DESIGNING A WINNING ARIMAA PROGRAM

David J. Wu[1]

ABSTRACT

In 2015, twelve years after the creation of the Arimaa Challenge, the program SHARP won the Challenge, defeating three strong human opponents with a total of 7 wins and 2 losses. This paper describes the design of the winning agent and the improvements and innovations that led to its recent success. Among these are several innovations greatly increasing the quality and selectivity of the search and several major improvements to the positional evaluation.

1. INTRODUCTION

Arimaa is a game invented in 2002 by computer engineer Omar Syed. According to Syed, the original motivation for Arimaa came from the defeat of the human World Champion in Chess, Garry Kasparov, in 1997 by IBM's DEEP BLUE (Syed and Syed, 2003). Syed set out to design a game playable using a Chess set that would be as fun and interesting for human players and yet be more difficult for computers. The result was a fascinating new board game called Arimaa. By designing Arimaa, Syed hoped to encourage new interest and research in artificial intelligence for strategic games (Syed and Syed, 2003).

To this end, every year an event known as the "Arimaa Challenge" has been held, in which the top computer program using common desktop hardware plays a series of matches against three different players chosen that year to "defend" the Challenge. Since the beginning of the Challenge in 2004 up until the previous year, despite significant progress in computer Arimaa, human players have convincingly defeated the top programs each year.

However, to everyone's surprise in 2015 the tables turned. In this year our own program SHARP swept both the Computer Championship and a preliminary blitz tournament undefeated with 18 wins in a row, outperformed the strongest other competing program in preliminary screening matches against human players with a record of 28-2, and proceeded to defeat the challenge defenders 7-2, winning the Challenge!

Such a strong performance was both exciting and unexpected, and judging from the games, likely some luck was necessary. But in retrospect, the result is not entirely surprising. Since its first win of the Computer Championship in 2011, SHARP has improved greatly, especially in the last two years. In self-play testing against older versions at blitz speeds, it improved at least 200 Elo rating points going into the 2014 competition. It then leapt far ahead of all other programs by gaining an additional 400 Elo rating points by the start of the 2015 competition[2]. If not already stronger now, SHARP is at least close to being on par with top players, and it seems that there is still plenty of room for further improvement.

The goal of this paper is to explain the design of SHARP and to present the most significant innovations and improvements responsible for its jump in strength in the last two years. In Section 2 we describe the game Arimaa and some of the properties that have made it computer-resistant. In Section 3 we give an overview of other work and research that has been done in Arimaa. In Section 4 we present the basic algorithms used in SHARP and in Section 5 the recent search improvements that made its success possible. In Section 6 we describe the development and design of the positional evaluation. Finally, in Section 7 we present some conclusions and possibilities for future work.

[1]email:lightvector@gmail.com

[2]In winning chances, 200 Elo implies about 3:1 odds, and 400 Elo implies 10:1 odds, quite a large improvement.

2. THE GAME

Below we describe the rules of Arimaa (2.1) and we briefly discuss some of the properties of the game that have made it computer resistant (2.2).

2.1 Rules

Arimaa is a deterministic two-player abstract strategy game played on an 8 x 8 board, the two players being *Gold* and *Silver*. The rows and columns on the board are labeled 1. . .8 and a. . .h, respectively, as shown in Figure 1. Below we describe five parts of the rules of Arimaa, including the move notation.

2.1.1 Setup

Prior to normal play, Arimaa begins with a *setup phase* where both players place their pieces in any desired arrangement within their starting two rows with Gold placing all pieces first. In order of decreasing strength, each player has 1 Elephant, 1 Camel, 2 Horses, 2 Dogs, 2 Cats, and 8 Rabbits.

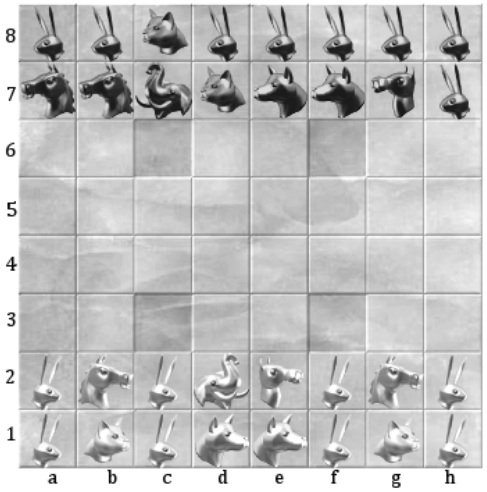

Figure 1: Example game position, just after the setup phase.

2.1.2 Movement

Following the setup phase, play begins and players take turns making moves of up to four *steps*, with Gold moving first. A *step* consists of selecting a piece of one's color and moving it to an empty adjacent square, that is, left, right, forward, or backward. All pieces move identically this way with one exception: rabbits may not move backward.

Pieces are differentiated in strength in that pieces can *push* or *pull* weaker opposing pieces using two steps at a time, as shown in Figure 2. Pushes can displace weaker opposing pieces into any adjacent empty square, and similarly pulls can displace weaker pieces from any adjacent square into the one just vacated. Simultaneously pushing and pulling with the same piece is illegal.

Additionally, stronger pieces can prevent weaker pieces from moving by *freezing* them. Whenever a piece is adjacent to a stronger opposing piece and is not *defended*, that is, when it has no friendly piece adjacent, then it is *frozen* and cannot move. (It can still be pushed or pulled by the opponent.)

Players must change the board position on their turn and also may not make any move that would result in a *third-time repetition* of the position and player-to-move.

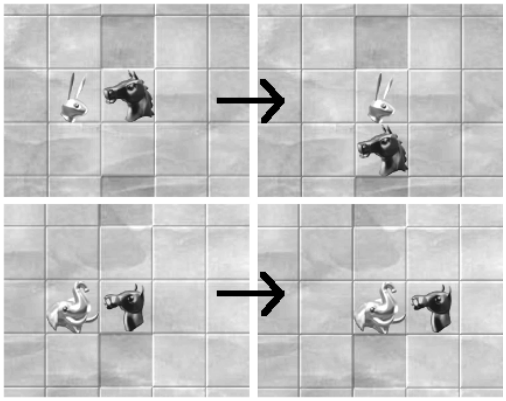

Figure 2: A silver horse steps down, pulling a gold rabbit. A gold elephant pushes a silver camel to the right.

2.1.3 Capture

The squares c3, c6, f3, and f6, are *trap* squares. Whenever a piece is on a trap square but is not defended, it is *captured* and immediately removed from the board.

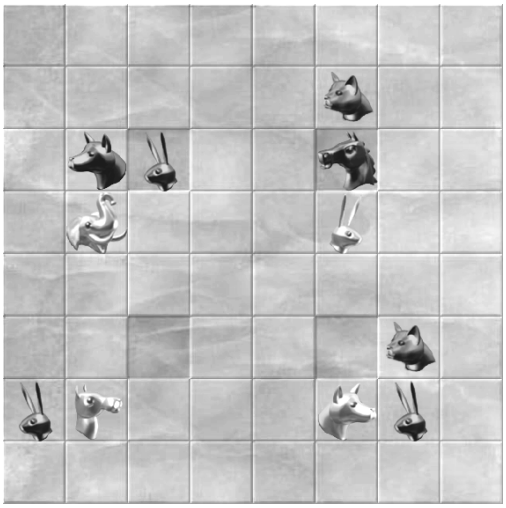

Figure 3: Examples of capturing and freezing. *Upper-left*: If the gold elephant pushes the silver dog, it captures the silver rabbit on the trap at c6. *Upper-right*: The silver horse at f6 can capture the gold rabbit by stepping left or right while pulling the gold rabbit onto the trap. *Lower-left*: The rabbit on a2 is frozen by the adjacent gold camel. *Lower-right*: The g2 rabbit is *not* frozen by the gold dog because it is guarded by the silver cat.

2.1.4 Game Objective

A player scores a *goal* and wins the game when one of that player's rabbits ends the turn on the opponent's back row, on the opposite side of the board[3]. A threat to win the game on the next move in this way is called a *goal threat*.

2.1.5 Notation

Arimaa move notation is used in a small number of places in this paper. For reference, it consists of tokens such as Ee2n indicating the piece (E $\in$ {E,M,H,D,C,R,e,m,h,d,c,r} uppercase for Gold and lowercase for Silver), the

[3]Much more rarely, a player can win by *immobilization* if the opponent is unable to make any legal move on their turn, or by *elimination* if the all of the opponent's rabbits are captured.

originating square (e2), and the direction of movement (n ∈ {n,s,e,w,x} with x indicating a capture). An example move in the middle of a game might look like: Ef4n Ef5w cf6s Hg4n.

2.2 Computer-Resistance

Why is Arimaa computer-resistant? We can identify two major obstacles.

The first is that in Arimaa, the per-turn branching factor is extremely large due to the combinatorial possibilities produced by having four steps per turn. Even after identifying equivalent permutations of steps as the same move, on average there are about 17000 legal moves per turn (Haskin, 2006). This is a serious impediment to search.

Obviously, a high branching factor alone doesn't imply computer-resistance, particularly if the standard of comparison is with human play: high branching factors affect humans as well. However, Arimaa has a property common to many computer-resistant games: that "per amount of branching" the board changes slowly. Indeed, pieces move only one orthogonal step at a time. This makes it possible to effectively plan ahead, cache evaluations of local positions, and visualize patterns of good moves, all things that usually favor human players.

The second obstacle is that Arimaa is frequently quite positional or strategic, as opposed to tactical. Capturing or trading pieces is somewhat more difficult in Arimaa than in, for example, Chess. Moreover, since the elephant cannot be pushed or pulled and can defend any trap, deadlocks between defending elephants are common, giving rise to positions sparse in easy tactical landmarks. Progress in such positions requires good long-term judgement and strategic understanding to guide the gradual maneuvering of pieces, posing a challenge for positional evaluation.

3. HISTORY AND EARLIER WORK

Despite these difficulties with branching factor and evaluation, the strongest Arimaa programs have generally resembled strong Chess programs in design, using alpha-beta search in combination with a slew of other enhancements and carefully-tuned evaluation functions (Zhong, 2005; Fotland, 2006; Cox, 2006). Developers Fotland and Zhong were among the first to do well with this approach and detailed many basic ideas used by nearly all strong Arimaa programs today, such as static capture and goal detection. Fotland in particular pioneered so much of the early groundwork that despite abandoning work on Arimaa after 2005, his program BOMB continued to win every Arimaa Computer Championship up through 2008[4](Syed and Syed, 2015b). Since then, the Computer Championship has been lively, with the title trading back and forth between a variety of ever-improving bots every year from 2009 onwards. In every case, the winning program was a "traditional" Chess-like searcher.

A few others have tried alternative approaches without significant success. These include Kozelek (2009) and Miller (2009), who made attempts to apply Monte-Carlo tree search in various ways. In another alternative but unsuccessful approach, Trippen (2009) investigated a pattern-matching and plan-based method for performing extremely selective search, examining only a tiny number of moves (as few as five!) in each position.

More recent and successful work has focused on ways of learning better move ordering or evaluation for use within the traditional alpha-beta search paradigm, including our own work on learning move ordering from expert game records (Wu, 2011) and similar other work (Vivek Choksi, 2013). In an exciting result, Hrebejk (2013) demonstrated that it is also possible to learn a strong evaluation function from expert game records . The following year, Hrebejk's program DRAKE, while not quite a contender for the top, performed fantastically well given its status as a newcomer.

The successes of these later approaches and the steady improvement in computer strength after 2009 made it a good bet that the traditional alpha-beta search paradigm would eventually be sufficient. And indeed, this is the path that SHARP followed. We begin next in Section 4 with an overview of the basic algorithms and enhancements taken from this paradigm, and then in Sections 5 and 6 elaborate on how we navigated the two obstacles of branching factor and strategic evaluation in Arimaa - the former via new enhancements to the search greatly increasing its efficiency and selectivity, the latter via determined engineering and a robust development methodology for improving the evaluation function.

[4]Even today, surpassing BOMB is viewed as a noteworthy achievement for any new developer!

4. BASIC SEARCH ALGORITHMS AND ENHANCEMENTS

SHARP follows the same fundamental design as strong Chess programs, using an *iterative-deepening depth-limited alpha-beta search* to explore the game tree and compute its *minimax value* out to increasing depths (see 4.1). The search is augmented with a variety of heuristics and enhancements for *ordering*, *extending*, *reducing*, and *pruning* to increase its efficiency and focus it in important parts of the tree (see 4.2 and 4.3). At leaf nodes, a carefully designed *evaluation function* is called to evaluate the game state, taking into account a large variety of positional features (described later in Section 6).

4.1 Alpha-Beta Search

In alpha-beta search, one computes the *minimax value*[5]of a position and finds a move that achieves that value by recursively finding the minimax value of each move in a depth-first manner and taking the maximum of the values returned for each move, negating whenever the side-to-move changes[6]. The algorithm's name derives from an optimization: it also accepts an interval (α, β) and returns from a node immediately if it can prove that its minimax value is outside the interval. The interval is passed down recursively, and after each move is searched (letting x be the value returned), two rules are followed to apply and update the interval:

1. Beta cutoff: If $x >= \beta$, return immediately, since the maximum move value must be $>= \beta$ and outside the window (α, β).
2. Improving alpha: Else set $\alpha := max(\alpha, x)$ when recursing on the remaining moves. A move value of x means that values $<= x$ can no longer affect the maximum move value at this node.

The efficiency of alpha-beta search improves greatly if it searches better moves first, obtaining beta cutoffs and stronger alpha-beta bounds earlier. As a result, alpha-beta is almost always used with a heuristic *move ordering* function to sort moves in order of likely quality. Since searching the entire game tree is infeasible, one stops at a limited depth and calls a heuristic *evaluation function* to estimate the value of the position based on features of the board state. In actual timed play, one usually also uses *iterative deepening* - iteratively re-searching the same position with increasing depth limits - to run until the desired amount of time is consumed.

It is worth noting that a variety of subtle details in the implementation of these algorithms can make a significant difference in performance. Like other strong programs, SHARP carefully manages these details. To give two examples:

- Upon a timeout during iterative deepening, the results of the interrupted search are used. In particular, if that search finds any move superior to the first move (which by ordering is always the best move of the previous iteration), that move is chosen. This gains a significant fraction of a ply in average effective search depth.
- When sorting moves for move-ordering, often a $O(n^2)$ *selection sort* is better than a more asymptotically-efficient sort. Selection sort can be done incrementally, eliminating the need to sort more than a few elements if a beta cutoff occurs early.

One other detail is that SHARP defines the depth of a search in steps rather than in moves. That is, it uses a formulation of the game where each *ply* of the search is a single step and the side-to-move only changes every four ply. Because of Arimaa's high branching factor, this improves performance by avoiding large lists of moves. However, as part of one of the recent innovations contributing to SHARP's success, the search also does not strictly adhere to recursing one step at a time. These details are elaborated on below in Section 5.4.

[5]The *minimax value* is the best result that the player-to-move can force from that position with optimal play assuming the opponent responds optimally.

[6]This is the "negamax" formulation of minimax search, which is the formulation virtually always used in practice.

4.2 Standard Enhancements

SHARP uses a variety of search enhancements that are standard in alpha-beta searchers for other games, notably Chess. We list and describe seven specific enhancements below.

4.2.1 Transposition Table

In Arimaa, different sequences of steps or moves that lead to the same final game state, or *transpositions*, are common. Therefore, a large hashtable, or *transposition table* is used to cache results to avoid repeating work[7]. As is usual, the contents of the table are preserved between top-level iterations of the search, and the best move is also stored so that in subsequent iterations that move can be searched first to improve move ordering. In SHARP, the table also caches the values of direct calls to the evaluation function[8].

In SHARP, transposition table entries are 128 bits, composed of 64 bits recording the full hash key, 32 bits for the best move, 21 bits for the evaluation, and 11 bits for the search depth and for flag values indicating the kind of result stored. In the event of a collision, a new entry always overwrites the existing entry. This is the simplest possible *replacement scheme*. A more sophisticated replacement scheme, such as a bucketed scheme based on entry age or depth, might be an avenue for future improvement.

4.2.2 Killer Moves

In wide variety of games, a heuristic that improves move ordering amazingly well is to guess that a move that performs well in one position will, if legal, be good in other positions. Accordingly, the *killer move* heuristic records a move whenever it causes a beta cutoff and ranks that move early in the ordering at any subsequent node at the same ply-level of the tree (Marsland, 1986). In Arimaa, this heuristic alone improves the speed of search by a factor of tens or hundreds or more depending on the depth of the search.

SHARP tracks 8 killer moves at each ply (2 per ply within the quiescence search), evicting older killers to make room for newer ones according to a FIFO (first-in, first-out) scheme. Unlike standard practice in Chess bots, killer moves are allowed to be captures.

4.2.3 History Heuristic

Another technique for move ordering based on a similar idea is known as the *history heuristic*. At the start of the search, a table with a counter for every possible move is initialized. After searching each node, the best move at that node has its counter incremented, and the values of these counters are used to order future moves (Schaeffer, 1989). Whereas the killer move heuristic captures tactics common to multiple local branches of the search, the history heuristic captures broader trends across the entire search about what moves tend to be better or worse.

Since in Arimaa a table of all possible legal moves would be too large and sparse, SHARP uses a table indexed only by the first step of the move (56 * 4 = 224 possibilities), or the first two steps for pushes and pulls (approx. 56*4*4 = 896 possibilities). A separate table is maintained for every ply in the search.

4.2.4 Quiescence Search

A depth-limited tree search will often return bad results because near the leaves, it will often misevaluate useless and losing threats as good because the search will end without the opponent having turns to refute these threats. This and similar problems are referred to as the *horizon effect*.

A well-known technique to fix this is to perform a *quiescence search* at leaf nodes, where the search is extended so long as the position appears to be tactically unstable (e.g. has a hanging piece), and to limit the cost of doing

[7] Note that in alpha-beta, frequently the result of a search is not an exact value, but rather an upper or lower bound. This necessitates care in determining whether a cached result is sufficient to prune the search of a repeated node when alpha-beta bounds have changed.

[8] This not always done in Chess programs. In SHARP, the evaluation function is by far the most costly part of the program, making it a clear win to cache it.

so, by examining only moves likely to resolve the instability (e.g. capturing moves) (Marsland, 1986).

In Chess, simply searching heuristically-promising capturing moves is already quite good, making quiescence search easy to implement. Most bad tactics in Chess that falsely appear favorable with shallow depth, such as capturing a defended piece, are refuted by a capture, such as recapturing with the defender.

By contrast, in Arimaa, many bad tactics are not simply refuted with captures. In practice, resolving tactical positions frequently involves determining whether a piece can be defended in four steps or whether an effective counterthreat can be made. However, the branching factor makes it difficult to include enough such moves without exploding the size of the search tree and degrading performance.

Because of this, SHARP, atypically compared to Chess programs, uses a quiescence search that is itself severely depth-limited. The quiescence search is never longer than 3 moves (12 steps). To further limit branching, only the first layer generates the full variety of moves needed to resolve common tactics, with deeper layers being more restricted. While not ideal, these choices strike a balance between performance and tactical strength.

The following describes the classes of moves generated in each layer of the quiescence search:

- Layer 0 (if initially 1-3 steps left): Goal defense, capturing moves, rare situational moves[9], goal threats, capture defense, capture threats, pushes and pulls of trap defenders, other "tactical" moves.
- Layer 0 (if initially 4 steps left): Goal defense, capturing moves, rare situational moves, goal threats, capture defense.
- Layer 1 (next 4 steps): Goal defense, capturing moves, rare situational moves.
- Layer 2 (next 4 steps): Goal defense, capturing moves, rare situational moves. Only invoked if layer 1 consumed all 4 steps.

4.2.5 Extensions

Aside from quiescence, it can be beneficial to extend the search in other situations. SHARP's extensions are simple. It extends the search depth by 1 step whenever a goal threat is played for which the minimum defense by the opponent requires at least 2 steps, and by 2 steps if the minimum defense requires at least 3 steps.

4.2.6 Late Move Reduction

Just as it is useful to extend the search in some cases, it is often useful to *reduce* the search depth of a move. With an effective move ordering, moves late in the ordering are unlikely to be good and therefore usually a waste of time to search. *Late move reduction* decreases the time spent on these moves by reducing the depth remaining by more than one ply when searching them, although if the move actually does appear to be good, the move is re-searched with the normal depth to verify the result (Romstad, 2007). One can view reduction as a "soft pruning" that doesn't entirely eliminate the cost of searching a move but mostly does so and avoids errors that would result from pruning it entirely. This technique is a key component in many strong Chess programs today.

Unfortunately, effective late move reduction is difficult in Arimaa because sufficiently effective move ordering is difficult. In the past, we have had at most limited success using this technique to improve the search. We also know of no other developer or published result so far that has mentioned any success with late move reductions in Arimaa[10].

However, in one of the biggest improvements for 2015, SHARP now *does* perform late move reduction with great effectiveness. This is due to the discovery of new and effective ways to identify good moves and improve the move ordering in Arimaa, which we will discuss further in Section 5.

[9] A few types of moves are generated only in special kinds of board positions according to various heuristics to fix situation-specific weaknesses, such in "elephant blockade" situations.

[10] Although, some authors have presented some methods for pruning, such as Zhong, who described a method involving the dependency structure of moves (Zhong, 2005) that is now used by many other Arimaa bots.

4.2.7 Multithreading

As with many other programs, SHARP takes advantage of multiple cores on a machine with a multithreaded implementation of alpha-beta. The algorithm used is conceptually similar to the "Dynamic Tree Splitting" algorithm developed for Chess and described by Robert Hyatt (1994). However it differs in that its communication methods are far simpler - rather than signaling other threads to request splitting of work, threads simply operate on a shared representation of the game tree, with operations synchronized with locks at each node.

As is typical with efficient multithreaded alpha-beta search implementations, the details of this implementation are fairly complex, and we refrain from describing them here.

4.3 Standard Arimaa-Specific Enhancements

SHARP also implements several Arimaa-specific search enhancements used by most other strong Arimaa programs. We list and describe three of them below.

4.3.1 Static Goal Detection

With four steps per move in Arimaa, it's nontrivial to check whether the player-to-move can win by scoring a goal on the current turn. Naively, doing so would appear to require a four-step search. However, as first described by David Fotland and elaborated on by Haizhi, it's possible to check this more efficiently by enumerating the classes of patterns that would allow a goal in four steps and writing specialized code (such as a decision tree) to directly determine whether one of these patterns is present on the board (Zhong, 2005; Fotland, 2006). While there are many patterns, there are few enough that enumerating them is barely feasible, and testing them directly provides a huge speedup over a search[11].

SHARP contains several thousand lines of code dedicated to testing whether goal is possible in up to four steps. The resulting code is fast enough that it can be called even on leaf nodes with no noticeable cost to performance, extending the effective search depth by a full four steps for goal threats in the deep endgame.

4.3.2 Static Capture Generation

For similar reasons, all strong Arimaa programs contain code for statically detecting whether a piece is capturable and for generating capturing moves. In SHARP, static capture generation is used in quiescence search and to improve move ordering.

4.3.3 Goal Relevance Pruning

SHARP also uses a technique first mentioned by Fotland and used in his early championship-winning program Bomb - when a goal threat is present on the board, moves that are too far away to defend it or to interact in dependencies with defending moves can be pruned (Zhong, 2005). As Fotland did not describe his method in detail, likely our implementation is different, and we describe it here:

Define a set of steps S as *relevant* to a goal threat if for every game state reachable by a move that stops the goal threat, at least one move reaching it begins with a step in S. Then, one can restrict move generation only to steps in S without any loss of non-losing moves.

The task is then to choose S as small as possible. The idea is that if the opponent threatens goal, S can be reduced to only a local set of steps that do prevent it or that due to local dependencies must be played before steps that would prevent it. Farther-away steps can be pruned because even if desirable, they can always be played later after the goal threat is dealt with.

[11] If it's not obvious why, consider the 1-step case. Whereas a search might involve generating and trying potentially dozens of steps, a direct test consists only of checking if there is an unfrozen rabbit on the 7th rank with an empty square in front, doable in just a few bitboard operations.

In general, given a goal threat, if the player to move has n steps left, all steps involving only squares more than $3(n-1)$ away from any squares involved in the goal threat (including squares necessary for unfreezing or trap defense for the goal threat move to be legal) can be excluded from S. Due to interactions involving the trap squares, this bound is tight in the general case, as shown in Figure 4.

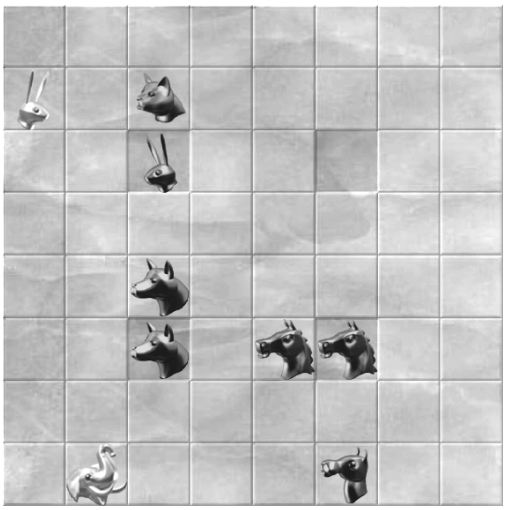

Figure 4: With 4 steps remaining, the step Mf1n, involving a square $3(4-1) = 9$ squares away from b7 (a square that if occupied could freeze the gold rabbit), cannot be pruned as irrelevant because it is part of the unique move Mf1n He3w Dc4n Cc7w that both prevents Gold's goal threat Ra7n and avoids sacrificing any pieces.

However, one can almost always restrict S much further. The "radius of relevance" can only expand by three squares per step as in Figure 4 around a trap with a singly-defended piece on that trap. Otherwise, one can use a radial bound of $2(n-1)$, or if freezing/unfreezing of pieces is irrelevant, only n. Due to a player's own blocking pieces occasionally interfering with ways to prevent a goal, often steps even closer than $n-1$ squares can be pruned, as shown in Figure 5.

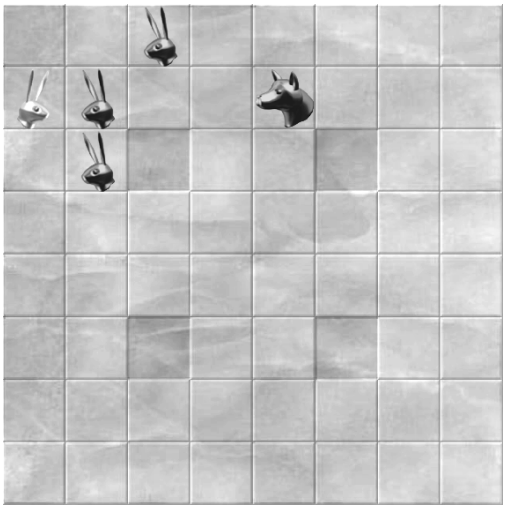

Figure 5: With 4 steps remaining, the step de7w, involving a square only 2 squares away from b7, can still be pruned as irrelevant. Silver's own rabbits prevent the dog from blocking or freezing the gold rabbit in 4 steps.

SHARP contains a couple hundred lines of bitmap-based code that takes advantage these details to compute a small S, so that when a goal threat is present, most steps can be pruned as irrelevant[12]. In the cases where a goal is actually unpreventable, this greatly speeds up the proof, and otherwise still serves to reduce transposition table load and search overhead in the endgame.

[12]If the goal threat is actually unstoppable, $S = \{\}$ by definition is a possible relevant set. And in fact, sometimes SHARP's local bitmap analysis is tight enough to generate the empty set for S and directly prove that the goal is unstoppable.

5. INNOVATIONS AND IMPROVEMENTS IN SEARCH

In just the last two years, SHARP has gained more than 600 Elo points in blitz games against its older versions[13]. Some of the gains were due to a plethora of minor changes and refinements, and some were due to large performance optimizations. However, a significant part was from a few key innovations directly aimed at solving the obstacle of Arimaa's large branching factor, one of the two major obstacles to strong computer play in Arimaa. The common thread between these innovations was improving the search's ability to order and filter good moves out from the thousands of other random and useless moves possible in any typical Arimaa position. We describe four innovations in 5.1 to 5.4.

5.1 Bradley-Terry Root Move Ordering

SHARP continues to use and benefit greatly from a move ordering function that we initially developed in 2011, detailed in Wu (2011). While not developed in the last two years unlike the other improvements discussed below, we present it again here due to the amount it continues to contribute to the playing strength.

This move ordering function is the result of training a slightly generalized *Bradley-Terry model* over thousands of expert Arimaa games to learn to predict expert players' moves. A Bradley-Terry model models the winner of a competition between two agents i and j probabilistically as:

$$P(\text{i wins}) = \frac{\gamma_i}{\gamma_i + \gamma_j}$$

where γ_i and γ_j are model parameters representing the "strengths" of i and j.

We can generalize this both to teams of agents and to competitions between more than two agents or teams by defining the strength of a team of agents $T \subset [1, ..., n]$ to be:

$$\gamma(T) = \prod_{i \in T} \gamma_i$$

And given competing teams $T_1, ..., T_n$, modeling the probability that T_j wins to be:

$$P[T_j \text{ wins}] = \frac{\gamma(T_j)}{\sum_{i=1}^{n} \gamma(T_i)}$$

We apply this model to predict expert moves by considering each position in an expert game to be a "competition" between the available moves, where the winner is the move actually played. Each move is a "team" of the various features associated with that move, where the possible features range from things like "moved a silver horse to d4" to "captured an opponent's horse" to "improves a heuristic trap-defensiveness score for c3 by 5 points".

Given a large number of training games, we then perform a maximum likelihood estimation to choose the model parameters $\gamma_i, i \in [1, ..., n]$ where γ_i is the "strength" of the ith possible feature, that maximize the likelihood of the data given the model[14]. The resulting model assigns to each move in a position a probability that the move would be chosen by an expert player. This can then be used for move ordering by ranking all moves in order of most likely to least likely.

Using a set of several thousand binary features, including features indicating source and destination and piece type, pushes and pulls, captures and goals, capture and goal threats and defense, trap defense, step dependency structure, and a few other types of features (see Wu (2011) for a more detailed list), and training on several thousand rated games by players rated 2100 or higher on arimaa.com, we obtain extremely good move prediction. The resulting move predictor captures the expert moves within the first 1% of the move ordering more than 80% of the time, and within the first 20% of the ordering more than 99% of the time!

At the root node, SHARP uses this move ordering function with great effectiveness. The search almost always finds a good move rapidly, particularly when the former best move is proven unsound on an iteration. Furthermore, with so few expert moves occurring past the first 10% or 20% of the ordering, it can reduce aggressively

[13] Implying winning chances of around 97%!

[14] One can also take logs, defining $\delta_i = \log \gamma_i$. This reveals that we are effectively performing a type of multivariable logistic regression with a linear model.

N	1	2	5	10	100	1000	1%	5%	10%	20%	40%	60%	80%
Expert%	15.0	22.5	34.2	44.4	78.3	96.8	81.0	94.4	97.5	99.2	99.8	100	100

Table 1: Out-of-sample percentage of expert moves falling within the top N moves of the root move ordering as of 2015, for different values of N. Test set is 21K game positions from players rated 2100+ on arimaa.com. Average branching factor was 16567.

with little risk of reducing good moves. All moves past the approximately first 10% of moves are reduced by 1 step and moves past the first 20% are reduced by 2 steps, gaining a large speedup.

Unfortunately, the resulting move ordering function is far too slow to be applied except at the root node. With the current implementation, ranking and ordering all legal moves in a typical position takes about 0.1 seconds, which is far too slow to apply deeper in the search tree.

Despite that, the success of this technique served as a proof of concept back in 2011 that it was possible to filter the set of legal moves down to only a tiny subset with hardly any loss of good moves based on only local features and properties of those moves. The knowledge that this was possible led to experimentation with many other ideas to see if something similar could be done with enough speed to be used within the search tree instead of only the root. Ultimately, this was what gave rise to the biggest improvements in 2014 and 2015.

5.2 Move Generation and Ordering Within-Tree

Whereas the method above solved the problem of move ordering and pruning at the root as early as 2011, it took until 2015 for SHARP to acquire similarly-effective methods suitable for use within the search tree.

The following list describes the current move generation and ordering structure for the main (non-quiescence, non-root) tree search. At each node, SHARP generates and searches the following moves in order from top to bottom[16]:

1. Transposition table move (if available)
2. Killer moves 1-8 (if legal)
3. Capture moves (ordered by size of piece captured)
4. Tactical moves (ordered by history heuristic + movetype[17])
5. Transposition table move prefix (first step, push, or pull only) (depth reduced by 1 step)
6. Other pushes, pulls, and single steps (ordered by history heuristic + movetype) (depth reduced by 1 step)

The key breakthrough was the development of a set of "tactical move generators", item 4 in the above list, which serve largely the same purpose as the Bradley-Terry ordering model for the root. They produce a set of "tactical" moves that, despite only amounting to a few percent of the legal moves in a position on average, contain a large fraction of the likely valuable moves in a position, enabling reductions of the rest.

Of particular note is the degree to which moves can be reduced. Each step, push, or pull in items 5 or 6 is reduced by only one step, but this reduction occurs at each level of recursion. As a result, moves composed entirely of steps that fail to be generated by earlier items (most moves), will be reduced by as many as four steps over the up-to-four levels of recursion for that turn!

A last thing to note is that the generated move list, rather than only containing single steps or only full moves, contains a mix of partial moves of all different lengths, mixing single steps together with two-step pushes and

[15] Having the best move almost always show up in the first few percent of searched moves also synergizes well with the minor detail in Section 4.1 about using the results of an interrupted search.

[16] In the presence of a goal threat, item 4 in this list is skipped, the moves from item 6 are generated using goal relevance pruning, and no reduction in depth is performed.

[17] A minor Arimaa-specific heuristic taking into account the direction and type of a move is used when the history heuristic is near zero and fails to distinguish moves.

pulls and potentially even longer moves, such as capture and tactical moves. This design choice itself is actually another major recent improvement, and also one of the key factors enabling the effectiveness of the late move reduction and boosting the gains from move ordering.

We describe both the tactical move generation and this design choice further in the next sections.

5.3 Tactical Move Generation

The within-tree counterpart to the Bradley-Terry root move ordering model, tactical move generation was perhaps the most significant single innovation in SHARP's search in 2015. It consists of a large set of move generators that generate a small set of "tactical" moves most critical and likely to be good in a position. These include the following classes of moves.

- Up to 4-step pushes and pulls of opponent's pieces on a square adjacent to a trap.
- Pushes and pulls of opponent's pieces that are heuristically likely to threaten them.
- Moves that add defenders to "poorly-defended" traps.
- Moves that unfreeze and/or run-away own pieces that are threatened with capture.
- Moves that freeze "important" opposing pieces.
- Steps around threatened pieces that are heuristically likely to block runaway attempts.
- Up to 4-step moves of the elephant that end the elephant on "useful" squares.
- Moves that match a variety of patterns for severe goal threats and/or are likely forced wins-in-2.
- Many more classes of moves involving further kinds of tactical conditions and patterns.

The design of these generators was directly motivated by the performance problems of the root move ordering. The root move ordering suffers in performance because it is "retroactive". It requires generating legal moves, playing them out, and sorting them after-the-fact. This is precisely analogous to the way that checking for 4-step goal or capture by searching and testing possible moves is far slower than a direct pattern-based check. It stood to reason that the performance of feature-based move ordering could be improved the same way - by directly generating candidate good moves in a forward-looking manner.

Just as with the Bradley-Terry model, we used data from expert games heavily. However, it became obvious early on that including all the kinds of good moves that strong Arimaa players play would be counterproductive. Often good moves in Arimaa are "quiet" moves, where there is no urgent fight and the precise choice of move does not make a major tactical difference. In these cases, the set of moves generated would need to be quite large to have a significant change of including the move actually played by an expert. And moreover, quiet moves are not ones that would benefit much from special search, precisely because of the absence of major tactics.

Therefore, rather than the obvious idea of maximizing the proportion of expert moves generated, we instead minimized the amount by which SHARP's evaluation of the best move produced by any generator was worse than SHARP's evaluation of the expert move[18]. In a bootstrapping fashion, this leveraged the program itself to judge when an expert move was tactically critical instead of merely one of many reasonable moves.

The development and testing cycle followed consisted of five steps.

- By hand, look at positions with large evaluation differences and identify the most common kind of critical move or tactic not being generated.
- Write a move generator that generates that class of moves.
- Observe that the generator is too inclusive and increases the branching factor too much, or that it is too restrictive and fails to produce enough instances of that class of moves.

[18] Treating the difference as 0 if SHARP believed a move generator's move was better than the expert move, and also capping it at 6000 millirabbits if it was worse by more than that. Evaluation consisted of a quick 5-step-deep search.

- Iterate on and refine the move generator and the heuristics used until it generates enough of the target class of moves while increasing the total number of legal moves generated by all generators by an acceptably small amount.
- Add the new move generator and repeat.

The results were a huge success. On average, the resulting set of move generators is capable of capturing about 97% percent of the evaluation difference between expert moves and passing, giving up an average of merely about 6 centirabbits per move[19], while only generating about 3% percent of all of the legal moves when run recursively over the course of the four steps of the turn to produce a complete move.

In the search, the presence of these move generators provides two major benefits. Just as with the root ordering, because of the effectiveness with which they identify critical moves, they allow aggressive reductions of all other moves. Secondly, some of them are used in the top layer of quiescence search, providing 1-4 steps of extra tactical depth at minimal cost. Adding these generators immediately gained 80 Elo rating points in self-play testing, the largest gain any single change to the search has produced in the last several years.

5.4 Game Formulation and Search Structure

The other major innovation in SHARP's search is a design choice that relates to the fundamental structure of the search. With four steps per move in Arimaa, there are two potential formulations of the game for the purposes of search:

- Movewise formulation: Each ply or recursion level in the search consists of a normal up-to-four-step legal move. The side-to-move changes every ply. Branching factor ≈ 10000.
- Stepwise formulation: Each ply or recursion level in the search consists of a single step. The side-to-move changes only every 4 ply. Branching factor ≈ 30.

Multiple authors have discussed the tradeoffs between these two formulations in earlier work, often favoring the stepwise formulation due to the overhead of generating and sorting large arrays of moves in the movewise formulation (Zhong, 2005). Among other things, generating tens of thousands of moves only to return due to a beta cutoff after the examining the first few is a huge waste of time, necessitating some form of batched incremental move generation, which itself poses other difficulties. Up until the last couple of years, for these reasons SHARP also used the stepwise formulation of search.

However, the stepwise formulation also has disadvantages. Since alpha-beta search is depth-first, it reduces the effectiveness of move ordering by forcing the search to consider moves that begin with the same steps together. If for example the four-step move Ee6s Ee5s Ee4s Ee3w is heuristically likely to be a good move but no other moves beginning with Ee6s are likely to be good, then in the event that the search first explores Ee6s, Ee5s, Ee4s, Ee3w and the move turns out not to be good, it will be forced to then uselessly explore every other move in the subtree beginning with Ee6s before being allowed to try better alternatives.

SHARP solved this problem in 2014 with the innovation of using a hybrid formulation. At any given node, the move list contains a mix of moves ranging from one to four steps[20]. This allows promising moves with three or four steps to be tried without committing to examining a large number of other moves with the same prefix. The transposition table move and killer moves are now always recorded with the full four steps[21], allowing them to be tried in rapid succession. Similarly, the static capture move generators also generate the entire capture sequence rather than merely the step that begins the capturing move, and pushes and pulls are always generated with both steps together.

Switching from an originally stepwise formulation to this hybrid formulation by itself resulted in about a 20% speedup for 9-step searches and much more for deeper searches. But more importantly, coupled with tactical

[19] That is, per move on average giving up an amount of advantage that SHARP judges to be 6% of the value of a rabbit in the opening. A rabbit is similar in value to a pawn in Chess.

[20] Unless reduced, moves still decrement the remaining depth of the search proportional to number of steps they contain, with pass steps decrementing by the number of steps left in the turn.

[21] This involves using a principal-variation recording mechanism to extract out the full 4 steps, since the first time these moves are discovered to be best, they are often played over more than one level of recursion.

move generation, this hybrid formulation enables effective late-move reduction. A hybrid formulation makes it easy to separate promising moves from other moves and protect them from being reduced. By contrast, a stepwise search gives only very coarse control over the search - one can only differentiate between moves in a forward manner by their first steps.

6. POSITIONAL EVALUATION

Recent improvements in the evaluation function have also played a major rule SHARP's strength and success. Whereas we managed the first obstacle to strong play in Arimaa, the branching factor, with a series of new innovations to greatly improve the move ordering and selectivity of the search, the second obstacle of crafting a positional evaluation accurate enough to guide the program through strategic positions was handled with simple dedicated engineering - a great deal of hand-tuning and testing guided by expert knowledge (see 6.3). To give the improvements a proper place we start with the description of a testing procedure (6.1) and of a developement cycle (6.2). We describe new components and improvements in 6.4.

6.1 Testing

Positional evaluation in Arimaa is difficult and complicated. Although recently there have been promising developments in automated evaluation learning and tuning (Hrebejk, 2013), the strongest Arimaa programs continue to have hand-written evaluation functions. Like them, SHARP's evaluation function is hand-written. It spans thousands of lines of code and involves dozens of tables and hundreds of numeric parameters and constants.

The only thing that made this complexity manageable was having an effective way to test and inspect the results of changes. Two methods in particular have proven highly valuable, and while these methods are well-known and by no means groundbreaking, they are important, and we present them next in Subsections 6.1.1 and 6.1.2.

6.1.1 Self-Play

Self-play was the most important way of testing changes to SHARP[22]. Every change to the evaluation function (as well almost any change to the search) was tested by playing at least several hundred games against a variety of older versions, often in multi-way tournaments. Some games used very fast controls (such as 4 seconds/move) and some used more normal time controls (such as 15 seconds/move), with faster time controls used to collect data rapidly and slower ones used to confirm differences with less bias.

To handle these multi-way tournaments and compare multiple versions of SHARP at once, we used the well-known BayesElo program (Coulom, 2010) and/or a set of custom tools[23]to analyze the game results assuming the Elo rating model and compute confidence bounds on the relative ratings of different versions. Changes were only accepted if statistically significant or if based on prior expectations the changes were likely to at least not cause harm (ex: a bugfix eliminating a source of random noise in the evaluation function).

While noisy and despite its possible biases, as far as we know, simply playing games is by far the most accurate way to evaluate the effect of a change. Development relied extensively on self-play to ensure that no changes significantly harmed the strength of the program.

6.1.2 Low-depth Play and Hand-Analysis

The second method used to test SHARP's evaluation was to play it against itself or another human with only a 4-step-depth search and watch the game by hand. Modulo details such as quiescence search, a 4-step search selects the move that directly maximizes the evaluation function. As a result, this testing method often revealed strange quirks and preferences by the evaluation function that were clearly wrong or pathological.

[22]Testing against a variety of other strong opponents rather than just oneself would likely be preferable, but at this point there are no other available programs strong enough!

[23]One such custom tool, for example, checks for any statistical intransitivities that would suggest the rating model used was a poor fit.

6.2 Development Cycle

All of these methods facilitated the basic development cycle for the evaluation function. The cycle consisted of the following five steps.

- Identify a deficiency, whether from an observed mistake in a game, expert commentary or feedback, seeing an obvious strategic misjudgment in low-depth play, etc., using search traces to trace bad play down to specific misevaluated board positions[24].
- Using instances of this problem as test positions, iterate on different possible implementations of a new evaluation term or a modification of an existing term to get the program to produce the right evaluation for those positions.
- Use low-depth-play testing to expose whether SHARP now appears to be doing unintended things as a result of side effects of the change. If so, go back and iterate further on the implementation.
- Check the performance cost of the new evaluation code, optimizing or simplifying and iterating further if the performance cost is too large.
- Once the intended evaluation is implemented, test it with a few thousands of games of self-play to determine whether it is ultimately effective or not.

6.3 Basic Evaluation Components

SHARP's evaluation function, for the most part, is linear, simply summing together terms for the various different components:

$$\text{Eval} = \text{Material} + \text{Piece-square tables} + \text{Trap control} + \ldots$$

Many of these terms are themselves sums of subterms where each subterm corresponds to a piece or square or board pattern and is a function of several features, often combined by applying lookup-table based transforms and taking a product. For example:

$$\text{Piece Threat Score} = \sum_{p} \text{Value}(p) * f(\text{TrapScore}(p)) * g(\text{ThreatDistance}(p)) * ...$$

Nine important top-level components are described below.

6.3.1 Material

In Arimaa, like in Chess, the material difference is the most important factor in evaluating a position.

One major difference between Arimaa and Chess is that in Chess, static piece values are a moderately good approximation throughout the game to the average value of a given material balance[25]. In Arimaa, this is not true. Since only relative piece strength matters for pushing, pulling, and freezing, the value of a piece depends heavily on the distribution of opposing piece strengths, with heavier pieces usually devaluing and weaker pieces usually increasing in value as pieces are traded.

To handle these nonlinearities, SHARP uses the "HarLog" material evaluation formula, developed by an anonymous arimaa.com user during a forum discussion thread (Anonymous, 2009):

$$\text{HarLog} = G * \log(\frac{\text{GR} * \text{GP}}{\text{SR} * \text{SP}}) \sum_{p \in \text{NonRabbitPieces}} (1 + C_p)/(Q + O_p)$$

[24] Having an easy-to-use search trace was invaluable, particularly one allowing interactive exploration of the game tree.
[25] Commonly, Pawn = 1, Knight = 3+, Bishop = 3+, Rook = 5, Queen = 9.

where GR and GP are the number of gold rabbits and pieces, SR and SP are the number of silver rabbits and pieces, O_p is the number of opposing pieces stronger that p, $C_p = 1$ if $O_p = 0$ and else $C_p = 0$, and $G = 0.6314442034$ and $Q = 1.447530126$ are constants.

While somewhat arbitrary, in practice this formula gives values for piece trades that agree closely with evaluations from expert players.

6.3.2 Piece-Square Tables

SHARP also uses the common technique of piece-square tables as a cheap and simple way to encourage different pieces to develop to useful locations, particularly the elephant. For each piece, a small value is added or subtracted using a table lookup based on its location and the number of stronger opposing pieces.

Additionally, for rabbits, to capture the effect that rabbit advances are of slight negative value early on but become good once pieces are traded, an additional table, indexed by the y-coordinate of the rabbit and the sum of twice the number of non-rabbit pieces of the opponent plus the number of rabbit pieces of the opponent, penalizes advancement when the opponent has a lot of material and rewards it when the opponent has less.

-450	-350	-250	-180	-180	-250	-350	-450
-160	-130	-95	-70	-70	-95	-130	-160
-90	-45	0	0	0	0	-45	-90
-60	-5	15	25	25	15	-5	-60
-60	9	35	35	35	35	9	-60
-90	-70	-10	35	35	-10	-70	-90
-180	-140	-115	-70	-70	-115	-140	-180
-450	-350	-250	-180	-180	-250	-350	-450

Figure 6: Piece-square table for the silver elephant

6.3.3 Trap Control

Perhaps the most important positional feature after material in Arimaa is trap control, referring to the ability and ease with which a player can locally defend against material threats in a trap, or conversely, the ability of a player to safely make material threats in that trap.

SHARP estimates trap control by summing over each piece the product of a factor that depends on the Manhattan distance of that piece from the trap with a strength factor that combines both the global and the local material rank of the piece. A variety of additional adjustments are applied, including:

- An adjustment for who has the strongest piece nearby for a couple different choices of radii.
- A bonus for controlling the defense squares closer to the edge of the board (pieces near the edge are less easily dislodged).
- An adjustment for having a weak piece on the trap square itself, varying from a bonus when trap control is otherwise weak (the piece helps defend) to a penalty when otherwise strong (the piece interferes with making threats).

For each trap, the resulting trap control score is fed through a logistic function $x \mapsto 1/(1 + e^{-kx})$ to produce the final evaluation. The trap control scores themselves are also used as inputs to a variety of other evaluation components.

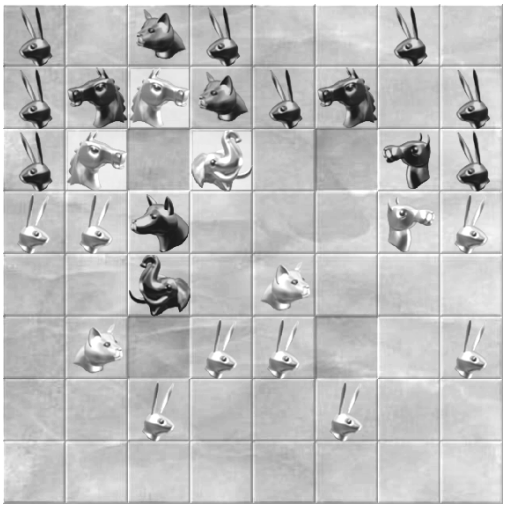

Figure 7: Silver has lost trap control at his c6 home trap, with Gold solidly holding the b6, c7, and d6 squares

6.3.4 Freeing and Domination Distance

For each piece, SHARP approximately calculates the number of steps (0-5) required to *free* a piece to move, where a piece is free if it is unfrozen and has a legal step, push, or pull available to it that does not sacrifice it[26].

Additionally, for each piece, SHARP approximately computes the number of steps (0-5) required by the opponent to *dominate* it, which is to place a stronger opposing piece adjacent to it such that the opposing piece is free.

These values are not used directly in the evaluation score, but rather are used as inputs into most other evaluation components, as they are important features to evaluate how defendable or threatenable a piece is.

6.3.5 Hostages

A common strategic pattern in Arimaa consists of one piece, frequently the elephant, holding another piece "hostage" by pinning and threatening it with capture. This ties the opponent's elephant to defense, ideally giving the hostage-holder the strongest remaining piece on the rest of the board.

SHARP evaluates hostages in a complicated manner involving a wide variety of features, some of which are:

- The trap control score of the relevant trap.
- The pieces involved and whether holding the hostage actually does result in the strongest free piece elsewhere.
- The number of steps required to capture if defense is abandoned.
- The degree of advancement of pieces by the hostaged side (a common counter against a hostage pattern is a swarm of the trap with smaller pieces to free the tied-down elephant from defense).

6.3.6 Frames

Another common strategic pattern in Arimaa is a "frame", where a piece is stuck on a trap, pinning another piece, usually the elephant, to defense. Frames are evaluated using a similarly wide variety of features, including:

- The strength of the piece being framed.

[26] An exception is made where a piece is considered free if it is unable to move only due to friendly pieces blocking it in, would still be unfrozen and unsacrificed if those pieces were to move away, and the situation is not part of a larger "blockade" formation.

- The domination distance of the pieces holding the frame.
- The strength and quantity of pieces needed to hold the frame.
- The ability of the opponent to rotate out and replace the stronger pieces with weaker ones.

6.3.7 Elephant Mobility and Blockades

Another relevant feature is how physically free the elephant is to move - having an elephant that can move and reach many parts of the board quickly is better than having one blockaded or hemmed in.

SHARP uses bitmap operations to estimate the number of squares reachable by the elephant using varying numbers of steps. These computations take into account steps required for stepping, pushing, or pulling blocking pieces, as well as the fact that the elephant cannot step through a trap by itself. The resulting counts, weighted by centrality of squares, are summed together and fed through a lookup table to produce a final score.

Additionally, in the event that the elephant is decentralized and nearly completely blockaded, additional routines are called to estimate the value of the blockade taking into account features very similar to those for a frame:

- The exact position of the elephant and the degree of blockadedness of the elephant.
- The domination distance of the pieces holding the blockade.
- The strength and quantity of opposing pieces needed to hold the blockade.
- The ability of the opponent to rotate out and replace the stronger pieces with weaker ones.

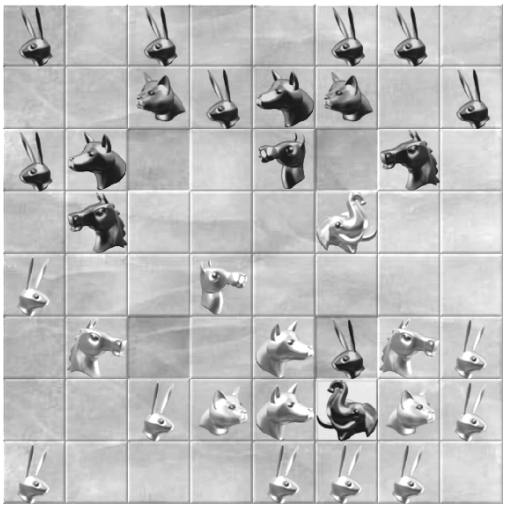

Figure 8: Silver's elephant is blockaded at f2. Silver is completely lost.

6.3.8 Piece Threats

Individual pieces, if threatened with hostaging or capture, can be liabilities. Features relevant in piece-threat evaluation include:

- The trap control score of the relevant trap.
- The number of steps required to capture the piece, including steps to push away defenders of the trap.
- Whether or not the piece is frozen, and whether there are friendly pieces in front of the piece.
- Whether or not it is a rabbit (rabbits can't retreat, making their defense uniquely difficult).

Additionally, the value of the piece itself is computed by considering what the HarLog score would be with and without the piece on the board, and is used to scale the threat. To avoid overpenalizing, in the event that the piece is already being penalized directly for another feature (such as being hostaged), only the maximum of the two penalties is used.

6.3.9 Goal Threats

Highly advanced rabbits can be extremely valuable if they are supported with other pieces or they are in sparsely-defended regions, so that they exert goal-threat pressure on the opponent. Features used in their evaluation include:

- The y-coordinate of the rabbit.
- A score indicating how well-blocked the rabbit is from advancing to goal.
- The trap control score of the nearest opponent trap to the rabbit.
- A heuristic measure of the amount of supporting friendly strength around the rabbit.

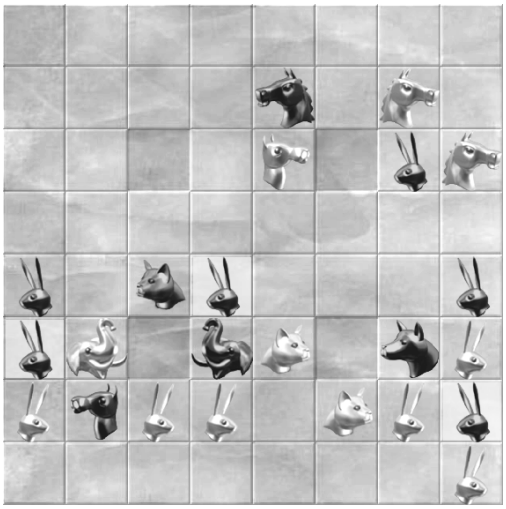

Figure 9: Silver is winning due to the severe goal pressure exerted by the marked rabbits.

6.4 New Components and Improvements

Prior to 2015, one of SHARP's major weaknesses was an understanding of how to place pieces efficiently and strategically. Such an understanding is necessary to play well in many non-tactical situations that involve gradual maneuvering and fighting for position. Although a variety of improvements were made in many areas of the evaluation function, the two largest improvements came from adding two major new components to fix this weakness, centering around the concepts of "piece alignment" (6.4.1) and "imbalances" (6.4.2). A third noteworthy new component involving nonlinearities and "swindles" is discussed in 6.4.3.

6.4.1 Piece Alignment

In Arimaa, "piece alignment" refers to the heuristic that pieces should placed near the opposing pieces that they most efficiently beat - camels should try to be near opposing horses, horses should try to be near opposing dogs, etc. Prior to 2015, a term for this did exist in the evaluation function but it was incomplete and failed to actually capture this heuristic and the nuances around it well.

The new term consists of a sum across each pair of non-rabbit pieces (p_1, p_2) where p_2 is (one of) the strongest opposing piece(s) weaker than p_1, taking into account the following features:

- The estimated number of steps needed for p_1 to dominate p_2, computed more approximately than the domination distance computation described earlier but with no cap on distance.
- The strength and quantity of the pieces of the two types.
- The y-coordinates of the pieces (attacking more advanced pieces is more valuable).

Piece alignment has long been recognized by human players to be important, and its absence from SHARP was one of the largest strategic weaknesses limiting the strength of the program. Implementing this feature properly resulted in an enormous improvement of nearly 80 Elo rating points in self-play testing. This is one of the largest gains ever observed for any single change in the last several years.

6.4.2 Imbalances

The other major new evaluation component in SHARP for strategic piece placement was a term for "imbalances", which refer to inefficiencies caused by having an overconcentration of pieces in the same region or side of the board.

SHARP computes this component of its evaluation in a very similar way to the way it computes the piece alignment value, but instead considers pairs of pieces owned by the same player rather than by different players, and instead of domination distance uses a slightly different notion of distance to assign penalties when many strong pieces are near each other or on the same side of the board.

Unfortunately, the current implementation of this feature fails to capture the strategic concept well enough to match expert opinion in some positions. It is likely that some more iteration and experimentation would produce a different design or feature set that would perform better. Nonetheless its addition was still a large improvement for 2015, gaining about 35 rating points in self-play testing.

6.4.3 Nonlinearities and Swindles

A final recent innovation in SHARP's evaluation function deals with a phenomenon of Arimaa that allowed human players to sometimes "swindle" computer programs out of otherwise won games.

In Arimaa, in situations involving goal threats and fights that threaten to end the game immediately, the values of most other positional factors, such as material, are reduced. Since pieces in Arimaa move slowly, in the event that a goal fight is occurring in a corner of the board, the game can easily be won or lost in a way determined only by the local state of that corner, making the global material balance of the board irrelevant.

Since most strong Arimaa programs, old versions of SHARP included, use evaluation functions that are essentially linear, they can sometimes be defeated in otherwise hopeless positions by sacrificing a large amount of material in order to initiate a goal attack. Often the program will happily take the offered material, oblivious to the fact it is made irrelevant by the attack.

Ideally, one would like to solve such a tactical situation primarily via search, but this is difficult. Moreover, the phenomenon still applies to a lesser degree even if the goal threats are not likely to end the game right away - they can still increase the overall volatility of the position.

SHARP now handles this by explicitly modeling the situation. In particular, the "goal threat" component of the evaluation function, in addition to outputting a term that gets linearly added into the score, also outputs probabilities g and s that Gold and Silver, respectively, win or otherwise turn the game around in the short term due to the volatility of their goal threats. The normal linear part of the evaluation, x is computed and then fed through a sigmoid to produce a probability $p = 1/(1 + e^{-kx})$ that Gold wins the game if the game is instead won over the long term. The final output of the evaluation function as a whole is simply the total probability that Gold wins:

$$\text{Final eval} = 1 * g + 0 * s + p * (1 - g - s)$$

In certain positions, this change makes it noticeably harder to swindle SHARP, since when ahead, SHARP is more likely to choose a "safe" and "conservative" move as opposed to a more "greedy" move. Additionally, it is more likely to attempt to start goal fights when sufficiently behind. And indeed, in self-play testing, this change resulted in an improvement of about 15 Elo points.

7. RESULTS AND FUTURE WORK

Over the last few years, SHARP has improved greatly in all areas of play. SHARP demonstrates that it is possible to achieve effective reduction/pruning in Arimaa by leveraging expert game data using local move features and patterns, and to meet the challenge of producing an accurate positional evaluation by employing good methods for iterative tuning and testing.

Year	2008	2010	2011	2012	2014	2015
Blitz (15s/move)	N/A	2045	2139	2256	2286	N/A
CC (2m/move)	1513	2035	1980	2117	2172	2487
Self-play gain	N/A	Untested	Untested	100-250?	≈200	≈400

Table 2: Elo ratings of versions of SHARP on arimaa.com as of July 16, 2015 at different time controls, along with estimated single-threaded self-play gain over the previous version at fast testing speeds.

However, SHARP's heuristics and algorithms are far from perfect. There are multiple promising avenues for further improvement in both search and evaluation. In the search, a variety of basic things have simply not been tried yet, ranging from a more sophisticated replacement scheme for the hashtable, to razoring and other heuristic pruning of nodes near leaves, to other enhancements such as "principal variation search" (Marsland, 1986). Tuning of the heuristics in components such as the tactical move generation could also yield improvements.

In the evaluation function, particularly with recent work demonstrating the feasibility of automated learning (Hrebejk, 2013), there is huge potential to improve SHARP by tuning the hundreds of different weights and coefficients on all of the features that have been implemented. Identifiable missteps and strategic misunderstandings in some of its recent games also suggest new features to add. Evaluation of positions in the opening remains a particularly weak part of SHARP's game, and almost certainly large improvements are possible.

While SHARP may have won the Challenge, there is still a small way to go to produce a bot that is unambiguously dominant over the top human players rather than merely competitive or favored, and beyond that there is still plenty of room to experiment and grow[27]. However, with our results Arimaa now crosses the threshold, leaving the ranks of the games holding out against effective computer play and joining the ever-growing pool of games in which computers are the strongest players.

8. ACKNOWLEDGEMENTS

I would like to thank my former undergraduate thesis advisor, Professor David Parkes, for his guidance and feedback on this paper. I would also like to thank Omar Syed as well as the Arimaa community as a whole for giving me the chance to work on this fun project and be involved in this amazing game.

9. REFERENCES

Anonymous (2009). Arimaa forum discussion thread. http://arimaa.com/arimaa/forum/cgi/YaBB.cgi?board=devTalk;action=display;num=1062013358;start=60. Accessed on 2015-07-12.

Coulom, R. (2010). Bayesian Elo Rating. http://www.remi-coulom.fr/Bayesian-Elo/. Accessed on 2015-07-15.

[27] As of now, July 2015, as a fun exhibition, SHARP with some human assistance is beginning a postal game against the "Mob", a team of human players including many top players, at time controls of a week per move. At such time controls the Mob is likely the favorite, but we'll see what happens as the game unfolds over the next year!

Cox, C.-J. (2006). Analysis and Implementation of the Game Arimaa. M.Sc. thesis, Maastricht University, The Netherlands.

Fotland, D. (2006). Building a World-Champion Arimaa Program. *Proceedings 4th International Computer and Games Conference* (eds. H. van den Herik, Y. Björnsson, and N. Netanyahu), Vol. 3846 of *Lecture Notes in Computer Science*, pp. 175–186, Springer Verlag, Heidelberg.

Haskin, B. (2006). A Look at the Arimaa Branching Factor. http://arimaa.janzert.com/bf_study/. Accessed on 2015-07-12.

Hrebejk, T. (2013). Arimaa challenge - static evaluation function. M.Sc. thesis, Charles University, Prague.

Hyatt, R. (1994). The DTS high-performance parallel tree search algorithm. https://cis.uab.edu/hyatt/search.html. Accessed on 2015-07-12.

Kozelek, T. (2009). Methods of MCTS and the game Arimaa. M.Sc. thesis, Charles University of Prague, Czech Republic.

Marsland, T. A. (1986). A Review of Game-Tree Pruning. *ICGA Journal*, Vol. 9, No. 1, pp. 3–19.

Miller, S. (2009). Researching and Implementing a Computer Agent to Play Arimaa. Thesis, University of Southampton, UK.

Romstad, T. (2007). An Introduction to Late Move Reductions. http://www.glaurungchess.com/lmr.html. Accessed on 2015-07-16.

Schaeffer, J. (1989). The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, pp. 1203–1212.

Syed, O. and Syed, A. (2003). Arimaa - A New Game Designed to be Difficult for Computers. *ICGA*, Vol. 26, No. 2, pp. 138–139.

Syed, O. and Syed, A. (2015a). Arimaa - The next challenge. http://arimaa.com/arimaa/. Accessed on 2015-07-12.

Syed, O. and Syed, A. (2015b). The Arimaa World Championship. http://arimaa.com/arimaa/wcc/. Accessed on 2015-07-12.

Trippen, G. (2009). Plans, Patterns and Move Categories Guiding a Highly Selective Search. *Proceedings 12th Advances in Computer Games Conference* (eds. H. van den Herik and P. Spronck), Vol. 6048 of *Lecture Notes in Computer Science*, pp. 111–222, Springer Verlag, Heidelberg.

Vivek Choksi, V. M., Neema Ebrahim-Zadeh (2013). Move Ranking in Arimaa. Accessed on 2015-07-12.

Wu, D. (2011). Move Ranking and Evaluation in the Game of Arimaa. Bachelor's thesis, Harvard College, Cambridge, Massachusetts, USA.

Zhong, H. (2005). Building a Strong Arimaa-playing Program. M.Sc. thesis, University of Alberta, Dept. of Computing Science.