagation of the result.

UCT
: Upper Confidence bound to Trees. Best-first search algorithm based on theory of multiarmed bandits. One of first representants of MCTS.