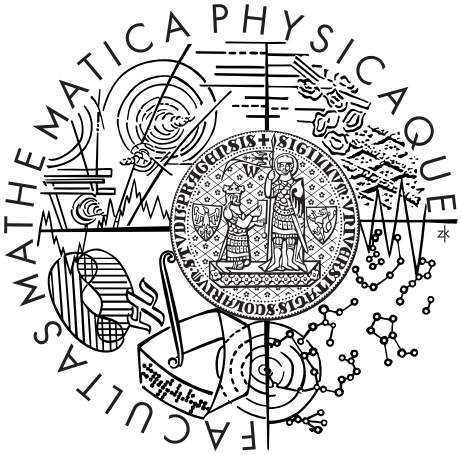
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Tomáš Hřebejk

Arimaa challenge - static evaluation function

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	Mgr. Vladan Majerech, Dr.
Study programme:	Computer Science
Specialization:	Theoretical Computer Science

Prague 2013

I would like to thank my supervisor, Mgr. Vladan Majerech, Dr., for his valuable expert advices and comments. I would also like to thank my family for their support and patience.

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

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

In ..... date ..... signature of the author

Název práce: Arimaa challenge - statická ohodnocovací funkce

Autor: Tomáš Hřebejk

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Vladan Majerech, Dr., Katedra teoretické informatiky a matematické logiky

Abstrakt: Arimaa je strategická desková hra pro dva hráče. Byla navržena tak, aby nebylo nebylo jednoduché vytvořit počítačový program, který by dokázal porazit nejlepší lidské hráče. V této práci jsme se zaměřili na návrh statické ohodnocovací funkce pro hru Arimaa. Úkolem ohodnocovací funkce je určit, který hráč má v dané pozici výhodu a jak je velká. Tento problém jsme rozdělili do několika částí, které jsme řešili samostatně. Nejvíce jsme se věnovali efektivnímu rozpoznávání významných vzorů na desce, například gólovým hrozbám. Základním prvkem navržené ohodnocovací funkce je mobilita. Pro každou figurku odhadneme počet kroků potřebný k tomu, aby se tato figurka přesunula na ostatní místa herní desky. Dále jsme se zabývali strojovým učením. Navrhli jsme nový algoritmus pro učení ohodnocovací funkce podle her expertů. Součástí práce je i implementace herního programu, který demonstruje navržené metody.

Klíčová slova: Arimaa, umělá inteligence, strojové učení, paralelizace

Title: Arimaa challenge - static evaluation function

Author: Tomáš Hřebejk

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Vladan Majerech, Dr., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Arimaa is a strategic board game for two players. It was designed with the aim that it will be hard to create a computer program that could defeat the best human players. In this thesis, we focus on the design of the static evaluation function for Arimaa. The purpose of a static evaluation function is to determine which player is leading in a given position and how significant the lead is. We have divided the problem into a few parts, which were solved separately. We paid most attention to the efficient recognition of important patterns on the board, such as goal threats. The basic element of the proposed evaluation function is mobility. For each piece, the number of steps that the piece would need to get to other places on the board is estimated. We also examined machine learning. We developed a new algorithm for learning a static evaluation function from expert games. An implementation of an Arimaa playing program, which demonstrates the proposed methods, is part of the thesis.

Keywords: Arimaa, artificial intelligence, machine learning, parallelization

Contents

1. Introduction

1.1. Arimaa challenge

In 1997, chess computer Deep Blue defeated world champion Garry Kasparov in a six-game match. It was a great success of artificial intelligence and a sign that computers will be able to overcome humans even in the disciplines that had been formerly considered to require "real intelligence".

In 2003, Omar Syed introduced a strategic board game called Arimaa [1]. It is not only an enjoyable game, but it is also a new challenge for artificial intelligence. Syed tried to create a game which is similar to chess, but which is resistant to AI techniques that were used in Deep Blue.

There are many possible positions from which a game may start and that makes it hard to create an opening book. A player has usually thousands of moves to choose from (about 17000 in average), so the branching factor of search algorithms is very high and the search cannot go very deep. Furthermore, the board is changing slowly, and it is much harder to capture a piece. This forces players to think more about strategic aspects, which is something that humans are good at, and lowers the importance of tactics.

In order to attract more people in the research of artificial intelligence, Syed offered reward $10000 for anyone who develops a program that is able to beat the best human players in Arimaa before the year 2020. Many Arimaa playing programs, which are commonly called bots, have been developed, but none of them has been able to win the Arimaa Challenge. Despite the growing strength of bots in the recent years, there is still a huge gap between computers and humans, and it is even not clear whether the gap is actually shrinking.

1.2. Arimaa rules and terminology

Arimaa is a game for two players - Gold and Silver. Both players have perfect information, and there is no randomness. The board consists of eight rows (1-8 from bottom to top) and eight columns (a-h from left to right). There are four special squares (c3, f3, c6 and f6) which are called traps. Each player has a set of pieces that initially consists of (in order from the strongest to the weakest):

1. elephant

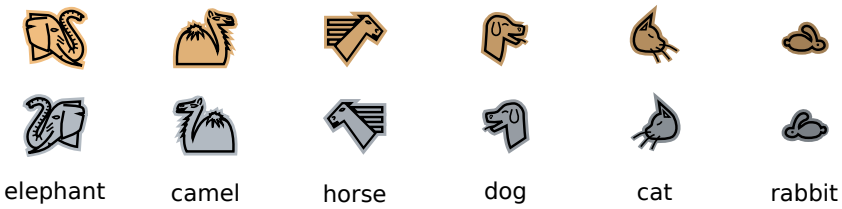


Figure 1.1.: Icons for Arimaa pieces created by Nathan Hwang.

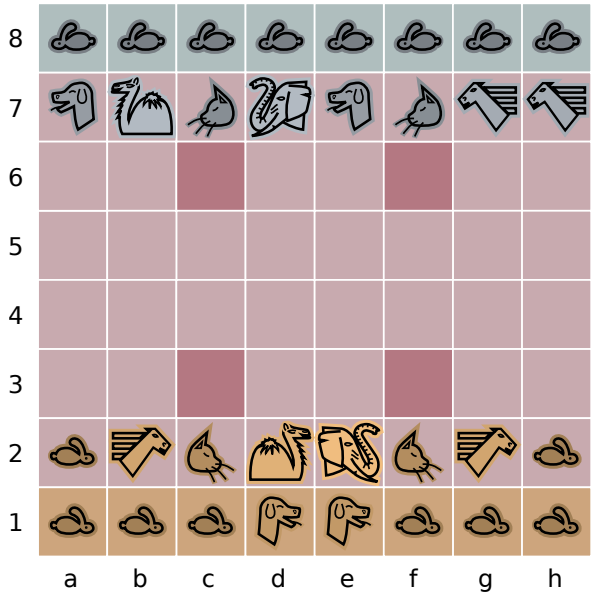


Figure 1.2.: An example of Arimaa position after the setup phase.

2. camel
3. horses (2)
4. dogs (2)
5. cats (2)
6. rabbits (8)

Figure 1.1 shows the icons that are used to represent pieces in this thesis. We use those icons, created by Nathan Hwang in 2011, instead of the original ones because the new icons seem more distinctive.

The game starts with a setup. As a first move, Gold places the gold pieces in any arrangement into the first two rows. Then Silver places the silver pieces in the last two rows. Figure 1.2 shows an example of an initial piece arrangement.

After the setup phase, players alternately move their pieces until a termination condition is satisfied. In one turn, a player can make four steps. Some steps may be passed, but the position has to be changed[1]. A step consists of moving a piece one square left, right, down or up. There are a few simple rules that say which steps can be performed.

If a piece has a stronger enemy next to it and there is no fellow (piece of the same color) around, the piece is prohibited from moving. In this case, we say that the piece is *frozen*. When a fellow is moved next to it, the frozen piece is unfrozen and can move.

There are 3 types of atomic moves, which we will call *full steps*:

1. Simple step
2. Pull
3. Push

A simple step means moving player's own piece from a square A to an adjacent square B. It can be done if:

1. The piece at A is not frozen.
2. The B square is empty.
3. The piece is not a rabbit, or B is not backwards (backwards means down for Gold and up for Silver) from A. (Rabbits must not go back.)

A pull consists of two steps. One step moves player's own piece from A to B, and the second step moves opponent's piece from C to A. Pull is possible if:

1. The piece at A is not frozen.
2. The B square is empty.
3. The piece at C is weaker than the piece at A.

A push also consists of two steps. First, opponent's piece is moved from B to C, and then player's piece is moved from A to B. The necessary conditions are similar as in the previous case:

1. The piece at A is not frozen.
2. The C square is empty.

[1] Swapping two pieces of the same type is not considered as a position change, and thus it is not a legal move.

3. The piece at B is weaker than the piece at A.

Both a pull and a push consist of two steps, so we will call them *double steps*. They are considered atomic, but they count as two steps with respect to the four steps limit of a move.

After each full step, the traps are checked. If there is a piece in a trap and it has no fellow next to it, then the piece is removed from the board. This is called a capture. Since captures happen after a complete full step, it is possible to pull an enemy even if the pulling piece moves into a trap where it is captured.

The main goal of a player is to get a player's rabbit on the other side of the board (row 8 for Gold, row 1 for Silver). This is the most common way of winning, but it is also possible to win by other means. All possible ways to win are:

1. A player wins if there is player's rabbit on the goal row.
2. A player wins if all opponent's rabbits are captured.
3. A player wins if it is opponent's turn and the opponent has no legal move.

These conditions are checked in the specified order after a turn and the first that is satisfied determines the winner. If the winning condition is met for both players, then the player who made the last move wins.

Finally, there is a rule that the player cannot make a move that would lead to a third repetition of the same position. This rule might seem artificial, but it guarantees (in theory) that the game will finish in a finite number of turns. A game might still last too long, so for practical reasons, there are usually some time limits for making a move and for the whole game.

1.3. Static evaluation function

Static evaluation function is a fundamental building block of any bot. Its purpose is to evaluate the winning chances of players. It is not necessary to give the exact probability of winning. What matters is the relative ordering of positions. We will use the convention that higher evaluation means higher winning chances for Gold.

If we had a very good static evaluation function, then it would be easy to implement a bot. It would be sufficient to evaluate all moves[2] and select the best. However, it is very hard to develop an evaluation function that would be so good that this simple algorithm would work well.

[2]By evaluating a move, we mean evaluating the position that is the result of the move. This simplification is used throughout the thesis. It should not lead to confusion, because we do not consider any other type of evaluation.

Instead of trying to make a perfect evaluation function, we can make a simple and fast evaluation function and use brute-force search to get a good bot. This is the approach that Deep Blue took. If the search is deep enough, then even a basic evaluation may be sufficient. Therefore, the main topic of research in game AI was how to make the search deeper. The most important result is the alpha-beta algorithm. In ideal case, it can reach the double depth in comparison with the straightforward minimax search. Many improvements to the alpha-beta algorithm were proposed, but that is not the subject of this thesis.

There are two extremes. The first is a sophisticated evaluation function with a simple search. The second is a simple evaluation function with deep search. We must find a good compromise between the quality of evaluation and the computational complexity to get a good combination of those approaches. Since the branching factor of the search is very high in Arimaa, we have decided to prefer a complex evaluation function over a simple evaluation function, even though the higher complexity may lead to a less effective search. An advantage of high-quality evaluation function is that poor moves might be recognized sooner and the search might be narrower. Consequently, it might be possible to search deeper even though it takes more time to evaluate a position. It is, unfortunately, very hard to develop a precise static evaluation function.

1.4. Thesis overview

The basic goal of this thesis is to boost the research of static evaluation functions for computer games. Static evaluation function is strongly dependent on a particular domain, and it is therefore hard to provide general results. We want to develop a good static evaluation function for Arimaa, but we take more attention to the aspects that might be useful also in other domains.

In Chapter 2, we describe a general concept of a static evaluation function. We divide the evaluation function into layers and we examine the opportunities for parallelization.

Very important part of the static evaluation function is recognition of interesting patterns. We study this topic deeply in Chapter 3. We focus mainly on the mobility of pieces, goal threats and capture threats.

There are many abstract strategic features in Arimaa, and they must be somehow incorporated into the static evaluation function. Chapter 4 discusses our solution to this problem.

Another goal that we had is to take advantage of machine learning. The ideal situation would be if the static evaluation function could be automatically learned without any human interaction, but that would be too ambitious. A more realistic

aim, which we have tried to achieve, is to develop a tool that makes it easier to make a static evaluation function. Our method is presented in Chapter 5.

We have also created a new bot, which we call bot_drake. Its main purpose is to demonstrate the abilities of the evaluation function that is presented in this thesis. Since the subject of this thesis was not a search algorithm, it uses only a simple alpha-beta search with a few basic extensions.

2. Evaluation function design

2.1. Programming environment

The first choice that we had to make when implementing the static evaluation function and other parts of the bot was which programming language to use. We have opted for C++ because it offers high-level constructs like classes, and at the same time it allows writing low-level code when we need to optimize important functions. In 2011, a new version of C++ standard (C++11), which brings many new features, was released. Some latest C++ compilers support it, so we have decided to use it. However, as a consequence, it is not possible to compile the code with an old compiler that does not support C++11. At the time of writing, the only two compilers that were able to compile the program were gcc (version 4.7 or newer) and clang (version 3.3 or newer). This led us to a decision that we will target only those two compilers. Consequently, we could use some extensions that are supported by both of them, because we are forced to use those compilers anyway.

Although the program was tested predominantly on a 64-bit Linux platform, we have tried it also on 64-bit Windows and it should be possible to compile it on any platform that is supported by gcc or clang. Build process is managed by CMake. For building bot_drake, the boost library is required. The tool for learning evaluation parameters has some additional dependencies. More information about building and using bot_drake is in Appendix B.

Documentation can be automatically generated with doxygen. All important functions and classes are briefly commented.

2.2. Decomposition

When we are facing a complex problem, it is usually a good idea to divide it into smaller subproblems. We have followed this philosophy, and thus the static evaluation function is composed of three layers:

1. Recognition of interesting patterns
2. Extraction of general features
3. Evaluation of the features

The first step in the position evaluation is to recognize important patterns. This may include capture threats, goal threats, blockades, hostages, etc. We have focused mainly on the analysis of piece movement. We want to determine where a piece can be moved in one turn. It is not hard to find these patterns, but it is hard to find them efficiently. Our solution is described in Chapter 3.

The output of the second layer is a set of parametrized features. In this context, a feature can be viewed as a function with arguments. It has a name and optionally some numeric parameters. For instance, we can have a feature indicating presence of a piece. That feature would probably have many parameters, including the color and the strength of a piece. One instance of this feature would be created for each piece on the board.

The purpose of the second layer is to generalize and to take into account complex relations. For example, if the first layer finds that a piece can be captured, it might be a serious threat, but it would not be a big problem if the trap could be easily protected. These two cases should be distinguished and the threat should be quantified. The second layer is based on human knowledge of Arimaa.

The final evaluation is the sum of values of features. Arbitrary function can be used to compute the value of a feature, thus it is possible to handle nonlinear relations between the parameters of a feature. However, dependence between two features cannot be captured. This limitation is necessary to make the evaluation tractable.

It is possible to automatically learn a good feature evaluation from a set of expert games. It is further explained in Section 5.

2.3. Parallelization

The static evaluation function has to be fast to compute, and at the same time it has to perform many nontrivial tasks. Both these requirements can be satisfied if we perform many operations in parallel. Current processors offer several ways to perform operations concurrently. We should analyze them and take advantage of them.

The simplest type of parallelism from programmer's perspective is instruction-level parallelism. Although programs are written as a serial stream of instructions, contemporary processors are usually able to automatically execute many instructions in parallel. That is possible only if several conditions are satisfied:

- It is clear what instructions will be executed next. If there are conditions (conditional jumps) in a code, then it is much harder to find opportunities for parallel execution.

- The instructions are independent. If the result of one instruction is an operand of a second instruction, then the second instruction has to wait for the first.
- The instructions are not restrained by the same resource. That is very platform dependent. For instance, it could be a problem if all instructions need to write to the memory.

A programmer does not usually need to care about these low-level details, that should be the job of an optimizing compiler. However, it is useful to know, for instance, that unpredictable conditions can make a code slow and it is generally better to avoid them. This advice may lead to a different choice of algorithms.

Second type of parallelism that we will discuss is bit-level parallelism. Processors work with numbers (words) of fixed length. The word length is commonly 32 bits or 64 bits. The key idea is that the word can be also interpreted as a vector of bits. One of the basic application is to use it to represent a subset of a small set. If the size of the set is not larger that the bit size of the machine word, then we can represent a set in one word and all basic set operations require only one machine operation. This data structure is called bitset. Especially 64-bit word is perfectly suitable for Arimaa because we can have one bit for each square of the board. We can represent an Arimaa position by a few bitsets. This representation is called a bitboard and it is widely used among, for example, chess and Arimaa engines.

In addition to scalar operations, modern processors often support vector operations. We may call them SIMD (single instruction multiple data) instructions because they concurrently perform the same operation on different data (elements of a vector). Processors with SIMD support have special registers that can hold vectors of numbers and special instructions that operate on these registers. The set of supported operations differs between different architectures, but all basic arithmetic operations are usually supported. A vector has a fixed bit-length. The number of elements in a vector depends on the size of a single element. A vector can contain integers of different sizes (1,2,4,8 bytes) or floating point numbers with single (32 bits) or double (64 bits) precision. If the vector size is 128 bits, then we can have, for instance, a vector of eight 16-bit integers or a vector of two double precision floating point numbers.

Nowadays, almost any processor supports some vector operations, even those processors that can be found in smartphones. The most important SIMD instruction sets are:

- SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2 - x86 architecture (Intel, AMD), 128-bit vectors

- AVX, AVX2 - x86 architecture, 256-bit vectors
- XOP - x86 architecture (AMD only), 128-bit vectors
- NEON - ARM architecture, 128-bit vectors
- AltiVec - PowerPC architecture (IBM), 128-bit vectors

There are two ways one can take advantage of SIMD instructions. The simplest one is to let a compiler vectorize the code automatically. Both gcc and clang can do this in some cases. The second option is to use vector operations explicitly. Unfortunately, C++ does not have any standard way to do this. One can use an assembler or intrinsic functions, which are special platform-dependent functions that correspond directly to machine instructions, but then it is hard to make the code portable across different architectures. There are some libraries which define vectors as classes with overloaded operators. They are easy to use, but we do not know of any such library that would support all important architectures and features. One example of a C++ library for SIMD programming is VC [2], which supports various instruction sets for x86 architecture, but unfortunately it does not support NEON and it does not define vectors of one-byte numbers, which we use a lot.

Another complication is that vector loads and stores must often be aligned, or it is at least more efficient. Aligned load of a 16-byte vector means that its address must be a multiple of 16. New C++11 standard brings some alignment support, but it is not fully implemented in gcc 4.7; therefore, we use a compiler specific extension instead. A lot of care must be taken to ensure that all data is properly aligned. It is even necessary to overload the operators new and delete for classes with over-alignment.

Although there is no standard way to use vectors in C++, gcc offers "vector extensions". It provides an intuitive definition of vector types and vector arithmetic operations. A big advantage of using this feature is that it is platform independent. The vector code is compiled differently according to the target architecture. That is exactly the way we want program. It is a step back to target single architecture or to write the code many times separately for each architecture. The main disadvantage is that it is not supported by all compilers. Since it is supported by both gcc and clang, which are the compilers we have chosen to use anyway, it is not a serious obstacle. We have chosen to use this extension, but we still need to use some architecture dependent intrinsic functions because there are many specialized instructions that cannot be used in any other way. The program is optimized primarily for the AVX instruction set. However, it can

be compiled for any architecture. If a vector instruction is missing, it is emulated with a scalar code.

The last type of parallelism, which we will consider, is multithreading. Processors usually have more than one core, thus they can run more threads of execution in parallel. We can use this parallelism best if we can split the computation into many independent tasks. The tasks should be big enough so that the overhead of running them concurrently is low. Therefore, it is better to make thread parallelization in a search algorithm, not in the static evaluation function.

3. Analysis of piece movement

3.1. Problem statement

One specific aspect of Arimaa is that a position does not change much after playing a move. A piece can move at most four squares from its original location. When it is frozen or blocked, it might not be able to move at all. So one of the basic questions that can be asked about a game position is whether it is possible to move a piece from one square to another provided that the player of its color is on the move. This information can be useful in various ways.

First of all, it is very important to know if a rabbit can reach the goal row. If the player on the move can move a rabbit to the goal row, then he can win. It is also important for the second player. A rabbit that can reach the goal row is a serious threat that may force the opponent to make some defensive steps, and it may eventually lead to a win in subsequent moves.

It is also important to know where strong pieces can get. For instance, the question whether an elephant can be moved next to a trap may determine if a piece can be saved from being captured. The squares around traps are generally crucial. Both players strive to occupy them with their strong pieces so that they can protect their pieces from being captured and threat to capture enemies. For this reason, it may be good to know if a piece can take control of such a square.

The last example involves a camel. It is the second strongest piece, and thus it only fears the elephant. If opponent's elephant cannot get next to it, then the camel is safe. Once again, the same question is asked: "Can that piece (opponent's elephant) be moved to this square (next to the camel)?".

Although the question may seem clear, it needs some clarification. An important point is that we may need to move other pieces in order to get a piece from one square to another. The piece may have to be unfrozen, we may need to make way for it, an enemy may need to be pulled, etc. This makes the question more challenging.

Finally, we need to define what it means to move a piece into a trap. In Arimaa, one can capture his own piece. It is legal to move a piece into a trap that is not protected by any other piece of the same color, but the piece is captured. Now we have to ask ourselves: "Is it useful to know whether a piece can move into a trap when we cannot say whether it will be captured?" In our opinion, it is not.

For example, we would not be able to say if a piece can be frozen when a stronger enemy can get to a trap that is next to the piece. Therefore, when we say that a piece can move into a trap, we imply that it will not be captured.

The precise definition of our problem is: *For each piece and for each square, find out if the player whose the piece is has a move that would leave the piece on the square.* For the player that is not on the move, we proceed as if the player was on the move.

That is the basic problem, but we might want to know a little more. If we know that there is a move that will move a piece somewhere, we might also wonder how many steps are needed. For $n \in \{1, 2, 3, 4\}$ the problem becomes: *For each piece and for each square, find out if the player whose the piece is has a move consisting of at most n steps that would leave the piece on the square.*

The player may also move with opponent's pieces. Push and pull moves are important because they are required to make captures. That gives us another problem: *For each piece and for each square, find out if the opponent of the piece has a move consisting of at most n steps that would move the piece to the square.* In this case, we do not imply that the moved piece will not be captured. In fact, the most important case is when the piece is captured. It is so important, that we also want the answer to the question: *Can a given piece be captured in n steps?*

The rest of this chapter will explore these problems. We start with a theoretical investigation of moves and steps in Arimaa. Then we describe the method that we applied and the actual implementation.

3.2. Moves in Arimaa

3.2.1. Move representation

In this section, we deeply discuss moves in Arimaa. They have some special properties, that can help us solve our problems more efficiently. First, we try to define precisely what a move is. Then (in Section 3.2.2) we analyze dependence between steps.

Although the question what a move is was already answered in Section 1.2, there are still some details that have not been covered. For instance, we would like to say that two moves in two different positions are the same. But what does it exactly mean? There is no single true answer. This question is strongly connected with a move representation. A natural way to define move equivalence is to say that two moves are equivalent if they have the same representation. However, there are many ways to represent moves, and each of them has some advantages and disadvantages.

In the official Arimaa move notation, each step is described by a moving piece,

an original square and a direction. The notation also includes information about captures. But should two moves be considered different only because in one case a piece was captured and in another case it was not? And is it really different if we move piece left and up or if we move it first up and then left? In both cases the effect is the same. Moreover, this representation is redundant. For instance, if we move an elephant, then it is not necessary to specify both the original square and the piece because there is only one elephant for a player and only one piece on a square.

If we try to reduce the official notation to a necessary minimum, then we do not have to include the information about captures and we do not need to say which piece is moving. That information can be easily reconstructed provided that we know the position from which the move was played. For each step, we only need to know a square and a direction. This representation requires only a single byte. If we use one special value for the null step, then we can store a whole move in four bytes. We will call this representation compact because of its memory-efficiency. We use this representation to store a list of all possible moves.

Previous representation does not distinguish simple steps and double steps. Sometimes, it is not even possible to say whether a step is moving a player's piece or an opponent's piece without knowing the position. Double steps should be considered atomic, so why they are stored as two entities? This flaw leads us to the idea of representing move as a sequence of full steps. An advantage of this approach is that if we omit the last full step of a valid move, then we always get a valid move (or a null move). That can make thinking about moves a little easier. A disadvantage is that it is more complex, and it requires more memory if it is stored directly as an array of full steps. Move generation in bot_drake is based on this representation.

All previous representations have the same problem, which has been already pointed out at the beginning. It is possible to have two moves with different representations that have the same effect. It is quite usual that a piece can be moved from one square to another along two or more different paths and it does not matter which path is chosen. We do not want to generate all possibilities, because that would be a waste of time. To resolve this problem, we may introduce another move representation, which is based on the effects of a move. For each piece that is involved in a move we keep the square where it was moved or a note that it was captured. A drawback is that it is not trivial to convert move in this notation to a move in the official Arimaa notation. Furthermore, it is trickier to test if a given move (e.g. killer move[1]) is valid. This representation is not directly

[1] That is a move that was good in other branch of search and we hope that it might be good also in the current position.

used in bot_drake, but we take care not to generate more variants of the same move.

3.2.2. Dependence between steps

In Arimaa, a move often consists of more independent parts. However, in real games, there are usually some dependent steps in a move. In his work [3], Zhong defines dependent steps ("step combo") by the condition: "If any 2 steps are swapped, we get a different board position or an illegal move." In this section, we give another definition, that is more formal and easier to work with. We also show applications of this concept to the problem that is described in Section 3.1.

When considering step dependence, we will always talk about full steps, even if it is not explicitly specified. This simplifies the problem a little because we do not need to deal with the dependence between two steps of a double step. A full step can be represented by a type and a square. The square is the location of a piece that belongs to the moving player. The type can have one of 28 values (4 simple steps, 12 pushes, 12 pulls) that specify in which direction the piece moves and if (and how) an enemy is moved. We do not include information about the type of the piece that is being moved, because that would significantly increase the number of variants of full steps. We want the set of all full steps to be small because then any subset of full steps can be efficiently represented with a bitset. Some combinations of step types and squares are invalid, but that is not a big issue.

When we are given a full step and a board, we may ask two questions. Can we make the step? And what are the effects of its execution? The interesting fact is that we do not need to know the whole board to answer both these questions. We define the *read set* of a full step as the set of squares that we might need to check to say if the full step can be made and what are its effects. The read set consists of:

- Original square. (We need to verify that there is a piece of the moving player.)
- All adjacent squares of the original square. (We need to verify that the piece is not frozen and that the destination square is empty.)
- If the full step is a push, then the square where the enemy is pushed. (It must be empty.)
- A trap square and all adjacent squares if we are moving a piece that was next to a trap. (We need to check if a piece was captured.)

Next, we define the *write set* of a full step as the set of squares that might be affected by the step. The write set consists of:

- Original square.
- Destination square.
- One additional square for double steps.
- A trap square if we are moving a piece that was next to a trap.

Finally, we can define step dependence. *We say that a full step A and a full step B are dependent if and only if the intersection of the read set of A and the write set of B is nonempty or the intersection of the write set of A and the read set of B is nonempty.*

The step dependence is a fixed property of steps that does not depend on a position. It follows from the definition that if two steps are independent (i.e. they are not dependent), then it does not matter in which order they are made. We can define step combo as a sequence of steps in which every two steps are connected through a chain of step dependence. Every move can be unambiguously dissected into one to four step combos. One can view this as connected components of a graph, where the vertices are full steps of the move, and the edges correspond to step dependence.

How can we apply this theory to the problem of determining if a piece can be moved to a selected square? All steps that move the piece must be in one step combo. We can omit all other step combos and the move will still be correct. This means, that if we want to solve the problem by generating all moves, then we only need to consider moves that consist of a single step combo. This could save us some computations, if we implement it efficiently.

3.2.3. Generating all step combos

The move generation in Arimaa game engines is usually based on backtracking. First, we find all possible steps in a position, then we take the first step and execute it. We continue recursively until we get to the depth of four steps. When we return, we select next step until all steps are tried.

The generation of step combos is similar, but we can reduce the number of steps that we have to try. It would be fine if we could just say that any step except the first must be dependent on a previous step, but that would miss some cases. For instance, a three-steps combo can have first two steps independent, and the third step may be dependent on both previous steps. To fix it, we need to introduce a supporting step. A supporting step is a simple step that is not dependent on

a previous step. It is connected to the combo by another step that is dependent both on a previous step and on the supporting step. This means that we have to try supporting steps only when we have spent one or two steps. It is not necessary to try all possible steps, we need to check only indirectly dependent steps. Steps A and B are indirectly dependent if and only if they are not dependent and there is a step that is dependent both on A and on B.

The implementation is a little tricky. We must be careful not to omit any cases, but on the other hand, we should not do unnecessary work. For instance, if we have generated a move ABC, where A and B are independent and C is dependent on both A and B, then it is not necessary to generate the move BAC, because it is the same move. Furthermore, after making the full step A, we may take advantage of independence of B and we do not have to check that B is correct. We only need to check that B was a correct step at the beginning of the search, and that is something we have already computed.

Although this approach can save a large part of computations, it is more complex and it is not obvious that it is really more efficient. We use bitsets to represent a set of (full) steps, and there is a function that computes the set of correct steps for a position using bitboards. Sets of dependent steps are stored in a precomputed table. This can be very fast or very slow depending on whether this table fits in a processor's cache (and is not forced out by other data).

We can make any move as a combination of some step combos. It is easy to check if two step combos are independent. All we need is to union read sets and write sets of all steps in a combo, then we can test the combo independence like the step independence. This means that this approach can be also used to generate all moves. It is actually used this way in bot_drake. Although it might not be faster, it enables some new opportunities. For instance, we could select only some good step combos and compose moves only of them.

3.3. Using bitboards

In Section 2.3, we have introduced bitsets. Our implementation of this concept is the `Bitset8x8` class. Instances of this class can be seen as 8×8 matrices of Boolean values. However, they are internally represented as a single 64-bit integer. There are many operations that can be done with bitsets:

- Element-wise logical operations: AND, OR, NOT, XOR (`&`, `|` , `~`, `^`)
- Assignment and shorthands for an operation and assignment (`=`, `&=`, `|=`, `^=`)
- Comparison of whole bitsets (`==`, `!=`)

- Check if a bitset is zero
- Read or change a single value (`get(index)`, `set(index, value)`, `setTrue(index)`, `setFalse(index)`, `toggle(index)`)
- Get the index of the first element that is set to true (`getFirst()`, `extractFirst()`)
- Count the number of true values (`popCount()`)
- Shift the elements horizontally and vertically (`shift(right, up)`, `shiftNotMasked(right, up)`)

Although it represents 2D structure, there is always a single index to an element as if it was a vector because of efficiency. The convention we use is that values are stored from left to right and from bottom to top. That means that the zero index is for the element in the left bottom corner, then the second value in the bottom row follows, and so on.

Logic operations correspond to the logic operations on the internal integers. The same is true for the equivalence. To access a single value, we can use either a few basic arithmetic operations or, if it is supported by the hardware and the compiler, a single machine instruction. The methods `getFirst` and `popCount` are harder to implement (efficiently) using only standard arithmetic operations, but some processors offer specialized instruction that can do these computations. The compilers we use (gcc, clang) have a built-in function for `popCount` that should be compiled to the best possible code for the target platform.

The `shift` method moves all values in a matrix by a given vector. Some values are discarded and some values are not defined after the shift. For instance, if the shift is two columns left and one row up, then the two leftmost columns and the top row are discarded and the two rightmost columns and the bottom row are undefined. The difference between the `shift` method and `shiftNotMasked` is that the `shift` method sets the undefined values to zero whereas the `shiftNotMasked` method leaves them undefined. The implementation of these methods is based on a simple integer shift. The `shift` method requires one additional AND operation with a mask that clears the undefined values.

Working with `Bitset8x8` instances can be seen as a kind of SIMD parallelism. We can imagine that we have a computer with 64 processors that are arranged in a 8×8 grid. Each processor has its own registers for Boolean values. Some of these registers contain input data. All processors always execute the same instruction. An instruction is either a logic operation on registers or access to a register of another processor. The other processor is given by a relative 2D

vector, that is the same for all processors and it is encoded in the instruction. For instance, all processors may look to their left neighbor's register. If there is no such neighbor (processors in the leftmost column), then the result is zero. In this analogy, an instance of `Bitset8x8` is a register, logical operations on bitsets are logical operations on registers, and the shift method means a communication between processors. Other methods do not fit this analogy well, because they operate with the whole bitset, but they are seldom used. This idea may help us better understand bitset operations because we can focus on a single value instead of the whole structure.

For instance, we have an expression:

```
Bitset8x8 empty = ~(gold | silver)
```

There are two input bitsets - `gold` is a `Bitset8x8` of gold pieces and `silver` is a `Bitset8x8` of silver pieces. We compute a third `Bitset8x8`, which says whether a square is empty. In a "SIMD terminology", we can say that each processor has the input in two Boolean registers. A processor is assigned a corresponding square of the board. The first register, called `gold`, is true if and only if there is a gold piece, and the second register, called `silver`, is true if and only if there is a silver piece. All processors concurrently compute the Boolean expression and store the result in the third register called `empty`.

We have already mentioned that an Arimaa board can be represented by several bitsets and that this is called bitboard representation. One way we can do this is to have a `Bitset8x8` for each piece type, which says whether a piece of that type is on a given square. It is sufficient, but we may add another two bitsets - one for Gold and one for Silver. They say whether a piece of a given color is on a given square. These bitsets are so often used that it may be a good idea to add them even though they are redundant. Actually, when we have these bitsets for colors, then we do not need to have a bitset for each piece type, but only for each piece strength. It is not very important which variant is chosen because each one can be easily transformed to any other. In bot_drake, we use the variant with $12 + 2$ bitsets.

Bitsets can be used to find out, for instance, which steps are legal. We will consider the case of simple steps, but it is easy to extend it to double steps. The conditions necessary for a step to be legal are listed in Section 1.2. All possible simple steps are represented by four bitsets, one for each direction. Listing 3.1 shows how to compute these bitsets.

Most conditions are easy to test. The only hard part is to say whether a piece is frozen. For each direction, we check if there is a stronger enemy in that direction. In order to compute that, we need to distinguish types of pieces. We start with

```
//Input: Bitboard bb - bitsets piece[12] and player[2]
//      int moving - number of the moving player
//      int enemy - number of the opposing player
//
//Output: steps[4] - valid steps
//

for(int i = 0; i < 4; i++)
    steps[i] = bb.players[moving]; //moving player?
steps[moving] &= ~bb.pieces[moving]; //rabbit can't go back
Bitset8x8 empty = ~(bb.players[0] | bb.players[1]);
Bitset8x8 destinationEmpty[4];
steps[0] &= empty.shift(0,1);
steps[1] &= empty.shift(0,-1);
steps[2] &= empty.shift(-1,0);
steps[3] &= empty.shift(1,0);
Bitset8x8 weaker[4] = { 0, 0, 0, 0 };
Bitset8x8 strongerEnemy[4] = { 0, 0, 0, 0 };
for(int p = 8; p >= 0; p -= 2)
{
    strongerEnemy[0] |= bb.pieces[p + 2 + enemy].shift(0,1);
    strongerEnemy[1] |= bb.pieces[p + 2 + enemy].shift(0,-1);
    strongerEnemy[2] |= bb.pieces[p + 2 + enemy].shift(-1,0);
    strongerEnemy[3] |= bb.pieces[p + 2 + enemy].shift(1,0);
    for(int i = 0; i < 4; i++)
        weaker[i] |= bb.pieces[p + moving] & strongerEnemy[i];
};
Bitset8x8 hasSupport = bb.players[moving].shift(0,1);
hasSupport |= bb.players[moving].shift(0,-1);
hasSupport |= bb.players[moving].shift(1,0);
hasSupport |= bb.players[moving].shift(-1,0);
Bitset8x8 dominated = weaker[0] | weaker[1] | weaker[2] |
       weaker[3];
Bitset8x8 notFrozen = hasSupport | (~dominated);
for(int i = 0; i < 4; i++)
    steps[i] &= notFrozen;
```

Listing 3.1: Computing valid simple steps using bitboards.

a camel and we check if there is the opposing elephant. Then we continue with horses, and we check if there is the opposing camel or the opposing elephant. We do the same for horses, dogs, cats and finally for rabbits. We incrementally update bitsets that say whether there is an enemy piece stronger than X, where X goes from camel to rabbit. When we know if there is a stronger enemy next to a piece, we only need to check if there is a friendly piece next to it. The rest of the computation is straightforward.

3.4. Quadbitset

We have decided to use bitsets extensively because operations with bitsets are very efficient on current processors. However, there is still space for improvement. First, we need to introduce better syntax for writing bitset operations. The code must be as brief and lucid as possible. At the same time, we cannot afford any performance penalty. Furthermore, we would like to take advantage of vector operations to make it even faster. To meet these goals we have developed a few helper classes.

The basic class for working with multiple bitsets at once is `Quadbitset`. One can look at `Quadbitset` as if it was an array of four bitsets. All elements in `Quadbitset` usually have the same meaning except that each element represents a different direction. For instance, the meaning of bitsets in a Quadbitset might be:

1. Piece can make step backwards.
2. Piece can make step ahead.
3. Piece can make step right.
4. Piece can make step left.

It is an important structure because Arimaa is very symmetric, and almost any interesting pattern can appear in all four directions. An example is Listing 3.1, where many operations must be repeated four times. The standard meaning of elements of `Quadbitset` is (the more general meaning is given in parentheses):

1. Down / Backwards (main direction)
2. Up / Ahead (opposite direction)
3. Right (rotated 90 degrees anticlockwise)
4. Left (rotated 90 degrees clockwise)

original	rotateClockwise()	rotateAnticlockwise()	mirror()
Down (1)	Left (4)	Right (3)	Up (2)
Up (2)	Right (3)	Left (4)	Down (1)
Right (3)	Down (1)	Up (2)	Left (4)
Left (4)	Up (2)	Down (1)	Right (3)

Table 3.1.: `Quadbitset` permutations

We may perform logical operations on `Quadbitset` instances like with bitsets. `Quadbitset` can be also constructed from four bitsets or from a single bitset. In the second case, the bitset is copied four times. Elements may be accessed for reading as if it was an array, but one must use the set method to change a single bitset because it may not be internally implemented as an array of bitsets. There are several other methods:

- Various methods for combining all four bitsets into one output bitset (methods: `all()`, `any()`, `atLeastTwo()`, ...)
- Permutations (methods: `rotateClockwise()`, `rotateAnticlockwise()`, `mirror()`)
- `Quadbitset` comparison, assignment, copy

The first kind of methods outputs a single bitset which is somehow determined from all four bitsets. The `all()` method returns the conjunction of all bitsets and the `any()` method returns the disjunction. For instance, if the `Quadbitset` says which pieces can make step in a given direction, then we can use the `any()` method to find pieces which can make a step.

Permutations permute elements of `Quadbitset` in a way that is suggested by their name. For instance, if the first element means down, then after performing `rotateClockwise()`, the first element will correspond to the direction left. The order of the elements after a permutation is described in Table 3.1.

Big advantage of `Quadbitset` is that it can be implemented with SIMD instructions. We can use either two 128-bit vectors or one 256-bit vector, if it is supported by the processor. We have focused on the 128-bit version because processors with 256-bit vectors are not yet widespread. Although it is not possible to take advantage of 256-bit vectors now, it will be easy to add this feature later - we will only need to change the implementation of `Quadbitset` and a few other helper functions. Using SIMD is optional. If it is not available, it will be replaced with a scalar code.

One may wonder why the order of elements is as it is. Why are the directions not listed in the clockwise order? The reason is, that it would make the permutations less efficient when it is implemented with 128-bit vectors. The swap of

whole vectors is usually free because the compiler can optimize it out at compile time. Instead of swapping the values of two registers, the compiler can only swap their names in following instructions.[2] The swap of the lower 64 bits and the upper 64 bits of a 128-bit vector requires an explicit operation, but there is an instruction that does that (at least on x86 architecture [4]). Both 90-degree rotations can be implemented with a vector swap (that is free) and one swap inside the vector, which costs one instruction. Mirror permutation requires two instructions. If the elements were ordered "clockwise", then it would be necessary to move numbers between 128-bit vectors in both rotations, which would probably cost four instructions. Mirror permutation would be free, but that would not help us much. Permutations are particularly useful in the implementation of the `Neighbourhood` class, which will be described in the next section.

The last missing feature is an instrument to make shifts on `Quadbitset`. The solution is the `lookAt<x,y>` function. The argument can be either `Bitset8x8` or `Quadbitset`. The result is `Quadbitset` in both cases. If the argument is a single bitset, then it is easy to understand what it does when we use the model with 64 processors. The first bitset is a value (the argument) of the processor with relative location $[x, y]$. For other bitsets in `Quadbitset`, the vector $[x, y]$ is rotated to $[-x, -y]$, $[-y, x]$ and $[y, -x]$. The simplest case is when $x = 0$ and $y = 1$, then `lookAt<0,1>(bs)` gives values of `bs` for all adjacent squares. If the argument is `Quadbitset`, then the shift is the same, but each output bitset is made from different input bitset.

For instance, if `empty` is a bitset that says whether a square is empty, then the result of `lookAt<0,1>(empty)` is a `Quadbitset` with elements that have the following meaning:

1. The square one step backwards from the piece is empty.
2. The square one step ahead from the piece is empty.
3. The square one step right from the piece is empty.
4. The square one step left from the piece is empty.

3.5. Neighbourhood

One of basic features that every programming language offers is an array. It is very useful to group variables because then we can, for instance, apply a function to all elements with a few commands. The `Neighbourhood` template class is

[2] This is what we expect the compiler should do, but we cannot guarantee that it is actually done.

N	Quadbitsets	bitsets
1	1	4
2	3	12
3	6	24
4	10	40

Table 3.2.: `Neighbourhood` size

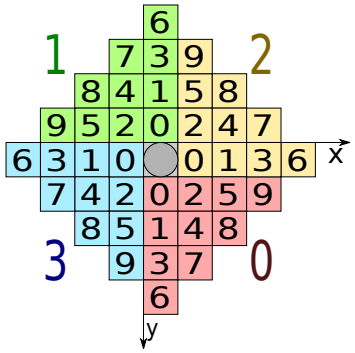


Figure 3.1.: `Neighbourhood` layout

basically an array of `Quadbitset` instances. In addition, it defines many methods and operators that make it possible to write short and intuitive code for complex operations. The template parameter `N`, which can take integer values from one to four, determines the number of elements (`Quadbitset`), but it is not exactly the count. There are three main purposes of this class:

1. Grouping variables with a similar meaning.
2. Transparent `Quadbitset` permutations.
3. Performing operations with whole arrays.

The first point is connected with the semantics of `Neighbourhood`. In Section 3.3, we have described bitsets with the model of 64 processors. Each processor is exploring its neighborhood, and it usually needs to keep the same information about all surrounding cells (squares). For instance, it probably needs to know which squares are empty. For adjacent squares, we could use one `Quadbitset`, but that is not sufficient if we want to include further squares. `Neighbourhood` with size `N` contains a bitset for all squares in distance at most `N`. An index of a bitset is a two-dimensional vector - the relative coordinates of the square which is associated with the bitset. The distance is computed as the sum of absolute values of coordinates (Manhattan distance). The number of bitsets for different `Neighbourhood` sizes is summarized in Table 3.2. Figure 3.1 shows the order in which elements of `Neighbourhood<4>` are stored in the memory.

`Quadbitset` permutations may look a little cryptic, but they are sometimes necessary. `Neighbourhood` does permutations automatically when they are needed. We do not ever need to use them explicitly. All work is done in `get<x,y>()` method. This method returns the `Quadbitset` at coordinates $[x, y]$. To be more precise, the first bitset of the returned `Quadbitset` corresponds to the given coordinates. Other bitsets in the returned `Quadbitset` correspond to rotated coordinates according to the semantics of `Quadbitset`. We can write the result symbolically as $([x, y], [-x, -y], [-y, x], [y, -x])$. The type of the required permutation is determined by the quadrant in which $[x, y]$ lies. In the main quadrant ($x \geq 0$ and $y > 0$), no permutation is done. In adjacent quadrants, one of the rotation is needed. In the opposite quadrant, the `mirror()` permutation must be done. The `get<x,y>()` method may be used for storing. In that case, the permutations are also applied automatically. The coordinates are template parameters, so they must be constants. This might look as a limitation, but it is much more efficient, and it is usually not a problem. In fact, there is one simple function for each coordinate. Therefore, it is possible to compute the required permutation and the index of `Quadbitset` at the compile time. We expect that the function call will be inlined by the compiler. That means that the function call is replaced with the function body, which consists of a few simple instructions. There should be no runtime penalty for using all these abstractions.

The last important purpose of `Neighbourhood` is to make the code shorter. One can write expressions that operate with the whole `Neighbourhood` instances. The idea is the same as in the case of `Quadbitset` - the same operation is executed element-wise. There are many possible operations:

1. Logical (AND, OR, NOT, XOR)
2. Slice (cast to `Neighbourhood` with smaller size)
3. Broadcast `Quadbitset` (convert `Quadbitset` to `Neighbourhood`)
4. `lookAround<N>` (construct `Neighbourhood` with the `lookAt` function)
5. Rearrange elements (`moveNbrBackwards`, `moveNbrRight`,...)

The first three types of operations do not need any comment. The `lookAround<N>` function is probably the most interesting. Suppose that bitset `empty` says which squares are empty. Then `lookAround<4>(empty)` creates `Neighbourhood` with size 4 that says which surrounding squares are empty. The argument can be single bitset or `Quadbitset` like in the `lookAt` function. It is so commonly used that there is one additional variant (static method `lookAround2`), which does two calls at once in a more efficient way.

The fifth type of operations is used less frequently, but it is still very useful. It is similar to `Quadbitset`'s permutations.

Although the idea of `Neighbourhood` expressions is similar to `Quadbitset` expressions, the implementation is different. `Quadbitset` have overloaded operators that directly compute the result. On the contrary, expressions that evolve `Neighbourhood` use technique called expression templates [5]. The result of such expression is a template object that encodes the whole expression. The expression is evaluated when it is assigned to a variable. The advantage of this technique is that no temporary results are needed, and therefore it is more efficient. We will use the following expression as an example:

```
Neighbourhood<4> nr = (n1 & n2) | (n3 & n4);
```

All variables are `Neighbourhood` objects with $N = 4$. If expressions were immediately evaluated, then two (or even three[3]) temporary `Neighbourhood` objects would be created and there would be three (or four) loops. With expression templates, there is just one loop that iterates over `Quadbitsets` and computes this expression. In theory, the compiler could do this optimization without our help, but in practice it is not so clever. The disadvantage of expression templates is that, as the name suggest, it heavily uses templates. The code is less clear, and compiler's error messages may be hard to understand. The whole expression is encoded in the name of a class, which means that there are many classes with really long names. Nevertheless, the performance is our priority, and therefore expression templates are the best choice.

Because `Neighbourhood` instances are used frequently, we use shorthands like `Nbr4` for `Neighbourhood<4>`, or `Nbr1` for `Neighbourhood<1>`.

3.6. Mobility analysis

We have used `Quadbitset` and `Neighbourhood` classes to write the function that gets a position and outputs information about mobility of pieces. The output is the `AnalyzerResult` structure, which contains several `Neighbourhood` objects and a few bitsets.

- `Nbr1 oneStepReach` - Squares the piece can reach in one step
- `Nbr2 twoStepsReach` - Squares the piece can reach in two steps.
- `Nbr3 threeStepsReach` - Squares the piece can reach in three steps.

[3] That depends on whether the compiler is smart enough to optimize assignment out and store the result of the OR operation directly into nr variable.

- `Nbr4 fourStepsReach` - Squares the piece can reach in four steps (one turn).
- `Nbr1 twoStepsMoved` - Squares where the piece can be moved by the opponent in two steps.
- `Nbr1 threeStepsMoved` - Squares where the piece can be moved by the opponent in three steps.
- `Nbr2 fourStepsMoved` - Squares where the piece can be moved in four steps.
- `Bitset8x8 twoStepsCaptured` - Can the piece be captured in two steps?
- `Bitset8x8 threeStepsCaptured` - Can the piece be captured in three steps?
- `Bitset8x8 fourStepsCaptured` - Can the piece be captured in one turn?

Both players are treated the same way. It does not matter who is actually on the move in this function. We act as if the player who owns the piece is on the move for the first four items and as if the opponent was on the move for the remaining items. The important point is that, in most cases, we do not need to compute the same thing twice for both players. The idea is that we write the code from the perspective of a single piece. Pieces are not distinguished as gold and silver, but as "fellows" (pieces with the same color) and "enemies" (opposing pieces). The piece has information about its neighborhood in several `Neighbourhood` objects, which are computed in the beginning of the function, for instance:

- `Nbr4 nEnemy` - "That square is occupied by an enemy piece."
- `Nbr4 nFellow` - "That square is occupied by a piece of the same color."
- `Nbr4 nEmpty` - "That square is not occupied by any piece."
- `Nbr4 nTrap` - "That square is a trap."
- `Nbr4 nWeakerEnemy` - "That square is occupied by a weaker enemy piece."
- `Nbr4 nStrongerEnemy` - "That square is occupied by a stronger enemy piece."

These `Neighbourhood` objects are then used in rules, which usually look like `A |= B`, where `B` is a `Quadbitset` expression and `A` is a part of the result or a temporary variable. The semantics of this expression was already described at

the level of `Quadbitset` (OR and assignment), but, on the higher level, it is an implication $B \Rightarrow A$. Although the code is written in procedural language, it looks as if it was written in a logic programming language. However, we must keep in mind that the code executes sequentially, and thus a variable can have different value in different parts of the code. In some rare cases, we even use commands in the form `A &= B`, which do not fit the idea of logic programming well, but they make the code shorter.

The function has about 6500 lines of code. Therefore, it is not possible to entirely describe it in this paper. Only some interesting examples will be given. We will start with the simplest one. We will show the code that does the same as the code of Listing 3.1 - it determines if a piece can make one simple step. Although the same code is executed for all pieces and all directions, we will describe it from the perspective of a single piece. The line of code that computes it is:

```
nOneStepSimple.at<0, 1>() = nCanStepAhead.at<0, 1>() & QuadBitset(
    notFrozen);
```

It is computed from one `Bitset8x8` (`notFrozen`) and one `Neighbourhood` (`nCanStepAhead`). The bitset `notFrozen` has a clear meaning - the piece is not frozen. It has been computed by this code:

```
Bitset8x8 dominated = nStrongerEnemy.at<0, 1>().any();
Bitset8x8 someSupport = nFellow.at<0, 1>().any()
Bitset8x8 notFrozen = ~dominated | someSupport;
```

`Neighbourhood nCanStepAhead` is harder to explain. Basically, it means that the piece can arrive at a given square by a step ahead provided that it can get to the previous square and it is not frozen there. It checks whether the destination square is empty and whether the piece is a rabbit that cannot move in that direction. In addition, it solves many other special situations that arise when we consider more steps and moving with enemies (pushing and pulling). It also handles traps and the possibility of being captured. It is complicated because of its generality. If we were only interested in the case of one simple step, then it could be defined as follows:

```
Nbr1 nWillSurvive, nCanStepAhead;
nWillSurvive.at<0, 1>() = ~nTrap.at<0,1> | nAtLeastTwoSupport.at
    <0,1>();
nCanStepAhead.at<0, 1>() = nEmpty.at<0,1>() & nWillSurvive.at
    <0,1>() & qCanMoveAhead;
```

The `nWillSurvive Neighbourhood` says if a piece can safely step on the square. The condition for the adjacent square is that it is not a trap or there are at least

two fellows next to it. Two pieces are required because the moving piece is one of them. The `qCanMoveAhead Quadbitset` forbids rabbits to move backwards. We have one such `Quadbitset` for each direction and we must not forget to use them.

It would be quite easy to write this function if we only allowed moving one piece. However, the amount of cases that we need to consider when we allow moving other pieces is enormous. The only way to tackle this problem is dividing it into smaller parts. During the development, we have always focused only on a subset of moves. When the code for the current subset was finished and tested, we moved on to another subset. Moves can be categorized by the triple of numbers:

1. The number of steps of the moving piece.
2. The number of steps of supporting fellows.
3. The number of steps of enemies (push or pull steps).

The example given above can be put into category (1,0,0). We first implemented (x, 0, 0) and (x, 0, y) cases. That was rather easy because the board does not change much when a single piece is moving and enemies can be pushed or pulled only by the moving piece, but it is definitely not trivial.

Figure 3.2 shows an Arimaa position that illustrates some strange situations. The gold cat at e3 cannot reach the g2 square, even though it has a clear path and every square on the path has a friendly piece next to it. On the contrary, it can reach the g4 square, but it needs to push the f4 rabbit into the f3 trap, where the gold camel initially sits. The point is that the gold camel will be captured if the gold cat moves and we must take it into account. The silver dog at d2 can capture the gold camel by pulling or pushing the gold cat. In addition, it can reach the f3 and e4 squares, which might not be obvious, because it would not be possible if the f3 square was not a trap. The last curiosity is that the silver elephant at b3 cannot get to a2 even though it can get to the adjacent square b2 in two steps and the a2 square is initially empty.

As an example, here is the code that solves another special situation:

```
QuadBitset qPattern5a = nTheOnlyGuardWeakerEnemy.at<1, 1>() &
    nCanMove1.at<1, 0>();
QuadBitset qPattern5b = nTheOnlyGuardWeakerEnemy.at<-1, 1>() &
    nCanMove1.at<-1, 0>();
QuadBitset qPattern5 = (qPattern5a | qPattern5b) & nTrap.at<0,
    1>() & nWillSurvive.at<0, 1>();
nFourStepsSimple.at<0, 1>() |= qPattern5;
```

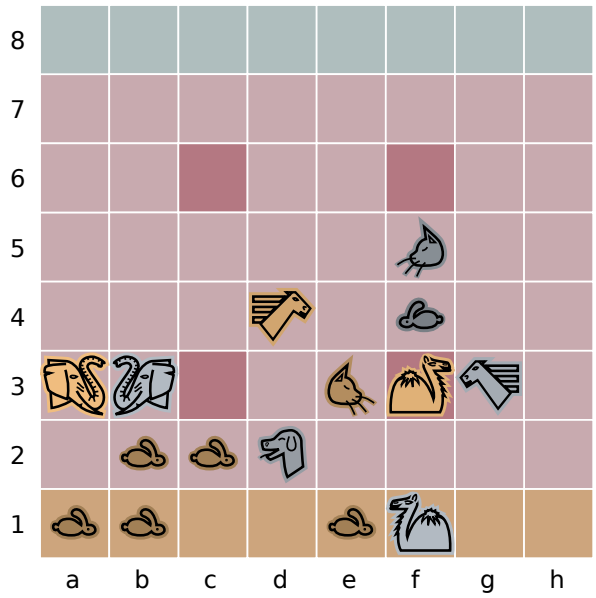


Figure 3.2.: Bizarre Arimaa position

The `nTheOnlyGuardWeakerEnemy` neighborhood says that there is a weaker enemy which is the only piece that protects a trap with another enemy. The `nCanMove1` neighborhood says that the piece can get to the given square in one step and it will not be frozen there. The expression on the first line means that the piece can make a step left (to [1,0]), then return while pulling a piece from [1,1], and an enemy will be captured. We know the piece is not a rabbit, because there is an enemy weaker than it. If it could be a rabbit, we would have to check that it can make a step back. The second line has a similar meaning as the first, but the direction is opposite. It starts with a step right (to [-1,0]) and the pulled piece is at [-1,1]. The whole pattern is that either the first case or the second case is true, there is a trap ahead (at [0,1]) and the piece will not be captured if it steps on the trap. We know that the captured piece must have been on the square [0,1]; thus, the square will be empty after the pull. The piece cannot be frozen after the pull, because it was not frozen initially and the only two changes are the capture of an enemy and a pull of a weaker enemy. Finally, we may conclude that the fourth step can be to [0,1]. The last line adds this rule to the `nFourStepsSimple` neighborhood, which is later added to the `fourStepsReach` neighborhood. This pattern is illustrated by Figure 3.3. The gold cat can reach the f3 square with the move Cf2w Ce2e re3s cf3x Cf2n.

The previous example also illustrates the importance of `Quadbitset` permutations. Although they are not explicitly used, they are actually applied three times[4]. For instance, the anticlockwise rotation is used in the expression:

[4]If the compiler was smart enough, it could optimize the code to use only two permutations. (If the same permutation is applied to both operands of a logical operator, then it can be

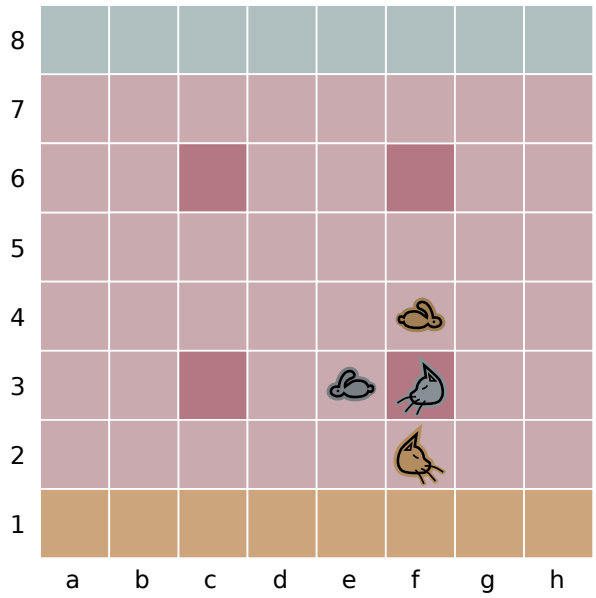


Figure 3.3.: Special pattern example

```
nCanMove1.at<1, 0>()
```

If we used a simple array of `Quadbitset` elements instead of the `Neighbourhood` class, then the same expression would look like:

```
nCanMove1[0].rotateAnticlockiwse()
```

The `Neighbourhood` class provides more readable code, and we do not have to think about `Quadbitset` permutations. However, we should still write the code so that it does not need many permutations. If all coordinates in the example code were rotated, then it would also work, but it would be less efficient because more permutation would be needed. The rules should be focused on the main quadrant, where no permutations are needed.

In next phases, we gradually added (3,1,0) moves, (2,x,y) moves and finally (1,x,y) moves. Each of this group was further divided into smaller groups. For example, the (3,1,0) moves can be divided according to the nature of the supporting step:

1. It unfreezes the piece before its first step.
2. It unfreezes the piece before its second step.
3. It unfreezes the piece before its third step.
4. It protects a trap for piece's first step.
5. It protects a trap for piece's second step.

applied on the result instead.)

6. It protects a trap for piece's third step.

7. It makes space for piece's first step.

8. It makes space for piece's second step.

9. It makes space for piece's third step.

This division is ambiguous because the supporting step can have multiple effects, but it does not matter. The important point is that every possible move is included in at least one of these cases. It is hard to make a clear categorization of all moves and some special cases must be often handled separately.

The last part, adding (1,x,y) moves, was solved specially. The piece makes only a single step, but a lot can happen around it. For instance, another piece may move three steps to unfreeze it. We should take advantage of the fact that we have already computed many interesting values in preceding part of the function. It would be perfect if we could use the information where a piece can get in three steps to say if the piece can be unfrozen in three steps. Unfortunately, it is more complicated because a move may have many side effects. For instance, a friendly piece may come to an adjacent square and unfreeze the piece, but it pushes an enemy to the square where the piece wants to move.

We distinguish two cases. The first is that the last full step is a double step (push or pull), and the second case is that the last full step is a simple step. If the last step is a double step, then only two steps remain and it is tractable to consider all the possibilities. In the second case, we took a different approach. Most moves in this group consist of a three-step move that gets a piece next to the piece or next to a trap that the piece is moving in. We make some generalizations of three-step moves. The first generalization is that we divide all three-step moves according to their side effects. We consider only side effects that may be significant. If a change is far from the destination square, then it may be ignored. Side effects are encoded in the name of a variable. There are more than one hundred types of moves, for example:

- `nThreeStep` - No significant side effects.
- `nThreeStep_aF` - A fellow is ahead. (The directions are relative to the destination square.)
- `nThreeStep_bL_blF` - A fellow has left the back square and it has come to back-left square.
- `nThreeStep_blS` - The back-left square has an additional support (fellow on an adjacent square).

- `nThreeStep_brEb` - An enemy has been moved from the back square to the back-right square.
- `nThreeStep_lL_blF_alD` - A fellow has moved from the left square to the back-left square and the ahead-left square has lost one support.

In the second phase, we act as if we have a fixed piece (or trap square) and a fixed adjacent square, where a friendly piece will be moved. The use of `Quadbitset` ensures that the same computations are automatically done for all pieces and all four adjacent squares. We consider all squares in distance one or two from the adjacent square one by one. In this phase, some types of moves may be fused. For instance, if we are considering moves from [-1,-1] to [0,-1] then it does not matter whether the move has any side effects on the right (square [0,-2]).

We must further distinguish three cases:

1. The piece is unfrozen.
2. The trap is protected for silver pieces.
3. The trap is protected for gold pieces.

Unfortunately, it is necessary to separate gold and silver pieces when we are dealing with the protection of a trap because both players may move in the trap. However, we delay the separation as long as possible. On the contrary, if a move is unfreezing a piece, then we know what piece's color is and there is no need of splitting pieces according to their color. For each of these three cases, we further distinguish the situations after the supporting move from the perspective of the piece that is unfrozen (the first case) or from the perspective of a trap that a piece will move into (second and third case). Again, there are many cases, for example:

- `nUnfreeze_rF` - The piece can be unfrozen from back, but there will be a friendly piece right.
- `nUnfreeze_rDD` - The piece can be unfrozen from back, but the right square will lose two protecting pieces (friendly pieces that are adjacent to the square). If the right square is a trap, then the unfrozen piece cannot move right, unless there was a friendly piece on all squares around the trap, because otherwise it would be captured.
- `nUnfreeze_aE` - The piece can be unfrozen from back, but there will be an enemy ahead.

- `nGoldTP_rS` - The trap can be protected from back for gold and the right square will get additional support. If there is a gold piece right, then it will be unfrozen.

- `nSilverTP_lD_rD_brEl` - The trap can be protected from back for silver, but the left and right squares lost one support and a gold piece is moved from the back square to the back-right square.

Each case for gold trap protection has a corresponding case for silver trap protection. In the final stage, we use a different approach for unfreezing and for trap protection.

When resolving unfreezing, we keep the idea that we have a fixed piece and a fixed adjacent square (back square), where an unfreezing piece is moved. For other three directions, we write rules which combine possibilities of unfreezing with necessary conditions. In a rule, all cases which have the same side effect in a given direction are grouped. For instance, if the piece wants to move ahead and that square has lost two supports by the unfreezing move, we must check that it is not a trap or there is still a supporting piece:

```
result.fourStepsReach.at<0,1>() |= nCanArriveDoubleDislodgePlus.at
    <0,1>() & (nUnfreeze_lF_aDD.at<0,1>() | nUnfreeze_rF_aDD.at
    <0,1>());
```

Trap protection is more complicated. There are a few additional steps of generalization. The output of the generalization is connected with a piece (not with a trap square) and there is no separation of gold and silver pieces. It says, for instance, that a trap in a given direction is protected for the piece and the piece has an additional support. The first step is not to consider from where a trap is protected, but only the effects of the protection for the piece in a given direction. In the next step, either the gold case or the silver case is selected based on the color of the piece. Finally, it is transferred to the piece with the `lookAt` function:

```
QuadBitset qProtected = lookAt<0,1>(nProtected.at<0,-1>());
```

Only a very small fraction of the function was presented in this chapter, but the same ideas that can be found in the examples are applied in other parts of code:

- The code is divided into small parts, which correspond to categories of moves.

- There are many abstract `Neighbourhood` variables, which make it easier to write rules and they make the code shorter.

- The rules are as general as possible.

3.7. Testing

It is generally a good idea to perform tests that check the correctness of a program. In this case, it is even more important. The problem is so difficult that it would be impossible to solve it without a tool that can automatically verify the program and tell us what is wrong.

The procedure for testing that we have applied is based on generating random inputs and checking the outputs. To check that the output is correct, we need another implementation of the function, which gives the correct output. At first, it might seem impossible to fulfill this requirement, but it is possible thanks to the fact that the verifying function does not need to be as fast as the main function. We might use a simpler algorithm, which is easier to implement. The natural choice is the algorithm which generates all step combos, that is described in Section 3.2.3.

Even the step-combo-generating algorithm is not trivial and we cannot prove that it is correctly implemented, but it is still far simpler and it was also thoroughly tested. Therefore, we have optimistically assumed that it gives correct output. Even if it was not absolutely correct, the chance that it will give the same wrong output in the same situation as the primary implementation, which is implemented in an entirely different way, is negligible. However, we must admit that there was initially a bug in a simple function that compares two outputs. It was fixed, but it shows that we may never be certain about correctness of a program.

The second problem is what the input data should be. There are two options:

- Positions from real games.
- Random (pseudo-random) positions.

Both of these options have some advantages and some disadvantages. The biggest disadvantage of real games is that we have only a limited number of them. Furthermore, the positions from real games may systematically miss some patterns. The files with records of all games that have been played on the Arimaa website can be downloaded, but it takes some effort to parse the files because they contain errors.

When we use random positions, some of these problems are eliminated. The question is, how a random Arimaa position should be generated. It is hard to achieve a probability distribution similar to the distribution of positions in real games. However, it is not important to generate natural-looking positions. The procedure that we have used placed pieces one by one on a random square. Not all pieces must be placed. The distribution of squares might not be uniform, and

it might be different for each type of a piece. The exact details were changed several times. The version that was used while testing used uniform distribution of squares whereas the current version places pieces in the player's half of a board with a double probability. The pieces which were placed on a trap and have no support are eventually removed.

In each stage of development, only some types of moves were considered. The function for generating moves gives the list of all (one-combo) moves; therefore, it is necessary to filter the moves before they are used to check the result of the function. The first step is to find more information about a move. The `MoveClassifier` structure defines several properties of a move. These properties are computed in the constructor, which takes a move and a position as arguments. The properties of a move are checked against a given condition, and if the condition is not satisfied, then the move is discarded. When we were advancing to a next stage of development, only the filtering condition had to be changed.

There are two types of errors:

1. False positive. The result of the function says that a piece can move to a square, but there is no such move.

2. False negative: The result of the function says that a piece cannot move to a square, but there is such a move.

When a false positive error is encountered, then the position, the piece and the square are printed. We use a C++ debugger to find the line which causes the error. Binary search method may be applied because there are only positive rules. We break the execution on a selected line and check whether the error has already showed. If it has, then we continue to search in the previous part of the code; otherwise, we focus on the next part. If the breakpoint is set in the middle of the interval, then we halve the interval. Using this method, the error is found quickly even though the function has thousands of lines. It would be possible to automatize this binary search, but we would need to emulate the code execution.[5]

When a false negative error is found, the position and the missing move are printed. According to the type of the move, we must identify the piece of code that should recognize the move and correct it. Sometimes, a new type of a pattern is discovered and a new section of code must be created. However, we must be careful not to duplicate the rules. We have always tried to fix existing code instead of introducing new code.

The function was tested on all positions from the game archive and on about 2 billion random positions. Hundreds of bugs have been found and fixed. Although

[5]The ideal solution would probably be to define the syntax as a special programming language and create custom compiler and debugger, but it would take too much time.

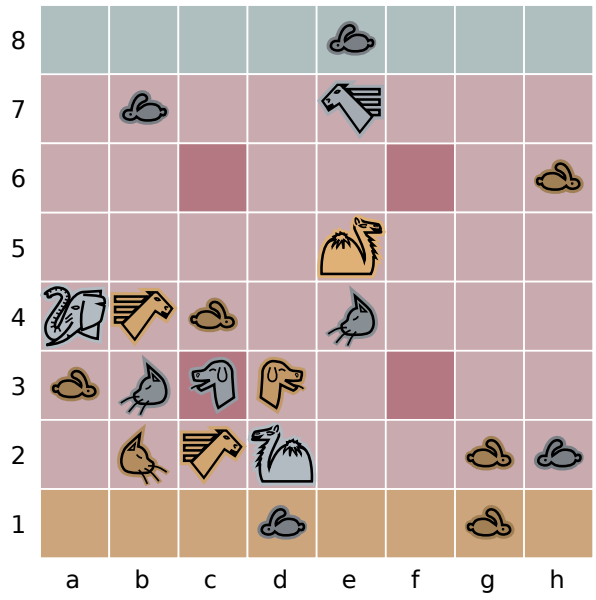


Figure 3.4.: Arimaa position illustrating a bug

we cannot prove that it is absolutely correct now, it should be good enough for any practical usage. Only a few bugs were found with the second billion of random positions.

Figure 3.4 shows a position in which the last bug was found. It is obvious that the position was randomly generated because, for instance, a silver rabbit is on the goal row, but it does not matter. The function claimed that Gold can move the gold dog from d3 to c3, which is not true. When the bug was found, the function has been already tested on more than a billion positions, so how could it make such a mistake?

The reason is that it is a quite unusual situation. First, it involves a capture, which is a rather rare phenomenon. Secondly, the answer would be right if the position was slightly different. If the b2 square or the a3 square was empty, then the gold horse at b4 could push the silver cat at b3 capturing the silver dog, the c4 rabbit could unfreeze the gold dog, and finally the gold dog could step into the c3 trap. In fact, the dog can be moved into the trap (by the move Hc2s dc3s Rc4e Dd3w Dc3x), but it would be captured and that does not count. The move that caused the confusion is Hb4n cb3n dc3x Rc4e Dd3w. It is not valid, because the gold rabbit is frozen by the silver cat that is pulled by the gold horse. If there was a different gold piece instead of the rabbit at c4, then the move would be possible and the answer would be right. Furthermore, if there was not the silver elephant, the rabbit could move first and the answer would be also right.

3.8. Possible extensions

Although we have accomplished our goals, we expect that more types of patterns will be recognized in a future version. The knowledge of Arimaa is still evolving and the static evaluation function will have to be continually updated to reflect this evolution. Even now, we have a few ideas how the pattern recognition could be extended.

Capture possibilities can be detected, but frame possibilities not. A frame is a situation when a piece is in a trap, it cannot move and it has only one guard protecting it from being captured. It is less devastating than a direct capture, but it is still a serious problem. Currently, a frame can be easily detected, but we cannot say if a player can make a frame in one turn. It would probably be quite difficult to recognize all possible patterns, but it might help to incorporate at least the most common ones.

When the moving player can reach the goal row with a rabbit, then the position is won for that player. This situation is detected by our pattern recognition function. However, we might go one turn further. If the moving player cannot win and the opponent has an immediate goal threat that the player cannot stop, then the position is lost for the player. To check whether this situation has occurred, it is currently necessary to enumerate all possible (unique) moves and evaluate the subsequent positions to see if the opponent still can goal after the move. That is very expensive. It would help very much if we could recognize this loss-in-two situations statically, like the win-in-one situations.

Although it is probably intractable to perfectly recognize loss-in-two without any search, it would be quite easy to write a heuristic that would recognize some simple cases. The pattern recognition function basically consists of many large Boolean expressions. It works only with Boolean values, and only logic operations are used. We perform many operations at once, but that does not change anything. The test for goal is, in fact, a single Boolean expression, which takes the bitboard as the input. All we need to do is to modify it so that it will use Kleene's three valued logic.

Kleene's logic works with three values:

- True
- False
- Unknown

The basic idea behind this logic is that we may evaluate logic expression with some variables having unknown values and the result may still be True or False.

AND	True	Unknown	False
True	True	Unknown	False
Unknown	Unknown	Unknown	False
False	False	False	False

Table 3.3.: AND operator in Kleene's logic

OR	True	Unknown	False
True	True	True	True
Unknown	True	Unknown	Unknown
False	True	Unknown	False

Table 3.4.: OR operator in Kleene's logic

In that case, it does not matter which value the unknown variables will have, the value will always be the same. The truth tables for AND, OR and NOT operators are showed in tables 3.3, 3.4 and 3.5.

Although we do not know which move the moving player will play, the board after the move may be described by a bitboard which uses the three value logic. Classic bitboard has a Boolean value for each square and each piece type. The value is true if and only if a piece of that type is on that square. The Unknown value means that a piece of that type may or may not be there. Another view on mobility analysis is that it tells us where a piece may be after one turn, and that is exactly what we need. Without loss of generality, we may assume that Gold is on the move. If a gold piece cannot move (and it is not in a trap), then we will assign True to the square where it is, as in classic bitboard. If a gold piece can move (or it is in a trap), then we will assign Unknown to all squares where it can be after a turn. Similarly, if a silver piece cannot be moved nor captured by Gold, then True is assigned to the square where it is, and otherwise Unknown is assigned to all squares where it can be moved by Gold. We must be careful with gold pieces in traps because we do not know whether Gold may sacrifice them. We could find this information, but it is probably not worth the effort.

If this bitboard with Unknown values is given to the function which is modified to work with Kleene's logic, it will output the same variables as the original function, except that some Boolean variables may have the Unknown value. The important point is that if a variable is True or False, it will have the same value

X	NOT X
True	False
Unknown	Unknown
False	True

Table 3.5.: NOT operator in Kleene's logic

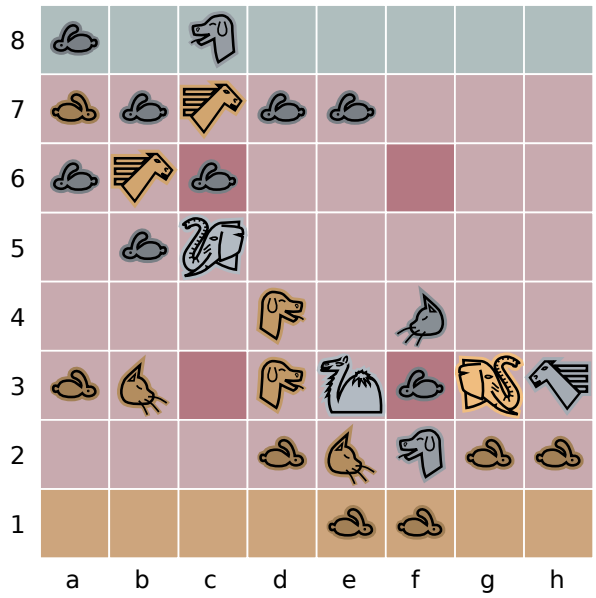


Figure 3.5.: Position after the turn 35s in the game between omar (Gold) and rabbits (Silver) from the 2013 Arimaa World Championship.

after one turn no matter which move will be chosen. For instance, if the goal detection test is True for the other player, then the moving player cannot prevent a goal. In other words, the position is lost for the moving player unless the player can also immediately goal.

The main drawback of this method is that it would probably often output Unknown. We can expect only the simplest loss-in-two situations to be recognized. This method could succeed in positions where a player has a goal move which cannot be disrupted by the opponent in any way. Unfortunately, the more usual case is that the opponent can prevent each single goal move, but it is impossible to prevent all the goal moves at once. However, it could still significantly improve playing strength against humans because humans can also recognize only the simplest situation without longer thinking. Furthermore, there may be some possible enhancements that would make it stronger.

We illustrate this topic with a game between omar (Omar Syed, Gold) and rabbits (Gregory Clark, Silver) from the 4th round of the 2013 Arimaa World Championship. Figure 3.5 shows the position after the turn 35s. Both players had a dangerous goal threat. If omar had played the move dc8e Hc7n rb7e Hb6n, which would have led to the position in Figure 3.6, he would have won in his next turn because he would have had an unstoppable goal threat. Unfortunately for him, omar chose a different move and lost the game a few turns later. Would it be possible to say that the position in Figure 3.6 is won for Gold using the suggested method?

The answer is that the basic version of the method would probably give the

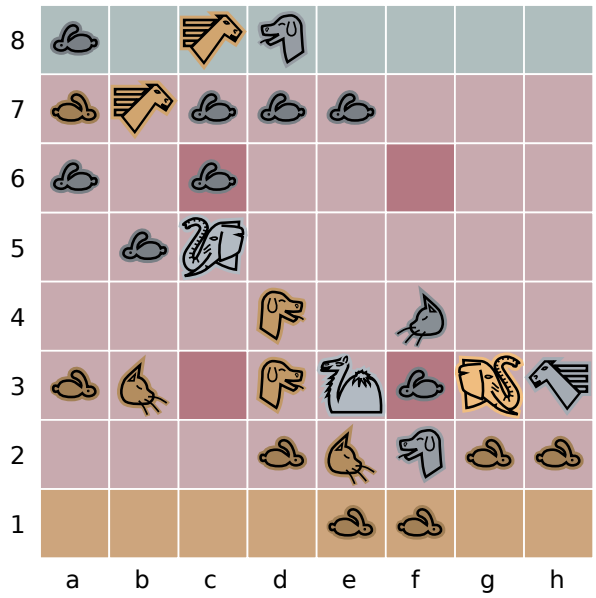


Figure 3.6.: Position after playing 36g dc8e Hc7n rb7e Hb6n instead of the move chosen by omar (Gold) in the game with rabbits (Silver) from the 2013 Arimaa World Championship.

answer Unknown. The reason is that the silver rabbit at a8 can move to b8, thus the rabbit has the Unknown value for both these squares and that is a problem. However, we know that the gold rabbit at a7 and both gold horses cannot be dislodged by Silver, thus the gold rabbit cannot be frozen. When a gold rabbit that is on the seventh row cannot be moved nor frozen by Silver, then the only possibility for Silver to stop the rabbit from goaling is to place a piece in front of it. In the discussed position, the only piece that can stand in front of the gold rabbit is the a8 rabbit. Accordingly, we can infer that the silver rabbit will stay at the a8 square and the b8 square will be empty. With this extension, the suggested method might be successful.

In conclusion, the pattern recognition part of the static evaluation function may be further extended in many interesting ways, and one of the most interesting is the recognition of loss-in-two positions. There should be more research in this field.

4. Important position features

4.1. Distance computation

The purpose of the middle layer of the evaluation function is to generalize. The first step is to conceal complicated rules about piece movement. We do not want to deal with low level intricacies, like piece freezing, at higher levels. However, we need some information about piece mobility. We believe that piece mobility is the key part of the evaluation. For instance, one of the main factors which determine the relevance of a goal threat is the distance of a rabbit from the goal row. It is also very important to know how many steps would an elephant need to get to a given square, especially next to a trap. Although we do not want to deal with tactical details anymore, we must take them into account when computing distance. The fact that a gold rabbit is on the seventh row does not necessarily mean that it is close to goaling, because it can be blocked by silver pieces. The proposed solution is to compute a matrix of distances for each piece.

A distance matrix has 8 rows and 8 columns like the Arimaa board. An element of the matrix is a distance in steps to the corresponding square. The distance can take values from 0 to 15. In the code, it is represented with the `MatrixB8` class. `MatrixB8` is used not only for distances, but for any matrix with size 8×8. The elements are stored in a single byte; therefore, they may be in the range 0 to 255. The implementation of `MatrixB8` uses SIMD instructions if they are available. It is stored as four 128-bit vectors and, in the future, it may use 256-bit vectors or even 512-bit vectors. `MatrixB8` has many methods and overloaded operators, that can be used to evaluate matrix expressions:

- Element-wise arithmetic operators (+, +=, -, -=).
- Element-wise logic operators (&, |, ~, &=, |=) .
- Element-wise comparison (<, >). The result is 255 if the comparison is true; otherwise, the result is zero.
- Element-wise minimum, maximum, average (methods min, max and avg).
- Element-wise division and multiplication by a constant (methods `div4`, `div8`, `mul16`).

- Element-wise addition and subtraction with saturation (methods `adds`, `subs`). Saturation means that if a result cannot be represented in the range $[0, 255]$, then the nearest possible value is chosen (0 for negative values, 255 for values greater than 255). On contrary, the standard arithmetic operators return the lowest byte ($255 + 1 = 0$).

- Conditional blending (`conditionallyStore` method). It selects each value from two matrices depending on the corresponding value (0 or 255) of the third matrix. The third matrix may be the result of a comparison, for instance.

- Matrix shifts (methods `moveUp`, `moveDown`, `moveRight`, `moveLeft`). They move elements in one direction. One row or column is discarded and one row or column is added. The default value for the new elements is zero, but there are overloaded versions that insert an arbitrary value.

- Table transformations (method `tableLookup`). Apply an arbitrary function $f : [0, 15] \to [0, 255]$ to all elements in a matrix. The function f is defined by the table (vector) with 16 values. It is used to map distance values to some other values. In SSSE3 instruction set, there is s special instruction (`pshufb`), which provides an efficient implementation.

- Aggregating functions (methods `sum`, `minimalElement`, `colMin`)

- The `nextTrap` method. It extracts 16 elements corresponding to the squares which are adjacent to a trap.

The computation of distance matrices is rather complicated. Initially, the matrices are set according to mobility analysis, which is described in Chapter 3. If a piece can reach a square in N steps ($N \leq 4$), then the value of the square is set to N. Otherwise, it is set to 15, which is the maximal value of a distance. Afterward, a heuristic algorithm is used to better approximate distances to the squares that cannot be reached in one turn. The meaning of distance for values greater than four is not precise. It should approximate the number of steps that a player would need to move a piece to the square. However, since a player can never play more than four steps without interference from the opponent, it might not be best to keep strictly to this definition.

The heuristic algorithm is based on the Bellman–Ford algorithm, which finds shortest paths from a vertex (the source vertex) to all other vertices in a weighted graph. The Bellman-Ford algorithm iteratively improves the solution. Initially, all vertices have infinite value except the source vertex, which has zero value. In each iteration, we try to lower a distance value along every edge. For instance,

```
MatrixB8 fromLeft = get[p].moveRight() + priceFromLeft;
MatrixB8 fromRight = get[p].moveLeft() + priceFromRight;
MatrixB8 fromAhead = get[p].moveDown() + priceFromAhead;
MatrixB8 fromBack = get[p].moveUp() + priceFromBack;
MatrixB8 m1 = fromLeft.min(fromRight);
MatrixB8 m2 = fromAhead.min(fromBack);
MatrixB8 m = m1.min(m2);
MatrixB8 newval = (m + one).max(five);
get[p] = get[p].min(newval);
```

Listing 4.1: A single iteration of the distance computation for a piece p.

if a vertex A has the value 6 and it is connected by an edge with the weight 2 to a vertex B with the value 9, then the value of B can be lowered to $6 + 2 = 8$. It can be proved that after $N - 1$ iterations, where N is the number of vertices, the value of any vertex is the length of the shortest path from the source to the vertex, provided that there are no negative cycles.

When computing distances, the vertices are squares and edges connect adjacent squares. The weight of an edge is a heuristic guess of the number of steps that a given piece would need to move along the edge. There are some modifications to the Bellman-Ford algorithm. Firstly, the distance matrix is already partially computed before the algorithm starts, as described above. Secondly, squares with value less than 5 are never updated and a distance can never be lowered to a value less than 5. Finally, the maximum value of distance is 15, which implies that only 10 iterations have to be performed because weights are at least 1, and because we start with a matrix computed to distance 4. In fact, even fewer iterations are performed because the result is still good enough and it saves time.

Listing 4.1 shows how a single iteration of distance computation is implemented with matrix operations. The input is the current distance matrix (`get[p]`) and the edge weights (`priceFromLeft`, `priceFromRight`, `priceFromAhead`, `priceFromBack`). Actually, the weights of edges are one greater. This one is added later because it is more efficient. For each direction, the number of steps required to get to the square from that direction is computed. Then the minimum of these values is computed. Next, one is added and the computed value is enforced to be at least 5. Finally, the distance value is set to the minimum of the current value and the computed value.

The most difficult part is the estimation of transition costs (edge weights). It must be simple because the evaluation must be fast. At the same time, it cannot be simple, because there are many special situations that must be handled if the result should be useful. A compromise must be found between these two conflicting requirements. We simplify the problem by dividing the cost computation into

two parts. It will be described from a perspective of one selected piece moving from a given source square to a given destination square. The same procedure is then applied for all pieces and all edges.

The first task is to estimate the number of steps that would be needed to unfreeze the piece at the source square. The basic idea is that if there is no stronger enemy adjacent to the source square or there is a friendly piece adjacent to the source square, then the unfreeze cost is zero. Otherwise, the cost is the minimal number of steps that another friendly piece would need to get to an adjacent square. However, this idea is too much simplified. It must be improved in several ways. First, we know that the destination square must be empty or there must be a weaker enemy before the piece moves there. Accordingly, we do not consider the destination square when it is determined whether the piece is frozen. Second, the situation is different when the source square was originally occupied by another piece.

If there was a friendly piece, then, when the piece is moved there, the friendly piece is probably at one of the adjacent squares, and therefore the piece cannot be frozen. However, it is even more complicated. The friendly piece might have been able to move only to the destination square, where it will be in the way. We try to cover this case by the following expression:

```
(leaveNotTo.min(unfreeze + leaveFar)).subs(leaveAnyDir);
```

The expression uses pseudo-code to make it clearer, but it is very similar to the actual code. The value of the `leaveNotTo` variable estimates the number of steps that the friendly piece would need to move to a square different from the destination square. The `unfreeze` value estimates the number of steps in which another friendly piece can come to an adjacent square which is different from the destination square. The `leaveFar` value estimates the number of steps that the friendly piece needs to move to a square that is not adjacent to the source square. Accordingly, the expression (`unfreeze + leaveFar`) estimates the number of steps which are needed to unfreeze the piece with a different piece than the piece that was initially at the source square. If we take the minimum of this expression and `leaveNotTo`, then we get the estimate of the number of steps that are needed to unfreeze the piece without blocking the destination square. However, `leaveAnyDir`, which estimates the number of steps that the friendly piece at the source square would need to move to any other square, is subtracted because only the additional penalty connected with the step should be considered. We apply the subtraction with saturation to ensure that it will not underflow. The current distance matrices are used to estimate the number of steps that a piece would need to get somewhere in all cases in which it is needed.

If the source square is occupied by an enemy that is not weaker than the piece, then the first part of the cost is computed with following expression:

```
pulledNotTo.subs(pulledAnyDir) + 1
```

The idea is similar to the idea in the previous case. If the enemy might be moved only to the destination square, then additional penalty is added. The `pulledNotTo` estimates the number of steps that are needed to move the enemy to a square different from the destination square. The `pulledAnyDir` value estimates the number of steps that are needed to move the enemy to any square. In fact, it is more complicated. There are several minor adjustments, that should solve some special situations. The subtraction with saturation is used for the same reason as in the previous case. The one is added in any case, since the enemy which is not weaker may cause some troubles.

If the source square was initially occupied by a weaker enemy, then we assume that the piece has pushed the enemy to an adjacent square. If it was pushed to the destination square, then we need to add a penalty that depends on the estimate of number of steps that it would take to push the enemy again. If the enemy was not pushed to the destination square, then we know that the number of steps that the piece needs to get to the source square must be greater than or equal to the estimate of the number of steps that are needed to push the enemy to an adjacent square that is different from the destination square. An estimate is computed for both these cases and the minimum is taken.

This brief description is missing many details. For instance, one interesting problem is how to compute the estimate of the number of steps that a different friendly piece would need to get to a square. That is needed to estimate the number of steps to unfreeze a piece. The difficulty is that the piece for which we compute the estimate must not be taken into account, because it cannot unfreeze itself. The problem is, in essence, how to transform efficiently a vector of N values so that each value is set to the minimum of all other input values. The values are distance matrices, in this case. A straightforward algorithm would compute N times minimum from $N-1$ values, which means performing $O(N^2)$ operations. A better algorithm uses dynamic programming. First, we compute all prefix minima and all suffix minima with a $O(N)$ operations. Then we can compute any output value as a minimum of a prefix minimum and a suffix minimum in a constant time. Overall, the minimum of two values is computed $O(N)$ times.

The second part of the cost is associated with the destination square. It estimates the number of steps that would be needed to free the destination square. For instance, if there is a weaker enemy that can be pushed to an adjacent square, then the cost is one. Four cases are distinguished:

1. The destination square is empty.
2. The destination square is occupied by a friendly non-rabbit piece or a weaker enemy.
3. The destination square is occupied by a friendly rabbit.
4. The destination square is occupied by an enemy of equal or higher strength.

In the first case, there is no additional cost. At first, it might seem strange that, in the second case, a weaker enemy and a friendly non-rabbit piece are examined together, but the situation is actually similar for both types of pieces. A weaker enemy can be pushed away if there is an empty square around, and a friendly non-rabbit piece may move to an empty square as if it was pushed because it will be unfrozen by the incoming piece. The situation is a little different when there is a friendly rabbit because a rabbit cannot move in one direction. The fourth case is the worst because another friendly piece must dislodge the enemy. In this context, we expect that an enemy is usually dislodged by pulling, because if it was pushed, then the pushing piece would be in the way.

In the second case and in the third case, there might be no empty square where the piece could be moved from the destination square. If that has happened, then we estimate the number of steps needed to free one of the squares where the piece could be moved. For a friendly piece, the estimate is equal to the minimum number of steps it needs to move to any square except the destination square. For an enemy piece, the estimate is equal to the estimate of the number of steps that are needed to pull it to any square except the destination square.

A technique, which is sometimes called bootstrapping, is used because we use the current distance matrices in the computation of weights, that are later used to compute more precise values of the distances, which can be subsequently used to compute better weights, and so on. We could continue until the distance matrices stop changing (the distances can only decrease), but that would be too expensive. The weights are only recomputed once.

The bot is able to generate interactive HTML log of a game. In the log, one can see, among other things, the distance matrix for each piece. The Figure 4.1 shows a position with information about mobility for the gold horse at g4. The number in the left upper corner of a square is the estimate of the number of steps that the gold horse would need to get to the square. The maximal value, which is also used when the piece cannot get to a square, is 15. For instance, the horse cannot get to e3, because there is the silver elephant; therefore, the distance value for e3 is 15. Unfortunately, there are some inaccuracies. For instance, the gold horse cannot reach h6 in 5 steps, it would need 6 steps. Nevertheless, the goal

Figure 4.1.: An example of distance matrix

Piece type	Evaluation
elephant	18
camel	11
horse	6
dog	3
cat	2.5
rabbit	1-12

Table 4.1.: Piece evaluation in bot_bomb

was not the perfect result but a reasonable estimation that can be computed in a reasonable time, and it seems that this goal has been achieved.

4.2. Relativity of piece strength

In Arimaa, the difference between various non-rabbit pieces manifests only in the interaction with opponent's pieces. From the perspective of a piece, it is only important which opponent's pieces are stronger because they can freeze the piece, and which pieces are weaker because the piece may dislodge weaker enemies. For instance, if one player looses the camel and both horses, then the opposing camel and the opposing horses are equal. They can be frozen by the same pieces (the elephant) and they can dislodge the same pieces (dogs, cats and rabbits).

Some Arimaa bots have hard-coded evaluation for each type of a piece. For instance, bot_bomb [6] uses the evaluation which is described in Table 4.1. Since the strength of pieces is relative, this evaluation must be inevitably wrong in some positions. Arimaa community noticed this problem long time ago, and several

solutions have been suggested on the Arimaa forum. We have adopted the idea of determining the strength of a piece by the number of stronger enemies. The strength of a piece can take values from the range 0 to 14. It is not defined for rabbits, because rabbits are so special that they are handled separately. The formula for strength is:

$$\text{strength} = 14 - 2 \cdot \text{strongerEnemies} - \text{equalEnemies}$$

The strongerEnemies variable is the count of stronger opponent's pieces and the equalEnemies variable is the count of opponent's pieces with the same strength. After the setup, a cat has the lowest strength 0 and an elephant has the strength 13. An elephant would gain strength 14, if the opposing elephant was captured, but that rarely happens. When many pieces are captured, the remaining pieces become stronger. For instance, if all horses are removed, then a dog has the same strength as a horse had at the beginning of the game ($14 - 2 * 2 - 2 = 8$).

The formula has been chosen for its simplicity. There are probably better ways to define piece's strength, but this definition seems good enough. We could try different constants or we could define the strength as the number of weaker enemies, but there is no clear evidence that it should be better. Generally, at this level of the evaluation, all concepts are vague and intuitive. The reason is that we are dealing with abstract terms which are intuitively understood by humans, but which cannot be easily defined.

When considering the relativity of piece strength, we can go even further. The fact that an opposing piece is stronger than a given piece is significant only when the stronger piece is on an adjacent square because then it can freeze or dislodge the piece. When the opposing elephant is far from the camel, then the camel becomes invulnerable as if it was an elephant. The actual strength of a piece is determined not only by the number of stronger enemies, but also by their distances. With distance, we mean the number of steps that the stronger enemy would need to get next to the piece, which was discussed in Section 4.1.

To express the local strength of a piece, we introduced the concept of fear. The fear is an integer value computed for each piece and each square. It is in the range $[0, 255]$. The low value means that there is no stronger enemy around and the high value means that there are many stronger enemies around. We use `MatrixB8` to compute the fear values for all squares. We only need to compute the matrix for each type of a piece. First, all distance matrices of non-rabbit pieces are transformed by a function that maps distances to the fear value that is caused by the piece. It is not obvious how the function should look like, but it should decrease with higher distance. The mapping which is currently used

distance	fear	support	protection1	protection2	danger	mobility
0	90	0	255	255	30	0
1	70	26	200	180	26	255
2	55	23	182	162	23	200
3	45	20	166	120	20	160
4	37	17	150	105	17	125
5	29	12	120	90	12	95
6	25	10	105	83	10	75
7	21	8	90	69	8	60
8	18	7	82	56	7	50
9	15	6	69	44	6	34
10	12	5	56	32	5	24
11	10	4	44	21	4	16
12	9	3	32	10	3	12
13	8	2	21	5	2	7
14	7	1	10	1	1	3
15	6	0	0	0	0	0

Table 4.2.: Mapping of distance to various values

is described in the second column of Table 4.2. The fear matrix for a piece is computed as the sum of all such transformed matrices which are associated with a stronger enemy piece.

The antithesis of the fear is support. The support value is computed for each player and for each square, and it is therefore represented as two matrices. It quantifies how many pieces of the given color are around a given square. If a piece has many friendly pieces around, then it can better fight with a stronger enemy because it can be easily unfrozen. It is computed similarly like the fear. Contribution of every piece is computed and then it is summed, separately for each player. The contribution is based on a table transformation of the distance matrix. The transformation is defined in the third column of Table 4.2. The strength of a piece is also considered. To be more specific, the fear value is used as a measure of strength in this case. If a piece has high value of the fear at a given square, then it contributes a little less to the support value of this square. One sixteenth of the fear value is subtracted from the support, but at least one half of the original value must remain.

4.3. Goal threats

A goal threat, which is a very important position feature, means that there is a rabbit close to the goal row. The situation that the moving player can immediately goal is recognized in the first layer of the static evaluation function, but that is definitely not sufficient. Long-term goal threats must be also evaluated.

The evaluation of goal threats is strongly connected with the evaluation of rabbits. We have decided to separately evaluate the goal prospect of each rabbit. This solution is not ideal, because two or more rabbits might be involved in a single goal threat, but it makes the problem much simpler. Another option is to compute a goal threat separately for each square on the goal row, but that has a similar problem because one rabbit can threaten to goal on more squares.

The reason for choosing the first option, i.e. to associate goal threats with rabbits, is that then we can consider interactions of goal threat feature with other features such as the threat of capturing the rabbit. If a rabbit that poses a serious goal threat can be captured, then the value of the goal threat is lower (or, from another point of view, the value of the capture threat is higher). It would be hard to handle that dependence if the goal threat evaluation was absolutely separated from the evaluation of the capture threat.

The feature that deals with goal threats is called goal prospects. It is computed for every rabbit. The value is an integer from the range $[0, 255]$. The higher the value is, the more serious the goal threat is. It is computed by the following formula:

$$\text{goalProspects} = \max_{\text{goal square } s} \{255 - 16 \cdot \text{stepsToGoal}(s) - (\text{fear}(s) - \text{support}(s))/4)\}$$

The prospects of goaling are evaluated for all goal squares and the maximal value is taken. In the expression, all subtractions are with saturation to ensure that the value cannot underflow zero. The rationale for the formula is that:

1. The most important factor is the estimate of the number of steps that the rabbit needs to goal.

2. The goal threat is less serious if the opponent has many pieces around the target square.

3. Friendly pieces can help to break the opponent's defense.

4.4. Capture threats

When there are all pieces on the board, it is almost impossible to get a rabbit to the goal row. Therefore, in the beginning, the objective of both players is to capture opponent's pieces. Capturing is much harder in Arimaa than in chess because pieces move slowly. To capture a piece, a player must take control of a trap and move the piece into the trap, which usually requires long preparation. It often takes tens of turns before the first capture happens. Therefore, a good

evaluation function should recognize not only immediate capture threats, but also long-term threats.

For each piece, a value called danger is computed. As usual, it is an integer in the range $[0, 255]$. The higher the danger is, the more serious the capture threat is. To simplify the problem, we compute the danger value separately for each trap, and then we sum these partial results. That seems right because if a piece is threatened in more traps at once, then it is harder to save the piece. The computation will be described for a given trap and a given piece. The evaluation of capture threats is based on two things:

1. How many steps would the opponent need to move the piece into the trap?
2. Can the player protect the trap? How many friendly pieces are close and how strong they are?

These two features are independently quantified. The danger value is the product of those values which is scaled to fit the required range.

The first part is simpler. At the start of the program, the Manhattan distance between every square and every trap is precomputed and multiplied by two. If the piece can be dislodged by an opposing piece, then all squares where it can be moved are examined and the result is the minimum of the sum of the distance from the square to the trap and the number of steps that the opponent needs to move the piece to the square. Otherwise, the result is the sum of the minimal number of steps a stronger enemy needs to get close to the piece and the distance to the trap. There are several minor details, which should make the estimate better. For instance, only three-step and four-step moves are taken into account and, in the second case, the stronger enemy is always considered to be at least 3 steps from the piece because we know it cannot dislodge the piece.

The second part is much more complicated. Supposing that the moving player may move the elephant to any trap in one turn, does it mean that all traps are safe? That is definitely not true, because the elephant cannot be at two places at once. Nevertheless, we still need a function that estimates which trap is safe and how much safe it is. The term *defensive square* will be used for a square adjacent to a trap. The method we use consists of four steps:

1. For each piece and for each defensive square, the ability of the piece to control the square is quantified. This value is called grasp.
2. For each player and for each trap, the priority of the trap is computed.
3. For each player, the best matching between pieces and defensive squares is found.

4. For each piece and for each trap, the trap protection value is computed.

The grasp value is based on the distance the piece has from the square and on the local relative strength of the piece. It is an integer value in the range $[0, 255]$. The exact formula for the grasp is rather complicated:

$$\text{grasp} = \max(\text{MinG}, g(\text{Distance}) - \text{DP} - \text{F1} - \text{F2}) - \text{RP}$$

All subtractions are with saturation. The most important part is the g function, which maps the distance to grasp. It is defined by the "protection1" and "protection2" columns of Table 4.2. It is different for the player on the move (protection1) and for the player who is not on the move (protection2) because, for the player on the move, it is less important whether a piece is directly on a defensive square or it needs a few steps to get there.

MinG value is 16 if the piece is on the defensive square, otherwise it is zero. The maximum ensures that if a piece is actually protecting a trap then the grasp is nonzero even if there are many stronger enemies around.

DP value penalize pieces that are on the defensive square but can be dislodged by the opponent. The penalty is 30, 35 or 40 depending on the number of steps the opponent would need. (The fewer steps are needed, the higher the penalty is.)

F1 is equal to the fear at a defensive square divided by four or eight. The divisor is four if the square is near the center of the board (d3, c4, c5, d6, e3, f4, f5, e6) and eight otherwise (c2, b3, b6, c7, f2, g3, g6, f7). The reason for bigger influence of the fear near the center is that weak pieces near the center are easy targets for the opponent. Elephants (or other locally dominant pieces) are not affected by the F1 value, since they have zero fear.

F2 is similar to F1. It is basically the fear divided by four, but it is lower if the support for the piece is high. 63 is subtracted from the support and the result is divided by two. This value is subtracted (with saturation) from the fear (divided by four). The rationale behind this formula is that if a piece has many friendly pieces around, it can more easily fight with stronger enemies, since it can be unfrozen by the friendly pieces and it can form phalanx.

RP, the last term, is the least significant. It is 5 for rabbits and 0 for other pieces. Its purpose is to slightly penalize rabbits as trap guards. The reason that rabbit should be considered worse than non-rabbits is that rabbits cannot move back.

The trap priority is a value computed for each trap and each player. Pieces are preferentially assigned to traps with higher priority. The basic reason that a trap should have high priority is that a piece is threatened in the trap. On contrary,

if it is easy to protect a trap (there are many pieces which can protect it), then the priority can be lowered because the trap will hopefully still be protected well. However, we should be careful with lowering priority in such cases. If the trap is evaluated to have bad protection and there are many pieces around the trap, then the evaluation of the position would be terribly wrong. Trap protection value is an integer in the range$[32, 127]$, but it is interpreted as a rational number from the interval $[0.5, 2)$.

The next step is matching of pieces and defensive squares. We need two matchings, one for each player. Why should be such a matching done? There are two reasons. The first is that a piece can protect at most one trap. The second is that a square can be occupied by at most one piece. The matching guarantees both these requirements. However, it brings the problem how the matching should be done. From the perspective of a player, it would be reasonable to choose the matching that gives the best evaluation for the player. There is one theoretical and one practical problem with this idea. The theoretical problem is that the best matching for one player might depend on the matching that was chosen for the opponent. Game theory studies this kind of problems and offers Nash equilibrium and the minimax theorem as a possible solution, but that solution would still be problematic. Nevertheless, we do not deal with this problem, because there is another, even more serious problem. It is impossible to efficiently evaluate all matching, and there is no way how to find the best matching without evaluating all matchings since the function that computes the final evaluation may be arbitrary. We are therefore forced to do a simplification.

The matching is computed independently for each player. It is chosen to maximize the total trap protection weighted by the trap priority. If we use the abstraction of graph theory, then we can imagine that we have a bipartite graph, where one part consists of pieces and the second part consists of defensive squares. An edge is between each piece and each square with a weight equal to the product of the trap weight and the grasp. Without loss of generality, we assume that there are all 16 pieces on the board, and thus both parts contain 16 vertices. (If a piece has been captured, all edges connected to it would have zero weight.) The task is to find the maximal perfect matching[1] in the graph. This problem is called "assignment problem" and we use the shortest augmenting path algorithm to solve it. The algorithm is described in Section 4.5.

The last step of trap protection computation is simple. The protection value of a trap is equal to the sum of grasp values of pieces that are assigned to an adjacent square. The only complication is that the grasp of a piece should not

[1] A perfect matching is a subset of edges that contains each vertex exactly once. Value of a matching is the sum of weights of edges in the matching.

be taken into account when its capture threat is considered. Consequently, there is a different value for each piece.

If we put together distance from a trap and trap protection, we get the danger value. The distance is mapped into the range$[0, 30]$ by the function which is defined by the "danger" column of Table 4.2. The value is interpreted as a fixed-point rational number from the interval $[0, 1)$ with a scaling factor 2^{-5}. Then it is multiplied with the complement of the danger value $(255 - \text{danger})$. Finally, values for all traps are summed (with saturation).

As the very last step, the danger value is modified if the piece can be captured. It is increased by 40, 60 or 80 depending on the number of steps that the capture requires. Furthermore, a minimal danger value for a piece that can be captured is defined. If the computed value is lower, it is increased to this minimal value.

4.5. Solving the assignment problem

The most complicated part of capture threat evaluation is solving the assignment problem. In Section 4.4 the problem is described in the language of graph theory. There is another way how the problem can be defined, which may be more intuitive. The input is a $n \times n$ matrix C (cost matrix). We will assume that the elements are non-negative integers, since that is true in the case we are interested in, but it is not a necessary restriction. The task is to choose elements of the matrix that satisfies these conditions:

1. Exactly one element is chosen from each row.
2. Exactly one element is chosen from each column.
3. The sum of chosen elements is maximal.

In some situations, it is still better to use the graph terminology. The vertices of the graph are rows and columns. The rows are the first part and the columns are the second part. An edge is between every row and every column. The weight of an edge is the value of the matrix in the given row and column. The selected values form a perfect matching.

Before we describe the algorithm, we start with the simple question: How can we prove that a given solution is optimal? It is easy to verify the first two conditions, but it is not obvious how the third condition should be verified. However, it would be easy if we were given additional information. The basic observation is that if we subtract the same value from all elements in a row or in a column, then the optimal solution is not affected, and the value of any solution is lowered by the subtracted value. The required information is a labeling $l(x)$

of vertices such that if the labels of rows and columns are subtracted from rows and columns, then the reduced matrix $C' : C'(x, y) = C(x, y) - l(x) - l(y)$ has all elements non-positive and the selected elements are zero. With that information, it is clear that the solution is optimal, since the reduced value of the solution is zero and it cannot be positive.

We have implemented the shortest augmenting path algorithm, which is one of the best algorithms for the assignment problem [7]. The result of the algorithm is an optimal solution with a labeling that is the certificate of the optimality. The algorithm start with an initial labeling and an initial matching, which is usually not perfect. Then it repeats the same operation that increases the size of matching by one until the matching is perfect. Several invariants are satisfied after the initialization and after every iteration:

1. The selected elements form a matching.
2. The reduced costs are non-positive.
3. The reduced costs of elements in the current matching are zero.

When the algorithm finishes, the matching is perfect. The invariants ensure that it is optimal. We only need to explain how the initial solution is computed and how the update procedure works.

The simplest way to make the initial solution is to set the row labels to the maximal value of the row, the column labels to zero and start with the empty matching. However, it is better to start with a nonempty matching so that fewer iterations must be performed. To get a better initial solution, the column labels are set to the highest value that does not break the second invariant. As a result, there is at least one element with zero reduced cost in each row and in each column. The initial matching is constructed with a greedy algorithm. All edges are enumerated and an edge is greedily added to the matching if it does not violate any invariant (vertices of the edge are unmatched and the reduced cost is zero).

A general way how to increase size of matching is to find an augmenting path. An augmenting path is a non-empty path in the input graph that starts with an unmatched vertex, ends with an unmatched vertex, and every second edge is from the current matching. To increase the size of the matching, the edges of the augmenting path that were in the matching are removed from the matching, and the edges of the augmenting path that were not in the matching are added. A complication is that labels must be updated so that the invariants stay true. An additional restriction must be put on the augmenting path. It has to be the shortest augmenting path. The length of a path is the sum of opposite reduced

costs of edges in the path. An opposite reduced cost is $w(x, y) = -C'(x, y) = l(x) + l(y) - C(x, y)$. The second invariant ensures that the opposite reduce costs are non-negative, so it is possible to use Dijkstra's algorithm to find the shortest path.

Dijkstra's algorithm finds the shortest paths from a vertex to all other vertices. The result consists of a distance array and the shortest path tree. The shortest path tree is defined by an array of pointers (indexes) to a parent vertex. The vertices are marked as open or closed. At the beginning, all vertices are open and they have infinite distance except the source vertex, which has zero distance. If a vertex is closed, its distance is minimal and it will not change. The algorithm repeats the following operations until all vertices are closed: An open vertex with the minimal distance is found and closed. For all vertices, it is checked whether the path through the closed vertex is shorter. (The distance array and the parent array are updated.)

The Dijkstra's algorithm may be terminated earlier. In that case, the shortest path tree is built for closed vertices, and we know that all open vertices have distance higher than or equal to the distance of the vertex that was closed last.

In the shortest augmenting path algorithm, the Dijkstra's algorithm is not run directly on the input graph. Only the column vertices are considered. We may imagine that there is a special vertex for the source (unmatched row), but it is not explicitly represented in the algorithm. The edges (and their weights) from a matched vertex are the same as the edges from the vertex it is matched to. It does not matter how edges from an unmatched vertex are defined, because the algorithm is stopped when an unmatched vertex (column) is chosen to be closed. The shortest path in this graph corresponds to the shortest augmenting path in the original graph.

To update the labels $l(x)$, we need the (partial) shortest path tree that has been computed and the distances to closed vertices (columns). The length of the shortest augmenting path is subtracted from the label of the source row. Furthermore, for each closed pair of vertices, a value is subtracted from the label of the row and the same value is added to the label of the column. The value is equal to the difference of the length of the shortest augmenting path and the length of the path from the source to the vertices[2]. After this update, all invariants are satisfied and all edges on the shortest augmenting path have zero reduced cost; therefore, the path can be used to augment the matching.

Because Dijkstra's algorithm is run for each row that was not in the initial matching, it is called $O(n)$ times. During each run of Dijkstra's algorithm, $O(n)$

[2]The length of the shortest path from the source is the same for both vertices, since the weight of a matching edge is zero.

vertices are closed, and it takes $O(n)$ operations to find the open vertex with minimum distance and update all other distances. Therefore, the total time spent in all calls of Dijkstra's algorithm is $O(n^3)$. The overall run-time of the shortest augmenting path algorithm is also $O(n^3)$ because all other parts of the algorithm take $O(n^2)$. Since Dijkstra's algorithm is the bottleneck, it should be implemented efficiently.

We can take advantage of the fact that we know $n = 16$ and make a special algorithm for this size of input. Furthermore, we know the input values are small integers that fit into short integer type (two bytes). In our implementation, the algorithm operates with vectors of eight values (128 bit). If 256-bit vectors were available, a whole row could be represented as a single vector. Some advanced vector operations are required. For instance, we need to do vector conditional store. Furthermore, we must pay attention to the possibility of overflow. In some places, a saturation operation were needed to ensure that there will be no error because of overflow provided that the input is in the range $[0, 32767]$.

4.6. Other features

The theory of Arimaa has already brought some advanced strategic concepts. One of the basic strategic patterns is the elephant blockade. When an elephant cannot move, the opposing elephant can take control of the board since all other enemies are weaker. Another important pattern, that is more common, is a hostage. The term hostage is used for many situations. The basic characteristic of a hostage is that a piece is frozen two squares from a trap and it is threatened to be captured in four steps. There are two advantages for the player that holds a hostage. The first is that the capture threat may force the opposing elephant to defend the trap. The second is that the hostage cannot move. The question is whether it is necessary to recognize hostages as a special pattern. Maybe, it is sufficient to only evaluate capture threats and mobility. The features should be as general as possible, and therefore we have decided to focus on mobility.

For each piece, the mobility value is computed. Mobility is an integer from the range $[0, 6000]$. Its computation is simple. The distance matrices are transformed with the function that is defined by the "mobility" column of Table 4.2 and the elements of the transformed matrix are summed. In the earlier stage of development, the mobility was equal to the number of squares that the piece can reach in one turn. We have changed the implementation, since the current is more general. It is possible to get the initial behavior by changing coefficients of the transform function.

A complement to the mobility is negative mobility. Whereas the mobility

considers how the piece can move by the player it belongs to, the negative mobility considers how the piece can be moved by the opponent. For each square where the piece can be moved, one or two points are added to the negative mobility value. Two points are added if there is no friendly piece on adjacent squares, because that means that the piece will be frozen. The range of the negative mobility is $[0, 24]$.

The location of the piece is also used in the evaluation. It would not be a good idea to add directly the square number as another type of a feature, because its evaluation would not be smooth. We want the dependence of evaluation on the feature value to be simple. The square number is divided into two values. The first value is called advancement and it is basically the row number except that it is reversed for Silver and it starts with 0 (7 is always the goal row). The second value is centralization. Since Arimaa is completely symmetric with respect to the left-right reflection, it is sufficient to have four values for horizontal location. There should be no difference in the evaluation of a piece at a1 and in the evaluation of the same piece at h1. The important information is the distance from the center. The centralization value is 0 for columns a, h and it is 3 for columns d, e.

The support and fear values are also taken into account. In both cases, the value of the square where the piece is located is taken. We expect it to be better to have a higher value of support and a lower value of fear.

4.7. Representation of features

The output of the second layer is a list of parametrized features. A parametrized feature is defined by its name and parameter list. A parameter is defined by following properties:

1. Name
2. Is quantitative? (True or False)
3. Maximal value.

The name is not important for the evaluation, but it may be used to produce log messages, for instance. Parameters are always non-negative integers. The difference between quantitative and qualitative parameters is in semantics. If a parameter is quantitative, then a small change of its value should result in a small change of the evaluation. In other words, there should be only a little difference in the evaluation of close values. On contrary, the numerical closeness of qualitative parameters does not mean anything. For instance, if a square

number	name	maximal value	qualitative
1	moving player	1	False
2	strength	15	True
3	mobility	6000	True
4	negative mobility	24	True
5	fear	255	True
6	support	255	True
7	danger	255	True
8	advancement	7	True
9	centralization	3	True

Table 4.3.: Parameters of the non-rabbit feature.

number was a parameter, then it would be qualitative, since there it is discontinuous. Whereas the advancement value, which is described in Section 4.6, can be considered quantitative. We could say continuous instead of quantitative and discontinuous instead of qualitative, but that could be confusing, since we are not talking about the actual mathematical definition of continuous functions. In Section 5, it is described why qualitative features are bad. The last property of a parameter is the maximal value it can take. Since the minimal value is always zero, the maximal value defines the domain of the parameter.

There is a method of the `Evaluation` class that returns the complete description of features. Furthermore, the type of the list of features is not dependent on the features. Therefore, it is possible to write functions that work with the output of this layer and that are not restricted to a particular set of parametrized features. For instance, there is a function that can store a feature list to a file. Thanks to this general interface, we will not have to modify the function if the set of features is changed. We expect that the feature set will be changing a lot. The evaluation function is therefore designed so that it can be easily modified.

In the current evaluation function, there are three types of features. The first type is called non-rabbit. One instance of this feature is added for each non-rabbit piece on the board. It has 9 parameters, which are described in Table 4.3. There is a similar feature for rabbits. Table 4.4 shows its parameters. The only difference is that rabbits do not have the strength parameter, but they have the goal prospects parameter. The last feature is added for each trap; therefore it is called trap. The only two parameters of the trap feature quantify the trap protection for Gold and Silver, which is taken from the algorithm that computes the capture threat.

number	name	maximal value	qualitative
1	moving player	1	False
2	mobility	6000	True
3	negative mobility	24	True
4	fear	255	True
5	support	255	True
6	danger	255	True
7	goal prospects	255	True
8	advancement	7	True
9	centralization	3	True

Table 4.4.: Parameters of the rabbit feature.

number	name	maximal value	quantitative
1	gold protection	1020	True
2	silver protection	1020	True

Table 4.5.: Parameters of the trap feature.

4.8. Shortcomings

The second layer is probably the most complex part of the evaluation function. The first part was very hard to implement, but we knew what the result should be. On contrary, we can only guess what the right set of features is. The primary aim was therefore to develop a solid base that can be easily extended. It requires long time to tune the evaluation function. Unfortunately, we did not have much time for experimenting with features, because the implementation of the other parts of the evaluation function, particularly the mobility analysis, was very difficult and time-consuming. We have many ideas how the feature set could be extended. Some of them will be discussed in this section.

The concept of fear is missing some complex relations between pieces. It ignores that a stronger piece might have other duties than chasing a given piece. For instance, if an elephant is forced to protect a trap where a camel is threatened, then the elephant is less dangerous for other pieces. In general, the objectives of a piece should be considered. If we knew that the objective of the elephant is to threaten the camel, then it would be reasonable to take the distance of the camel from the elephant into account in the static evaluation function. A similar problem occurred in the computation of trap protection. The assignment of defensive squares to pieces might be understood as an assignment of objectives to pieces, with the restriction that only the trap protection is considered. Even with that simplification, it was a difficult problem that was solved only approximately. There are many reasons why it would be difficult to extend the method used for the trap protection to other types of objectives. First, it is hard to compare

different types of objectives. Even the top human players might not be sure what a piece should do in some situations. Second, a piece may have more than one objective. For instance, it can protect a trap and threaten to capture an enemy at the same time. However, it is possible to perform two tasks at once only if they are located in the same area.

Trap control is another field that should be studied more. The trap feature is too simple. It should evaluate who is more powerful around a given trap and how many steps would each player need to take control over the trap. To control a trap, a player usually needs to place three pieces at three defensive squares around the trap. It would be possible to estimate the minimal number of steps that a player needs to achieve that control. Another very important factor is who has the strongest piece in the area.

Finally, many new features could be added. Although it is probably not necessary to have a special feature for hostages and frames if all consequences of these patterns are handled correctly, it might still be better to include these patterns explicitly at this stage. Another important pattern that should be probably handled specially is a frame. A frame is a situation when a piece is in a trap, cannot escape, and has only one guard. If the only guard leaves, the framed piece is immediately captured. If the defending piece is weak, then the framed piece will be probably captured soon. If it is an elephant, then the framed piece cannot be captured, but the elephant is immobilized, which is a big handicap. From the perspective of the second player, the frame has the disadvantage that it usually requires many pieces because three squares must be occupied by a piece which cannot be dislodged by the framed piece. It is usually necessary to relocate ("rotate") pieces to take advantage of the frame. To sum up, it is not easy to decide whether a frame is an advantage or not, and how big the advantage is.

5. Machine learning

5.1. Learning methods

The last step of static evaluation is to transform all features to one value, which corresponds to the probability of winning. The zero value should mean that both players have 50 percent chance of winning provided that their game strength is the same; neither side has any advantage. The higher the evaluation value is, the higher the advantage for Gold is. On contrary, a negative value means that Silver has an advantage. Our aim is to find such an evaluation automatically using machine learning.

There are three basic types of machine learning.

- Supervised learning
- Unsupervised learning
- Reinforcement learning

Supervised learning is learning from examples. In a typical task of supervised learning, we are given training data that consists of pairs of values (x_i, y_i). The first value of a pair might be a single value, but usually it is a vector. It describes an observation. The second value is usually a scalar value, that describes the right reaction to the observation. In our case, x_i would represent features and y_i would be the static evaluation. The training data would consist of evaluated positions. The main problem is how to obtain such evaluated positions. We can access the database of all games that were played on the Arimaa server to get the positions, but there is no evaluation associated with them.

The evaluation cannot be easily computed. If we had a function that can accurately evaluate positions, then we would not need the static evaluation function. Another option would be to let an expert evaluate positions, but that would be very time-consuming. Furthermore, it is hard for humans to give an accurate numeric evaluation of a position.

The difference between supervised learning and unsupervised learning is that there are no labels for the observations in the unsupervised learning. Training data consists only of observations. An algorithm for unsupervised learning must be able to derive output only from those observations. The most common form of

unsupervised learning is clustering, which divides the data into groups according to their similarity. Unsupervised learning is not suitable for learning a static evaluation function, but it could be used to get an insight into the structure of the features. For instance, we could use clustering to find the most common feature arguments. It could also be used as a part of some other learning methods.

Reinforcement learning can be put somewhere between supervised learning and unsupervised learning. There are no learning data in reinforcement learning. Instead, the learning agent can interact with the environment and get a feedback, which indicates whether agent's actions were appropriate or not. In the case of playing Arimaa, the agent is the bot that plays against another player. An action is a played move and the feedback is the result of the game. It is necessary to play many games to learn something useful. The opponent can be a different program, human, or the same program. The advantage of self-play is that the opponent is evolving together with the agent. If the opponent is fixed, then there is a risk that the agent learns to play well against a particular opponent, but it plays poorly against a different opponent. There are many algorithms for reinforcement learning. The simplest algorithm adjusts the evaluation of all positions in a game towards the final reward. Temporal difference learning uses a more sophisticated formula, which takes into account the difference between successive estimates.

Reinforcement learning, especially TD(λ)[1], is probably the most popular method for learning a static evaluation function. It gained its popularity in 1992, when G. Tesauro successfully used TD(λ) to learn a static evaluation function for Backgammon[2] [8, 9]. His program, which is called TD-Gammon, was able to beat even the best human players. The same method has been applied to many games. Unfortunately, it was not always so successful. Haizhi Zhong tried to tune an evaluation function for Arimaa with TD(λ), but no improvement was achieved. David J. Wu examined four different reinforcement learning algorithms, including temporal difference learning [10]. Although the results seemed promising, the learned evaluation function was worse than the hand-tuned one. Furthermore, the learning took a long time and it was not reliable.

Since previous research shows that reinforcement learning is probably not so good for Arimaa as for some other games, we have focused on supervised learning instead. There are several reasons why supervised learning should be better for Arimaa. The first is that it should generally be easier to learn with a teacher than without. That is only an intuitive argument, but it is based on the experiences from the real life. The second reason is that we can access the database of all

[1] TD(λ) is a variant of temporal difference learning.

[2] Backgammon is an old two-player board game. Unlike Arimaa, Backgammon involves chance, because possible moves depend on the roll of dice.

games played at the Arimaa server. There are thousands of high quality games to learn from. We should take advantage of that data. The most important reason is that human players are very good at position evaluation, better than bots. It is easier to learn from better players. Finally, the algorithm that we have developed is suitable for games with a high branching factor and a strong emphasis on accurate evaluation, which is the case of Arimaa.

Traditional supervised learning requires positions to be labeled with the correct evaluation. In Section 5.4, we present the LP-learning algorithm, which does not have this requirement. We show that a static evaluation function can be learned from records of games. By playing a move, an expert expresses that one position is better than other, and this information can be used to learn a static evaluation function. It is not a new idea. The same concept, which is called comparison training, was described by Paul E. Utgoff and Jeffery A. Clouse in 1991 [11]. Recently, it was successfully used to train Shogi[3] evaluation function and it was the topic of a few research papers [12, 13, 14]. Although the basic idea of LP-learning is not new, it is different from all similar algorithms that we know. It was designed from scratch specifically for Arimaa[4]. The main difference is that LP-learning does not perform minimax search. That simplifies the problem so that the solution can be efficiently computed.

5.2. Representation of functions

The input of the last layer of the static evaluation function for a position p is a list of parametrized features $L(p)$. Each feature instance $l \in L(p)$ has a name $n(l)$ and a vector of arguments $v(l)$. The final evaluation $f(p)$ is computed as the sum of evaluations of the features:

$$f(p) = \sum_{l \in L(p)} f_{n(l)}(v(l))$$

There is a function f_i for each type of feature i. These functions can be arbitrary, so it is possible to reflect nonlinear dependence between parameters of a feature. On contrary, dependence between features cannot be represented in this model. It is a compromise between simplicity and generality. If the model was too general, then it would be hard to learn.

Since feature parameters are integers from a finite range, the whole domain of feature parameters is also finite. Therefore, it would be possible to represent any function with a table. However, the table would be too large. A more

[3] Shogi is a two-player board game similar to chess, which is very popular in Japan.
[4] Although it was designed for Arimaa, it might be useful also in other domains.

compact representation is needed. It is easier to find a compact representation for quantitative parameters because they are more regular. For this reason, the qualitative parameters are extracted, and there is a function for each combination of values of qualitative parameters. In other words, the table approach is used only for qualitative parameters. Consequently, features should not have many qualitative parameters, and their domain should be as small as possible. Let $v'(l)$ denote the vector of quantitative arguments of a feature l and $v''(l)$ denote the vector of qualitative arguments, then

$$f(p) = \sum_{l \in L(p)} f_{n(l),v''(l)}(v'(l))$$

Functions $f_{i,j}$ have only quantitative arguments. The arguments are treated as real numbers and scaled to the interval $[0, 1]$. Thus, the problem is how to represent a function $f_{i,j} : [0, 1]^{m(i)} \to \mathbb{R}$, where $m(i)$ is the number of quantitative parameters of the feature i. At first, it might seem that it is even harder problem than the original one because the domain is infinite. However, it is not a problem in practice, and we do not have to deal with different domains of parameters.

The key advantage of qualitative features is that they enable interpolation. For instance, if we knew that $g(1) = 4$, $g(2) = 6$ and $g(4) = 10$, we could guess that $g(3) = 8$. On contrary, if we knew that $g'(\text{blue}) = 3$, $g'(\text{red}) = 6$ and $g'(\text{green}) = 5$, then we would have no clue what $g'(\text{black})$ should be, except that it would probably be in a similar range. Interpolation is a case of generalization, which is the fundamental part of supervised learning. Supervised learning would not work without generalization, because the learned function must be able to correctly evaluate inputs that were not in training data. When choosing a class of functions $F_{i,j}$ from which $f_{i,j}$ is taken, a compromise must be found between simplicity and precision. If $F_{i,j}$ was too restrictive, then the evaluation would be inaccurate even for training data. On contrary, if $F_{i,j}$ was too extensive, then it would generalize poorly because over-fitting would occur.

Another requirement that we have on every $F_{i,j}$ is that every function $f_{i,j} \in F_{i,j}$ can be written as

$$f_{i,j}(x) = \sum_{k=0}^{C_{i,j}-1} (w_{i,j,k} h_{i,j,k}(x)) = w_{i,j} \cdot h_{i,j}(x)$$

where the constant $C_{i,j}$ and the functions $h_{i,j,k}$ are fixed for all $f_{i,j} \in F_{i,j}$. The function $f_{i,j}$ is defined by a vector of weights $w_{i,j}$. The function $h_{i,j}$, which can be arbitrary, maps the input space to another space with dimension $C_{i,j}$. The k-th element of $h_{i,j}(x)$ is $h_{i,j,k}(x)$.

This requirement is necessary because many machine learning algorithms, in-

cluding LP-learning, can learn only an evaluation function that is a dot product of a weight vector and a feature vector. If every function $f_{i,j}$ can be written as a dot product $w_{i,j} \cdot h_{i,j}(x)$, then the whole evaluation function $f(p)$ can be also written as a dot product $w \cdot h(p)$. The final weight vector w is the concatenation of all weight vectors $w_{i,j}$, and

$$h(p) = \sum_{l \in L(p)} h'_{n(l),v''(l)}(v'(l))$$

where $h'_{i,j}(x)$ is $h_{i,j}(x)$ extended with zeros to the size of w, so that $f_{i,j}(x) = w \cdot h'_{i,j}(x)$.

At first, the requirement might seem limiting, but actually many suitable function representations can be used. The only example of a function representation that we were considering to use and that does not fulfill this requirement is the multilayer perceptron. It would be necessary to use a different learning algorithm if we wanted to use it. Nevertheless, many other good options remain. We have considered three representations:

1. Interpolation on a regular grid.
2. Polynomial function.
3. Radial basis function network.

In the first method, the weights are associated with points that form a regular grid. The function value is interpolated from the corner-point of the cell that the input lies in. It is equal to the weighted average of values of corner-point, thus it can be written as a dot product. An advantage of this method is that it can be easily computed. It can approximate any continuous function. The precision depends on the density of the grid. The main drawback of this method is that the number of weights is exponential in the number of feature parameters.

The second alternative is a polynomial with degree d. Each $h_{i,j,k}$ corresponds to a different term with degree less than or equal to d. For instance, if $d = 2$ and there are two parameters x_1 and x_2,then

$$f(x_1, x_2) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2 + w_5 x_1 x_2$$

The number of weights $C_{i,j}$ is equal to the number of terms, which can be computed as a number of d-combinations with repetition from $n + 1$ elements, where n is the number of (quantitative) parameters of feature i.

$$C_{i,j} = \binom{n+d}{d} = \frac{(n+d)!}{d!n!} = \frac{(n+1)(n+2)\cdots(n+d)}{d!}$$

For fixed d, the number of weights is polynomial in the number of parameters, which is a significant improvement in comparison to the exponential dependence of the interpolation method. However, there are also some drawbacks. Since there are fewer parameters, the learned function might be inaccurate because complex relations between parameters cannot be captured. Another disadvantage of the polynomial representation is that many arithmetic operations are needed to compute the evaluation.

Radial basis function network is a type of artificial neural network. It has one hidden layer with $C_{i,j}$ neurons. Each neuron in the hidden layer is associated with a point from the input space, which is called center. The output of a neuron in hidden layer is a function of the distance of its center $z_{i,j,k}$ from an input vector. The final output is a linear combination of outputs of neurons from the hidden layer.

$$f_{i,j}(x) = \sum_{k=0}^{C_{i,j}-1} w_{i,j,k}\psi(\|x - z_{i,j,k}\|)$$

The function ψ is usually Gaussian.

$$\psi(d) = e^{-\beta d^2}$$

If the centers were parameters that should be learned, then it would not be possible to write the function as a dot product. However, the centers can be learned with unsupervised learning before the main learning phase. For instance, we can find clusters in the training data and represent each cluster with its center as a neuron in the hidden layer. After this first phase, the centers are fixed and only the linear coefficients have to be learned.

Radial basis function network is similar to the interpolation in a grid. The main difference is that the control points can be placed arbitrary. The function can be more detailed in the areas where data occur with higher probability. The cost of this flexibility is higher computational complexity.

From these three options, we have chosen to implement the polynomial representation. The reason is that it needs fewer weights that the interpolation in a grid, and it can be evaluated faster than the radial basis function network. It would not be hard to add the other methods in the future.

To overcome the problem of high computational complexity, we have developed a method of using vector instruction to make a fast implementation of a polynomial function. The problem for which we were trying to find the most efficient solution is how to compute all terms with a given degree d. All terms with the degree less than or equal to d can be computed by calling the function for $1, 2, \ldots, d$.

The straightforward approach of computing each term separately is inefficient for two reasons. The first is that the same computations are done many times. For instance, if $d = 3$ and there are 10 variables, then there are 10 terms that include x_1^2. It is unnecessary to compute x_1^2 10 times. This flaw can be fixed with the backtracking algorithm. The process of computation can be imagined as a tree search. The state is a term, initially 1. An action adds a variable to the term. Only the variables with the index higher than or equal to the index of the variable that was added last can be added. When the depth d is reached or when all actions from the current state have been examined, we backtrack to the previous state. This method can solve the first problem. The second problem is that we are not taking advantage of vector instructions, which modern processors offer.

The proposed method works with vectors of the same size as the input vector (n). We can imagine that the variables are arranged in a circle. The successor of x_i is $x_{(i+1) \bmod n}$. Any term can be expressed as a pair (v, s), where v is an index of a variable that occurs in the term and s is a sequence of d non-negative integers such that the sum of all elements is equal to n. Such (v, s) pair will be called t-pair and the sequence s will be called t-sequence. The values in a t-sequence determine how many steps we should move in a circle of variables to get to the next variable of the term. Since the sum of values is n, the last step completes the circle at v The variable v is not added to the term in the last step. Let a_i denote the index of the i-th variable in a term[5]. Then the term can be computed from a t-pair (v, s) by a recursive formula:

$$\begin{aligned} a_0 &= v \\ a_i &= (a_{i-1} + s_{i-1}) \bmod n, d > i > 0 \end{aligned}$$

Furthermore, it is true that

$$a_i = (v + \sum_{j=0}^{i-1} s_j) \bmod n$$

and

$$a_0 = (a_{d-1} + s_{d-1}) \bmod n = v$$

Each t-pair determines a term and every term can be expressed by a t-pair. However, there are d different t-pairs for a term. If every variable has degree one in a term, then there are d options for the first variable v and each of them leads

[5] We assume that a term is represented as a sequence of variables.

x_5^3	$x_1x_4x_5$	$x_2^2x_6$
(5,(0,0,8))	(1,(3,1,4))	(2,(0,4,4))
(5,(0,8,0)	(4,(1,4,3))	(2,(4,4,0))
(5,(8,0,0)	(5,(4,3,1))	(6,(4,0,4))

Table 5.1.: Example of terms and corresponding t-pairs. (There are 8 variable x_0,...,x_7)

to a different t-pair. If a variable x has a degree b in a term, then there are b t-pairs with $v = x$ because the occurrences can be distributed in b ways to the beginning and to the end of the t-sequence. Table 5.1 shows an example of three terms and corresponding t-pairs for $n = 8$ and $d = 3$.

An interesting fact, that can be observed in Table 5.1, is that the t-sequences in all possible t-pairs for a term are the same except that they are rotated. That is not a coincidence. If (v, s) generates a term t, then (v', s')

$$\begin{aligned} v' &= (v + s_0) \bmod n \\ s' &= (s_1, s_2, \ldots, s_{d-2}, s_{d-1}, s_0) \end{aligned}$$

also generates t because

$$a'_i = a_{(i+1) \bmod n}$$

and the order of the variables in a term does not matter. All possible t-pairs that describe a term can be generated in this way from a single t-pair.

Let S is the set of all possible t-sequences for n variables and degree d.

$$S = \{(s_i)_{i=0}^{d-1} | \forall i : s_i \in \{0, 1, \ldots, n\}, \sum_{i=0}^{d-1} s_i = n\}$$

We can define equivalence relation $\sim$ on S. Two t-sequences are equivalent, if one is a rotation of the other.

$$s \sim s' \Leftrightarrow \exists k \forall i : s_i = s'_{(i+k) \bmod d}$$

Next, we select $S' \subseteq S$ so that S' contains just one t-sequence from each equivalence class. It is not very important which t-sequence is chosen, but some choices might lead to slightly better code. A natural rule is to choose the lexicographically minimal sequence. Another good rule, that may be better in some sense, is to choose the maximal sequence in the reverse lexicographic order.

The algorithm computes a term for each t-pair from $T = \{0, 1, \ldots, n-1\} \times S'$.

All terms will be computed, because there is a t-pair for every term in T. However, some terms might be computed more than once. It happens when there are two different t-pairs with the same t-sequence for a term. There are d t-pairs for a term and all t-sequences of these pairs are the rotations of one t-sequence, thus there might be two different t-pairs with the same t-sequence only if there is a nontrivial rotation that is identity. For instance, if $n = 8$ and $d = 2$, then term x_0x_4 can be expressed either as $(0, (4, 4))$ or $(4, (4, 4))$. The same is true for x_1x_5, x_2x_6 and x_3x_7. The t-sequence $(4, 4)$ generates only 4 terms.

This problem occurs only if $\gcd(n, d) > 1$. If $\gcd(n, d) = 1$, then a t-sequence cannot be identical with its nontrivial rotation. Proof by contradiction: We assume that $\gcd(n, d) = 1$ and there is a nontrivial r rotation of a t-sequence s that is an identity ($\forall i : s_i = s_{(i+r) \bmod d}$). Let $G_i = \{(i + kr) \bmod d | k \in \mathbb{N}\}$, then $\exists c \forall i : |G_i| = c$ because $G_i = \{(a + i) \bmod d | a \in G_0\}$ and $a \mapsto (a + i) \bmod d$ is a bijection on $\{0, 1, \ldots, d - 1\}$. Since the rotation is nontrivial ($0 < r < d$), there must be at least two elements in G_i; thus $c > 1$. The set $\{0, 1, \ldots, d - 1\}$ can be divided into groups with c elements; thus c must divide d. Since s_j is the same number for all $j \in G_i$ and $n = \sum s_i$, c must also divide n. That is a contradiction because we get $\gcd(n, d) \geq c > 1$.

One step of the algorithm consists of computing the terms for one t-sequence and all variables. This computation can be done in parallel using SIMD instructions. All that we need is an implementation of vectors with size n that supports following operations:

1. Load/Store operations.
2. Element-wise multiplication
3. Rotations of a vector

There are two variants of such a function. The first variant takes a vector of coefficients and values of the variables as the input and evaluates the polynomial. It is used to evaluate a feature (it computes $f_{i,j}(x)$). The second variant does not evaluate the polynomial, but it outputs all terms (not multiplied by coefficients) as a vector. It is used to transform feature parameters x to $h_{i,j}(x)$. The first function could be implemented with the second function and a dot product, as $f_{i,j}(x) = w_{i,j} \cdot h_{i,j}(x)$, but it is more efficient to do it at once because then it is not necessary to store the intermediate result (terms) to the memory.

The functions for $n \in \{4, 8\}$ and $d \in \{1, 2, 3, 4\}$ were implemented. Listing 5.1 shows the code for $n = 4$ and $d = 3$. First, the variable values are loaded into the vector r0. Next, the rotations are computed (r1, r2, r3). Finally, all 20 terms are computed with only 7 vector multiplications. The function was constructed according to these t-sequences:

```
v4sf r0 = v4sf_load_aligned(inBuffer);
v4sf r1 = v4sf_rotate<1>(r0);
v4sf r2 = v4sf_rotate<2>(r0);
v4sf r3 = v4sf_rotate<3>(r0);
v4sf a0 = r0 * r0;
v4sf a1 = r0 * r1;
v4sf_store_aligned(outBuffer  , a0 * r0);
v4sf_store_aligned(outBuffer + 4, a0 * r1);
v4sf_store_aligned(outBuffer + 8, a0 * r2);
v4sf_store_aligned(outBuffer + 12, a0 * r3);
v4sf_store_aligned(outBuffer + 16, a1 * r2);
```

Listing 5.1: Computation of all terms with degree 3 for 4 variables.

1. (0,0,4)
2. (0,1,3)
3. (0,2,2)
4. (0,3,1)
5. (1,1,2)

In this example, there are no symmetric t-sequences, because $\gcd(4, 3) = 1$. In some other cases (for instance $n = 4$ and $d = 2$), there are symmetric t-sequences that must be handled specially. It would be harder and less efficient to implement the functions for different values of n because the hardware support only the vectors with size 4 and 8. It would be even harder to write a general function that takes n and d as part of the input. However, as we only need functions for small values of n and d, it is sufficient to implement only the specializations for these small values.

5.3. Training data

Supervised learning need some training data, i.e. games of experts, to learn from. The quality of the data has a strong influence on the quality of the learned static evaluation function.

The database of all games that were played at the Arimaa server is available. It is stored in text files. There is usually one separate file for each month. We used the data from January 2007 to April 2013. The games from the earliest period were omitted because the playing skills of Arimaa players were not as good as they are now. The first line is a header and every other line represents one game.

The data are arranged into columns. There is detailed information about time control, players and the game.

The first task is to parse the database. The `GameArchiveReader` class, which we have implemented, offers a comfortable interface to the database. The game records can be retrieved with the C++11 "for each" loop as the `GameRecord` structure type. The `GameRecord` structure contains all the available information in a parsed format. The data are checked during parsing. If there is an error, then a flag is set, but the process can continue. There are several corrupted data fields in the database, and therefore it was necessary to handle wrong inputs.

Only the high quality games should be used for learning. A procedure that filters games was implemented.

First, the games that do not look normal are removed. Only the games that were naturally finished are further considered. The games that ended because of a time limit or resignation are ignored. The games that are marked as non-rated are ignored too. The length of a game is examined. If only a few moves were played or if the game lasted less than 5 minutes then a game is ignored. If it is too long (more than 400 turns) than it is also ignored. If there is a capture in first two turns after the setup, then it is probably a handicap game and the capture was a deliberate sacrifice. These games are also omitted.

After the initial filtering, each game is rated. Only games with the rating higher than a selected threshold are kept. The rating is based on the rating of the two players[6]. It is computed as

$$\text{game\_rating} = 0.8 * \text{lower\_rating} + 0.2 * \text{higher\_rating} + \text{other\_factors}$$

The lower rating of the players is more important because the evaluation function is learning from both players. The quality of players is important, but even a top player may play poorly if he has little time to think. The time per move is therefore taken into account. The average time in minutes used for one turn t is computed. Then the value $100 \ln(\min(t, 30))$ is added to the rating. In postal games, t can be very high, but the actual thinking time is much shorter; therefore, the value of t is limited to 30 minutes. If the average thinking time is one minute, then the expression is zero. Doubling the thinking time brings approximately 70 points. Postal games receive the bonus of approximately 340 points.

A penalty of 30 points is given if one of the players is bot and penalty of 60 points is given if both players are bots. We slightly prefer human games because

[6]The rating of a player is automatically computed for every player on the Arimaa server, and it is included in the description of a game in the game archive. Beginners start at 1400 and the rating of top players is around 2500.

minimal rating	number of games
2400	145
2300	391
2200	700
2100	1644
2000	3355
1900	6488
1800	11392

Table 5.2.: Number of expert games with a high rating.

we assume that humans are better at positional evaluation than bots even if they have the same rating. Furthermore, bots may tend to repeat the same mistakes and this mistake might be transferred to the evaluation function in the learning process. The penalty is rather small because it is not certain that bot games are worse for learning. Maybe, the opposite is true.

The last component of the game rating is the penalty for games that were played long time ago. Twenty points are subtracted for every year. More recent games are preferred because the knowledge of Arimaa and the corresponding playing strength is consistently improving.

Table 5.2 shows how many games remain after the filtering. It depends on the minimal rating that is required. Generally, there are thousands of high-quality games available. We could get even more training data if games that terminated with resignation (or because of a time limit) were added, but then we would probably need a better heuristic for choosing good games.

5.4. LP-learning

5.4.1. Overview

LP-learning is a supervised learning algorithm that uses linear programming to find weights that fit the training data. It was designed from scratch, but some similar algorithms have been already successfully used. Relationship to other learning algorithms was discussed in Section 5.1. We assume that the evaluation function is in the form of dot product of weights w and features $h(p)$.

$$f(p) = w \cdot h(p)$$

In Section 5.2, we have shown how to transform the evaluation function to this form.

LP-learning is not one algorithm, but it is a family of algorithms that differ in the definition of the loss function and in the implementation details. The loss

function quantifies how well an evaluation function fits training data. In Section 5.4.2, we show how the loss function can be defined.

LP-learning is based on linear programming. In Section 5.4.3, we give a brief introduction to linear programming and we present the software that we have used for solving linear programs.

Two algorithms were implemented. The first, which is described in Section 5.4.4, is simpler and faster. It can compute reasonable weights in a few minutes. The second algorithm, which described in Section 5.4.5, is more complicated and slower, but it should produce better results. In addition, an algorithm that uses quadratic programming instead of the linear programming is presented in Section 5.5. The algorithms are compared in Section 5.6.

5.4.2. Loss function

Training data for supervised learning usually consists of observations that are labeled with the correct output. However, there is another concept of supervised learning that is more general. In supervised learning, we are given some data and we try to find a function that fits the data. All that we need is a function that measures how good a candidate solution is with respect to the training data. That function is called loss function. If we had the correct labels, then the loss function might be the sum of quadratic differences between the correct labels and the outputs of the model. Although the correct labels are not available, it is still possible to define a reasonable loss function for learning an Arimaa evaluation function.

Supposing that we have an evaluation function f and a set of expert games, how can we measure the quality of f with respect to the games? Answering this question was a fundamental step in developing LP-learning. We have found that there are many possibilities and it is hard to tell which is the best.

One thing that can be considered is the relation between the static evaluation and the outcomes of the games. Since a positive value of the evaluation should mean that Gold is winning, the evaluation of positions from a game that Gold won should be positive, and the evaluation of positions from a game that Silver won should be negative. Although it is a reasonable rule, it is not always true. One player may lead throughout the game, and then make a fatal mistake that results in a loss at the end of the game. We can say nothing about a particular position, but we can use some statistics that take into account many positions because it will reduce the impact of random noise. For instance, we can sum evaluation of all positions from games that Gold won and subtract the evaluation of all positions from games that Silver won. Let $r(p)$ is 1 if the game that the position p is from was won by Gold and -1 if it was won by Silver. Then the loss

L_{outcome}, which measures how well the outcome of the game fits the evaluation, can be defined:

$$L_{\text{outcome}} = -\sum_{\text{position}p} r(p)f(p)$$

There is the negative sign before the sum because we want the sum to be high. This loss function is far from being perfect. One problem is that it might be dominated by a few extreme values. If the static evaluation function is able to say that a game is won for one of the players, then it may evaluate it with extremely high (low) value. Another problem is that there is no scale. If the values of static evaluation are doubled, the loss function is doubled too, although it is still the same static evaluation, because only the relative order of positions matter. The problem of scaling evaluation function is strongly connected with the definition of loss function.

There are better ways to define loss function. In all remaining, cases we focus on a particular position p and we give a formula for the loss of that position. The total loss is the sum of losses for all training positions. We will assume that Gold is on the move in p. The situation is analogous, if Silver is on the move. It is usually only necessary to change the sign.

Although we suppose that the training data are not labeled with a correct output, there might be some cases when we know the correct value $e(p)$. For instance, it is possible to compute the game-theoretical value of some positions from the end of the game. In these cases, the loss may be defined as $|f(p) - e(p)|$. If we know that a position is won for Gold, then it is probably not a problem if the actual evaluation $f(p)$ is higher than $e(p)$, because that only means that the evaluation function is even more convinced that the position is won for Gold. Therefore, it would be better to define the loss function as $\max(0, e(p) - f(p))$. This loss function is not currently used in the algorithm, but it can be added in the future.

How can the loss function be defined if there is no label for p? Figure 5.1 illustrates the situation. The current position p is represented with a full black circle. There are many possible moves from the position p. One of them is the move that was played by an expert. It is marked with the green color in the figure and it is called "played". Furthermore there is a move that leads to the position with the best evaluation. It is marked with the red color and it is called "best". The average evaluation of a position after a move is also represented in the figure because it is very important. We are interested in differences between evaluation of these significant positions (and the average, which can be understood as a pseudo-position):

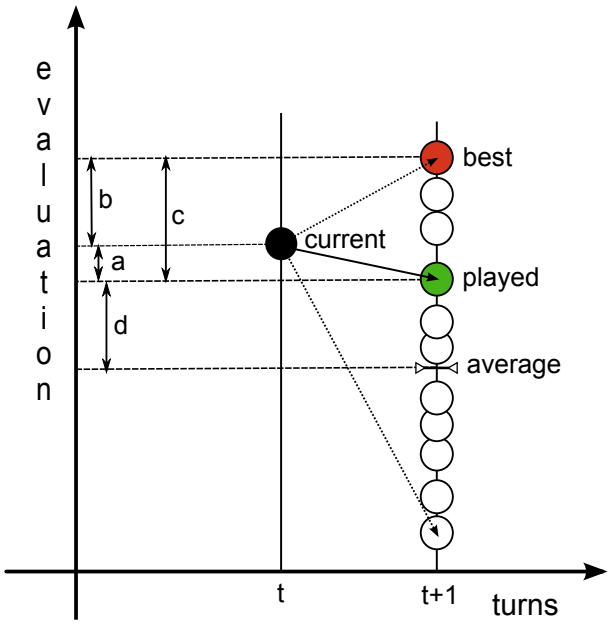


Figure 5.1.: Loss function for a position

$$
\begin{aligned}
a &= |f(\text{current}) - f(\text{played})| \\
b &= |f(\text{current}) - f(\text{best})| \\
c &= f(\text{best}) - f(\text{played}) \\
d &= f(\text{played}) - \text{average}
\end{aligned}
$$

Supposing that an expert has chosen the objectively best move, the evaluation of p should be the same as the evaluation of the position after the played move. The evaluation of p should not be lower, because if there is a move that leads to a good position, then the current position is good too. The evaluation of p also should not be higher, because if the objectively best move leads to a bad position, then the current position is bad too. The conclusion is that a should be small for a good evaluation function.

In this case, our requirements on an evaluation function might be stricter than it is necessary. In the standard alpha-beta algorithm, all evaluated positions are in the same depth, and therefore the moving player is also the same. If we add an arbitrary constant to the evaluation of all positions where Gold is on the move, then the result of the alpha beta algorithm is not affected. Accordingly, it might seem that there is no reason that the evaluation of two subsequent positions should be similar. Actually, many evaluation functions ignore this condition. For instance, a basic static evaluation function that computes the evaluation as a sum of constant values for each piece systematically underestimates the player

who is on the move because it does not take into account which player is moving, and it is usually an advantage to be on the move. Nevertheless, it is probably a good idea to try to achieve this consistency. If a search algorithm that searches deeper in some branches is used, then positions with a different moving player are compared, and the consistency condition should be satisfied. Furthermore, it makes the learning process easier because we can use it in the loss function.

The value b should be small for the same reason as a. The only difference is that we use the best move according to the evaluation function instead of the best move according to an expert. It has the advantage that it is not affected by poor moves of experts. Furthermore, there is a strong evidence that an evaluation function with low b values should be good. If b is zero for all positions and the evaluation of decided positions is correct, then the evaluation function is perfect because it is equal to the minimax value. Despite this evidence, it is not clear that b is a better measure of evaluation function quality than a. Since the move that was chosen by an expert is usually different[7] and it should be the objectively best move, we can deduce that the succeeding position with the best evaluation is probably overrated. If it is really overrated, then the evaluation of p should be lower[8], otherwise it would be overrated too.

The difference between evaluation of position after the actually played move and the best evaluation is equal c. It should be zero because the move played by an expert is supposed to be the best move. Since there are usually many possible moves, there might be many moves with similar quality as the played move. Therefore, it is not wrong if another move gets the same evaluation as expert's move, but no move should be significantly better, so c should be close to zero.

The values a, b and c can be used to measure the quality of a static evaluation function, but they are relative. They have meaning only if they are compared with another value because they depend on the scale of the evaluation function. Zero evaluation function, which is a static evaluation function that evaluates every position to zero, would be considered perfect with respect to these characteristics because the values would be zero. A scale value s must be defined. It should measure the amplitude of a static evaluation function. The incorrectness of a static evaluation function can be expressed as the quotient of a characteristic (a, b, c) and the scale value s.

We can use d as a scale. It is equal to the difference between played move and

[7] If the move chosen by an expert is the same as the best move according to the evaluation function, then $a = b$ and it does not matter which of these two values we use.

[8] We are still supposing that Gold is on the move in p. If Silver is on the move, the best move is the move that leads to a position with lowest evaluation and the evaluation of the best move is not too high, but too low.

average evaluation of all moves. We assume that the evaluation of expert's move is higher than the average evaluation. If it is not true, then either the evaluation function or expert's move is wrong; this case must be handled specially because negative scale value does not make sense. This solution is natural because it can be used in any game. No special knowledge about Arimaa is used. The scale value $s = d$ can be interpreted as the expected loss of playing a random move instead of the best move.

Unfortunately, it is not universal. It might fail in some situations. The problem is that the average does not say anything about the distribution of values. One move may be very bad. For instance, it can be an elephant sacrifice. If that move has extremely low evaluation, then the scale would be high even if the other moves have zero evaluation. If the elephant sacrifice was possible in all training positions and if it was never chosen by an expert, then any evaluation function that evaluates elephants with an extremely high value would be considered very good according to the suggested method. This problem can be fixed in various ways. Values of a static evaluation function may be limited so that extreme values cannot occur. The clearly bad moves can be ignored when computing the average. Another option is to consider only a random sample of moves when computing the average. That could help because it would be unlikely that an extremely bad move would be in the selection if extremely bad moves are rare.

The scale can be defined in many other ways. For instance, L_{outcome}, which was defined in the beginning of this section, could be also used as the scale.

A standard way to scale evaluation functions is to fix the values of pieces. For instance, the documentation of AEI[9] states that an initial rabbit capture should be worth 100. The rule should probably ensure that it is possible to compare evaluation of different bots. A complication is that a static evaluation function is a black box for us. We cannot extract the value of rabbit from it. There might even be no constant that would specify the value of a rabbit. However, the value of rabbit can be estimated in a general way. The evaluation of a position can be compared with the evaluation of the same position with one rabbit removed. The difference is the estimated value of a rabbit. This process should be repeated with different positions, and then the estimates should be averaged to produce the final estimate.

We use d as the scale in our algorithms. When the evaluation function is being learned, a required scale value G is chosen. The loss function should penalize the evaluation function if its scale is lower than G. The loss associated with scale for one position is computed as $\max(0, G - d)$. If d is equal to or greater than G, then no loss is added.

[9] AEI (Arimaa Engine Interface) is a standard protocol for communication with Arimaa bots.

The loss $l(p)$ for a training position p is a weighted sum of all characteristics:

$$l(p) = \alpha a + \beta \max(a, b) + \gamma c + \delta \max(0, G - d)$$

We use $\max(a, b)$ instead of b because if we used b, then it would be harder to optimize the loss function. The values α, β, γ, δ and G are parameters of the LP-learning algorithm.

The final loss function L is the sum of losses $l(p)$ for each training position p plus a regularization term.

$$L = \sum_{\text{training position} p} l(p) + \rho R$$

Regularization term R measures how complex the evaluation function is; ρ is a parameter that determines the importance of regularization term. Generally if we have many explanations, then we should choose the simplest one. This rule is known as Occam's razor. At first, the rule might seem strange, but it is an essential part of any reasoning. People use it all the time. In machine learning, it is useful to prevent over-fitting. If the function is simpler, that there is higher chance that it will evaluate new positions correctly. Since the evaluation function is defined by the vector of weights w, the complexity can me measured as the sum of absolute values of weights.

$$R = \sum_i |w_i|$$

The LP-learning algorithm finds the evaluation function (weight vector w) for which the loss function is minimal. The problem has a global optimum that can be found in polynomial time because it can be written as a linear program.

5.4.3. Linear programming

In this section, we give a brief introduction to linear programming. Linear programming is not a main topic of this thesis; it is only a tool that we use inside learning algorithms. Therefore, we will not go into details. Much of what we use can be found, for instance, in the Wikipedia article about linear programming [15].

Linear programming is a field of mathematical optimization that deals with the problem of optimizing linear objective function subject to linear constraints. Such problem is called linear program. The standard form of a linear program is:

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & Ax \leq b, x \geq 0 \end{array}$$

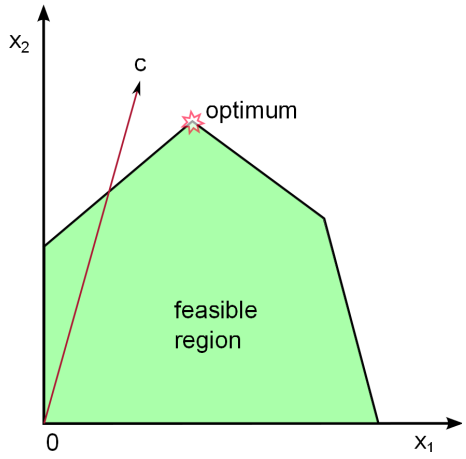


Figure 5.2.: Illustration of a linear program

where x is the column vector of n variables; c is a column vector of n coefficients of the objective function ($c^T$is its transpose); A is a $n \times m$ matrix of coefficients, and b is a column vector of m right-hand sides. The multiplication is a standard matrix multiplication and a vector inequality means the inequality between all components of vectors.

In the standard form, there is the constraint that the variables are nonnegative. Variables in a linear program can generally have any real value, but every linear program can be converted to the form with nonnegative variables because a real variable can be expressed as a difference between two nonnegative variables. A linear program can also contain equalities. They can be replaced by two opposite inequalities or eliminated by substitution to get the standard form. The objective function can be either maximized or minimized. Since maximizing $c^T x$ is the same as minimizing $-c^T x$, the choice is not important. The variant that most naturally fits the problem is usually chosen.

A linear constraint defines a half-space. The intersection of all half spaces that are defined by constraints of a linear program is a convex polyhedron, which contains all feasible solutions. The task is to find a point from this polyhedron with optimal objective function value. If the polyhedron is empty, then there is no solution. If the polyhedron is unbounded, then there might be no optimal solution, because it might be possible to get arbitrary good feasible solution (it also depends on the objective function). Figure 5.2 shows an example of a linear program with two variables.

Every linear program has its dual. The dual linear program for a (primal) linear program which is given in the standard form is:

$$\begin{array}{ll} \text{minimize} & b^T y \\ \text{subject to} & A^T y \geq c, y \geq 0 \end{array}$$

y is a vector of dual variables. There is one dual variable for each constraint of the primal linear program, and there is one constraint for each variable.

A basic fact about dual is that any dual feasible solution is an upper bound on the solution of primal. This is known as the weak duality theorem, and it is easy to prove. If we multiply each constraint of primal with the dual variable and then sum the constraints, we get the constraint

$$(A^T y)^T x \leq b^T y$$

Since any feasible solution x of primal is nonnegative and $A^T y \geq c$,

$$c^T x \leq (A^T y)^T x \leq b^T y$$

The left-hand side is the primal objective for a feasible primal solution x and the right-hand side is the dual objective for a feasible dual solution y. The strong duality theorem states that if an optimal solution exists for one problem, then an optimal solution also exists for the second problem, and they are equal[10]. The proof of the strong duality is much harder; therefore we will omit it.

Primal and dual are strongly connected. When one of them is solved, then the solution for the second can be easily computed. The result that we get from a solver usually contains both the primal solution and the dual solution. Although we are usually interested in the primal solution, the dual solution can give us some additional information. The value of a dual variable can be interpreted as importance of the corresponding constraint. From the proof of the weak duality theorem, it follows that if a dual variable is zero, then the corresponding inequality is not used to bound the optimal primal solution. If it is removed, then the optimal solution would not improve. This corollary is used in the advanced LP-learning algorithm.

An example of a problem that can be written as a linear program is the assignment problem, which was described in Section 4.5. It could be solved by any algorithm for solving linear programs, but the specialized algorithm is probably much faster.

The assignment problem was one of the first problems that were solved with the simplex algorithm. The simplex method was developed by George B. Dantzig in 1947. Although it has exponential worst case time complexity, it is very fast in practice and it is still one of the best algorithms for linear programming. The basic idea of the simplex method is to move from one vertex of the feasible region to a neighboring vertex with a higher value of the objective function.

[10] It also says what can happen if a solution does not exist, but that is not important for our purposes.

Interior point method is an alternative to the simplex method. The principal difference is that interior point methods work with a temporary solution that is inside the feasible region. It can be implemented with the polynomial worst case time complexity. In practice, modern variants of interior point method (barrier method) are often faster than the simplex method for large problems, but it depends on the type of the problem. One of the advantages of the barrier method over the simplex method is that it can take advantage of multi-core processors.

Although the general idea of algorithms for solving linear programming might seem simple, it is difficult to implement them correctly and it is very difficult to implement them efficiently. Fortunately, we do not need to implement new solver. We can choose from many existing solvers. There are a few free solvers and many commercial solvers.

According to information that we found on the Internet, one of the best free (and open source) solvers is the CLP solver from the COIN-OR project. It offers a C++ implementation of the simplex method. It has been being developed for many years and it is still maintained, and thus the algorithm should be reliable. CLP also offers an implementation of barrier method, but it is not as mature as the simplex method and, in our experience, it does not work well.

There are two commercial solvers that offer free license for academic purposes: Gurobi and Mosek. We have tried Mosek 6, Mosek 7 and Gurobi 5.5. In comparison with CLP, they have many additional functions, but we were interested mainly in their implementation of barrier method for linear programming.

We want to be able to easily switch between the solvers. Unfortunately, each solver has a different interface, and consequently we must either write a different version of code for each solver or use an intermediate layer that encapsulates the interface of each solver. This problem has been already recognized by other researchers and as a result, OSI (Open Solver Interface) was developed. OSI comes from the COIN-OR project, like CLP. It is a C++ library that contains a generic class for linear programming solver. There is a derived class for each supported solver, but most operations can be done directly through the general interface. All solvers that we wanted to use are supported. At first, this solution seems perfect, but it has also some disadvantages. First, the documentation is not very clear and we had some problems to configure and install the library. Second, there is very limited support for the barrier method. Finally, an intermediate layer may add some overhead. Nevertheless, we have used OSI and, thanks to it, the LP-learning algorithms are not dependent on a particular solver.

5.4.4. Basic algorithm

We have developed two learning algorithms that use linear programming to find the best evaluation function with respect to the loss function that was described in Section 5.1. The first algorithm is simpler and faster because it is based on a simplified loss function with $\beta = 0$ and $\gamma = 0$. The algorithm is called `ExpertMoveLearn`3 (E3). The number 3 indicates that it is the third version of this algorithm. The first two versions were similar, but they had some flaws, that were fixed in the third version.

The algorithm is straightforward. It reads the file with game records, then it constructs a linear program, which is subsequently transferred to the solver and solved. The solution of the linear program directly contains the weights of the evaluation function.

The linear program contains four types of variables. If n is the number of weights and m is the total number of positions then there are:

- n variables for the weights (w_i)
- n variables for regularization loss (r_i)
- m variables for the loss of type a (a_i,a'_i)
- m variables for the loss of type d (d_i)

The meaning of the first type of variables is clear. They directly refer to the weights of the evaluation function. Generally, they could have any real value, but in this algorithm, lower bound L is defined. The reason is that it seems to be more efficient.

For a weight w_i, we need to add $\rho\,|w_i|$ to the objective function. There are several ways how that could be achieved. A straightforward option would be to add a variable for absolute value of each weight. Although it is not possible to have constraints with absolute values in linear program, it is possible to have two constraints that together ensure that variable r'_i is greater than or equal to the absolute value of a variable w_i.

$$\begin{aligned} r'_i &\geq w_i \\ r'_i &\geq -w_i \end{aligned}$$

If r'_i is minimized (with positive coefficient), then in optimal solution, it must be equal to $|w_i|$. Therefore minimizing $\rho r'_i$ is equivalent to minimizing $\rho\,|w_i|$.

Although it would work, we have taken slightly different approach. The regularization variable r_i for a weight variable w_i is a nonnegative value that is greater

than or equal to w_i. In the linear program, there are the following constraints for each weight variable w_i and corresponding regularization variable r_i:

$$\begin{aligned} r_i &\geq w_i \\ r_i &\geq 0 \end{aligned}$$

Minimizing $2r_i - w_i$ is equivalent to minimizing $|w_i|$; therefore $\rho(2r_i - w_i)$ is added to the objective function for every i. This variant was preferred over the previous because the constraint $r_i \geq 0$ is simpler than the constraint $r'_i \geq -w_i$.

The variables a_i and a'_i for $i \in \{1, 2, \ldots, m\}$ correspond to the loss of type a. Let x_i denote the feature vector of the training position number i and let y_i denote the feature vector for the position after expert's move. For every $i \in \{1, 2, \ldots, m\}$ the following constraints are added:

$$(x_i - y_i)w = a_i - a'_i, a_i \geq 0, a'_i \geq 0$$

The left-hand side is equal to the difference of evaluation between the positions. The variables a_i and a'_i are added to the objective function with coefficient α ($\alpha > 0$). In the optimal solution, a_i and a'_i cannot be both nonzero, because then we could subtract $\min(a_i, a'_i)$ from both variables and get a better feasible solution. Therefore, the sum of the variables is equal to the absolute value of difference between evaluation of positions.

To measure the loss of type d, we need to compute the average of feature vectors of all positions that can be reached in one turn. Since it would be very time-consuming to evaluate all moves, we approximate the average with a random sample of K moves, where K is a parameter of the algorithm. Let z_i denote such an approximated average vector. For every training position i, the following constraints are added:

$$s_i(x_i - z_i)w \geq S - d_i, d_i \geq 0$$

s_i is 1 if Gold is on the move in the position i and it is -1 if Silver is on the move. S is a constant that determines the scale of the evaluation function. The d variables are added to the objective function with the coefficient δ ($\delta > 0$).

The last part of the linear program is the constraint that the average evaluation of training position is zero. That should ensure that the zero evaluation will mean that both players have equal chances of winning.

Table 5.3 is an overview of all parameters. Every parameter can be changed with a command line option. The corresponding option is given in the second

parameter	option name	default value
α	eqQ	0.7
δ	diffQ	1
ρ	regQ	1
K	randomSamples	50
S	move_gap	100
L	wLowerBound	-2000

Table 5.3.: Parameters of the basic algorithm

column of the table. If no explicit value is given, then the default value, which is the third column of the table, is used.

5.4.5. Advanced algorithm

The second algorithm is more complex because it works with the general loss function as it was defined in Section 5.4.2. The difference from the basic algorithm is that loss of type $\max(a, b)$ and loss of type c is considered in addition to all other types of loss.

It is possible to construct a linear program that leads to the optimal weights for the full loss function. The linear program from the basic algorithm can be extended to the general case by adding some variables and constraint.

To add the loss of type c to the objective function of the linear program we must first introduce new variables $c_i \geq 0$ for $i \in \{1, 2, \ldots, m\}$; c_i should be equal to the difference between the best move and the move that was chosen by an expert. Let Y_i denote the set of feature vectors for all position that can be reached in one turn from training position i excluding the position that is reached by expert's move. Then for every i and for every $y' \in Y_i$ the following constraint is added:

$$s_i(y' - y_i)w \leq c_i$$

c_i is added to the objective function with coefficient γ. Since it is minimized, c_i must be either equal to $\max\{s_i(y' - y_i)w | y' \in Y_i\}$ or zero in the optimal solution. Therefore c_i corresponds to the loss of type c for position i as it was defined in Section 5.4.2.

The loss of type $\max(a, b)$ is handled similarly. Variables $b_i \geq 0$ for $i \in \{1, 2, \ldots, m\}$ are added. There are two types of constraint. The first is analogous to the constraints for loss of type c. For every i and every $y' \in Y_i \cup \{y_i\}$:

$$s_i(y' - x_i)w \leq b_i$$

The second type of constrain is:

$$s_i(x_i - y_i)w \leq b_i$$

We will assume that Gold is on the move in position i. The case for Silver is analogous. All the constraints of the first type together ensure that b_i is greater than or equal to the difference between best evaluation and evaluation of position i. The constraint of the second type ensures that b_i is greater than or equal to the difference between evaluation of position i and evaluation of expert's move. Since the best evaluation is higher than or equal to the evaluation of expert's game, b_i is greater than or equal to the loss of type $\max(b, c)$. Variables b_i are added to the objective function with positive coefficient β, and therefore b_i is equal to the loss in the optimal solution.

The solution of this linear program would directly give us the optimal weights. The problem is that the linear program is too big. The number of constraints for one training position is higher than the number of unique moves from that position, which is about 17000 in average for Arimaa. If we wanted to learn from a thousand games and if each game consisted of 100 positions, then the linear program would have billions of constraints. The model must fit into memory, but we would even not be able to store the model to the hard disk, because it would have many terabytes, depending on the format and number of weights.

Although the model is huge, most of the constraints are redundant in some sense. We know, that there must be an optimal solution to the model that gives the optimal weights w^*. If we knew w^*, then we would only need to consider positions with best evaluation according to w^* instead of Y_i. It would not change the result[11], because when a constraint is satisfied with positive slack, then it can be omitted. If follows directly from the fact that the feasible region is convex and the objective function is linear. It can also be clarified with the dual problem. The corresponding dual variable must be zero[12] and that means that it is possible to construct tight upper bound without that constraint. Although we do not know w^* when we are constructing a model, we assume that it is a good static evaluation function for Arimaa. Therefore, we could guess what moves might be best according to w^* because it should be the moves that humans would consider to be the best. If the guess was correct, then we would get the optimal solution to the complete model by solving a much smaller problem. Otherwise, the learned evaluation function will consider different moves, which we had not taken into account, to be best. We could add constraints for these moves and solve the updated model. That is the basic idea of the second LP-learning algorithm.

[11] It might only allow some additional optimal solutions with the same objective.

[12] That is a corollary of complementary slackness, which is not described in this thesis, but it can be found in any textbook about linear programming.

Algorithm 5.1 Advanced LP-learning

```
Input:

    Training positions
    Initial weights
    Several parameters

1) Perform initial computation:

    1a) Compute average vectors.
    1b) Determine position importance.
    1c) Construct an initial linear program.
    1d) Solve the initial linear program.

2) Repeat several times:

    2a) Remove constraints with zero dual variable.
    2b) Add new constraints to the linear program.
    2c) Solve the updated linear program.

3) Return the weights from the last solution.
```

The complete linear program is modified. All loss variables for a position are merged into one. Two loss variables e and e' can be merged by combining all their constraints. Without loss of generality, we assume that the coefficients of both variables in the objective function are equal to one. Let nonnegative loss variable e is determined by k constraints

$$q_i w \leq e, \text{for } i \in \{1, 2, \ldots, k\}$$

and let nonnegative loss variable e' is determined by k' constraints

$$q'_i w \leq e', \text{for } i \in \{1, 2, \ldots, k'\}$$

Then merging e and e' means replacing variables e and e' with new variable e'' and adding kk' new constraints

$$(q_i + q'_j) w \leq e'', \text{for } i \in \{1, 2, \ldots, k\}, j \in \{1, 2, \ldots, k'\}$$

The coefficient of e'' in the objective function is one. The modified linear program is equivalent to the original. It has one less variable and kk' more constraints. The number of new constraints is huge because k and k' correspond to the average number of moves in Arimaa, which is about 17,000 and, furthermore, more than two variables are merged. However, it is not a big problem in practice, because the model is not explicitly created.

Algorithm 5.1 is the overview of the advanced LP-learning algorithm. The

linear program it works with is always a subset of the complete linear program. The partial linear program contains all constraints connected with regularization and the equality that ensures that the average evaluation of training positions is zero. On contrary, it contains only a small fraction of constraints for loss variables. The algorithm performs several iterations. In each iteration it solves the linear program, and then updates the set of active constraints. The result is an approximation of the optimal solution of the complete linear program.

The algorithm starts with the initial phase. Firstly, it computes auxiliary data structures. We need to compute the average of feature vectors for all training positions. Next, for each training position, the average of all possible successors is computed. In fact, it is only approximated because, in the initial phase, random part of moves is discarded to speed up the computation.

Although the algorithm should generally work with any initial weights and the result almost does not depend on them, we assume that reasonable initial weights are provided. We expect that they will be computed with the basic algorithm. There are two purposes of initial weights.

First, initial weights are used to compute importance values of training positions. The importance value is a coefficient of corresponding loss variable in the objective function. High importance value means that the algorithm will pay more attention to that particular position. The importance value is computed according to the initial weights (initial evaluation function). If the position is balanced, then the importance is high. If one side has a big advantage, then the importance is lower. The idea behind this rule is that player might play poor moves if the game is already decided. It is hard to precisely classify positions as decided, and therefore the "fuzzy" importance value is used. The formula for the importance p is:

$$p = \frac{1}{1 + qe^2}$$

where q is a nonnegative constant and e is the evaluation of positions according to initial weights. The value is in the range $(0, 1]$. The importance of positions where the winner is on the move is increased to $0.5 + 0.5p$ because we expect that the winner was playing good moves to the very end of the game. The addition of the importance was one of the last changes, but it does not seem to be very significant.

Second case where the initial weights are used is the computation of the initial linear program. For each position (loss variable), the constraint that leads to the highest value of loss variable with respect to the initial weights is added. However, only a random part of moves is considered. The linear program is solved as the

last step of the initial phase.

The main phase of the algorithm consists of a loop. In one iteration, the linear program is updated and solved. The loop ends when the linear program does not change much or the limit of the number of iteration is reached.

The linear program is modified in two steps. First, the constraints for loss variables that have zero dual variable are removed. Second, some constraints that are not in the current linear program are checked and if they are violated, then they are added to the model. For each position, at most one constraint is added. If more than one constraint is violated for a single position, then the constraint that forces the highest value of the loss variable is selected.

It would take too much time to check all constraints (a few hours) because it is equivalent to performing 1-ply search from every training position. For this reason, only a fraction of positions is completely recomputed. The rest of the positions are recomputed only partially. We take advantage of the fact that the same positions are evaluated many times. The feature description of some positions is stored to the hard disk so that we can skip the first two phases of evaluation. Unfortunately, this approach is limited by the performance of hard disk and by its capacity. The speed of reading from the hard disk is similar to the speed of the computation. Nevertheless, it brings some speedup. The choice of positions that should be stored to the hard disk is random, but a position with high evaluation is selected with a higher probability. Furthermore, the positions that were once best are always stored. All stored positions are recomputed in every iteration and the set of stored feature vectors is updated for the positions that are completely recomputed.

There are some additional implementation details. For instance, there is a limit for the number of constraints that can be added in one iteration. On contrary, if there is not enough new constraints, then some constraints that would be normally removed are kept. Since we are working with finite precision of numbers, there are rounding errors. Therefore, instead of testing that a dual variable is equal zero, we test that the variable is in a small interval around zero. Furthermore, some randomness is added to the process of removing and adding constraints.

The solutions of the partial linear program converge to the solution of the complete linear program. The reason is that the objective value of the optimal solution of the partial linear program is increasing with each iteration. There is an unlikely possibility that the optimal objective value stays the same, but it does not seem to be a problem in practice and if it was a problem, then it could be easily fixed with a minor modification to the algorithm[13]. The optimal objective

[13]The modification is that constraints are removed only if the objective increases. Then there might be only a finite number of consecutive iterations that does not increase the objective.

option	default value	description
regQ	1.5	ρ (regularization)
eqQ	0	α
diffQ	1.0	δ
playedQ	0.5	γ
consQ	0.4	β
diffGap	100	S (scale)
winThreshold	500	$q = 1/\text{winThreshold}^2$
wLowerBound	-2000	L (lower bound for weights)

Table 5.4.: Most important options for the advanced LP-learning algorithm.

value cannot decrease, because we only remove constraints with the zero dual variable. There is only a finite number of subsets of constraints; therefore, if the optimal objective value always increases, then we must reach the maximum, which is equal to the optimal objective of the complete linear program. Although we cannot provide any theoretical estimation of the number of iterations that are necessary, the convergence is rather fast in practice.

Table 5.4 shows the most important options for the algorithm together with their default value. There are some other options. For instance the maximum number of iterations or the fraction of the games that are completely recomputed in each iteration.

5.5. Quadratic programming

An extension of linear programming is quadratic programming. The difference is that, in quadratic programming, the objective function is quadratic:

$$\frac{1}{2}x^T Q x + c^T x$$

Quadratic programs can be efficiently solved if Q is positive definite. OSI does not support quadratic programming, so we have directly used the Gurobi solver.

The quadratic loss function can be used instead of the linear. We have implemented two algorithms that are based on quadratic learning to compare it with the LP-learning. However, as we have focused more on the LP-learning, the description of the quadratic algorithms will be only brief.

The first quadratic algorithm is called `QuadraticMoveLearn4` (Q4). It is similar to the basic LP-learning algorithm. It finds the weights w that minimize the objective function:

$$\sum_i (x_i^T w - y_i^T w)^2 + \rho \sum_i |w_i| - \delta d^T w$$

option	default value	description
regQ	30000	ρ
winGap	1000	S
diffQ	40	δ
randomSamples	50	like in E3

Table 5.5.: Options for Q4

subject to:

$$\begin{aligned} g^T w &= 0, \text{ where } g = \sum_{\text{setup position i}} x_i \\ v^T w &= S, \text{ where } v = \frac{1}{m} \sum_{\text{position}i} r_i x_i \end{aligned}$$

The objective function minimize the quadratic difference between evaluation of following positions. There is also a regularization term. The expression $d^T w$ is equal to the average difference between the played move evaluation and average evaluation. It is subtracted because it is better to have higher difference. It is similar concept like the loss of type d.

The constrains ensures that the evaluation of the setup training position is zero in average and that the scale L_{outcome} is equal S. There are only $n+2$ constraints, no matter how many training positions there are. This is a big advantage because this algorithm can handle a large set of training games. The parameters are listed in Table 5.5.

The second quadratic algorithm is called `AdvancedQuadraticLearn2` (AQ2). It is based on an older version of the advanced LP-learning algorithm. The main difference is the model that is optimized. Its objective function is similar to the objective function of the previous algorithm:

$$\sum_{i=1}^{m} (x_i^T w - y_i^T w)^2 + \rho \sum_{i=1}^{n} |w_i| - \sum_{i=1}^{m} (\gamma c_i^2 + \gamma' c_i')$$

There are two loss variables (c_i and c_i') for every position. These variables are associated with the loss of type c. The sum of two variables $c_i + c_i'$ is on the right side of the loss constraints instead of a single variable. Since the objective is minimized, the function that maps the loss of type c for a position to the value of the objective function is quadratic around zero and linear from a point that depends on the choice of γ and γ'. There is only one constraint for scaling:

$$s^T w = S, \text{where } s = 2d + \sum_{\text{game}g} s_g((\sum_{i=1}^{m_g - 1} x_{g,i}) - m_g x_{g,0})$$

option	default value	description
regQ	40	ρ
playedQ	40	γ'
pquadQ	10	γ
scaleGap	100	S

Table 5.6.: Basic options for AQ2

name	algorithm (solver)	training data rating	time (h:m:s)
Expert3	E3 (Mosek 7)	2200	0:8:49
Expert3G	E3 (Gurobi 5.5)	2200	0:10:28
Advanced3	A3 (Mosek 7)	2200	8:42:59
Q2200	Q4 (Gurobi 5.5)	2200	0:1:22
Q2000	Q4 (Gurobi 5.5)	2000	0:6:30
Q1800	Q4 (Gurobi 5.5)	1800	0:22:38
AQ2	AQ2 (Gurobi 5.5)	2200	22:35:45

Table 5.7.: Learned evaluation function weights.

Scale vector s is a combination of vector d, which was also used in the previous algorithm, and a vector that corresponds to the average difference between the evaluation of a position and the evaluation of the position after the setup from the same game, from the perspective of the winner (if Silver won, then the sing s_g is opposite). In the formula, m_g denotes the number of positions in the game g and $x_{g,i}$ denotes the position i turns after the setup in the game g.

Table 5.6 lists the basic options for AQ2. There are some other options that control the learning process, the same like in the original linear algorithm.

5.6. Experimental results

In this section, we present the experimental comparison of various learning methods. We compare the time of learning and the quality of the result. We also examine how robust the methods are, and what is the impact of adding more training data. In addition, we compare the linear programming solvers.

Table 5.7 presents the learning methods that were compared. Default values of parameters were used. In all cases, the features were expanded to the polynomial with degree 3. Expanded feature vectors have 670 elements. The degree 3 was chosen as a good compromise between the number of weights that must be learned and the precision of the evaluation function. Increasing the degree to 4 did not bring any significant improvements, but we did not make many experiments with it.

Before we started to work with learning algorithms, we implemented a hand-tuned evaluator. We spent a few days trying to set the coefficients so that the

	E3	E3G	A3	Q2200	Q2000	Q1800	AQ2	Sim.	**SUM**
Expert3	-	102	63	129	112	104	56	118	**684**
Expert3G	97	-	61	122	113	101	59	127	**680**
Advanced3	136	139	-	142	141	121	95	137	**911**
Q2200	71	78	58	-	88	62	50	110	**517**
Q2000	88	86	59	112	-	91	64	109	**609**
Q1800	96	99	79	138	109	-	75	121	**717**
AQ2	144	141	105	150	136	125	-	146	**947**
Simple	80	73	63	90	91	79	54	-	**530**

Table 5.8.: Tournament results. A row contains points that a bot got in the matches with other the bots. The last column shows the total score.

bot can play reasonably well. This hand-tuned evaluator was included in the comparison under the name Simple.

The problem of comparing two static evaluation functions is more complex than it might seem. A natural way to experimentally compare evaluation functions is to let them play against themselves. However, if the evaluation functions and other parts of a bot are deterministic, then all games between two bots would be identical. We could add some random noise. For instance, a random move might be played with a given probability, or we can add a random value to the evaluation function. The problem with this approach is that one of the bots may be favored. The randomness might harm one bot more than the other. Furthermore, it might lead to artificial positions.

The method that we used for comparison introduces variability by other means. The games are started from a selected position. The position is chosen from the first half of a game from the Arimaa game archive. It might be the position after the setup, but usually it is a position after several turns. Two games are always played from the same positions. The players switch colors between the games to ensure that no player can get an advantage. If one color has significant advantage, then each player probably wins the game when playing with that color, and the final score will be 1:1. There is a limit on the number of turns that a game may have (300). If the limit is reached, then both players loose. Consequently, the sum of points might be slightly lower than the number of games. A possible problem of this method is that bots that play better in the endgame than in the opening have an advantage. Nevertheless, the result of this method should be meaningful.

We let the competitors play with each other. A simple algorithm for choosing the move is used. The move with best evaluation is played. No search is performed. The only difference is in the evaluator of features. Each duel consisted of 100 pairs of games.

The results of the tournament are summarized in Table 5.8. The winner is AQ2, but it took almost one day to learn the weights with this algorithm. Advanced LP-learning algorithm was slightly worse, but it finished in less than 9 hours. Every automatically learned evaluation function was better than the hand-tuned except Q2200. Q2200 was the worst, but it was computed extremely quickly (in less than two minutes).

Expert3 and Expert3G are two evaluation functions that were learned with the same algorithm, with the same parameters and with the same data. The only difference was in the linear programming solver. The learning algorithm uses randomness and we wanted to check what effect it has. Although the learned weights were not exactly the same, the playing strength is almost identical. The conclusion is that the basic LP-learning algorithm gives stable results. We expect that the same is true for all algorithms because the role of randomness is negligible.

All algorithms except Q4 can process only a limited amount of training data because their memory requirement increases with the number of training positions. The computer we used for experiments has 6 gigabytes of memory, which allowed us to train weights from the set of 700 games with rating 2200 and more. The algorithm Q4 is not limited in this way, so we could try to use more training data. The experiment clearly shows that it is better to have more training data even if they have lower quality. Q1800, which was learned from the set of 11392 games is stronger than Q2200, which was learned from 700 games.

We did not use CLP solver in the experiments, because it is too slow and it requires too much memory. It is orders of magnitude slower than the commercial solvers Gurobi and Mosek. One reason is that it does not have good implementation of the barrier method, which seems to be superior to the simplex algorithm for this class of problems.

We have also found that it is more efficient to let a solver solve the dual problem. The solvers can transparently transform the problem to its dual, solve it and then transform the solution back to the original problem. All solvers have an option which determines whether the dual should be used instead of the primal. Mosek chose the dual problem automatically, whereas Gurobi needs to set the option explicitly.

The standard behavior of solvers is that they switch from the barrier method to the simplex method when they are close to the optimum. One of the reasons is that the solution of the simplex is somehow special. The simplex algorithm transfers from vertex to vertex; therefore, the optimum is also a vertex of the feasible region. On contrary, the solution of the barrier method is inside the feasible region, thus the variables might have values like $4.11546e-11$ instead

of zero. However, Gurobi solver had some difficulties with the crossover phase. It took too much time, so we used pure barrier method with Gurobi 5.5. There were no such problems with Mosek 7.

In the second experiment, we compared bot_drake with other bots that are available on the Arimaa server. bot_drake uses standard alpha-beta search with some basic extensions, like move ordering and killer moves. It can search to the depths one, two or three. If there is not enough time to complete the search to the full depth, then the search is interrupted, and the best move that was found so far is played. We tried the evaluation functions learned by AQ2 and A3 algorithms. The evaluation function learned by the AQ2 algorithms seemed to be better even against other bots; therefore, we eventually used only this evaluation function. The usage of bot_drake is described in Appendix B.

When we finished and debugged bot_drake, we let its P2 version[14] play against other P2 bots. There is almost always enough time to complete the search to depth two; therefore, the differences between bots should be caused mainly by the differences in their static evaluation functions. After 92 games again various P2 bots, the rating of bot_drakeP2 steadied around 1,950. For comparison, the highest rated P2 bot currently is bot_Sharp2012P2 with rating 2,038, and bot_drakeP2 is the second highest rated P2 bot. This is very good result, but the significance of the rating must not be overestimated. For instance, bot_drakeP2 lost all four games against bot_Sharp2013P2, which has lower rating. Furthermore, the static evaluation function in bot_drake is probably more computationally demanding that in other bots[15]; thus the search will be less effective. We also tried bot_drake without restriction on depth. After some bugs were fixed, it could reach rating over 2,000, but it usually lost with the best bots.

[14] P2 means that the depth of search is limited to two plies.

[15] In almost every game, the average thinking time of bot_drakeP2 was lower than the average thinking time of its opponent, but bot_drakeP2 used multiple threads, whereas bots on the Arimaa server are probably limited to a single thread. A notebook with quad-core processor (Intel Core i7-3612QM, 2.10GHz) can evaluate approximately 120,000 positions per second.

6. Conclusion

We have implemented a static evaluation function for Arimaa and a new bot called bot_drake. The bot can play reasonably well, but there are still many things that could be improved.

We have divided the problem into several parts. The most difficult part was the mobility analysis. We have managed to implement an extremely complex function that determines where a piece can move and whether a piece is threatened to be captured. It was possible only thanks to the testing method that we used. Unfortunately, it took more time than we expected. As a consequence, less time remained on the other parts of the static evaluation function. Mobility analysis is the first layer of the static evaluation function.

The second layer deals with abstract strategic aspects of a position. The basic concept of this layer is distance. It is an estimate of the number of steps that a piece would need to move from its current location to a given square. Most of position features are based on the distance, for instance the evaluation of goal threats. It would probably be necessary to add more features to get a better evaluation function. The output of the second layer is not an evaluation, but an abstract description of position with parametrized features.

We have shown that the final evaluation of features can be automatically learned from games of experts. The learning algorithm uses linear (or quadratic) programming. State of the art commercial solvers can easily solve the optimization problems that arise during learning. Unfortunately, the free solver (CLP) was too slow to be practically usable. There are still many opportunities for improvement. The learning algorithm has many options and we could not have tried all combinations, because the learning process takes a long time.

The speed of computation was one of our main concerns. Almost whole static evaluation function is accelerated with vector instructions. Despite the heavy optimization, it is probably slower than evaluation functions of other bots because it performs a lot of complex computation.

Bibliography

[1] SYED, Omar. Arimaa: A New Game Designed to be Difficult for Computers. *Journal of the Internation Computer Games Association*, vol. 26 (2), (2003), pp. 138–139. ISSN 1389-6911. Available from: http://arimaa.com/arimaa/papers/030801ICGA/.

[2] KRETZ, Matthias and Volker LINDENSTRUTH. Vc: A C++ library for explicit vectorization. *Software: Practice and Experience*, vol. 42 (11), (2012), pp. 1409–1430. Available from: http://dx.doi.org/10.1002/spe.1149.

[3] ZHONG, Haizhi. *Building a Strong Arimaa-playing Program*. Master's thesis, University of Alberta, Edmonton, 2005. Available from: http://arimaa.com/arimaa/papers/HaizhiThesis/haizhiThesis.doc.

[4] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, September 2013. Available from: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[5] VELDHUIZEN, Todd. Expression Templates. *C++ Report*, vol. 7 (5), (1995), pp. 26–31. ISSN 1040-6042.

[6] FOTLAND, David. Building a World-Champion Arimaa Program. In: *Computers and Games*, Springer Berlin Heidelberg, *Lecture Notes in Computer Science*, vol. 3846, pp. 175–186. 2006. Available from: http://dx.doi.org/10.1007/11674399_12.

[7] DELL'AMICO, Mauro and Paolo TOTH. Algorithms and codes for dense assignment problems: the state of the art. *Discrete Appl. Math.*, vol. 100 (1-2), (2000), pp. 17–48. ISSN 0166-218X. Available from: http://dx.doi.org/10.1016/S0166-218X(99)00172-9.

[8] TESAURO, Gerald. Practical Issues in Temporal Difference Learning. *Machine Learning*, vol. 8, (1992), pp. 257–277. ISSN 0885-6125. Available from: http://dx.doi.org/10.1007/BF00992697.

[9] TESAURO, Gerald. Temporal difference learning and TD-Gammon. *Commun. ACM*, vol. 38 (3), (1995), pp. 58–68. ISSN 0001-0782. Available from: http://dx.doi.org/10.1145/203330.203343.

[10] WU, David Jian. *Move Ranking and Evaluation in the Game of Arimaa*. Master's thesis, Harvard College, Cambridge, Massachusetts, May 2011. Available from: http://arimaa.com/arimaa/papers/DavidWu/djwuthesis.pdf.

[11] UTGOFF, Paul E. and Jeffery A. CLOUSE. Two kinds of training information for evaluation function learning. In: *In Proceedings of the Ninth Annual Conference on Artificial Intelligence*. Morgan Kaufmann, 1991, pp. 596–600.

[12] URA, Akira, Makoto MIWA, Yoshimasa TSURUOKA and Takashi CHIKAYAMA. Comparison Training of Shogi Evaluation Functions with Self-Generated Training Positions and Moves. In: *Proceedings of the 8th International Conference on Computers and Games*. 2013.

[13] HOKI, Kunihito and Tomoyuki KANEKO. The Global Landscape of Objective Functions for the Optimization of Shogi Piece Values with a Game-Tree Search. In: *Advances in Computer Games*, Springer Berlin Heidelberg, *Lecture Notes in Computer Science*, vol. 7168, pp. 184–195. 2012. ISBN 978-3-642-31865-8. Available from: http://dx.doi.org/10.1007/978-3-642-31866-5_16.

[14] KANEKO, Tomoyuki and Kunihito HOKI. Analysis of Evaluation-Function Learning by Comparison of Sibling Nodes. In: *Advances in Computer Games*, Springer Berlin Heidelberg, *Lecture Notes in Computer Science*, vol. 7168, pp. 158–169. 2012. ISBN 978-3-642-31865-8. Available from: http://dx.doi.org/10.1007/978-3-642-31866-5_14.

[15] WIKIPEDIA. Linear programming — Wikipedia, The Free Encyclopedia, 2013. Available from: http://en.wikipedia.org/w/index.php?title=Linear_programming&oldid=577920529. [Online; accessed 24-October-2013].

List of Figures

List of Tables

Appendix A.

Content of the attached CD

bin/linux64/bot_drake bot_drake executable for 64-bit Linux system. It was tested on Ubuntu 12.10.

bin/windows64/bot_drake.exe bot_drake executable for 64-bit Windows. It was tested only on Windows 7, but it may also work on other Windows systems. If it does not work, then it is necessary to build bot_drake.exe from the source code.

doc/html/ HTML documentation for the source code. It was automatically generated with doxygen.

doc/refman.pdf PDF version of the doxy documentation.

experiment/ Directory that contains outputs of experiments with learning.

experiment/commands.sh Linux shell script that can be used to repeat the experiments. (It is necessary to first build drake_learning and create configure file drake_learn.cfg with options set to correct values.)

importantFiles/ Directory with files that bot_drake needs at runtime.

importantFiles/drake.cfg Configuration file for bot_drake.

importantFiles/viewerTemplate.html HTML template for game log.

importantFiles/weights.txt Weights for the static evaluation function.

src/ Directory with the source code. The most important files are placed directly into this directory. Code that is used only by one target is placed in a subdirectory.

src/analyzer_test Tool for testing mobility analysis code.

src/bot Source code for bot_drake.

src/learning Source code for learning algorithms and other tools. Everything is assembled into one executable - drake_learning.

src/tests Directory for unit tests. Tests can be automatically run with `make test` command after they are built.

src/unused Archive for files that are no longer used, but which might be used in the future.

Appendix B.

Using bot_drake

B.1. Building bot_drake from source

bot_drake has the following requirements:

1. C++ compiler (GCC 4.7 or newer, Clang 3.3 or newer)
2. CMake build tool (version 2.8.9 or newer)
3. Boost library (it was tested with several recent versions, e.g. 1.50).

For building drake_learning, some additional libraries are needed:

1. At least one linear (quadratic) programming solver (CLP, Gurobi 5.5, Mosek 7)
2. OSI (Open Solver Interface) configured for the selected solvers.

CMake is used to configure the build. There is a comfortable user interface for this purpose (ccmake on Linux, cmake-gui on Windows). Several options can be set:

- What components should be build (bot_drake, drake_learning, analyzer_test, tests).
- Build type (Release / Debug)
- Options for compiler (force static linking, target architecture).
- Enable / Disable profile guided optimization. (For gcc only; gcc 4.8 has a bug that causes crash of the compiler if PGO is enabled.)
- Paths to the libraries. (Libraries are automatically found if they are in standard locations.)
- If drake_learning is build, then which solvers should be used. (If more solvers are enabled, then the solver can be chosen at runtime with a command-line option.)

CMake outputs a makefile (or a project for an IDE). The executables are build by executing the `make`. Complete build process on Linux may look like this: (We assume that the source directory src is located in the current working directory.)

```
mkdir build
cd build
cmake ../src
ccmake .
make
```

B.2. Running bot_drake

bot_drake uses AEI (Arimaa Engine Interface) to communicate. However, the program options are not set through AEI, but with command line parameters or in a configuration file. The list of all options is displayed when bot_drake is executed with `-h` option. Configuration file is called `drake.cfg`. It must be present in the current directory, or the environmental variable `DRAKE_DIR` must be set to the directory where it is placed. The configuration file should contain a path to the weights (importantFiles/weights.txt) and a path to the log template (importantFiles/viewerTemplate.html). For instance:

```
log_template=/path/to/viewerTemplate.html
weights=/path/to/weights.txt
```

There are four different modes in which bot_drake may run:

1. Play random moves (`./bot_drake -e random`)
2. Search limited to depth 1 (`./bot_drake -e simple`)
3. Search limited to depth 2 (`./bot_drake --max_depth 2`)
4. Search limited to depth 3 (`./bot_drake`)

If logging is enabled (default), then HTML file lastGame.html is created after a game is finished. It contains the record of the game and detailed description of the evaluation for each position. Features of a piece can be displayed by clicking on the piece. The images of pieces are located on the Arimaa server, so it will not work without Internet connection.