Move Ranking and Evaluation in the Game of Arimaa

A thesis presented
by

David Jian Wu

To
Computer Science
in partial fulfillment of the honors requirements
for the degree of
Bachelor of Arts
Harvard College
Cambridge, Massachusetts

March 31, 2011

Abstract

In the last two decades, thanks to dramatic advances in artificial intelligence, computers have approached or reached world-champion levels in a wide variety of strategic games, including Checkers, Backgammon, and Chess. Such games have provided fertile ground for developing and testing new algorithms in adversarial search, machine learning, and game theory. Many games, such as Go and Poker, continue to challenge and drive a great deal of research.

In this thesis, we focus on the game of Arimaa. Arimaa was invented in 2002 by the computer engineer Omar Syed, with the goal of being both difficult for computers and fun and easy for humans to play. So far, it has succeeded, and every year, human players have defeated the top computer players in the annual "Arimaa Challenge" competition. With a branching factor of 16000 possible moves per turn and many deep strategies that require long-term foresight and judgment, Arimaa provides a challenging new domain in which to test new algorithms and ideas.

The work presented here is the first major attempt to apply the tools of machine learning to Arimaa, and makes two main contributions to the state-of-the-art in artificial intelligence for this game. The first contribution is the development of a highly accurate expert move predictor. Such a predictor can be used to prune moves from consideration, reducing the effective branching factor and increasing the efficiency of search. The final system is capable of predicting almost 90 percent of expert moves within only the top 5 percent of its choices, and enables an improvement of more than 100 Elo rating points. The second contribution is a comparison of several algorithms in reinforcement learning for learning a function to evaluate the long-term value of a position. Using these algorithms, it is possible to automatically learn an evaluation function that comes close to the performance of a strong hand-coded function, and the evidence presented shows that significant further improvement is possible. In total, the work presented here demonstrates that machine learning can be successful in Arimaa and lays a foundation for future innovation and research.

Acknowledgements

I would like to thank my adviser, David Parkes. His advice throughout the process of researching and presenting this thesis has been invaluable, and he has been a supportive and encouraging mentor. Additionally, I would like to thank Omar Syed and the Arimaa community for the development of this excellent game.

Contents

Chapter 1

Introduction

Arimaa is a game that was invented in 2002 by Omar Syed, a computer engineer. According to Syed, the original motivation for designing Arimaa came from the defeat of the world champion Chess player, Gary Kasparov, in 1997 by IBM's Deep Blue [31]. Syed wanted to see if it was possible to design a game that could also be played 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 [31].

To this end, every year an event known as the "Arimaa Challenge" is held, in which the current top computer program, running on 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, human players have soundly defeated the top programs in every match, with the exception of the most recent Challenge in 2010, where for the first time the top program defeated one of the defending players, Patrick Dudek, in a match with a score of 2-1 [28].

It appears there is still some distance to go to develop a computer program that can stand against the best human players. This is not due to a lack of effort either, as the online Arimaa community is fairly programmer-dense. Yet currently on the server at Arimaa.com, the top computer programs appear to be around 300 Elo points weaker than the best players.[1] And although the programs are steadily improving, due to the newness of the game, the population of human players is still growing and improving as well. It is not entirely clear how long human dominance in Arimaa will last, but for now, Arimaa joins the increasingly small group of abstract strategy games that resist strong computer play.

[1] The Elo rating system is a common way to compare the relative skill levels of players in two player games [11]. 300 Elo corresponds to a winning chance of around 85% under the Elo model.

However, it is also true that many approaches have not been tried yet. As far as we know, nobody has yet made a major effort to apply the wide range of tools in machine learning to this difficult game. Therefore in this thesis, we investigate several machine-learning algorithms for ranking moves and evaluating positions in Arimaa. Our goal is to take the first step in exploring and applying these techniques, which we believe have great potential for raising the level of computer play.

In the remainder of this chapter, we give an overview of the problem and our contribution. We begin by explaining the rules of Arimaa in Section 1.1. In Section 1.2, we discuss the properties that make Arimaa challenging. In the context of these properties, we describe in Section 1.3 our approach and contribution to the field. Finally, in Section 1.4, we survey the techniques currently being used by top programs and the alternatives that have been tried so far.

In Chapter 2, we attack the problem of heuristic move ordering in Arimaa. We apply an innovative method developed by Rémi Coulom [6] and later used also by Harrison [12] to predict expert moves in Computer Go, and we achieve similar success with it in Arimaa. Our resulting function is capable of high-quality move ordering and is good enough to be used for direct pruning during a tree search. Indeed, we were able to use it to great success in our own Arimaa program, Sharp, and as a result of this and other improvements, Sharp recently won the 2011 Arimaa Computer Championship [30].

In Chapter 3, we turn to the problem of learning a strong evaluation function for Arimaa. We present a simple framework for evaluating board positions using weighted linear combinations of features, and within that framework, we compare four different algorithms for learning the appropriate feature weights. While none of the algorithms is quite able to achieve the same level of performance as the hand-coded evaluation function we currently use in our Arimaa-playing program, our best result comes close, and we believe that with some improvements to the algorithms and the learning procedure, it should be possible to match or surpass it.

1.1 Rules of Arimaa

Arimaa is a deterministic two-player abstract strategy game played on an 8 x 8 board. The rows and columns on the board are labeled 1...8 and a...h, respectively, as shown in Figure 1.1. The two players are named *Gold* and *Silver*.

1.1.1 Setup

Prior to normal play, Arimaa begins with a *setup phase*, where both players place their pieces on an initially empty board. The players each begin with sixteen pieces of their color: 1 Elephant, 1 Camel, 2 Horses, 2 Dogs, 2 Cats, and 8 Rabbits.

Both players may place their pieces in any desired arrangement within the two rows closest to their side of the board. Gold places all gold pieces first, then Silver places all silver pieces.

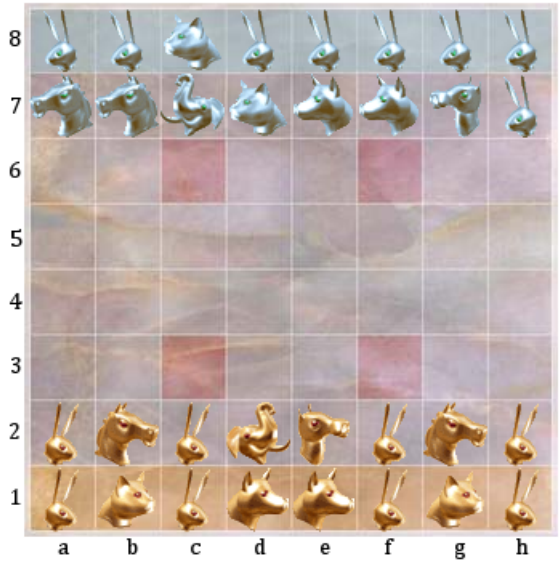

Figure 1.1: Example game position, just after the setup phase.

1.1.2 Play

Following the setup phase, play begins and the players alternately take turns, beginning with Gold.

Movement

During a player's turn, that player may make up to four *steps*. A *step* consists of selecting a piece of one's color and moving it to an empty adjacent square. Adjacent squares include only the squares immediately left, right, forward, or backward. All pieces move the same way, with one exception: rabbits may not move backward.

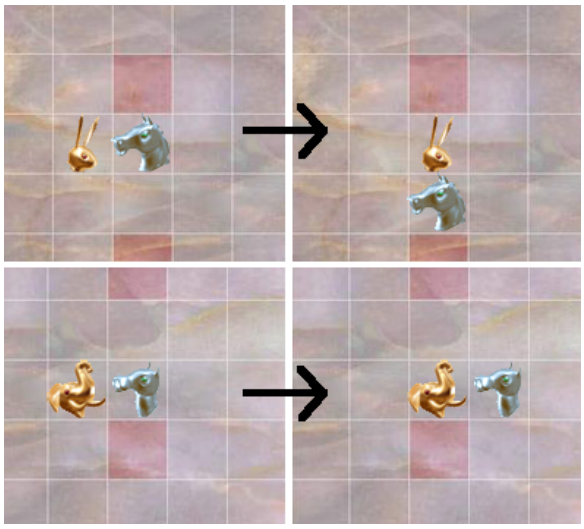

Figure 1.2: A silver horse steps down, pulling a gold rabbit. A gold elephant pushes a silver camel to the right.

Players may also use steps to *push* or *pull* their opponent's pieces by using their own pieces, as shown in Figure 1.2. A *push* consists of displacing an opposing piece to any adjacent empty square, followed by moving a friendly piece into the square previously occupied the opposing piece. A *pull* consists of moving a friendly piece into any adjacent empty square, followed by moving an opposing piece into the square just vacated from any square adjacent to the vacated square. In each case, if there are no adjacent empty squares, then pushing or pulling is not possible. [2]

Moreover, pieces are ordered by *strength*, and pieces may only push or pull pieces that are strictly weaker than they are. From strongest to weakest, the order of strength is elephant, camel, horse, dog, cat, rabbit.

Pushes and pulls count as two steps, since they involve moving both one's own piece and an opponent's piece one step each. Pushing and pulling simultaneously with the same piece is forbidden, but otherwise pushes and pulls can be made in any desired combination along with regular steps, up to a limit of four steps per turn.

Capture

The squares c3, c6, f3, f6, are *trap* squares. Whenever a piece is on a trap square but is not *guarded* by a player, it is *captured* and removed from the board. A piece or square is

[2]Note also that when pushing and pulling, the two pieces do not have move in the same direction. Any adjacent empty square is legal.

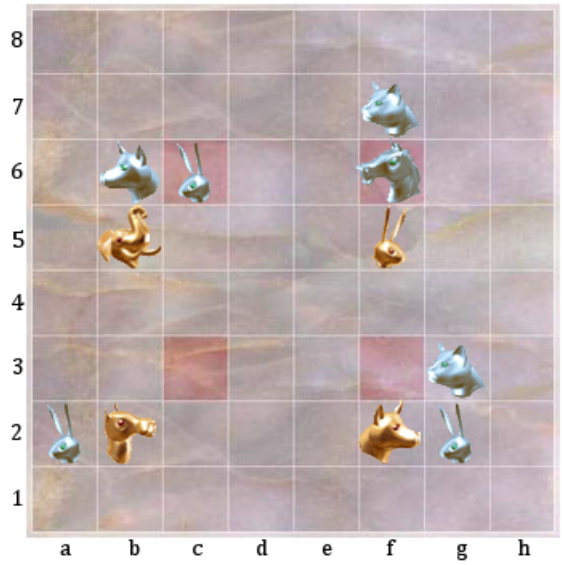

Figure 1.3: Examples of capturing and freezing.
Upper-left corner: If the gold elephant pushes the silver dog up or left, it captures the silver rabbit on the trap at c6 because the dog was the only silver defender of that trap.
Upper-right corner: The silver horse on the f6 trap can capture the gold rabbit by stepping left or right while pulling the gold rabbit north onto the trap.
Lower-left corner: The rabbit on a2 cannot move because it is frozen by the adjacent gold camel.
Lower-right corner: The g2 rabbit is *not* frozen by the gold dog because it is guarded by the silver cat.

guarded or *defended* when there is a friendly adjacent piece, or *defender*.

The rule of capture is enforced continuously. In particular, it is enforced between steps within the same turn, rather than just at the beginning or end of a turn. See Figure 1.3 for examples.

Freezing

If a piece is adjacent to a stronger opposing piece and is not guarded, then it is *frozen* and cannot move. That is, it cannot step, push, or pull, until the stronger opposing piece moves away or is pushed or pulled away, or until a friendly piece moves adjacent. See Figure 1.3 for examples.

Winning

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. Alternatively, 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.

Repetition

Players must change the board position on their turn. This means they must make at least one step, and may not make a sequence of steps that leaves the board unchanged. Additionally, players may not end their turn making a move that causes a *third-time repetition*, where the same resulting board position with the same player to play next has occurred twice before in the game.

1.1.3 An Example Position

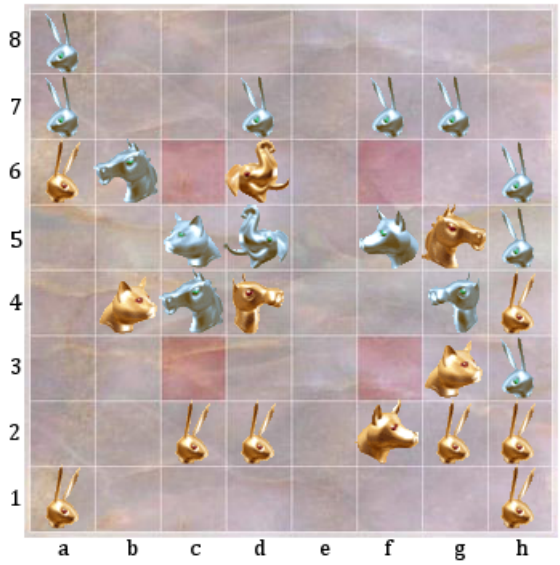

Figure 1.4: Example game position, Gold to play.

In Figure 1.4, we show another example to further illustrate the game rules.

Currently, it is Gold's turn, and Silver is threatening to capture the horse at g5. If Gold does not do anything about it, then on the next turn, Silver can move his camel from g4 to g5, pushing the gold horse to g6, and then move his camel from g5 to g6, pushing the gold

horse into into the f6 trap. Since there are no other gold pieces guarding the f6 trap, the gold horse would be captured. In total, this would also use all four steps of Silver's turn.

Gold has a capture threat of his own: the gold camel on d4 can capture the silver horse on c4 in by pushing it into the c3 trap. However, the camel is not immediately able to do this, because it is frozen by the silver elephant on d5. Gold needs to spend one step first to advance a rabbit from d2 to d3, guarding the camel and thereby unfreezing it and allowing it then to push the horse. This accomplishes a capture in three steps.

A good move for Gold might be to capture the silver horse in this way, and then use the fourth step of his turn to move the gold elephant from d6 to e6. This would prevent Silver from capturing the g5 horse next turn, because the f6 trap would now have a gold defender, namely the elephant now on e6. In standard Arimaa move notation, Gold's move would be Rd2n hc4s hc3x Md4w Ed6e.[3]

1.2 Why Is Arimaa Challenging?

Playing games has long been a topic of interest to computer AI research, and in the past few decades, substantial progress has been made. Many games have been solved with explicit optimal strategies found, including Checkers [26] and Go-moku [1]. Other games, such as Chess, Backgammon, and Reversi, have not been solved, but top programs have surpassed world-champion human players [17, 32, 5]. In still more games, such as Shogi and heads-up limit Poker, computers are reaching or rapidly approaching champion-level play [21, 16]. Only a small number of classic games continue to resist expert-level computer play, including Go, many other variants of Poker, and Bridge [22, 23, 3].

As a two-player adversarial game, Arimaa shares many properties with these previous games. In particular, Arimaa is:

- *Finite* - Players have only a finite number of possible moves each turn. And since there are a finite number of possible configurations of the 8x8 board, the rule prohibiting three-fold repetition guarantees that eventually, any game must terminate.[4]
- *Sequential* - Players make moves alternately, rather than simultaneously.
- *Perfect Information* - Both players observe the complete game state, that is, the move history, current board position, and the current player to move.
- *Deterministic* - Any move in a given state uniquely determines the subsequent state.

[3] See [29] for a description of the notation used to record positions and moves.

- *Zero-Sum* - Any winning outcome for one player is equally a losing outcome for the other. Both players' interests are exactly opposed.

Together, these properties make Arimaa trivial from a game-theoretic perspective. A standard minimax argument proves that one player must have a deterministic winning strategy, since there are no draws. However, this by no means indicates that Arimaa is easy to play well. It may easily be infeasible to find the optimal strategy because the number of states in the game might be extremely large. Therefore, we consider the state-space and game-tree complexity of Arimaa next.

1.2.1 State-Space Complexity

The state-space complexity of a game is defined as the number of game states that can be reached through legal play.[5]

The state-space complexity of Arimaa is approximately 10^{43} [8]. Since Arimaa can be played using a standard Chess set, it is not surprising that this is comparable to that of Chess, which has been estimated to be around 10^{47} [34]. It is slightly less than in Chess, because there is no possibility for piece promotion in Arimaa.

A state-space complexity of 10^{43} is actually surprisingly small in that most popular games with state-space complexity up to this size have seen at least world-class computer play, including Chess itself. Those that continue to be challenging tend to have much larger state spaces, most notably Go, with a state space complexity of 10^{172} [1]. Therefore, we turn to the game-tree complexity next for an explanation of Arimaa's difficulty.

1.2.2 Game-Tree Complexity

The game-tree complexity of a game is defined to be the number of nodes in shallowest full-width game tree that proves the value of the initial position. For most popular games, this is infeasible to compute, so it is very roughly approximated by taking the average branching factor to the power of the length of the average game in practice.

[4]This is not actually a useful practical bound, since the number of configurations is huge. One can even construct positions that force games of ridiculous length, ending only due to prohibition of repetition. But in practice, Arimaa games do not seem to exhibit this problem! Despite the lack of a "fifty move rule" as in Chess, serious problems of repetition have occurred in almost none of the thousands of games played by people since Arimaa was invented.

[5]Depending on one's definitions, the full game state might include not just the board position and the side to move, but also the move history, because differences in the move history could affect what moves are legal later due to prohibition of third-time repetitions. Usually, when computing the state-space complexity of a board game, we ignore the move history.

In Arimaa, the per-turn branching factor is extremely large, due to the combinatorial possibilities produced by having four steps per turn. Given a board position in Arimaa, we define the number of legal moves for a player to be all legal combinations of up to four steps by that player (possibly including pushes and pulls), except that we identify moves as the same if they produce the same resulting board position.

To estimate the average branching factor and average length of a game, we computed the average of both over the collection of all rated games played on the server at Arimaa.com [28] between January 2005 and December 2010 between players with a rating of at least 1800, ending in goal, elimination, or immobilization (and not by time or resignation). This was a total of 14115 games.

On average there were about 16064 distinct legal moves per turn, and the average game length was about 92 turns, or 41 complete turn alternations. This is in reasonably close agreement with earlier studies on the branching factor of Arimaa [13]. We note that this disagrees with the much larger branching factor of 276386 computed by Christ-Jan Cox in [8]. This is because Cox considers different sequences of steps that lead to the same board position to be distinct moves. In general, many of the possible permutations of the steps in a given move will be legal and will produce the same outcome, giving a substantially higher number if counted separately.

Since each player may arrange their pieces in the two rows on their side of the board in any desired fashion during the setup phase, we multiply by the number of possible setups for each player when computing the game-tree complexity. There are a total of 64864800 possible setups for each player, although we may reduce the number by a factor of 2 for the first player due to symmetry. This gives a game tree complexity of approximately:

$$(64864800^2/2) * 16064^{92} \approx 1.8 * 10^{402}$$

The enormous branching factor of about 16000 and the game-tree complexity of about 10^{402} give a reason for Arimaa's computational difficulty, for they are substantially larger than in most other popular games. For instance, the average branching factor and game-tree complexity in chess have been estimated to be around 35 and 10^{123}, respectively [1]. In fact, the values for Arimaa are most comparable to those for Go, which has a branching factor of around 250 and a game-tree complexity of around 10^{360} [1]. While the branching factor of Go is smaller than Arimaa, Go games tend to last much longer, giving a game-tree complexity nearly as large. These properties are summarized in Table 1.1.

Game	State Space	Branching Factor	Avg. Length	Game-Tree Complexity
Chess	10^{47}	35	80	10^{123}
Go	10^{172}	250	150	10^{360}
Arimaa	10^{43}	16064	92	10^{402}

Table 1.1: Game complexities for Chess, Go, and Arimaa

1.2.3 The Branching Factor in Practice

The large branching factor in Arimaa is a serious impediment to game-tree search. Indeed, under standard tournament time controls of 1-2 minutes per turn, current programs are incapable of searching typical positions to deeper than 3 turns at full width. That is, any branch of the search ends after player 1 makes a move, then player 2 responds, then player 1 responds. Without special extensions for captures and other critical moves, this is barely enough to see some of the common tactics in the game.

Although human players are also strongly affected by the branching factor, they are able to compensate by recognizing good patterns of piece movements. Moreover, human players are much more capable of decomposing the problem by identifying the most important subgoals. The top human players are so effective at identifying the best moves that they are able to consistently play well even when considering only a bare handful of moves each turn (even as few as 2-3), out of the tens of thousands possible. As a result, the large branching factor in Arimaa favors humans over computers.

1.2.4 Strategic Properties

In addition to a huge branching factor, empirically, Arimaa also exhibits some specific strategic properties that appear to benefit human players over computer players, compared to other Chess-like games.

Most notably, capturing pieces and winning material is somewhat more difficult in Arimaa than in Chess. In Arimaa, a piece on one of the four trap squares is only captured when there are no friendly adjacent pieces, or *defenders*. This is a moderately stringent condition, since the opponent can often just add more defenders or just retreat the threatened piece away from the trap. Moreover, the opponent can make a trap safe simply by placing his elephant adjacent to it. Since no piece is stronger than the elephant, it cannot be pushed or pulled away, so no piece can be captured in that trap until the elephant leaves on its own.

Therefore, it is necessary to make multiple threats at the same time, which usually requires first that a strong positional advantage to be obtained over many turns. This increases the importance of long-term strategy over short-term tactics.

Additionally, since pieces of equal strength cannot push or pull one another, they can deadlock in local fights. Such deadlocks frequently make it necessary to shift the global distribution of one's pieces to make progress, which again requires significant long-term planning and judgment.

Altogether, this benefits human players over computers, since humans are often better at making plans over long time horizons in large spaces and learning to evaluate uncertain tradeoffs. And while Arimaa positions can easily be highly complicated and combinatorial, overall the greater difficulty of capture and lessened importance of short-term combinations make it slightly easier for experienced human players to avoid tactical mistakes.

It is worth noting cases where these observations fail in Arimaa, such as in many endgames. Once numerous pieces have been exchanged and the board opens up, Arimaa play usually becomes extremely wild and sharp due to multiple threats to score goal on either side and a relative lack of pieces with which to defend traps. In fact, this is one area of the game where current Arimaa programs frequently outperform the top human players, unlike the midgame and opening.

1.2.5 Conclusion

Altogether, Arimaa does appear to be a challenging domain. The massive branching factor and the long time-period over which moves exert their strategic effects make it difficult to design a strong computer agent. Nonetheless, progress is still being made. Especially in the last several years, the top computer programs have been improving significantly, with a gain of a couple hundred rating points in total on the server at Arimaa.com [28]. While top human players continue to be rated another several hundred rating points higher and are still improving, it is our belief that steady continued innovations will be sufficient to bridge the gap in the next one or two decades. We believe that such innovations will include the types of learning algorithms described in this thesis, as well as incremental improvements in existing algorithms for evaluation and search.

1.3 Our Contribution

The two most significant challenges facing strong computer play in Arimaa are the massive branching factor and the difficulty of accurate positional evaluation. We are the first to make a major effort to apply the tools of machine learning to solve these problems, and our primary contribution is an investigation and comparison of a variety of algorithms for doing so. Our goal is to show how learning can be highly effective in Arimaa and to lay the groundwork for future innovation and research.

In Chapter 2, we describe our investigation into the problem of learning a good move ordering function for Arimaa. With tens of thousands of legal moves per position, any current method of search will be rapidly overwhelmed by the possibilities, making it important to find better ways to order, select, and prune moves. We apply an innovative method first used by Rémi Coulom [6] in computer Go for learning to predict expert moves in games. Our resulting function is capable of ordering the legal moves in a position in Arimaa well enough such that almost 90 percent of the time, the actual move played by the expert player in that position is within the top 5 percent of the ordering. We then demonstrate that our ordering function is also effective at combating the branching factor in practice by using it to prune moves in our Arimaa program. Even at levels of pruning as extreme as 95 percent, the strength of our program does not decline—indeed, it increases by over 100 Elo,[6]and using this function for pruning, our program recently won the 2011 Arimaa Computer Championship [30]. In this way, we demonstrate how a high-quality move ordering can be learned in Arimaa and then directly applied to increase the effectiveness of search.

In Chapter 3, we attack the challenge of accurate positional evaluation in Arimaa by applying and comparing four different algorithms for learning evaluation functions. These algorithms include the well-known TD(λ) algorithm and a variant of it developed by Baxter, Tridgell, and Weaver [2], as well as two algorithms recently developed in 2009 by Veness, Silver, Uther, and Blair [35] that learn directly using the results of a minimax or alpha-beta search. We show that beginning only from material weights, it is possible to automatically learn an evaluation function that comes fairly close to the performance of the hand-coded evaluation we use in our Arimaa program, Sharp. While we unable to match the performance of Sharp, our best learned agent comes close, winning 74 out of 200 games when searching to an equal depth. Moreover, we give evidence that substantial improvements are possible, potentially with only minor changes and improvements to our existing methods.

[6]Under the Elo rating model [11], 100 Elo corresponds to approximately to a winning chance of around 64%, and while not large, is a substantial gain.

Overall, our contribution is one that opens several new avenues for research in Arimaa. Our investigations push forward on the state-of-the-art in this new game and lay groundwork for the development of new algorithms and techniques in the future. We believe that techniques like the ones investigated here will play a major role in ultimate triumph of computer play in Arimaa.

More broadly, we believe Arimaa has the potential to serve as a testbed for the development of new game-playing and learning algorithms that perform well in the presence of a high branching factor. As a tactical game in which deep search is almost impossible, and yet one in which long-term evaluation is critical, Arimaa provides a unique and challenging domain. There is substantial room for improvement on our results, and such improvements may have the potential to uncover new ideas in search and evaluation.

1.4 Current Arimaa Programs

We give a survey of a few of the current techniques used by top computer Arimaa programs, as well as some of the alternatives that have been tried.

Alpha-beta Search

Despite the difficulties with the large branching factor, all of the current best Arimaa programs use *iterative-deepening depth-limited alpha-beta search* at their core, although they incorporate varying degrees of additional pruning and other search enhancements [36, 9, 8]. Overall, this makes them very similar in structure to Chess programs.

Briefly, in *alpha-beta* search, one searches the game tree in a depth-first manner and computes the minimax value of each node, while also tracking lower (alpha) and upper (beta) bounds on the values of subtrees and pruning whenever a subtree provably cannot affect the minimax value of the root node. Since it is impossible to search the entire game tree, one imposes a *depth limit*, where one terminates the recursion and applies a heuristic *evaluation function* to estimate the minimax value of the position. Additionally by *iterative-deepening*, where one iteratively searches with an increasing depth limit until some time limit is exceeded, one can also search for a specified amount of time rather than to a fixed depth. To gain better alpha-beta pruning, a heuristic *move-ordering function* is applied to sort the moves at each node in order of likely quality [24].

Evaluation functions for Arimaa can be very complex. Generally, they are based on material advantage (which player has more/stronger pieces), but take into account a wide

variety of positional features, such as control of traps, advancement of pieces, goal threats, and various identified strategic configurations in Arimaa known as *blockades*, *hostages*, and *frames* (see Appendix B).

Step-based Search

In Arimaa, due to the fact that multiple steps occur in a turn, there is a nontrivial choice of how to perform a search. One can view Arimaa in a *turn-based* fashion, as we have so far, where there are thousands of legal moves per turn composed of different combinations of steps. But one can also view Arimaa in a *step-based* fashion, where there are only twenty to forty legal "moves" per turn, namely the individual steps, pushes, and pulls, and where the turn switches only every four "moves" of the game.

Although these two views are equivalent from a game-theoretic perspective, they have practical differences for search. For instance, it can be faster in a search to generate and handle lists of twenty to thirty moves in a step-based search, rather than lists of thousands of moves. On the other hand, heuristic move ordering can sometimes be less effective in a step-based search. This is because even when a move is good, making only part of the steps of the move or making different steps at the end of the move could be very bad, and a step-based recursion forces one to search moves together that share their initial steps.

Most strong Arimaa programs use some variant of the step-based search [36, 9]. Empirically, the speedup afforded by step-based search seems to outweigh the disadvantages in many cases. Moreover, it can enable more effective iterative deepening. For instance, if there is not enough time to search 3 turns (12 steps) deep, but there is excess time after searching 2 turns (8 steps) deep, it is often possible to do an intermediate search 9 or 10 steps deep, gaining a better evaluation than otherwise possible.

Hash Tables

In Arimaa, like in many games, many different sequences of moves can lead to the same position. In Arimaa, this even occurs with different permutations of steps within a single turn. By caching results in a hash table to avoid multiple searches of these repeated positions, one can achieve dramatic improvements in speed. Other data can also be cached such as data about best moves in each subtree, which can be used to dynamically improve the move ordering [19].

Quiescence Search

Depth-limited alpha-beta search frequently suffers from the *horizon effect*, where threats beyond the depth limit are ignored by the search, leading to very poor results. This frequently manifests in Arimaa when a program is faced with the inevitable capture of a strong piece, such as a horse. It will often respond by placing a weak piece as a defender next to the relevant trap for the opponent to capture for free, simply because that delays the loss of the horse one turn beyond the depth limit where the search can see it. In this way, the horizon effect can cause a program to meaninglessly sacrifice pieces.

Quiescence search attempts to mitigate this problem by extending the search beyond the depth limit when it appears likely that a position will be misevaluated [19]. For instance, if pieces remain capturable when the depth limit is reached, a quiescence search might extend the search to include additional capturing or defending moves for that piece, so that the evaluation function can return a more accurate result.

Tree-Based Capture and Goal Detection

In Arimaa, due to the combinatorial possibilities of having four steps per turn, it is nontrivial even to determine whether a piece can be captured or whether a rabbit can reach goal and win the game on the current turn. Recursive search is costly and undesirable. Therefore, most strong Arimaa programs use a decision tree generated by backwards induction to answer such questions. Such decision trees can be applied very quickly and can allow certain tactics to be discovered a full four steps earlier in the search [9].

1.4.1 Other Approaches

Various approaches other than alpha-beta search have been tried for Arimaa, although generally with less success.

Monte-Carlo Tree Search

Recently, Kozelek [18] and Miller [20] both independently attempted to apply Monte-Carlo tree search (MCTS) to Arimaa, a recent search paradigm that has resulted in major breakthroughs in computer Go [10]. In Monte-Carlo tree search, a game tree is expanded in memory by repeated playouts from the root node. In each playout, moves are chosen at each node according to a *tree policy* that favors moves that have previously performed the best (exploitation), but chooses other moves at least some of the time (exploration). When

a playout reaches the end of the tree, it is played pseudorandomly until a terminal position is reached and the result, such as a win or loss, is used to update the statistics of the nodes involved. High-performing leaves of the tree are expanded periodically, so that the tree grows asymmetrically with strong bias towards the best moves.

Unfortunately, random playout turns out to be a poor way to evaluate positions in Arimaa, even to the point of valuing an extra rabbit more than the elephant, because the extra rabbit better improves the chances of randomly reaching the goal [18]! Because of this, a straightforward implementation of MCTS is barely better than uniform random play [20]. Kozelek attempted to solve this by using only a short random playout followed by a deterministic, traditional evaluation function, but even so, the resulting agent was relatively weak [18].

Our own observation is that Arimaa in general seems poorly suited to random playout. There are a large number of moves in Arimaa that damage one's own position greatly, such as sacrificing one's own pieces in traps or opening paths for opposing rabbits to reach goal. These moves add a lot of noise and there is enough variety that it is extremely challenging to classify and prune them. Moreover, it is frequently the case that maintaining one's position requires a piece *not* to move. A simple example is that an elephant may not want to move because it is guarding a trap. This means that Arimaa positions are somewhat "unstable" in that random movement will rapidly destroy the strategic features of the position. Because of this, we believe that MCTS-based methods are unlikely to succeed.

Planning and Selective Search

In another alternative approach, Trippen [33] investigated a pattern-matching and plan-based method for performing extremely selective search. Rather than doing a traditional full-width search, the resulting playing algorithm examines only a small handful of moves (as few as 5!) in each position, which are generated according to positional analysis that suggests possible long-term plans. As proof of concept, the Trippen demonstrated a simple plan based around a certain type of long-term threat in the opening that is capable of defeating several weaker Arimaa programs.

However, a drawback of the system is that it requires plans to be manually specified and implemented. Moreover, the limited way in which the planning system was tested makes it difficult to evaluate its potential. And given that many midgame and endgame positions in Arimaa are intensely combinatorial and do seem to require broad tactical search, it is probable that this type of system could only be used as a component of a more general playing algorithm.

1.4.2 Conclusion

Unfortunately, barring a major breakthrough, we believe that the alpha-beta will continue to be the most effective general method of search in Arimaa. While the high branching factor is problematic for alpha-beta search, it is a similar problem for most other search algorithms. Moreover, although obtaining an evaluation function capable of accurate long-term evaluation is difficult, the current poor performance of methods involving playout greatly limits the alternatives.

Chapter 2

Move Prediction and Ordering in Arimaa

In this chapter, we attack the problem of the immense branching factor in Arimaa by investigating ways of learning a good move ordering. To do this, we draw from the very closely related problem of expert move prediction.

In Section 2.1, we motivate and define the problem. In Section 2.2, we present a simple feature-based learning approach to solve this problem. Following that, in Section 2.3, we present two algorithms for learning to predict expert moves based on these features, the first using a naive Bayesian model, and the second using a generalized Bradley-Terry model [6]. In Section 2.4, we describe the actual features used. Finally, in Section 2.5, we present our training method and experimental results.

2.1 The Problem of Move Ordering

It is well-known that in alpha-beta search, the order that moves are searched can drastically affect the efficiency of the search. In particular, if at each node the optimal move is searched first, then the minimax value of a game tree can be computed in $O(b^{d/2})$ time, where b is the branching factor and d is the depth of the tree, while if moves are searched in order from worst-to-best, alpha-beta search degenerates down to full-width minimax search, running in time $O(b^d)$ [24].

With a good move ordering heuristic, it may even be possible to prune beyond alpha-beta by discarding moves that come late in the ordering. If the ordering heuristic is good enough, these moves will very likely be worse than the earlier moves in the ordering and therefore will not affect the minimax value. In this case, the increased search depth resulting from

the time saved by discarding the moves can more than compensate for the occasional errors introduced.

Move ordering is useful in other search paradigms as well, such as the more recent family of Monte-Carlo tree search (MCTS) algorithms. Ordering heuristics have been used successfully in computer Go for progressive widening of the MCTS tree, as well as for biasing moves in the random component of the playout to enable more accurate estimation of the value of positions [22].

How might we learn a good move ordering? Informally, given a board position and a set of legal moves, the task is to find an ordering of the moves that maximizes the likelihood that the best move is near the front of the ordering.

More formally, consider a finite two player deterministic zero-sum game.

- Let S be the state space of the game.
- Let M be the set of all possible moves that could be made at any point in the game.
- Let $M(s) \subset M$ be the set of legal moves in state $s \in S$.
- Let $m^*(s) \in M(s)$ be the optimal move in state $s \in S$.[1]
- Define an *ordering function* to be a function $O : S \times M \to \mathbb{R}$.
- Given an ordering function O, for any state $s \in S$, define the induced ordering $>_{O,s}$ on the legal moves $M(s)$ by $m >_{O,s} m'$ iff $O(s, m) > O(s, m')$.

Typically we are not interested in the particular values of O so much as we are interested in the relative ordering of values, as given by the induced ordering.

The task is to learn the best possible ordering function according to some evaluation metric. In general, we want to rank the optimal move as highly as possible, although the specific metric is open to choice. For instance, we might be interested in the frequency that the ordering ranks the optimal move first:

$$\frac{1}{|S|} \sum_{s \in S} \chi \left(\forall m \in M(s) \text{ s.t } m \neq m^*(s),\ \ m^*(s) >_{O,s} m \right)$$

Or we might be interested in the average rank of the optimal move:

$$\frac{1}{|S|} \sum_{s \in S} |\{m \in M(s) : m <_{O,s} m^*(s)\}|$$

[1] For notational simplicity, we assume the optimal move is unique.

We may also be interested in how the ordering ranks "good" moves that aren't necessarily optimal, so that in cases where the optimal move isn't ranked first, the top few moves might still be strong. And of course, in practice we may also not care too much how it performs on arbitrary game states, so long as it performs well on all the states that could "reasonably" occur in real games.

Unfortunately, it is infeasible to find the optimal move $m^*(s)$ in general. However, we can substitute the moves played by expert human players in recorded games, since game records are readily available. In Arimaa, expert human play will probably be stronger than anything we can achieve with a static learned heuristic, and therefore from our point of view, a fine replacement for optimal.

This transforms the problem into one of *expert move prediction*. Rather than learning an ordering function that tries to rank the optimal move highly, we instead learn an ordering function that tries to rank the expert move highly. Equivalently, we are trying to learn a function that given a game state, will attempt to predict the move that the expert will play and order the legal moves by how likely they are to be the expert move.

By attacking this problem, we hope to obtain a good move ordering function that approximates the optimal ordering, which will then allow us to prune much more effectively within a game-tree search.

2.2 Using Features to Generalize

Since the number of state-move pairs is very large, the space of possible ordering functions is also enormous, and it would be infeasible to learn such a function without some sort of inductive bias or ability to generalize. After the first few moves in a game of Arimaa, it is likely that every board position thereafter will be unique, having never occurred in any other game yet played, so the ability to generalize between positions is essential.

However, a major challenge for generalization is that the space of game states is highly discontinuous with respect to what moves are good and bad. For example, in Figure 2.1 we see that in one position, a particular move by Gold captures Silver's camel, making it a very good move, whereas with a slight change in the position, the move no longer captures, and is in fact relatively pointless. The tiny change in the position has caused a drastic shift in how good a move is.

Therefore, rather than train on the state-move pairs themselves, we instead map them into a *feature space* composed of various identified *features* that correlate more strongly with how good a move is likely to be than the raw position or move, such as whether a

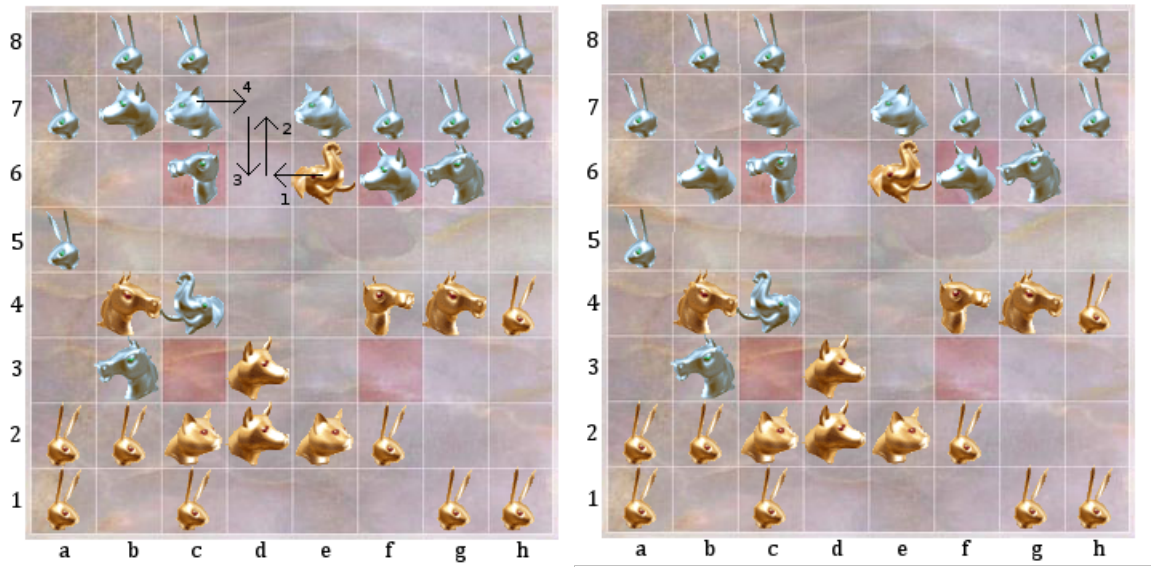

Figure 2.1: On the left, Gold can capture Silver's camel on c6 in four steps by moving his elephant from e6 to d7 and then pulling the c7 cat to d7, removing the only silver defender of the trap. On the right, Silver's b7 dog has been moved a single space. Gold's move no longer captures the camel, since the trap has one more defender.

move captures a piece, or how many resulting defenders will be next to a given trap square. This allows us to distinguish cases where superficially similar moves behave very differently, since the two moves can differ greatly in the resulting feature space. Moreover, this lets us effectively generalize between different positions by allowing us to learn trends that are exhibited across many positions, such as the fact that capturing opposing pieces is generally good.

In our particular case, given a state $s \in S$, for each possible legal move $m \in M(s)$, we compute a *binary* feature vector $v(s,m) \in \{0,1\}^n$, where the value of any binary entry indicates the presence or absence of a particular feature.

Then, given a training instance (s,w) where s is the game state and w indicates that out of all the legal moves $m_1, ..., m_j$, m_w was the one played by the expert, the input to the learning algorithm is $((v_1, ..., v_j), w)$ where $v_k = v(s, m_k)$. We take our ordering function O to be of the form $O(s,m) = h(v(s,m))$ where $h : \{0,1\}^n \to \mathbb{R}$ is an ordering function on feature vectors, rather than state-move pairs.

Our task is now as follows: Given training instances $(V_1, w_1), ..., (V_n, w_n)$ where each V_j is a list of feature vectors $(v_{j1}, ..., v_{jm})$, find a function h with induced ordering $>_h$, such that given any new list of feature vectors for moves in an Arimaa position, h orders or ranks

the vector corresponding to the expert move as highly as possible.

2.3 Algorithms

We implemented and tested two learning algorithms for solving this task. We describe them in the following sections.

2.3.1 Naive Bayes

As a baseline, we implemented a Naive Bayesian classifier, using a standard conditional independence assumption to try to distinguish between expert and non-expert moves. Although Naive Bayes is often an overly simplistic model, it is easy to implement and fast to train, and is known in a variety of settings to give results that are surprisingly not too bad.

Consider the task of trying to learn a binary classification $C : \{0,1\}^n \to \{0,1\}$. Given a feature vector $v \in \{0,1\}^n$ with feature values $f_1(v) = a_1, ..., f_n(v) = a_n$, the probability that $C(v) = 1$ can be expanded using Bayes's theorem:

$$P[C(v) = 1 \mid f_1(v) = a_1, ..., f_n(v) = a_n]$$

$$= \frac{P[C(v) = 1]\ P[f_1(v) = a_1, ..., f_n(v) = a_n \mid \text{C(v)=1}]}{P[f_1(v) = a_1, ..., f_n(v) = a_n]}$$

$$= \frac{P[C(v) = 1]\ P[f_1(v) = a_1, ..., f_n(v) = a_n \mid \text{C(v)=1}]}{\sum_{c=0}^{1} P[C(v) = c]\ P[f_1(v) = a_1, ..., f_n(v) = a_n \mid C(v) = c]}$$

If we assume that all of the v_j are conditionally independent given $C(v)$, then we have:

$$= \frac{P[C(v) = 1] \prod_{j=1}^{n} P[f_j(v) = a_j \mid C(v) = 1]}{\sum_{c=0}^{1} P[C(v) = c] \prod_{j=1}^{n} P[f_j(v) = a_j \mid C(v) = c]}$$

From here, all of the component probabilities can be estimated directly by counting their frequency in the training data.

To apply this to our problem of expert move prediction, we classify $C(v) = 1$ if v is the feature vector for an expert move and $C(v) = 0$ if not. For each training instance $((v_1, ..., v_j), w)$, we provide the expert move vector v_w as an input with $C(v_w) = 1$, and for every other $i \neq w$, we provide v_i as an input with $C(v_i) = 0$.

Note that since we are interested in a move ordering rather than the classification of any particular move alone, we do not use the Naive Bayes classifier the typical way. Typically,

one receives a single vector v at a time, and returns the most probable class:

$$\text{argmax}_c \; P[C(v) = c \mid f_1(v) = a_1, ..., f_n(v) = a_n]$$

In our case, we instead order all the possible legal moves in a game state by the independent probability that they are in the class of expert moves. Our ordering function h is:

$$h(v) \; = \; P[C(v) = 1 \mid f_1(v) = a_1, ..., f_n(v) = a_n]$$

2.3.2 Bradley-Terry Model

Our other learning algorithm for Arimaa is based on a generalized Bradley-Terry model, a simple model for associating the skill level of agents in a competition with the probability that any particular agent will win the competition [15].

Such a model was first used in computer Go by Coulom [6] to predict expert move choices using the innovative idea of treating them as outcomes of competitions between moves. Each move, viewed as a team whose members are its features, competes to be the expert move. The moves with the best features are the most likely to "win" the competition and be chosen by the expert. By optimizing the model over a set of expert game records, we can learn how good or bad each feature is in order to predict moves in future games.

Model

In a Bradley-Terry model [15], each agent $i \in [1, ..., N]$ in a series of competitions is assigned a certain strength $\gamma_i \in (0, \infty)$. In any single competition, there is exactly one winner, and we model the probability that an agent wins so that each agent wins with probability proportional to its strength. For instance, in a pairwise competition between two agents i and j, the probability that i wins is:

$$P[i \text{ wins}] = \frac{\gamma_i}{\gamma_i + \gamma_j}$$

In a generalized Bradley-Terry model, we consider competitions between teams of agents as well as between multiple teams at once. Given a team $T \subset [1, ..., N]$, we define the strength of the team by:

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

Then, in a competition between teams $T_1, ..., T_n$, the probability that T_j wins is:

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

We apply this model to the problem of expert move prediction by considering each individual feature v_i to be an agent with a certain unknown strength γ_i. Each binary feature vector v is a team of its component features, with strength equal to the product of its features' strengths:

$$\gamma(v) = \prod_{i:f_i(v)=1} \gamma_i$$

Each training instance $((v_1, ..., v_j), w)$ is modeled as the result of a team competition between the binary feature vectors, where the feature vector v_w corresponding to the expert move is the winner of the competition. Our training consists of a standard maximum likelihood estimation, where we use Bayesian inference to find the most likely model parameters $\gamma_1, ..., \gamma_n$ given the observed data.

To predict the expert move in a new game state, we simply order each legal move $m_i \in M(s)$ by the probability that its associated feature vector v_i wins the competition:

$$h(v_i) = P[v_i \text{ wins}] = \frac{\gamma(v_i)}{\sum_{i=1}^{|M(s)|} \gamma(v_i)}$$

Since the normalizing factor in the denominator is the same for any feature vector for this state, we can omit it. Therefore, our final ordering function is:

$$h(v_i) = \gamma(v_i)$$

Performing the Maximum Likelihood Estimation

We use the same optimization procedure described by Coulom for computer Go [6].

To perform the maximum likelihood estimation, we wish to optimize the model parameters to maximize the probability of the observed results given those parameters. Formally, let $\gamma \in (0, \infty)^n$ be a vector of all the model parameters γ_i and let R be a collection of all the observed results $R_1, ..., R_n$. Then, we wish to maximize $P[\gamma \mid R]$, which by Bayes Theorem is:

$$P[\gamma \mid R] = \frac{P[R \mid \gamma]P[\gamma]}{P[R]}$$

The normalizing constant $P[R]$ can be ignored for the purposes of maximization since it does not depend on γ. Additionally we can choose a prior $P[\gamma]$ that is of the same form as that produced by the model itself. That is, we may choose a prior of the form $P[\gamma] = P[R' \mid \gamma]$ where R' are a set of virtual competition results. A natural choice is to give every feature n wins and n losses against a virtual opponent. A prior of this form has the advantage that:

$$P[R \mid \gamma]P[\gamma] = P[R, R' \mid \gamma]$$

and therefore we may simply focus on maximizing $P[R, R' \mid \gamma]$.

We assume that all the results $R_k \in R$ are independent of one another. This is not actually true, since we are extracting a result from every state in each game, and features of many legal moves will be very similar between successive states in a game. Moreover, human players typically do not consider each situation in a game independently, but will rather cache and update their evaluation of the board position based on their evaluations in previous turns. However, independence is a fair assumption to impose, given that the game state itself should still by far be the most important factor in the expert's move choice. The effects of short runs of correlations between positions from the same game should be washed out over a data set of hundreds or thousands of games.

To perform the maximization of $P[R, R' \mid \gamma]$, we initialize all γ_i to the initial values 1.0, and then perform successive iterations of updates. Each iteration, we sequentially update each feature by the following update rule (see Appendix A for a detailed derivation):

$$\gamma_i \leftarrow \frac{W_j}{\sum_{j=1}^{N} \frac{C_{ij}}{E_j}}$$

C_{ij} is the total strength of the other features on all teams that feature i is a member of, E_j is the total strength of all participants (given the current value of γ_i), and W_j is the number of times feature i is on the winning team. We continue iterating until convergence. In practice, on the type of data generated for our expert move training, convergence is rapid and consistent, requiring only a few tens of iterations.

Note that if we fix the values E_j, the denominator in the update rule has a natural interpretation, namely the expected number of wins that would be scored by the teammates of feature i without the help of feature i, over all the competitions. Thus, our update rule simply consists of setting γ_i to be the ratio of the number of wins with feature i to the expected number of wins without feature i.

2.4 Features Implemented

In the following section, we describe the actual features used by our learning algorithms. The features were chosen according to expert knowledge about what types of properties of a move might make it more or less good, as well as by examining the results of preliminary tests for systematic mispredictions that suggested additional useful features to add.

All features describe the results of the full move of a given turn, as opposed to one of the four possible steps of a turn in Arimaa. All features are binary.

Symmetry

Since the rules are preserved by left-right symmetry and also by the symmetry of simultaneously swapping the two players and vertically reflecting the board, all features are normalized accordingly by the player to move, and where relevant, mirroring locations on the board are considered equivalent (giving only 32 locations, rather than 64).

Piece Type

In many of the following features, it is important to distinguish how strong a piece is. However, the strength of a piece is only relative to other pieces. For instance, if both players have 1 elephant, 2 horses, and 2 rabbits, this is exactly the same as the case where both players have 1 elephant, 2 dogs, and 2 rabbits.

Therefore, we do not consider the absolute type of a piece such as "horse" or "dog", but rather we count the number of opposing pieces stronger than it, and whether or not it is a rabbit (because rabbits cannot move backwards). The piece types we used are as follows:

Type 0: Non-rabbit, 0 opponent pieces stronger
Type 1: Non-rabbit, 1 opponent piece stronger
Type 2: Non-rabbit, 2 opponent pieces stronger
Type 3: Non-rabbit, 3-4 opponent pieces stronger
Type 4: Non-rabbit, 5-6 opponent pieces stronger
Type 5: Rabbit, 7-8 opponent pieces stronger
Type 6: Rabbit, 5-6 opponent pieces stronger
Type 7: Rabbit, 0-4 opponent pieces stronger

Position and Movement

Different pieces in Arimaa tend to work most effectively in certain key areas of the board, such as squares adjacent to trap squares. However, the best location for different pieces varies significantly with the type of piece. Moreover, the importance of pushing and pulling opponent pieces depends strongly on its type and location. Therefore, we add the following features:

SRC(*p,type,loc*): Piece owned by player *p* (0-1) of type *type* (0-7) moved from location *loc* (0-31). (512 features)

DEST(*p,type,loc*): Piece owned by player *p* (0-1) of type *type* (0-7) moved to location *loc* (0-32), where location 32 indicates captured. (528 features)

Trap Status

The number of defending pieces (0-4) next to each trap in Arimaa is critical in considering what possible tactics are available. We also distinguish whether or not the elephant is next to the trap or not, except in the case of four pieces next to a trap, since the presence of an elephant as a defender is very important for considering how strongly that trap is held. This gives 8 possible statuses for the defense of a trap.

> TRAP_STATUS(*p,trap,status0,status1*): Trap *trap* (0-3) from perspective of player *p* (0-3) changed from *status0* (0-7) to *status1* (0-7). (512 features)

Capture Threats and Defense

Threatening and defending the capture of a piece plays a key role in many tactics.

> THREATENS_CAP(*type,s,trap*): Threatens capture of opposing piece of type *type* (0-7) in *s* (1-4) steps , in trap *trap* (0-3). (128 features)
>
> INVITES_CAP_MOVED(*type,s,trap*): Moves own piece of type *type* (0-7) so that it can be captured by opponent in *s* (1-4) steps , in trap *trap* (0-3). (128 features)
>
> INVITES_CAP_UNMOVED(*type,s,trap*): Own piece of type *type* (0-7) can now be captured by opponent in *s* (1-4) steps, in trap *trap* (0-3), but it was not itself moved. (128 features)
>
> PREVENTS_CAP(*type,loc*): Removes the threat of capture from own piece of type *type* (0-7) at location *loc* (0-31). (256 features)

The way in which one defends the capture of a piece is also important.

> CAP_DEF_ELE(*trap,s*): Defends own piece otherwise capturable in *s* (1-4) steps in trap *trap* (0-3) by using the elephant as a trap defender. (16 features)
>
> CAP_DEF_OTHER(*trap,s*): Defends own piece otherwise capturable in *s* (1-4) steps in trap *trap* (0-3) by using a non-elephant piece as a trap defender. (16 features)
>
> CAP_DEF_RUNAWAY(*trap,s*): Defends own piece otherwise capturable in *s* (1-4) steps in trap *trap* (0-3) by making the threatened piece run away. (16 features)
>
> CAP_DEF_INTERFERE(*trap,s*): Defends own piece otherwise capturable in *s* (1-4) steps in trap *trap* (0-3) by freezing or blocking the threatening piece. (16 features)

Goal Threats

Making and defending against threats to goal (which wins the game) is also very important.

THREATENS_GOAL(s): Threatens goal in s (0-4) steps. (5 features)

ALLOWS_GOAL: Allows opponent to goal next turn. (1 feature)

Piece Advancement and Threats

Choosing when to advance pieces is important, given the balance of strength in that area. We use a measure of "influence" in a region to determine this.

Influence is determined by a simple heuristic calculation. The square under each piece is assigned a value in $\{75, 55, 50, 45, 40, 35, 30, 25, 15\}$ according to the number of opposing pieces stronger than that piece, using negative values for the opponent and zero for empty squares. The values are then diffused four times, where for each adjacent square, a diffusion transfers 0.16 of the value of that square. The final value is scaled down into a range (0-8).

RABBIT_ADVANCE(*y,inf*): Moves own rabbit to row y (0-7) with the area around the goal directly in front of the rabbit having influence *inf* (0-8). (72 features)

PIECE_ADVANCE(*type,inf*): Advances piece of type *type* (0-7) with the opponent's nearest trap on the opponent's side having influence *inf* (0-8). (72 features)

PIECE_RETREAT(*type,inf*): Retreats piece of type *type* (0-7) with the opponent's nearest trap on the opponent's side having influence *inf* (0-8). (72 features)

For the last two of the previous three classes of features, we average the influence of the opponent's traps when the piece ends on either of the two central columns of the board. Additionally, we do not consider movements just in the far or near side of the board to be advancements and retreats—pieces must cross through one of the middle four rows.

We also consider movements of pieces relative to the nearest piece stronger than them. That is, the nearest *dominating* piece.

ESCAPE_DOMINATOR(*type,inf,dist*): Piece of type *type* (0-7) moved away from nearest dominating piece, with source influence *inf* (0-8), ending at manhattan distance *dist* (1-4) from the dominating piece. (288 features)

APPROACH_DOMINATOR(*type,inf,dist*): Piece of type *type* (0-7) moved closer to nearest dominating piece, with source influence *inf* (0-8), ending at manhattan distance *dist* (1-4) from the dominating piece. (288 features)

DOMINATES_ADJ(*type,inf*): Moves own dominating piece adjacent to opponent's piece of type y (0-7) with the opponent's nearest trap on the opponent's side having influence *inf* (0-8). (72 features)

Stepping On Traps

Voluntarily stepping on a trap square can be dangerous when the trap has too few defenders. However, it can also be good to place a piece on a trap square because it allows that piece in the future to access any of the four surrounding squares for defending that trap, and a piece on a trap is hard to push away.

We heuristically define a "safe" trap for a player as one that is guarded either by two defenders of that player, or the elephant. Otherwise, the trap is "unsafe".

> UNSAFE_STEP_ON_TRAP(*trap,type*): Piece of type *type* (0-7) stepped on unsafe trap *trap* (0-3). (32 features)
>
> SAFE_STEP_ON_TRAP(*trap,type*): Piece of type *type* (0-7) stepped on safe trap *trap* (0-3). (32 features)

Freezing

Freezing the opponent's pieces is beneficial for restricting their possible moves, while having one's own pieces frozen is detrimental.

> FREEZE_TYPE(p,*type*,f): Piece owned by player p (0-1) of type *type* (0-7) had its frozen/unfrozen status changed to f (0-1). (32 features)
>
> FREEZE_LOC(p,*loc*,f): Piece owned by player p (0-1) at location *loc* (0-31) had its frozen/unfrozen status changed to f (0-1). (128 features)

Blocking

A number of important tactics in Arimaa involve using pieces to block the opponent's piece from entering or leaving a certain square, such as a trap square, a square adjacent to a trap square, or the goal line. Either a piece of greater or equal strength can be used, or a *phalanx* formation composed of weaker pieces (see Appendix B for an illustration).

> BLOCK_TYPE(*type*,b,*isphalanx*): The blocked/unblocked status of an opposing piece of type *type* (0-7) in some adjacent square changed to b (0-1), and *isphalanx* (0-1) indicates whether a single piece or a phalanx was used. (32 features)
>
> BLOCK_LOC(*loc*,*dir*,b,*isphalanx*): The blocked/unblocked status of a piece at location *loc* (0-31) in the direciton *dir* (0-2) changed to b (0-1), and *isphalanx* (0-1) indicates whether a single piece or a phalanx was used. (96 features)

Previous Moves

The location of the previous move and the move before the previous move are moderately correlated with the position of the next move, so we can use these to aid our prediction.

For both the previous move and the move before the previous move, we compute a score indicating how close in location the current move is. Letting $s_1, ..., s_4$ and $t_1, ..., t_4$ be the four steps of two given moves, the closeness score of those moves is computed as:

$$\sum_i \sum_j \max(0, 4 - M(s_1, t_1))$$

where $M(s, t)$ is the manhattan distance between the square that step s moved from and the square that step t moved from. This gives a value bounded in the range $[0, 63]$, where higher values indicate that more of the steps of the two different moves took place closer together. We add the following features:

LAST_CLOSENESS(c): The closeness score of the current move to the previous move is c (0-63). (64 features)

LAST_LAST_CLOSENESS(c): The closeness score of the current move to the previous previous move is c (0-63). (64 features)

Dependency Structure

Lastly, we include a very interesting feature first proposed and used by Zhong [36] in one of the earliest Arimaa programs. This is the dependency structure of the different steps within a move. Although one can make four independent steps in a turn, often some of the steps will depend on others, and may not be arbitrarily permuted. For example, this occurs when moving the same piece twice, or pushing or pulling, or using one piece to unfreeze another. It turns out that by far the vast majority of moves played by expert players are not composed of four independent steps, even though a plurality of the legal moves are, and in general, the number and type of the dependencies is correlated with expert move choice.

For our features, we approximately count the number of independent components in the steps in a move by considering one step to be dependent on another if their sources or their destinations are adjacent, or if they both involve the adjacent squares of the same trap (because the trap square mechanic can cause dependencies between steps that aren't immediately adjacent).

Additionally, the number of steps in a move itself is a useful feature. Usually, it is bad to use fewer than four steps, so we add features that distinguish this as well.

MOVE_STRUCTURE(s,c): A total of s (1-4) steps were made, with c (1-4) independent components. (16 features)

2.5 Experimental Results

2.5.1 Data

We gathered all games played from January 2005 to March 2010 on the online server at Arimaa.com by human players and certain selected bots, with the restriction that both players had a rating on the server of at least 1800. Note that we included sufficiently strong bots in our testing data rather than only human players, because many moves made by strong bots are still reasonably strong moves, and still well-beyond what we could hope to achieve with a static heuristic.

Since the number of games involving bots was somewhat higher than the number of human-versus-human games, (due to the relatively small human population online at any given time and the continuous availability of a variety of bots), we added a mild bias towards human-played games in that we randomly excluded a game with probability 0.3 for each bot player in the game.

This left a set of 4560 games. Approximately 10 percent of these games were randomly selected to be used as a testing set and the remaining 90 percent were used as the training set. This gave a training set of 4121 games and a testing set of 439 games, on which we trained and tested both the Naive Bayes and Bradley-Terry predictors.

We attempted to do some filtering of the games for "bad moves" to try to reduce some of the noise in the data. Arimaa players will sometimes make self-damaging moves, such as self-capturing their pieces near the end of the game when they have no way to avoid losing in the next few turns, as a way of indicating concession. Additionally, players occasionally self-capture their own pieces at the beginning of the game to begin with fewer pieces, as a way of playing a handicap game against a weaker opponent. We crudely attempted to filter these moves out by excluding any sequence of consecutive self-capturing moves contiguous with the beginning of the game, as well as any self-capturing move made in the last two turns of the game by the player who lost.

2.5.2 Implementation and Training

Naive Bayes

The Naive Bayesian classifier was trained with a straightforward single pass over the training games, counting the number of observations of each feature among expert moves and the number of observations among non-expert moves to estimate the conditional class probabilities.

Additionally, to handle features that never occurred in the training games, letting the background frequency of expert moves be p, we added fractional virtual observations of weight p of the feature occurring and not occurring with an expert move, and fractional virtual observations of weight $(1 - p)$ of the feature occurring and not occurring with a non-expert move.

Training took a little more than 14 hours, the vast majority of which was spent generating the features for each move. The memory usage was negligible.

Bradley-Terry

The Bradley-Terry model was trained by repeatedly iterating over the feature data to perform the necessary updates. A total of 20 iterations of updates were performed. This was plenty to achieve convergence, as most features appeared to reach within a small range of their final values after only 5 to 10 iterations.

We chose our prior to be an observation of 1 win and 1 loss for each feature against a virtual opponent of strength $\gamma = 1$.

One major drawback of the training process is that we were forced to hold all the feature data in memory. This is because the process for computing E_j and C_{ij} in the update procedure requires a pass over all of the data, iterating over each competition, team, and feature to compute the appropriate sums of products of γs. Since these values change after every update, this requires that the features for each move either be kept around or regenerated every update. However, given that generating the features takes more than 12 hours and tens of thousands of updates are required, it was not possible to regenerate them each update, so we were forced to store them in memory.

Unfortunately, even with a sparse vector representation, this meant that memory limitations made it impossible to store even a small fraction of the training set in memory. We estimate that the memory required for storing the full training set would have been on the order of 1 terabyte, which was much more than we had available.

Therefore, we took the radical approach of discarding almost all of the non-expert moves in each position. In each position, we randomly permuted all the legal moves that weren't chosen by the expert, then randomly discarded all but 0.5 percent of them beyond the first twenty.[2]Additionally, to further reduce the memory requirements, we also randomly discarded all but one third of the training instances. However, these training instances were still chosen from all 4121 games, to retain the diversity of game states and provide better independence between training instances.

Of course, a scheme like this loses a substantial amount of information. However, one would intuitively expect the correlations between features and expert moves to remain strong—features that correlate positively with expert moves will continue to do so in the reduced data, and features that correlate negatively with expert moves will do so by virtue of showing up more frequently in remaining random moves than in the expert move. As such, one might plausibly expect the Bradley-Terry model to still produce good feature weights, and this appears to be empirically supported by our results. Moreover, in informal testing, this approach appeared to give better results than the alternatives, such as discarding no data but only training on a few thousand instances (the equivalent of only 20-40 games).

There may be other ways around the problem of memory, and we propose a few in Section 2.5.5 where we discuss possibilities for future work. However, here we did not pursue them.

Unlike memory, computation time was much less of an issue, especially after reducing the data. On the reduced data, the training took around 70 minutes, 25 of which were spent computing the features themselves. Training used about 1.5 gigabytes of memory. For comparison, running Naive Bayes on the reduced data takes 30 to 35 minutes in total with almost no memory usage.

2.5.3 Prediction Accuracy

For both algorithms, we used the ordering function learned by the algorithm to order the legal moves in each position in our testing set of 439 games. For different values of X, we counted the proportion of times the top X moves contained the expert move. For $X = 1$, this is the proportion of times that the ordering actually ranked the expert's move first. We also did the same for X equal to a percentage of the legal moves in a position, rather than an absolute number. Our results are summarized in Tables 2.1 and 2.2.

Both algorithms performed very well at predicting expert moves. More than 85 percent of the time for Naive Bayes, and almost 95 percent of the time for Bradley Terry, the expert

[2]Keeping at least twenty moves ensures that we have a reasonable number of moves remaining in the rare case that there are actually very few legal moves in a position.

Top X Moves	1	2	5	10	50	100	500	1000
Naive Bayes	4.4	7.0	12.8	18.7	38.0	49.3	75.4	84.8
Bradley Terry	12.0	18.1	28.9	38.0	61.0	70.7	88.6	93.6

Table 2.1: Percentage of expert moves falling within the top X moves of the ordering, for different values of X. Average branching factor was 16567.

Top X Percent	5%	10%	20%	40%	60%	80%
Naive Bayes	76.5	85.7	92.7	97.5	99.2	99.7
Bradley Terry	89.4	94.3	97.7	99.3	99.7	99.9

Table 2.2: Percentage of expert moves falling within the top X percent of moves of the ordering, for different values of X.

move was within the top 10 percent of the ordering. A substantial fraction of the time, the expert move was within the first 5 to 10 actual moves, and the Bradley-Terry predictor managed to predict the exact move 12 percent of the time. This is surprising, considering the branching factor of around 16000. These results indicate that our feature-based approach is highly effective, and that our feature set contains many features that correlate well with expert move choices.

The Bradley-Terry predictor clearly outperformed Naive Bayes, despite being trained on only about 1/600 of the data (because of the memory constraints). This is because even with much less data, Bradley-Terry is able to adjust for correlations between different features that violate the conditional independence assumption in Naive Bayes.

For instance, in the current feature set, there are strong correlations between features such as altering the number of defenders at a trap and moving to a destination square adjacent to a trap, or making a goal threat and advancing a rabbit forward. The Bradley-Terry predictor is able to handle such correlations because the degree to which a choice of move reflects on the strength of a feature of that move depends on the strengths of the other features of the same move.

This is especially important for some good moves that include some very bad features. For instance, to allow a rabbit to reach the goal and win the game, it is sometimes necessary to sacrifice a piece by leaving it on the trap square while the rabbit, the only defending piece, moves away. The Bradley-Terry predictor would score this move very well despite the negative feature of the piece sacrifice. This is because such piece sacrifices often occur during

goaling in expert games, which causes the evaluation of reaching goal to be correspondingly higher in the maximum likelihood estimate to "explain away" the fact that this type of move is frequently played.

It is interesting to note that the ordering function for both predictors is essentially the same, a linear function, after an appropriate transformation. Both predictors can be written in a form with only a single real parameter for each feature, where the ordering function is simply the sum of the parameter values corresponding to the features that occur in a move. This is obvious for Bradley-Terry when one takes the logarithm of the feature strengths γ_i, and becomes apparent for Naive Bayes when one takes the logarithm of the "odds-likelihood" formulation of Bayes's Rule. The only difference is that the model backing the linear weights in the Bradley-Terry predictor is more effective.

The only downside is that the Bradley-Terry predictor is significantly more costly to train on the same amount of data, with the primary limiting resource being the memory needed to store all of the feature vectors for all training instances. But even though we were forced to discard vast amounts of data to reduce the memory usage enough to train the Bradley-Terry predictor, it proved to be very good at learning to predict the expert moves anyways.

2.5.4 Testing in Play

We tested the Bradley-Terry move predictor in play in our computer Arimaa program, Sharp, to evaluate its effectiveness as a move ordering function. We hypothesized that by using the learned function to order moves at the root node of each search, we could increase the effectiveness of alpha-beta pruning, thereby allowing the search to reach deeper given the same amount of search time and increasing its strength. Moreover, with the high levels of prediction accuracy we observed for expert moves, we hypothesized that we could in fact use it to prune moves outright and speed up the search further without losing too many good moves, so we tested various degrees of root-level pruning.

Briefly, Sharp is a standard alpha-beta searcher using most of the search enhancements described earlier in Section 1.4, including a limited quiescence search and a hand-coded evaluation function. Prior to our testing, Sharp performed no root-level ordering except for the simple improvement in iterative deepening of searching the best move of the previous iteration first during the next iteration. We also continued to do this during the testing of Bradley-Terry, but aside from this, no other ordering or pruning was done at the root. For some additional details about Sharp, see Section 3.4.2 in Chapter 3.

We played five versions of Sharp using various levels of ordering and pruning in a round-robin tournament at a time control of 5 seconds per move, totaling 1600 games in all. One version did not use the learned ordering at all, one version used it only to order but not prune at the root node of each alpha-beta search, and three other versions used it to prune all but 30, 10, and 5 percent of the legal moves at the root node. The tournament results are summarized in Table 2.3.

	Base	BTOrder	BTPruneTo30	BTPruneTo10	BTPruneTo5
Base	-	80-80	69-91	48-112	49-111
BTOrder	80-80	-	55-105	66-94	42-118
BTPruneTo30	91-69	105-55	-	76-84	70-90
BTPruneTo10	112-48	94-66	84-76	-	69-91
BTPruneTo5	111-49	118-42	90-70	91-69	-

Table 2.3: Pairwise results of tournament between versions of Sharp, playing using 5 seconds per move, with 160 games played per pair. Each entry is of the form "x-y" where x is the number of wins by the row player, and y is the number of wins by the column player. The Base version did not use the learned ordering, BTOrder used it to order moves at the root but not prune, and BTPruneToX used it to order moves and prune all but X% of moves after the first ten.

The results in Table 2.3 are a strong confirmation of our hypothesis that the learned ordering function could be used effectively to prune moves. We see a progression where the versions using increasing degrees of pruning generally won increasing numbers of games against versions that pruned fewer moves, or that did not prune. Against the non-pruning base version of Sharp, increasing the level of pruning from none to 95 percent raised the number of games won out of 160 from 80 up to 111 or 112. Moreover, in every direct comparison between a version that used greater pruning with one that used fewer, the one that used greater pruning won the majority of games.

However, the results from ordering alone ran contrary to our expectations. We hypothesized that the learned ordering function would improve performance even without pruning, yet when comparing the base version of Sharp and the version using the learned ordering, it does not appear that one is clearly better than the other. Of the 160 games they played directly against one another, both won 80, and their results against the other versions are mixed, with one sometimes winning more games and sometimes the other.

To lend additional support to this observation, we computed the strength ratings of each of these versions that would be implied by their match results under the Elo rating

model.[3]We obtained the results in Table 2.4.

	None	BTOrder	BTPruneTo30	BTPruneTo10	BTPruneTo5
Elo rating	-66 ± 23	-69 ± 23	20 ± 22	35 ± 22	81 ± 23

Table 2.4: Relative Elo ratings implied by the tournament results in Table 2.3, 95% confidence intervals (note: each individual confidence interval was calculated assuming all other ratings were accurate)

Indeed, surprisingly, there was no statistically significant difference between the playing strength of the base version and the version using the Bradley-Terry ordering without pruning.

There are a few possible reasons why we failed to observe any improvement in strength using only ordering without pruning, even though the earlier results for expert move prediction accuracy indicate that the ordering function is very good. One possibility is that the limited form of ordering that we do already in the base version of Sharp (searching the previous iteration's move first during iterative deepening) catches the best move frequently enough that the additional gain in alpha-beta pruning from ordering alone isn't very large. Another possibility is perhaps that the gain from ordering consistently fails to allow the search to complete the next iteration of depth. This could occur if the 5 seconds of search time we used happens to be a time limit that consistently allows the search to reach a certain depth and such that reaching the next depth consistently requires more than the gain from ordering alone.

However, the fact that we were able to use our learned ordering function to achieve gains in performance even with extreme pruning demonstrates that it is effective at pruning and filtering moves in actual search. If this were not the case, one would expect to see gradually worsening performance as the pruning became more and more severe and lost more and more good moves. We confirmed this by running a second round-robin tournament using versions of Sharp that performed the same degrees of pruning, but *without* using the learned ordering. The results are displayed in Tables 2.5 and 2.6.

As expected, we observed a catastrophic decrease in playing strength when we pruned moves without the ordering. This is a sharp contrast to the *rise* in strength with the ordering. Drawing from the earlier data on expert move prediction accuracy in Table 2.2, where 89.4 percent of expert moves were caught within the top 5 percent of ordered moves

[3]Using the free program BayesElo [7], with a prior of one win and one loss against a virtual opponent of Elo 0.

	Base	PruneTo30	PruneTo10	PruneTo5
Base	-	33-7	39-1	40-0
PruneTo30	7-33	-	30-10	36-4
PruneTo10	1-39	10-30	-	31-9
PruneTo5	0-40	4-36	9-31	-

Table 2.5: Pairwise results of tournament between versions of Sharp, playing using 5 seconds per move, with 40 games played per pair. Each entry is of the form "x-y" where x is the number of wins by the row player, and y is the number of wins by the column player. The versions labeled PruneToX prune all but X% of moves after the first ten, *without* using the learned ordering.

	None	PruneTo30	PruneTo10	PruneTo5
Elo rating	359 ± 100	73 ± 63	-118 ± 62	-314 ± 81

Table 2.6: Relative Elo ratings implied by the tournament results in Table 2.5, 95% confidence intervals (note: each individual confidence interval was calculated assuming all other ratings were accurate)

and 10.6 percent were lost, it is likely that at the most extreme level of pruning tested here, we are losing the best move about once every 10 turns. However, the increased search depth more than makes up for this.

One must take some care in interpreting these results. For one, 5 seconds per move is a relatively short search time, and it is possible that the improvement from pruning will be lessened for longer search times due to diminishing returns. In the extreme case, if we had enough time to search the entire game tree and compute the precise minimax value, then obviously performing such pruning at the root would give no improvement, and could only hurt the result. Additionally, testing by self-play can sometimes be misleading.

Despite these concerns, the sheer degree of pruning enabled by the learned ordering function and the clear performance gains it provides give a strong affirmation of its effectiveness in practical search. In fact, using a slightly more conservative version method of pruning (where "pruned" moves are still searched, but to a lesser depth), our program Sharp recently won the 2011 Arimaa World Computer Championship [30], and the efficiency gains from this method of ordering and pruning were one of the major improvements that allowed it to do so.

2.5.5 Conclusion and Future Research

In summary, we demonstrated a highly effective method for expert move prediction and move ordering in Arimaa. Using a simple feature-based framework with a generalized Bradley-Terry model, we were able to capture almost 90 percent of all expert moves within only the top 5 percent of our learned ordering. Moreover, in an actual search, we were able to prune up to levels as extreme as discarding 95 percent of all moves while observing a clear gain in playing strength. By showing that machine learning can be successfully applied to the task of ordering and pruning moves in Arimaa, we believe we have opened a door for the exploration of further techniques for handling the explosive branching factor and improving the effectiveness of search.

Additionally, we believe our current algorithm can still be improved, and we offer a few possible ways for doing so.

While the actual application of the ordering function is extremely fast, computing the features themselves is moderately expensive. Currently, ordering the typical 16000 moves or so at the root takes roughly 1/10 to 1/5 of a second, which is too slow to be applied within a search tree unless the search time is extremely large. However, if one could train a faster version of the ordering function using a reduced feature set, one might be able to order and prune within the search tree as well. Finding a way to prune within the tree, rather than just the root, could be a huge additional gain.

Additionally, one could investigate ways to parallelize the Bradley-Terry training process or ways of circumventing the memory limitation. While it is not possible to regenerate the feature data every update, it may be possible to compute it once and store it on disk, where the estimated terabyte of required space becomes easily possible. Assuming there is no overhead for reading the data from disk each time, we estimate that the the training process would take between a few weeks and a month on the complete training set. This is certainly feasible, especially if the training could be parallelized. Training on the complete data would have the potential to improve the learned feature values further, particularly for the features that occur less commonly.

Lastly, one could investigate the addition of other feature types, such as pattern-based features. Pattern-based features were notably absent from our feature set, because in informal testing, we found that a straightforward addition of radius-two patterns did not improve the prediction at all! Yet there is the possibility that larger or alternative types of patterns or other new classes of features could increase the performance. A study of the moves currently mispredicted could help lead to the discovery of these new features.

Chapter 3

Evaluation Functions in Arimaa

Whereas in Chapter 2, we examined ways to combat the extreme branching factor by learning to order and predict moves, in this chapter, we attempt to solve the other main challenge facing search in Arimaa, that of finding a good evaluation function.

To do so, we apply the tools of reinforcement learning and investigate whether it might be possible to automatically learn a good evaluation function. In Section 3.1, we give some background and explain our approach. In Section 3.2, we describe four algorithms for learning evaluation functions, two of them using temporal differences and two of them learning directly from a minimax search. In Section 3.3, we describe the features used for evaluation, and in Sections 3.4 and 3.5, we discuss the details of our implementation, training and results.

3.1 The Problem of Evaluation

Since it is impossible to search the game tree to any significant depth, any method of search in Arimaa requires some way to evaluate the board position and estimate the minimax value of positions well before the end of the game. As discussed in Chapter 1, playout-based methods are unlikely to work. Therefore, we need a heuristic *evaluation function* that can estimate how much of an advantage one player or another has in a given game state. For this task, we apply the framework of reinforcement learning, which has been used successfully in many other games to learn effective evaluation functions [32, 2].

We consider the problem of reinforcement learning and reward estimation as applied to two player adversarial games. As before, let S be the state space of the game, let M be the set of all possible moves that could be made at any point in the game, let $M(s) \subset M$ be the set of legal moves in state s, and let $m^*(s) \in M(s)$ be the optimal move in state s. Then, additionally:

- Let $T(s, m)$ be the state resulting from making move $m \in M(s)$ in state $s \in S$.
- Let $V(s)$ be the minimax value of state $s \in S$.

Our goal is to find some heuristic evaluation function $H : S \to \mathbb{R}$ that approximates V as well as possible, yet that is feasible to compute. We do this by selecting H from some class of functions parametrized by some values $\theta \in \mathbb{R}^k$. For instance, we could consider the class of linear evaluation functions (over some particular feature set or representation of the board), where θ would be the weights of the linear components. Or, we could consider the class of functions representable by a neural network with a certain structure, where θ would be the weights of all the connections between each node in the network.

We consider the case of a learning agent playing a large number of games, possibly against itself, or possibly against one or more other opponents. In each game, the agent plays according to some policy and observes a sequence of states $s_0, s_1, ..., s_n$, where each subsequent state depends on the agent's choice and the response of the opponent. The final state s_n gives a reward $V(s_n)$ equal to the outcome of the game, such ± 1 for a win or loss.

Following the observation of a state or a game, the agent may perform an update $H_\theta(s) \xleftarrow{\theta} *$, changing the value of θ so that the evaluation H_θ more closely agrees with some target value on the state s. This update could be a single step of gradient descent or backpropagation on the parameters θ. Then, by varying when and how to perform this update, we can obtain any of several different algorithms. We describe some of these algorithms in the next section.

3.2 Algorithms

We tested four different algorithms for performing the appropriate updates, each of which we present below. Each of these algorithms has previously been successful for learning evaluation functions in other games, and we will briefly mention these results as we proceed.

3.2.1 TD-Root

Given an opponent that plays according to some policy π, we want $H_\theta(s)$ to estimate the *optimal expected return* $V^\pi(s)$, obtained by maximizing at nodes where it is the agent's turn, and taking the expectation over the opponent's choice of move according to π at nodes where it is the opponent's turn. If the game is stochastic, we also take the expectation over the state transition probabilities of the moves of the game.

In the case where opponent's policy π is optimal and the game is deterministic, V^π is simply the minimax value V, so that we update H_θ to approximate the optimal value of the game, as desired.

In temporal difference (TD) learning [27], we perform updates by considering the *temporal differences* between the estimated expected values at each time step:

$$\delta_i = H_\theta(s_i) - H_\theta(s_{i-1})$$

Specifically, the agent plays a game against the opponent with policy π, and chooses the action with greatest expected return according to H_θ on each turn. Then, at the end of each game, having observed the sequence of states in the game $s_0, s_1, ..., s_n$, we attempt to update the value of each state to match the value of future states, under the assumption that the evaluations of states closer to the end of the game are more accurate than that of the current state.

In the simplest form of temporal-difference learning, for each observed state s_i, we perform the update:

$$H_\theta(s_i) \xleftarrow{\theta} H_\theta(s_i) + \delta_{i+1} = H_\theta(s_{i+1})$$

That is, we adjust the parameters to move the current evaluation towards the evaluation of the immediate next state.

We may also consider the other extreme, that of the *Monte-Carlo update*, where we adjust the evaluation towards the final value[1]:

$$H_\theta(s_i) \xleftarrow{\theta} H_\theta(s_i) + \sum_{j>i} \delta_j = H_\theta(s_n)$$

The algorithm TD(λ) interpolates between these two behaviors according to a parameter $\lambda \in [0, 1]$ by performing the update:

$$H_\theta(s_i) \xleftarrow{\theta} H_\theta(s_i) + \sum_{j>i} \lambda^{j-i} \delta_j$$

Or equivalently:

$$H_\theta(s_i) \xleftarrow{\theta} \sum_{j=i+1}^{n-1} (1-\lambda)\lambda^{j-i}(H_\theta(s_j) - H_\theta(s_i)) + \lambda^{n-i}(H_\theta(s_j) - H_\theta(s_i))$$

Intuitively, TD(λ) works by assigning an ever-decreasing proportion of the "blame" for results further and further into the future. The algorithm attributes an unexpected change

[1]For simplicity of notation, we assume always that $H_\theta(s) = V(s)$ if s is a terminal state of the game.

in the estimated value mostly to the moves immediately preceding the change in value, and exponentially less to more distant moves, which are presumed not to have been the "cause" of the change in value. As such, one might expect TD(λ) to perform best when λ is set to match as closely as possible the distribution of times over which a mistake becomes "evident" and starts being reflected in later scores.

Most notably, TD(λ) was used successfully by Tesauro in the early 1990s to create the world-class Backgammon player TD-Gammon [32]. However, succeeding with temporal difference learning has been much more difficult in more tactical games [2, 4].

One additional modification to the algorithm that is sometimes used in adversarial games when training against fallible opponents is to train only on the negative temporal differences [2]. This is because a positive temporal difference $\delta_i = H_\theta(s_i) - H_\theta(s_{i-1})$ indicates that the agent's estimation of the value unexpectedly rose, which could easily have occurred because the opponent made a mistake and not because the agent misevaluated the position. By contrast, an unexpected drop in the estimated value of a position indicates a clear misevaluation by the agent.

Just as done by Veness et al. [35], we call this version of the TD(λ) algorithm *TD-Root* to emphasize that the update is performed using the actual game positions, which are the root nodes of the minimax searches that one might perform in order to play the game. This is to distinguish it from the next algorithm.

3.2.2 TD-Leaf

The TD-Leaf algorithm is a variation of TD-Root that was developed and used successfully in Chess by Baxter, Tridgell, and Weaver [2]. It is essentially the same as TD-Root, except that the algorithm is applied to the leaf nodes of the *principal variations* of a depth D minimax search from each of the game states. The principal variation is the branch of the search tree resulting when both players choose the optimal move at each node, where the optimal move is determined according to the (depth-limited) minimax values computed by the search for each node.

Specifically, let $H_\theta^D(s)$ be the minimax value of a depth D search starting at state s, using θ as the evaluation function. Let $T_D(s)$ be the state at the leaf node of the principal variation the search, so that $H_\theta(T_D(s)) = H_\theta^D(s)$. Then, TD-Leaf updates using the rule:

$$H_\theta(T_D(s_i)) \xleftarrow{\theta} H_\theta(T_D(s_i)) + \sum_{j>i} \lambda^{j-i}\delta_j$$

where

$$\delta_i = H_\theta(T_D(s_i)) - H_\theta(T_D(s_{i-1}))$$

The motivation for using TD-Leaf rather than standard TD(λ) derives from the difficulty of obtaining an accurate evaluation without any lookahead. Unlike in Backgammon, it is extremely difficult to obtain an accurate static evaluation in games such as Arimaa, Chess, or Othello, due to the sharp variations in evaluation produced by short-term combinatorial tactics. In practice, such tactics are always resolved with extensive search [2].

This means that we can instead focus on making the minimax value $H_\theta(T_D(s_i))$ of a search match the expected value of the game, rather making the value $H_\theta(s)$ of the current state match the future value of the game. Since the minimax value $H^D_\theta(s)$ is the value of the leaf node of the principal variation $H_\theta(T_D(s_i))$,[2]and the derivative of the minimax value $\frac{d}{d\theta}H^D_\theta(s)$ is (usually) the same as $\frac{d}{d\theta}H_\theta(T_D(s_i))$, we can adjust the minimax value by adjusting the value of the leaf.

Note that the derivatives may *not* agree in the case where a change in θ causes the principal variation itself to change. In that case, the derivative of the value of the search with respect to θ can be discontinuous, and we lose any theoretical guarantee of convergence. This can occur even with an arbitrarily small change in θ, if another node along the principal variation has precisely same minimax value. However, in practice, even if we ignore these problems, it is still possible to obtain good results [2].

Intuitively, the advantage of performing updates using the leaf nodes is that the values between successive positions will be more stable or accurate, because we are performing a minimax search at each step to resolve the short-term tactics. This partially relieves the learning algorithm from the difficult task of having to accurately predict the result given the noise introduced by the immediate tactics in a position.

3.2.3 Rootstrap

Unfortunately, as discussed by Veness, Silver, Uther, and Blair [35], there are some potential drawbacks to the temporal-difference algorithms. One such drawback is that they depend heavily on the actual sequence of positions in a game, and in particular, on the opponent. That is, for any fixed opponent policy π, TD(λ) optimizes against π by attempting to estimate V^π, rather than estimating the true minimax value V. In theory, this is not necessarily an obstacle if the algorithm trains against an improving opponent, such as itself, but in practice, training by self play can be unreliable.

[2]This requires that the game be deterministic.

Another drawback they point out is that the distribution of positions observed in actual games is likely to differ greatly from the distribution of positions observed within the search tree. This could lead to inaccurate evaluations for types of positions that are rare in play but common in search. Additionally, none of these algorithms takes advantage of the large amount of data in the search tree resulting from a minimax search, only performing updates using the root positions, or in the case of TD-Leaf, only the principal leaf.

Therefore, the authors present two new algorithms to address these issues, called Rootstrap and Treestrap.

In the Rootstrap [35] algorithm, upon observing any state s, we perform the update:

$$H_\theta(s) \xleftarrow{\theta} H_\theta^D(s)$$

That is, we simply adjust the value of the current state to better correspond to the minimax value of a search from the current state. This gives a simple for learning that removes the dependence on the opponent.

3.2.4 Treestrap

In the Treestrap algorithm [35], upon observing any state s, we perform a depth D minimax search and perform a update for every state in the search tree. Let T be the set of non-leaf states occurring in the search tree, and let $d(s')$ be the depth of state s' in the tree. Then, in the process of performing the search, for all $s' \in T$, we will have computed $H_\theta^{D-d(s')}(s')$, so for every $s' \in T$, we perform the update:

$$H_\theta(s') \xleftarrow{\theta} H_\theta^{D-d(s')}(s')$$

In practice, rather than perform a full-width minimax search, we may wish to use alpha-beta pruning. In this case, many of the values computed for interior nodes in the search tree will be upper or lower bounds rather than exact values. Let $\alpha_{\theta,s}^D(s')$ be the upper bound computed for s' in a depth D alpha-beta search starting from the root s using the evaluation function H_θ, and let $\beta_{\theta,s}^D(s')$ be the lower bound, so that $\beta_{\theta,s}^D(s') \leq H_\theta^{D-d(s')}(s') \leq \alpha_{\theta,s}^D(s')$. For any particular node, only one bound will be relevant - in the case of an alpha cutoff, we have $\beta_{\theta,s}^D(s') = -\infty$, and in a beta cutoff, we have $\alpha_{\theta,s}^D(s') = \infty$, and for exact nodes, both are equal to $H_\theta^{D-d(s')}(s')$.

Then we instead perform the updates:

$$H_\theta(s') \xleftarrow{\theta} \alpha_{\theta,s}^D(s') \quad \text{if } H_\theta(s') > \alpha_{\theta,s}^D(s')$$

$$H_\theta(s') \xleftarrow{\theta} \beta^D_{\theta,s}(s') \quad \text{if } H_\theta(s') < \beta^D_{\theta,s}(s')$$

By performing updates using nodes in the search tree, we obtain many more training instances, potentially allowing for much faster learning. Moreover, we also train on a distribution much closer to the one that the evaluation function will be applied on, namely the positions occurring within a search tree. In Chess, both the minimax and alpha-beta versions of the Treestrap algorithm were shown to greatly outperform Rootstrap and the temporal difference algorithms, both in in speed of learning and the final quality of the evaluation function [35].

3.3 Performing Updates using Features

Just as for move ordering, we apply all of the above algorithms in Arimaa in a feature space of various strategic and tactical features of the board, since learning to evaluate from the raw board representation is likely to be extremely difficult. By mapping into a feature space, we can extract out the otherwise discontinuous and hard-to-learn properties of the game and expose them for direct evaluation, greatly increasing the smoothness of the function that we wish to learn.

Because it is important that the evaluation function be computable extremely efficiently to be usable in a tree search, we chose our evaluation functions to be linear functions over the feature space, where θ is the vector of weights for each feature. Evaluation functions of this form are simple to optimize and have been successful for other games [35]. Formally, given a state s, we compute a feature vector $\phi(s) \in \mathbb{R}^n$ and then we define:

$$H_\theta = \theta^T \phi(s)$$

Then, given a learning rate α, we perform an update $H_\theta(s) \xleftarrow{\theta} K$ by gradient descent:

$$\theta := \theta + \alpha(K - H_\theta(s)) \frac{dH_\theta}{d\theta}(s)$$

3.3.1 Features Implemented

Many of the features we use for evaluation are similar to the ones we used for move ordering. However, for evaluation, we use features that are real-valued, rather than binary. This is because in evaluating the whole board, we are interested in the number of occurrences of different features across the board, whereas for ordering moves, we were more often interested in whether a move had some certain property or not. While some of the features

we specify below are still effectively binary, they are converted into real values by setting them to 0 if not active, and 1 if active.

As with our move ordering features, locations are normalized according to left-right symmetry where relevant, and according to the player to move.

To evaluate a position, features are always computed *twice*, once from the perspective of each player. Then, the feature vector for the opponent is negated and added to the feature value of the current player. This allows a symmetric evaluation, and is also a natural way to evaluate a position. For instance, if having a certain strategic formation is worth a certain value to the current player, then it should be just as valuable to an opponent if held by the opponent, and therefore of negative value to the current player in this case.

Since the various components of the raw feature vector can differ greatly in magnitude and in rarity, for each feature, we computed its sample variance over every position in the 4121 games of the training set we used for move ordering in Chapter 2, and for the purposes of the gradient descent, we normalized the magnitude of each feature so that it had a sample variance of 1. To handle the small number of features that never occurred in the training set and therefore had sample variance zero, we added a small value of 0.01 to the computed variance of each component prior to performing the normalization. Additionally, at the start of training, we initialized the feature vector to begin with reasonable values for material features, set according to current expert judgment.

Piece Type

Just as in the move ordering features, it is important to distinguish how strong a piece is relative to other pieces. However, for the purposes of evaluation, it is more important to make finer distinctions between the precise strength of a piece, since while pieces with similar but distinct strengths may generally prefer to move in and be in the same areas of the board, the long-term value of having such pieces could differ greatly. We distinguish 12 different types, rather than only the 8 types earlier, as follows:

Type 0: Non-rabbit, 0 opponent pieces stronger
Type 1: Non-rabbit, 1 opponent pieces stronger
Type 2: Non-rabbit, 2 opponent pieces stronger
Type 3: Non-rabbit, 3 opponent pieces stronger
Type 4: Non-rabbit, 4 opponent pieces stronger
Type 5: Non-rabbit, 5 opponent pieces stronger
Type 6: Non-rabbit, 6 opponent pieces stronger
Type 7: Rabbit, 24-22 opponent pieces plus opponent non-rabbit pieces
Type 8: Rabbit, 21-19 opponent pieces plus opponent non-rabbit pieces

Type 9: Rabbit, 18-16 opponent pieces plus opponent non-rabbit pieces
Type 10: Rabbit, 15-12 opponent pieces plus opponent non-rabbit pieces
Type 11: Rabbit, 11-0 opponent pieces plus opponent non-rabbit pieces

Material

As in Chess, the most significant factor in Arimaa evaluation is the amount of material each player has.

Unfortunately, using static values for different types of pieces does not work well in Arimaa, because aside from rabbits, pieces are only differentiated by their strength relative to one another. In practice, the value of a piece increases greatly as pieces stronger than it are captured, and increases somewhat when equally strong pieces are captured. Moreover, the value of rabbits increases the fewer pieces total the opponent has, because they are much more able to threaten goal. Therefore, the following material features depend only on relative quantities, rather than absolute piece types:

MAT_PIECE(s,e): Number of non-rabbit pieces with s (0-6) stronger opponent pieces and e (0-2) equal opponent pieces.

MAT_RABBIT(s,e): Number of rabbit pieces with s (0-8) stronger opponent pieces and e (0-8) opposing rabbits.

Additionally, the relative value of rabbits increases as the number of rabbits becomes very few, as losing all rabbits means losing the game by *elimination*. However, this change in value appears to be nonlinear, being unnoticeable when there are many rabbits, and highly noticeable when there are just one or two left. So we add the features:

RABBIT_COUNT(c): Number of rabbits is c (0-8). (binary)

We also add in a heuristic material evaluation known as "HarLog" which is used by some top bots, including our own. While the formula itself is arbitrary with no theoretical justification, it was chosen and tuned by its developer in order to produce values that reasonably closely agree with expert opinion on material values. The HarLog score for a single player is computed as follows:

- Let $Q = 1.447530126$ and let $G = 0.6314442034$
- For each friendly non-rabbit piece with zero stronger opponent pieces, add $2/Q$.
- For each friendly non-rabbit piece with n stronger opponent pieces, add $1/(Q + n)$.
- Let r be the number of friendly rabbits and t the total number of friendly pieces.

- Add $G(\log r + \log t)$.

We add in the HarLog score as a feature as well.

> HARLOG: The HarLog score of the player.

Piece-Square Tables

Different pieces are better or worse in different areas of the board. For example, elephants tend to be strongest in the center, where they can reach and fight for any trap, whereas rabbits and weak pieces are usually best remaining near the edges of the board. Additionally, the squares adjacent to traps are often more valuable.

> PIECE_SQUARE(*s*,*loc*): There is a friendly non-rabbit piece on location *loc* (0-31) with *s* (0-6) opponent pieces stronger than it. (binary)
>
> RABBIT_SQUARE(*t*,*loc*): There is a friendly rabbit on location *loc* (0-31), and the number of opponent pieces plus the number of opponent non-rabbit pieces is *t* (0-24). (binary)

Trap Control

Maintaining "control" of traps is very important, and usually taking over one of the opponent's traps is a large advantage. As in our feature set for move ordering, we distinguish 8 different statuses of defense around a trap, based on the number of defending pieces (0-4) whether or not the elephant is next to the trap. Additionally, we distinguish whether the trap is on our own side of the board or the opponent's.

> TRAP_STATUS(*s*,*ownside*): Number of traps with status *s* (0-7) that are on own side of board if *ownside* (0-1) is true, or opponent's side if it is false.

We also include a heuristic calculation that estimates the current player's control of a trap, taking into account not just the pieces immediately defending the trap, but the other nearby pieces as well.

The trap control heuristic is relatively complex. Its dominant component is computed by iterating over all pieces within a manhattan distance of four from the trap. Each piece is assigned a strength value in $\{32, 22, 17, 13, 11, 10, 9, 9, 9\}$ according to the number of opponent pieces stronger than it. This strength value is multiplied by a distance value in $\{18, 20, 13, 5, 2\}$ depending on that piece's manhattan distance. The resulting values for all friendly pieces are added together, and the resulting values for all opposing pieces

are subtracted. Some additional corrections are added depending on whether pieces are immediately threatened or frozen, for control of key squares around the trap (the squares near the edge of the board are typically safer to hold), and for weak pieces standing directly on the trap (which is good when the trap is threatened by the opponent, since that piece can easily step off to become another defender, but is bad when the trap is already strongly held, since it interferes with one's own ability to capture).

The final value is included as a feature for the traps on the opponent's side only, since the traps on one's own side will be accounted for when all of the features are computed separately from the opponent's perspective.

In practice, trap control appears to behave in a nonlinear fashion, in that increasing one's control of a trap that is under question is more important than increasing one's control of a trap that is already very strongly controlled, or very weakly controlled. Therefore, we also include the same value with a logistic transform $L(x) = 1/(1 + e^{-x})$ applied. For the logistic transform, the values are scaled so that roughly the full range of trap control goes from $x = -5$ to $x = 5$.

TRAP_CONTROL_LINEAR: The sum of the trap control heuristics for the opponent's traps.

TRAP_CONTROL_LOGISTIC: The sum of the logistic-transformed trap control heuristics for the opponent's traps.

Piece Advancement

We include some features that take into account the advancement of different pieces as well as the different amounts of "influence" in their area.

Influence is determined by the same calculation as in the move ordering features. The square under each piece is assigned a value in $\{75, 55, 50, 45, 40, 35, 30, 25, 15\}$ according to the number of opposing pieces stronger than that piece, using negative values for the opponent and zero for empty squares, and then values are diffused four times, where for each adjacent square, a diffusion transfers 0.16 of the value of that square.

ADVANCED_BASE(*type,y*): The number of pieces of type *type* (0-11) at row *y*.

ADVANCED_INFL(*type,y*): The total influence of pieces of type *type* (0-11) at row *y*.

Goal Threats

Rabbits are more valuable if they are threatening to reach goal soon. For each friendly rabbit, we compute an approximation of the number of steps that it would take for that

rabbit to reach goal if the current player were allowed to make any number of steps in a row, without being limited to just four steps per turn. These steps including possibly using other friendly pieces to push opposing pieces out of the way or unfreeze the rabbit.

GOAL_THREAT(s): The number of rabbits threatening to reach goal in approximately s (0-9) steps.

Capture Threats

For each piece, we compute an estimate of the number of steps it would take the opponent to capture that piece, as well as the number of steps it would take for the opponent to "attack" it by placing a stronger piece adjacent to it.

CAPTURE_THREAT(*type,s,thisturn*): Number of pieces of type *type* (0-11) threatened with capture in s (0-6) steps, where *thisturn* indicates whether or not it is actually the current player's turn and the capture can be performed immediately.

ATTACK_THREAT(*type,s*): Number of pieces of type *type* (0-11) such that the opponent is threatening to place an stronger piece next to it in s (0-6) steps.

ATTACK_THREAT_ROW(*type,s,y*): Number of pieces of type *type* (0-11) such that the opponent is threatening to place an stronger piece next to it in s (0-6) steps, where the piece is on row y.

Blocking

Just as in move-ordering, we compute features for pieces that are blocked from entering or leaving each square, either by a single sufficiently strong piece, or by a *phalanx* of multiple weak pieces.

BLOCKED(*type,dir,isphalanx*): Number of pieces of type *type* (0-11) blocked in direction *dir* (0-2), where *isphalanx* (0-1) indicates whether a single piece or a phalanx was used.

Frames

There are several important long-term strategic configurations that can occur in Arimaa that can have very high value, even comparable to the material value of a strong piece. One such type of configuration is called a *frame* (see Appendix B for an illustration). In a frame, a piece is stuck on a trap square with only one defender and is unable to step or push its way off that trap square. This means that the single defending piece, usually the elephant,

is unable to move without sacrificing that piece. In this way, the holder of the frame can pin down the opposing elephant using only weaker pieces, leaving his own elephant as the strongest free piece.

Not all frames are advantageous, however. In particular, if maintaining the frame requires the elephant as well, then the holder of the frame does not always gain the strongest free piece. Therefore, we compute a heuristic value that attempts to estimate whether a frame provides an advantage, given the current material balance on both sides. This is roughly done by pairing off the remaining pieces against the opponent's pieces, adding or subtracting values in {500,140,50,20,10,5,2,1} for each pairing depending who has the strongest piece in that pairing, with some minor corrections and adjustments depending on the details of the frame.

In general, the opponent's only option against such a frame, aside from sacrificing the framed piece, is to attempt to *break* the frame by moving other strong pieces to push or pull away the pieces that are keeping the framed piece stuck on the trap. Therefore, it is important to consider also how many steps the opponent is away from breaking the frame when assessing its value. To enable better generalization, since frames are relatively rare, we include redundant features below that apply regardless of the number of steps to break the frame.

FRAME_BASE(*type*): Number of pieces of type *type* (0-11) framed.

FRAME_VALUE(*type*): Heuristic value of framed pieces of pieces of type *type* (0-11).

FRAME_BASE_BREAK(*type,s*): Number of pieces of type *type* (0-11) framed, breakable in *s* (1-8) steps.

FRAME_VALUE_BREAK(*type,s*): Heuristic value of framed pieces of type *type* (0-11), breakable in *s* (1-8) steps.

Hostages

Another important long-term strategic configuration is called the *hostage* (see Appendix B for an illustration). In a hostage situation, a stronger piece can holds a weaker opposing piece hostage near a trap square so that it cannot escape and so that it is threatened with capture so long as the trap is not defended. Such a situation can also give the strongest free piece. For instance, in a *camel hostage*, the elephant holds the opposing camel hostage, forcing the opposing elephant to defend. This effectively takes both elephants and the opposing camel out of play from the rest of the board, leaving the friendly camel as the strongest free piece.

Not all hostages are profitable, however. In the same manner as for frames, we also compute a heuristic value for hostages to try to estimate whether holding the hostage gives a long term advantage, such as, most notably, having the strongest free piece.

HOSTAGE_BASE(*heldtype,holdertype*): Number of instances of piece of type *holdertype* (0-6) holding opponent piece of type *heldtype* (0-11) hostage.

HOSTAGE_VALUE(*heldtype,holdertype*): Heuristic value of piece of type *holdertype* (0-6) holding opponent piece of type *heldtype* (0-11) hostage.

Elephant Blockades

Another important long-term strategic configuration is called the *elephant blockade* (see Appendix B for an illustration). It is possible for a large number of weak pieces to surround the opposing elephant near the edge of the board and prevent it from moving. This is often massively advantageous for the side holding the blockade.

Again, not all blockades are valuable, particularly if the elephant itself is required to hold the blockade. Just as with frames and hostages, we compute a heuristic estimate that tries to determine if holding the blockade leads to an advantage by estimating to what degree it provides the strongest free piece.

The position of a blockade matters greatly, because on the edge, they require many fewer pieces to hold, and on the opponent's side of the board, the opponent's own pieces can be used to help the blockade. Additionally, partial blockades can also be profitable, but to a lesser extent. Therefore, we add a heuristic parameter called "tightness", where 0 indicates that the elephant can't move at all, 1 indicates that the elephant can move, but not very usefully, and 2 indicates that the elephant can move and free itself, but could take some time to do so.

As with frames, the opponent's only recourse is often to try to *break* the blockade, by advancing other strong pieces to push or pull away the numerous weak pieces that form the blockade. Therefore, as with frames, we compute an estimate of the number of steps that the opponent is away from breaking the blockade.

To enable better generalization, we separate out many of these features into separate classes.

EBLOCKADE_CENTRAL(d,t): Elephant blockaded at manhattan distance d (0-6) from the center four squares of the board, and the blockade has a tightness of t (0-2). (binary)

EBLOCKADE_ROW(y,t): Elephant blockaded on row y (0-7), and the blockade has a tightness of t(0-2). (binary)

EBLOCKADE_CENTRAL_VALUE(d,t): Heuristic value of elephant blockaded at manhattan distance d (0-6) from the center four squares of the board, and the blockade has a tightness of t(0-2).

EBLOCKADE_ROW_VALUE(y,t): Heuristic value of elephant blockaded on row y (0-7), and the blockade has a tightness of t(0-2).

EBLOCKADE_CENTRAL_BREAK(d,s): Elephant blockaded at manhattan distance d (0-6) from the center four squares of the board, and the blockade can be broken in s (1-8) steps. (binary)

EBLOCKADE_ROW_BREAK(y,s): Elephant blockaded on row y (0-7), and the blockade can be broken in s (1-8) steps. (binary)

EBLOCKADE_CENTRAL_BREAK_VALUE(d,s): Heuristic value of elephant blockaded at manhattan distance d (0-6) from the center four squares of the board, and the blockade can be broken in s (1-8) steps.

EBLOCKADE_ROW_BREAK_VALUE(y,s): Heuristic value of elephant blockaded on row y (0-7), and the blockade can be broken in s (1-8) steps.

Elephant Mobility

In general, the ability of the elephant to move quickly around the board is critical, since it is the strongest piece. This motivates some additional features, measuring various aspects of the mobility of the elephant.

EMOBILITY_3(d,n): The elephant is at a manhattan distance d (0-6) from the center of the board and can reach n (0-31) different squares on the board by making 3 steps, including pushes/pulls. (binary)

EMOBILITY_4(d,n): The elephant is at a manhattan distance d (0-6) from the center of the board and can reach n (0-31) different squares on the board by making 4 steps, including pushes/pulls. (binary)

E_TRAP_DIST(d,t): The elephant is at a manhattan distance of d (0-5) from trap t (0-3).

3.4 Implementation and Training

3.4.1 Algorithm Parameters and Implementation

We implemented all four algorithms, TD-Root, TD-Leaf, Rootstrap, Treestrap, using the above feature set with a linear evaluation function, updating weights by gradient descent.

A series of preliminary experiments were performed to determine reasonable learning rates. For each algorithm, we settled on a learning rate of $\alpha = 10^{-6}$ (surprisingly, the same for each algorithm). For each algorithm, we found that this was roughly the highest rate that gave relatively consistent results. Higher rates tended to give unreliable results, sometimes giving moderately higher rates of improvement but also frequent unlearning and occasional divergence of weights.

For both of the temporal difference algorithms, we set $\lambda = 0.7$, a value that has been used successfully by in reinforcement learning in Chess [2]. It seems plausible that a similar value might work as well in Arimaa, given that the total game length in turns is comparable and that the time-scale in turns over which various tactics "pay off" is not too different. However, we did not attempt to tune the value of λ.

TD-Root and TD-Leaf only performed updates only at the ends of games, and additionally, both algorithms used the modification of only training on negative temporal differences, which appeared to perform at least as well, and possibly slightly better during informal testing.

Treestrap was implemented using an alpha-beta search, and the alpha-beta version of the algorithm was used for training. Additionally, a *step-based* search was used, as described in Chapter 1. This means that the internal nodes of the tree used for updates include numerous positions where only part of the steps of the current turn have been made. From the perspective of achieving a good distribution of training instances, this is desirable, because in performing a step-based search, the evaluation function is frequently called on positions with only part of the steps made.

Unlike the temporal-difference methods, both Rootstrap and Treestrap performed updates on every position. Since they do not depend directly on the temporal sequence of positions within a game, the updates were also performed at the end of each move, rather than at the end of each game.

For Treestrap, although all the interior nodes of the tree were used for updates, this was not done until the end of search, so as not to change the evaluation function while the alpha-beta search was ongoing. Nodes within a quiescence search (described below) were not used for updates, although the results of quiescence searches were used. Additionally, the learning rates for updates within the tree were normalized by the number of nodes in the tree, as we found that without doing so, the learned weights behaved erratically, even sometimes diverging.

3.4.2 Playing Algorithm

For each of the algorithms, we played a learning agent using that algorithm against our hand-coded program *Sharp* over a series of 2000 games. We then evaluated the learned weights by playing a non-learning agent using those weights in a series of 100 games against Sharp running at various depths, as well as an older 2010 version of Sharp. Draws were counted as half of a win.[3]

The search algorithm used for the learning agents was the same search algorithm as used by Sharp. This is an iterative-deepening alpha-beta search that incorporates the standard enhancements described earlier in Section 1.4 of Chapter 1, namely hash tables and quiescence search. In addition to pruning repeated positions, the hash table also caches the best moves in each subtree, which are used for move ordering on subsequent iterations. The remaining moves are sorted by the *history heuristic* [25]. Additionally, the Bradley-Terry ordering developed in Chapter 2 was used to order (but not prune) the moves at the root node of the search.

The quiescence search extends critical lines of the search involving captures to improve the tactical evaluation. It is allowed to perform up to 3 steps in a single turn that capture pieces or defend against possible captures by the opponent. These 3 steps are followed by up to 4 steps of just captures, without any capture defenses, for a maximum 7 steps of quiescence search.

Additionally a static decision tree is applied at each node in the search for detecting whether a goal by any rabbit is possible within four steps. Whenever the player to move can goal, the branch of the search is terminated immediately with the appropriate winning score returned. Otherwise, if the opponent can goal, the move generator is restricted to only a subset of moves within a certain radius of the goaling rabbit, a radius for which it is provable that any more distant move cannot be relevant to stopping the rabbit. These goal defense moves are also added to the quiescence search when goals are threatened during the quiescence search.

[3]The Arimaa rules actually have no provision for draws because draw-like situations essentially never happen in human games. In computer games, they can sometimes occur when both programs reach a simultaneous local maximum in their evaluations and remain in it for a very long time, permuting pieces slightly to avoid third-time repetitions. To handle this, we consider a game a draw if it lasts more than 500 moves.

3.4.3 Opponents

Our primary opponent for both testing and training was Sharp. Although Sharp uses the same search algorithm as the learning agents, its uses a hand-coded evaluation function, rather than a learned one. Some of the heuristic components used in the function are similar to the features in our feature set above, most notably much of the trap control heuristic, but the evaluation overall is different, and is in fact somewhat more complex than our feature set above.

The 2010 version of Sharp from a year ago was also used as a testing opponent. It differs enough from the current version that it is not too unreasonable to consider it a different program entirely. Most of its evaluation function is fundamentally different, as is its method of quiescence search. Additionally, its playing style is more defensive and "home-oriented", whereas the current version plays a very aggressive "attack-oriented" style. Having the 2010 version as a testing opponent helps confirm that any differences in playing strength among the learning agents are general, as opposed to exploitative of a single opponent or playing style. Sharp 2010 is slightly stronger than the current Sharp when searching to equal depth, but the current version is stronger when searching for the same amount of time.

3.4.4 Search Depth

Despite the speedup of alpha-beta pruning and the use of good move ordering and hash tables, it remains extremely difficult to search deeply in Arimaa. In order to run enough games, we were only able to perform searches to a depth of 5 steps during training. This is a lookahead of only one turn, followed by one step of the opponent's turn. However, in lines of the search involving capture or goal threats, the quiescence search extends this up to a maximum of 12 steps, or three turns of lookahead. This is still very limited, but enough to see many of the basic tactics in the game.

Games for testing the weights were performed with Sharp searching at depths of 4, 6, and 7 steps, and the agent using the learned weights searching at a depth of 6 steps.

3.4.5 Ensuring Randomness

One thing we strongly wish to avoid in both training and testing is for identical or nearly identical games to be played over and over due to a lack of randomness by the players.

We avoid this in a few ways. Firstly, Sharp varies its moves sometimes. This is done by adding a small random value to its evaluation of each position, drawn from an approximately

Gaussian distribution, with standard deviation scaled to around 1/40th of the value of a rabbit. This causes Sharp to sometimes choose different moves in the same position if those moves lead to positions that it judges to be very close in value. Additionally, we gain some amount of variation during training games simply as a result of the learned weights changing over time, although not during testing. Finally, a significant amount of randomness comes from our procedure for the setup phase, as described below.

3.4.6 Setup Phase

One thing we have neglected to address so far is the process of setting up the board. Up until now, we have only considered algorithms and methods for making moves on the board. However, as explained in the rules, Arimaa actually begins with a *setup phase* where both players place their pieces within their two home rows.

Fortunately, as long as a setup is used that is not too bad, it does not matter too much, since the actual fighting and maneuvering in the middle of the game during normal play is much more important in determining the outcome of the game, especially with the very low depth limits used in our training and testing games. Therefore, we are relatively free to vary the setup, allowing it to become a tool for us to introduce additional randomness into the game.

We use a randomized method that generates numerous random setups and keeps the one with the best score according to a scoring method. Setups with stronger pieces in front, rabbits on the sides and back, and the elephant near the center are given higher scores and are significantly more likely to be chosen. Manual inspection indicates that while many of the setups produced by this method may not be great, they are mostly reasonable and not too disadvantageous. At the same time, there is enough random variation that nearly every setup during the course of training and testing is unique.

3.5 Experimental Results

3.5.1 Comparison of Algorithms

We tested all four algorithms as described above, training against our program Sharp for 2000 games, with both the learning agent and Sharp searching to a fixed depth of 5 steps. Each run took approximately a week. At various points, we evaluated each agent against both the current and the 2010 versions of Sharp searching to various fixed depths. For the testing games played to determine the Elo ratings, the learned agents searched to a fixed

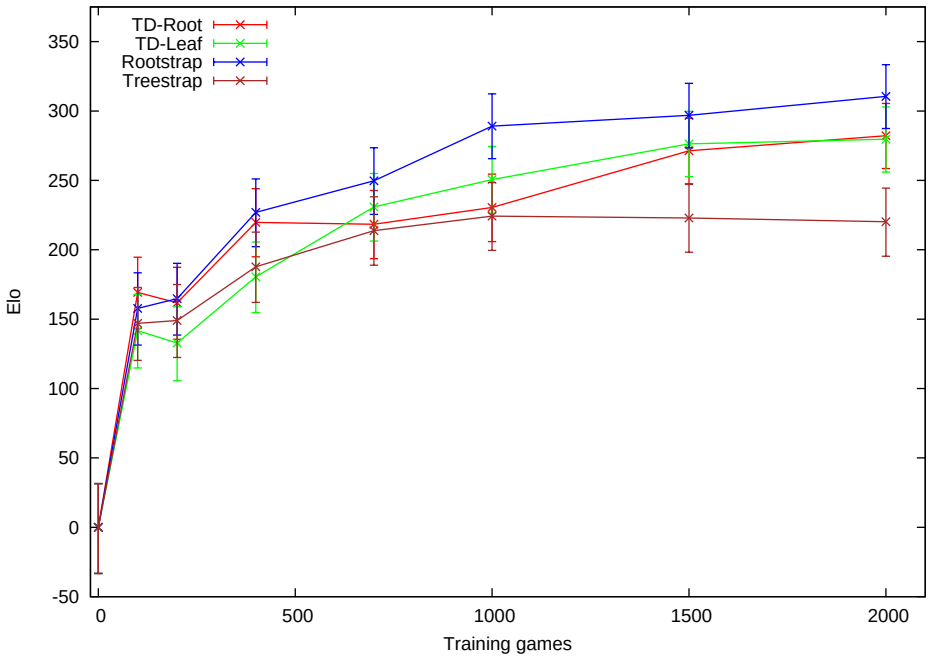

Figure 3.1: Elo ratings over the course of training with 95% confidence intervals. For all algorithms, $\alpha = 10^{-6}$, and for TD-Root and TD-Leaf, $\lambda = 0.7$. Training opponent was Sharp, training searches were done to depth 5, testing at depth 6.

Sharp Version & Depth	D4	D6	D7	2010D4	2010D6	2010D7
Untrained	70	10	3	46	9	3
TD-Root	132	45	28	128	47	27
TD-Leaf	130	48	22	118	46	40
Rootstrap	137	51	34	135	45	40
Treestrap	132	29	16	106	27	25

Table 3.1: Evaluation of the final agents produced by each algorithm, searching to depth 6 against both current and 2010 versions of Sharp searching to various depths. Each box indicates the number of games won by that agent against Sharp out of 200 games.

depth of 6 steps. The results are displayed in Figure 3.1 and in Table 3.1.

Out of all the algorithms, Rootstrap performed the best, and in particular, outperformed the Treestrap algorithm. This is surprising, because it is opposite to the results obtained by Veness et al. [35], where the Treestrap algorithm outperformed the Rootstrap algorithm by a large margin in Chess. One would intuitively expect Treestrap to outperform Rootstrap because it is essentially the same algorithm performing the same updates, except that it uses the intermediate results in the search tree to perform additional updates and thereby learn faster.

This was not quite the case for our implementation, since we always normalized by the number of nodes in the search tree. However, we found that even at higher learning rates, Treestrap continued to underperform, and like the other algorithms, became unstable and likely to unlearn or diverge.

We believe the poor performance of Treestrap occurred because the search depths were too shallow. As discussed earlier, due to computational limits, we were only able to search 5 steps deep for each move, which is barely greater than 1 turn (although this was extended by the quiescence search at many nodes). As a result, it appears that the additional updates performed by Treestrap using the interior nodes did not add any significant useful information for learning, because they were always from very similar positions, and rather merely acted as a source of additional noise. Moreover, because they were searched to an even shallower depth than the root, the resulting updates would be of much lower quality.

Both of the temporal difference algorithms performed very well, almost at the level of Rootstrap, and it is easily conceivable that as parameter settings and the precise training methods are varied, Rootstrap and the temporal difference algorithms could easily alternately overtake one another. We suspect that a reason for the strong performance of the

temporal difference methods relative to Rootstrap is the relatively low search depth, making learning from minimax search not as effective. Furthermore, it is possible that they were more effective at learning longer-term strategic features. Given this, it is not surprising that they did well, even in a moderately tactical game. Also notable is the fact that both TD-Leaf and TD-Root performed very similarly, and again, we attribute this to the relatively shallow depth of the search tree.

Unfortunately, while significant learning did occur with all algorithms, none of them were quite able to meet the performance of our hand-coded evaluation in Sharp, and the most games any agent won against Sharp at an equal search depth of 6 was the Rootstrap agent's 51 wins out of 200. However, as seen in the next section, it is possible to achieve even better results with a higher learning rate, although not consistently.

3.5.2 Higher Learning Rates

As we mentioned earlier, our learning rate of $\alpha = 10^{-6}$ was chosen because it was the highest learning rate that appeared to give relatively consistent results for each algorithm.

At higher learning rates, all algorithms exhibited much less stability, frequently unlearning good results and plummeting in performance due to the weights diverging or wandering to poor values due to noise. However, they also occasionally achieved results that were significantly better than the versions with lower learning rates.

Our best result was achieved by one particularly spectacular run using Rootstrap with a significantly higher learning rate of $\alpha = 10^{-5}$ rather than $\alpha = 10^{-6}$, and is depicted in Figure 3.2.

Sharp Version & Depth	D4	D6	D7	2010D4	2010D6	2010D7
Rootstrap, $\alpha = 10^{-5}$	165	74	59	144	63	54
Rootstrap, $\alpha = 10^{-6}$	137	51	34	135	45	40

Table 3.2: Evaluation of the agents in 3.2 at their respective peaks. Each box indicates the number of games won by that agent against Sharp out of 200 games.

What is exciting about this result is that, as seen in Table 3.2, the performance of the agent using the higher learning rate, at its peak, was not too far from the performance of our hand-coded evaluation in Sharp. At an equal search depth of 6, the agent won 74 games out of 200 against Sharp, indicating that it was indeed not much weaker than Sharp. We find these results strongly encouraging.

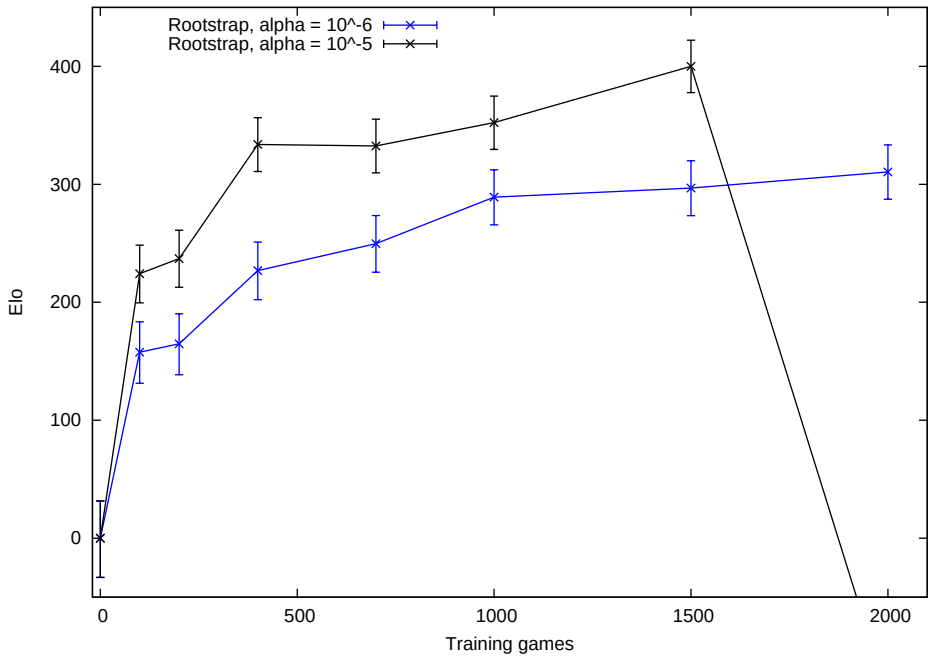

Figure 3.2: Elo ratings over the course of training, with 95% confidence intervals. A learning rate of $\alpha = 10^{-5}$ performs very well, until a sudden drop after 2000 games!

This and one or two similar results in our testing indicate that there exist much stronger evaluation functions over the current feature set that are not yet being found. Furthermore, they suggest that all it may take to achieve them are improvements in the training process, such as a larger number of training games, adaptive learning rates, and deeper search depths during training.

The only limitation so far has been time, since playing thousands of games to test various parameters is costly, as is increasing the search depth or the number of training games. Because of this, we have not yet had the time to fully pursue such alternatives after our initial testing and tuning of the learning rates and our obtaining of these results. However, we believe that with additional work, we have every chance to surpass the current performance.

3.5.3 Future Directions

We offer some additional directions by which we might be able to surpass our hand-coded evaluation and improve the state of evaluation in Arimaa.

Even though Rootstrap performed the best in our testing, it only did so by a small

margin, and even among just these four algorithms, it is not clear yet which will prove to be the most effective for Arimaa. In particular, we believe that there is substantial potential for the Treestrap algorithm and variants of it still to succeed, particularly if we increase the search depth. Even with the relatively shallow search depths, we believe there is much information in the search not being exploited optimally yet. It is also possible that there is a bug in our current implementation, and if so, we might observe substantially better performance if it were fixed.

Moreover, we believe there is potential for considering a hybrid algorithm that uses both temporal difference methods and direct backups from the search tree as in Rootstrap and Treestrap. Under the hypothesis that direct backups using minimax values from the tree are more effective detecting short-term tactics and learning weights for features that correlate with them, and under the hypothesis that temporal difference updates are more effective at weighting strategic features that lead to long-term losses or gains in the future, perhaps a hybrid algorithm would be able to get the best of both worlds.

Thirdly, we believe that improvements to the current feature set may help significantly. Currently, there are a number of straightforward tactical and strategic features that we are missing, such as frozen pieces, specific features for goal threats and defenses, and combined features for trap control and advancement. One might also try the addition of pattern-based features, or look at methods for automatic detection of potential features.

Finally, given the relative success of these algorithms at learning when initialized with only raw material weights, one wonders how much better they might do if used to optimize an existing evaluation function. Granted, there are additional challenges associated with tuning some of the nonlinear components of evaluation functions like the one currently used by Sharp. However, we believe that there are many possibilities here, and this is something we hope to try with Sharp's evaluation in the near future.

Chapter 4

Conclusion

In Chapter 2, we showed that using a generalized Bradley-Terry model, it is possible to learn a highly accurate move-ordering function for Arimaa, and moreover that such a function can be used effectively for pruning to speed up search and improve the level of play. Our move ordering function succeeds almost 90 percent of the time in ranking the expert move within the top 5 percent of the ordering, and we have used it to great success in our current program Sharp to prune moves and improve the efficiency of the search, increasing the strength of the program by around 140 Elo at short time controls.

In Chapter 3, we showed that any of several algorithms is capable of substantial automated learning in Arimaa, beginning with only raw material weights. While none of them were able to achieve the same level of performance as our hand-coded evaluation, our best result came quite close, winning 74 out of 200 games against our bot Sharp at equal search depths. Additionally, there is evidence that significant improvement is possible, and that with future work, it will be possible to use such techniques to surpass the best current methods for evaluation in Arimaa.

In both move ordering and evaluation, we gave several possible avenues by which our results might be improved further. For move ordering, there is a possibility for massive improvement in finding a way to compute the move ordering more efficiently so as to use it to prune within the tree as well. There is also the potential to improve the prediction accuracy by resolving the memory usage of the Bradley-Terry model optimization to allow the full data set to be used. For learning an evaluation function, it is almost certainly the case that our current training procedure can be improved, and we believe also that there is the potential for improvements in the algorithms themselves. And in both of these areas, we believe that there are gains to be made by adding new types of features to the feature sets we used. All of these have the potential to further the results we have achieved so far.

Overall, we believe our work is a vindication for the ability of machine learning to improve

the state-of-the-art in Arimaa. We have demonstrated that there is much potential for algorithms and techniques in ranking, classification, and reinforcement learning to raise the level of computer play, and there remains much to be explored and tried. We believe through a combination of such techniques and continued improvements in search and evaluation, it will be possible to achieve world-class-level play in Arimaa, perhaps even within the next decade.

Bibliography

[1] Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, The Netherlands, 1994.

[2] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Learning to Play Chess Using Temporal Differences. *Machine Learning*, 40(3):243–263, September 2000.

[3] Paul M Bethe. The State of Automated Bridge Play. January 2010.

[4] Marco Block, Maro Bader, Ernesto Tapia, Marte Ramrez, Ketill Gunnarsson, Erik Cuevas, Daniel Zaldivar, and Ral Rojas. Using Reinforcement Learning in Chess Engines. *Research in Computing Science*, 35:31–40, 2008.

[5] Michael Buro. Takeshi Murakami vs Logistello. *ICGA*, 20(3):189–193, 1997.

[6] Remi Coulom. Computing Elo Ratings of Move Patterns in the Game of Go. *ICGA*, 30(4):198–208, December 2007.

[7] Remi Coulom. Bayesian Elo Rating. `http://remi.coulom.free.fr/Bayesian-Elo/`, 2010. Accessed on 2011-03-27.

[8] Christ-Jan Cox. Analysis and Implementation of the Game Arimaa. Master's thesis, Maastricht University, The Netherlands, May 2006.

[9] David Fotland. Building a World-Champion Arimaa Program. *Lecture Notes in Computer Science*, 3846:175–186, 2006.

[10] Sylvain Gelly and David Silver. Combining Online and Offine Knowledge in UCT. pages 273–280, 2007.

[11] Mark Glickman. A Comprehensive Guide to Chess Ratings. *American Chess Journal*, 3:59–102, 1995.

[12] Brett Harrison. Move Prediction in the Game of Go. Undergraduate thesis, Harvard University, April 2010.

[13] Brian Haskin. A Look at the Arimaa Branching Factor. `http://arimaa.janzert.com/bf\_study/`, 2006. Accessed on 2011-03-29.

[14] Tzu-Kuo Huang, Ruby C. Weng, and Chih-Jen Lin. Generalized Bradley-Terry Models and Multi-Class Probability Estimates. *Journal of Machine Learning Research*, 7:85–115, 2006.

[15] David R. Hunter. MM Algorithms for Generalized Bradley-Terry Models. *The Annals of Statistics*, 32(1):384–406, 2004.

[16] Michael Bradley Johanson. Robust Strategies and Counter-Strategies: Building a Champion Level Computer Poker Player. Master's thesis, University of Alberta, October 2007.

[17] Larry Kaufman. Pawn and Two Move Handicap Match IM Eugene Meyer vs. Rybka 3. `http://www.chessbase.com/newsdetail.asp?newsid=4837`, August 2008. Accessed on 2011-02-20.

[18] Tomas Kozelek. Methods of MCTS and the game Arimaa. Master's thesis, Charles University of Prague, Czech Republic, December 2009.

[19] T.A. Marsland. A Review of Game-tree Pruning. *ICGA*, 9(1):3–19, 1986.

[20] Sam Miller. Researching and Implementing a Computer Agent to Play Arimaa. Thesis, University of Southampton, UK, May 2009.

[21] Hun Nagashima. *Towards master-level play of Shogi.* PhD thesis, Japan Advanced Institute of Science and Technology, March 2007.

[22] Arpad Rimmel, Olivier Teytaud, Chang-Shing Lee, Shi-Jim Yen, Mei-Hui Wang, and Shang-Rong Tsa. Current Frontiers in Computer Go. *IEEE Transactions on Computational Intelligence and Artificial Intelligence in Games*, 2(4):229–239, December 2010.

[23] Jonathan Rubin and Ian Watson. Computer Poker: A Review. *Artificial Intelligence*, 2011. preprint.

[24] Stuart J. Russel and Peter Norvig. *Artifcial Intelligence, A Modern Approach.* Prentice Hall, 2 edition, 2003.

[25] Jonathan Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:1203–1212, 1989.

[26] Jonathan Schaeffer, Neil Burch, Yngvi Bjornsson, Akihiro Kishimoto, Martin Muller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, September 2007.

[27] Richard S. Sutton. Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3(1):9–44, 1988.

[28] Omar Syed and Aamir Syed. Arimaa - The next challenge. `http://arimaa.com/arimaa/`. Accessed on 2011-03-31.

[29] Omar Syed and Aamir Syed. Notation for Recording Arimaa Games. `http://arimaa.com/arimaa/learn/notation.html`. Accessed on 2011-02-20.

[30] Omar Syed and Aamir Syed. The 2011 Arimaa World Computer Championship. `http://arimaa.com/arimaa/wcc/2011/`. Accessed on 2011-03-31.

[31] Omar Syed and Aamir Syed. Arimaa - A New Game Designed to be Difficult for Computers. *ICGA*, 26(2):138–139, June 2003.

[32] Gerald Tesauro. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[33] Gerhard Trippen. Plans, Patterns and Move Categories Guiding a Highly Selective Search. pages 111–222, 2009.

[34] John Tromp. Number of chess diagrams and positions. `http://homepages.cwi.nl/~tromp/chess/chess.html`, 2010. Accessed on 2011-02-20.

[35] Joel Veness, David Silver, William Uther, and Alan Blair. Bootstrapping from Game Tree Search. In Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1937–1945. 2009.

[36] Haizhi Zhong. Building a Strong Arimaa-playing Program. Master's thesis, University of Alberta, Dept. of Computing Science, September 2005.

Appendix A

Derivation of the Bradley-Terry Update Rule

We reproduce the analysis and optimization procedure described in [6].

In summary, in our generalized Bradley-Terry model, we consider competitions between teams of agents where agent i has strength γ_i. The strength of a team $T \subset [1, ..., N]$ is $\gamma(T) = \prod_{i \in T} \gamma_i$, and in a competition between teams $T_1, ..., T_n$, the probability that T_j wins is:

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

Given a series of results R_j of competitions and the winners of those competitions, we seek the strengths $\gamma = (\gamma_1, \gamma_2, ..., \gamma_m)$ that optimize the likelihood:

$$P[R|\gamma]$$

Assuming independence of the results:

$$P[R|\gamma] = \prod_{R_j \in R} P[R_j|\gamma]$$

Then, by the model, every individual $P[R_j|\gamma]$ is of the form:

$$\frac{\gamma(T_w)}{\sum_{k=1}^{n} \gamma(T_k)} = \frac{\prod_{i \in T_w} \gamma_i}{\sum_{k=1}^{n} \prod_{i \in T_k} \gamma_i}$$

where w is the index of the winning team.

Focusing on a single γ_i that we wish to update, and fixing the values of all other parameters in γ as constants, we may write this as:

$$\frac{\gamma(T_w)}{\sum_{k=1}^{n} \gamma(T_k)} = \frac{A_{ij}\gamma_i + B_{ij}}{C_{ij}\gamma_i + D_{ij}}$$

where $A_{ij}, B_{ij}, C_{ij}, D_{ij}$ are constants and either $A_{ij} = 0$ or $B_{ij} = 0$. In particular $A_{ij} \neq 0$ iff the winning team contains feature i.

For example, if we have have three teams $T_1 = \{1, 2, 3\}, T_2 = \{2, 4\}, T_3 = \{1, 5, 6, 7\}$, and T_1 is the winner, then:

$$\begin{aligned} & P[R_j|\gamma] \\ &= \frac{\gamma_1\gamma_2\gamma_3}{\gamma_1\gamma_2\gamma_3 + \gamma_2\gamma_4 + \gamma_1\gamma_5\gamma_6\gamma_7} \\ &= \frac{(\gamma_2\gamma_3)\gamma_1 + 0}{(\gamma_2\gamma_3 + \gamma_5\gamma_6\gamma_7)\gamma_1 + \gamma_2\gamma_4} \end{aligned}$$

so that $A_{ij} = \gamma_2\gamma_3$, $B_{ij} = 0$, $C_{ij} = \gamma_2\gamma_3 + \gamma_5\gamma_6\gamma_7$ and $D_{ij} = \gamma_2\gamma_4$.

Then, we use minorization-maximization on the log probability to update γ_i. We have:

$$\begin{aligned} & \log P[R|\gamma] \\ &= \log \prod_{j=1}^{N} \frac{A_{ij}\gamma_i + B_{ij}}{C_{ij}\gamma_i + D_{ij}} \\ &= \sum_{j=1}^{N} \log(A_{ij}\gamma_i + B_{ij}) - \sum_{j=1}^{N} \log(C_{ij}\gamma_i + D_{ij}) \end{aligned}$$

Since for every j either $A_{ij} = 0$ or $B_{ij} = 0$, for the left sum, we obtain terms of the form $\log B_{ij}$ or $\log A_{ij} + \log \gamma_i$. In each case, we may drop all the constants $\log A_{ij}$ and $\log B_{ij}$ since they do not affect the maximization, and letting W_j be the number of times $A_{ij} \neq 0$, we may maximize:

$$W_j \log x - \sum_{j=1}^{N} \log(C_{ij}x + D_{ij})$$

where currently $x = \gamma_i$. We minorize the right sum by taking the tangents to the logarithms at $x = \gamma_i$, and after simplification, the expression to be maximized becomes:

$$W_j \log x - \sum_{j=1}^{N} \frac{C_{ij}x}{C_{ij}\gamma_i + D_{ij}}$$

Letting $E_j = C_{ij}\gamma_i + D_{ij}$ be the total strength of all participants in competition j, the maximum is achieved at:

$$\frac{W_j}{\sum_{j=1}^{N} \frac{C_{ij}}{E_j}}$$

And therefore, our update rule for a single γ_i is:

$$\gamma_i \leftarrow \frac{W_j}{\sum_{j=1}^{N} \frac{C_{ij}}{E_j}}$$

We do not prove any theoretical results here for the convergence of our generalization of the Bradley-Terry model. For a discussion on the general convergence properties of this type of minorization-maximization on Bradley-Terry models, see [15], as well as [14].

Appendix B

Strategic Formations in Arimaa

There are several strategic formations in Arimaa that can be very valuable over the long-term. Most of them involve various ways of limiting the mobility of stronger opposing pieces using weaker pieces, so that one's own pieces become the strongest on the remainder of the board. We describe a few such formations here.

Phalanxes

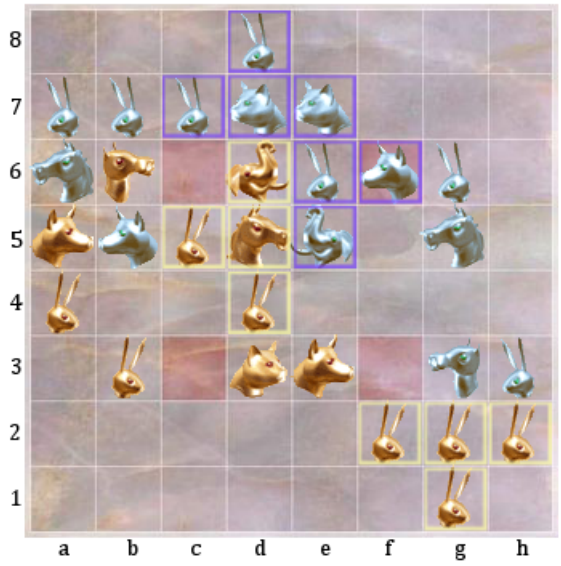

Figure B.1: Both Gold and Silver have formed phalanx formations to block the movement of opposing pieces.

A large number of strategic formations in Arimaa use subformations called *phalanxes* in order to block pieces from reaching certain squares. In a phalanx, a cluster of weak pieces

blocks every available space around a certain piece. This prevents a stronger opposing piece from pushing into that square, simply because there is no empty square for the pushed piece to be displaced to. Such formations are created very frequently during the game.

For example, in Figure B.1, the highlighted cluster of gold rabbits around g2 blocks the silver camel at g3 from pushing its way south, because there is no adjacent empty space where the gold rabbit at g2 could be pushed to. This greatly delays the threat of silver's h3 rabbit, because if the camel could push its way south, it would not be long before it could push enough pieces out of the way to allow the silver rabbit to reach goal.

Similarly, there is a gold phalanx at d5, preventing the silver elephant from going west to defend the c6 trap. This guarantees that gold will be able to start capturing silver pieces in that trap.

Silver has his own phalanxes at e6 and d7 preventing the gold elephant from moving into those squares, but unfortunately, they are not helpful, since Gold's elephant is happy to remain where it is, capturing pieces in the c6 trap. This illustrates that sometimes phalanxes may not be so useful. Nonetheless, they are an important component of tactics in Arimaa, and form key pieces of some of the subsequent formations.

Frames

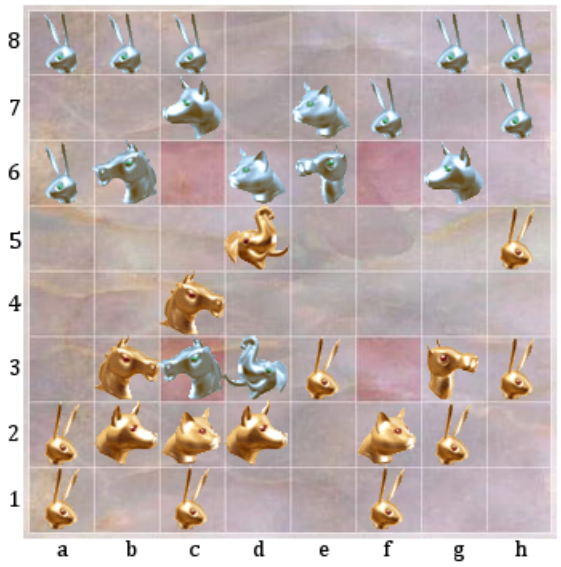

Figure B.2: The silver elephant on d3 cannot move without sacrificing the horse framed on c3.

Figure B.2 illustrates a the strategic formation in Arimaa known as a *frame*. In this position, the silver horse cannot push its way off of the c3 trap square because it is framed by the two gold horses and a phalanx of weaker pieces to the south. This means that the silver elephant cannot move either, because it is the only defender of the horse, and moving would cause the horse to be captured. Therefore, Gold's elephant is now the strongest free piece on the remainder of the board, giving Gold a large advantage. Silver's only options are to try to *break* the frame by bringing his camel to remove the gold horses, or to sacrifice his horse, accepting a major material loss to regain the mobility of his elephant.

Hostages

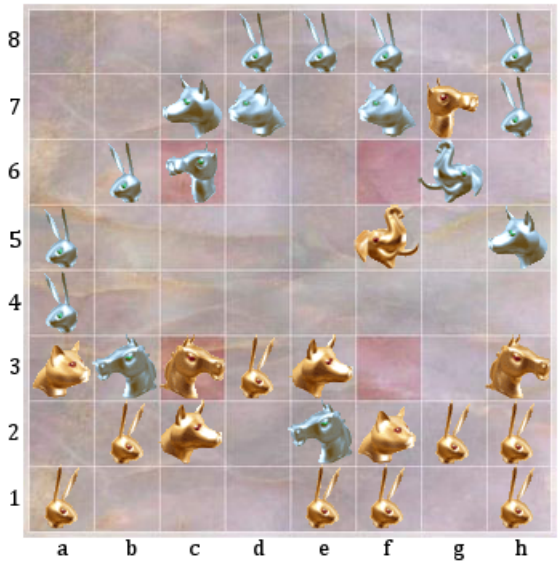

Figure B.3: The silver elephant holds the gold camel hostage next to the f6 trap, forcing the gold elephant to defend.

Figure B.3 depicts another common strategic formation in Arimaa, known as the hostage. The gold camel on g7 is frozen by the silver elephant and is continually threatened with capture in the f6 trap. This prevents the gold elephant from moving away from that trap, since otherwise Gold would lose one of his most valuable pieces. Therefore, Silver's camel becomes the strongest free piece on the rest of the board. Moreover, Silver's horses can attack freely, since the only pieces that could threaten them, the gold elephant and camel, are occupied.

Possibly Gold's best option is to advance a large number of pieces to try to defend the f6 trap with multiple weaker pieces and free his elephant again, before the silver horses

and camel cause too much damage. If that does not work, Gold can only settle for moving his elephant away to attack one of Silver's horses, accepting an unfavorable horse-for-camel trade.

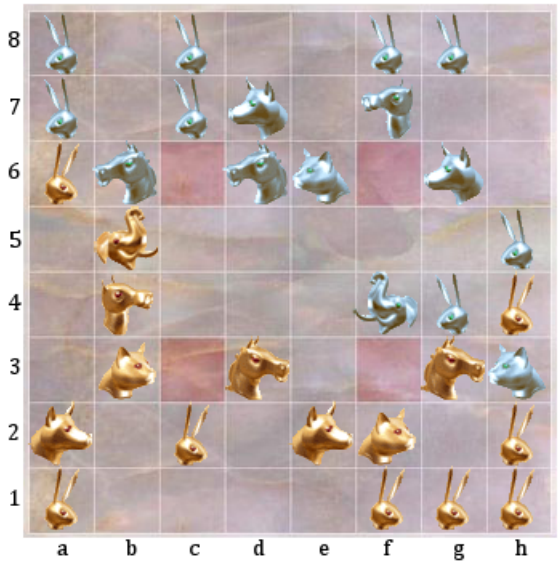

Figure B.4: The gold horse on g3 holds a silver cat hostage. The silver horse on b6 holds a gold rabbit hostage.

Other types of hostages are also common, such as in Figure B.4, where gold holds a cat hostage on h3 using his horse. Silver also holds a rabbit hostage on a6 using his own horse, but not for much longer, since the gold elephant and camel are about to pull the horse away.

Elephant Blockades

A particularly devastating formation in Arimaa is the elephant blockade. In Figure B.5, Gold's elephant is blocked on three sides by silver phalanxes, and the only remaining direction it can step is into a trap, where it would immediately be captured. With Gold's elephant unable to move, Silver's elephant dominates the board. Gold's only hope is to try to free his elephant by using his horses and camel to break the blockade, but with both the silver elephant and camel active in the center, Silver will almost surely be able to win something in the meantime.

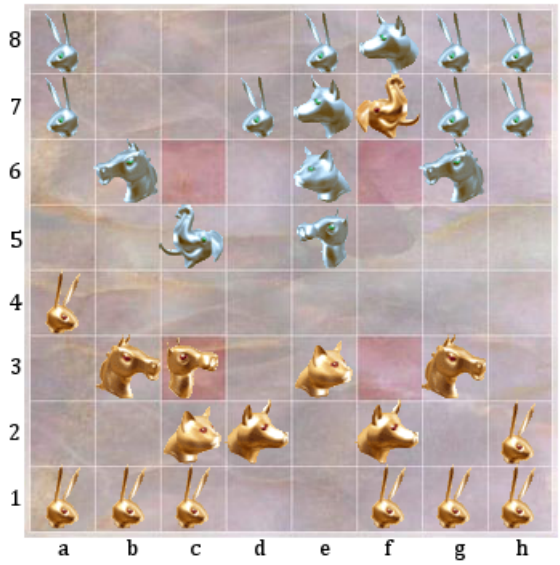

Figure B.5: The gold elephant is blockaded.