Leveraging Game Phase in Arimaa

Vivek Choksi, Neema Ebrahim-Zadeh, Vasanth Mohan
Stanford University

Abstract

Past research into AI techniques for the game Arimaa has dealt with refining the evaluation function used in minimax tree search as well as defining a move ordering to prune the number of moves considered at each game board. In this paper, we describe a function to identify the game phase of a board in Arimaa. Intuitively, game phase captures the idea of progression throughout a game. We apply this knowledge of game phase to the evaluation function of botFairy—an open-sourced bot—and to move ordering. Preliminary results for applying game phase to board evaluation are negative: botFairy outperformed our modified bot when both bots used a 2-ply minimax search with alpha-beta pruning. However, applying game phase to move ordering yielded slight improvements, especially for the later phases of the game.

1. Introduction

Arimaa is a two-player strategy board game played on an 8x8 checkerboard. As in chess, each player controls 16 pieces. Omar Syed, Arimaa's creator, designed the game to be intuitive for humans but computationally difficult for computers [1]. This difficulty for computers arises in part from the game's enormous branching factor, which results from a player's ability to make four steps[1] in one turn. Researchers have estimated an average of 16064 distinct legal *moves* per turn, with an average of 46 turns per player per game [2]. As a result of Arimaa's large branching factor, standard AI search techniques have not been nearly as successful in Arimaa as in other games.

To combat this large branching factor, researcher David Wu wrote his undergraduate thesis on the application of *move ordering* to Arimaa [2]. In our past work, building almost entirely upon Wu's work in move ordering, we applied various machine learning models (Naive Bayes, logistic regression, and SVM) to move ordering in Arimaa [3]. We used these models to calculate the "expertness" of moves, and we defined "expert" moves as moves played by players with an Elo of 2100 or higher.

However, it is unlikely that there exists a single, abstracted, "expert-move cutout" that applies to the entire spectrum of game scenarios. Rather, it is more likely that experts tune their tactics and strategy depending on the scenario, and a game-phase model could bring this context-based play to the forefront.

The exploration of game-phase is also applicable to the evaluation function in game-tree traversal. By giving a game playing bot a representation of what phase of the game we are currently in, the bot can make a more informed decision as to what strategies it should adopt. In other words, knowledge of game phase in a bot's evaluation function would allow it to mimic human game-phase-based play (for example, by assigning high value to advancing rabbits in the endgame).

With these points in mind, we researched the notion of game phase in Arimaa, the tactics that are used in

[1] A step is defined as moving a piece in one of the four cardinal directions.

these phases, and the application of game phase to the two objectives outlined above. We begin in the next section by describing a few important rules in Arimaa for readers unfamiliar with the game. In Section 3, we discuss the definition of game phase and present two methods of implementing game phase discrimination. Section 4 details how we applied game phase to an existing Arimaa bot, and Section 5 details our application of game phase to move ordering. The last section concludes with a discussion of possible future research paths.

2. Rules of Arimaa

In Arimaa, the two players are referred to as *gold* and *silver*. Each player has 16 pieces, homologous[2] to the game of Chess. The hierarchy of pieces is as follows, from weakest to strongest: eight rabbits, two cats, two dogs, two horses, one camel, and one elephant. Each player's pieces all start on the two home ranks (ranks 1 and 2 for gold, ranks 8 and 7 for silver). The game's objective is to advance any of one's rabbits to the opponent's home rank.

To be clear, in Arimaa, as in Chess, *rank* is a synonym for row, and *file* is a synonym for column. The rank number increases from bottom to top, and the file letter increases from left to right. (The silver camel in the board in Figure 1 is said to be on (file, rank) = h7.)

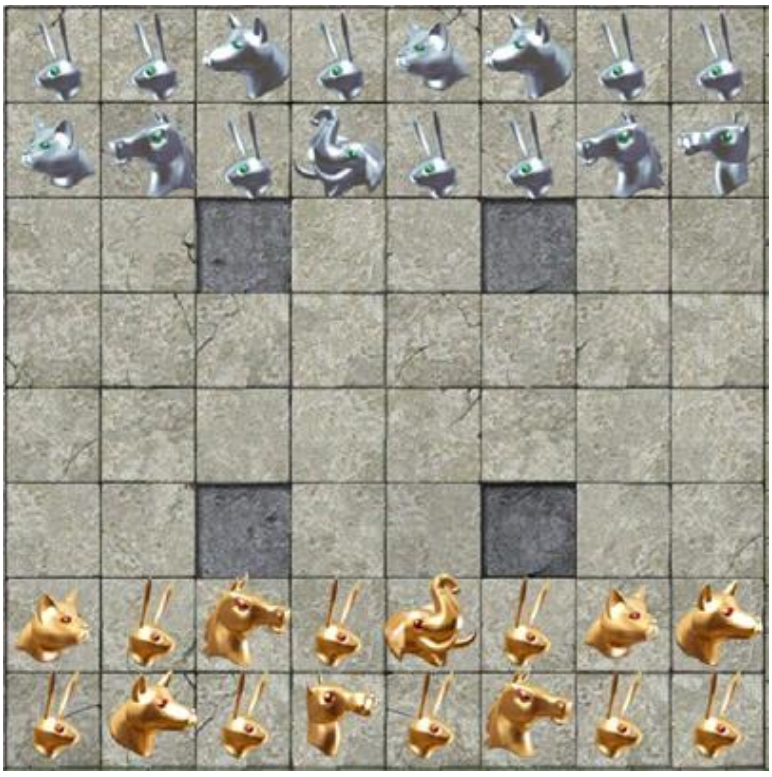


Figure 1: An Arimaa board in one of many possible starting configurations. [4]

Unlike in Chess, where the starting positions for each piece are set by the rules of the game, Arimaa players are allowed to place their pieces on their two home ranks in any starting permutation. Gold begins play. Each player's *move* consists of up to four *steps*. A *step* consists of moving any piece in one of the four cardinal

[2] Actually homologous—there is a one-to-one correspondence between Chess pieces and Arimaa pieces.

directions, with the exception of rabbits, which can never move towards the home rank on their own volition.

Additionally, an Arimaa player must keep the following facts in mind when making a move.

1. Stronger pieces are able to dislodge (*push* or *pull*) strictly weaker pieces, consuming two *step*s.
    a. In a *push*, the pushing piece assumes the location of the pushed piece, which can be displaced in any of the valid[3] cardinal directions.
    b. In a *pull*, the pulling piece moves in any of the valid cardinal directions, and the pulled piece assumes the former location of the pulling piece.
2. Stronger enemy pieces can *freeze* a player's weaker piece if it is *isolated*—that is, if the weaker piece has no touching allied pieces.

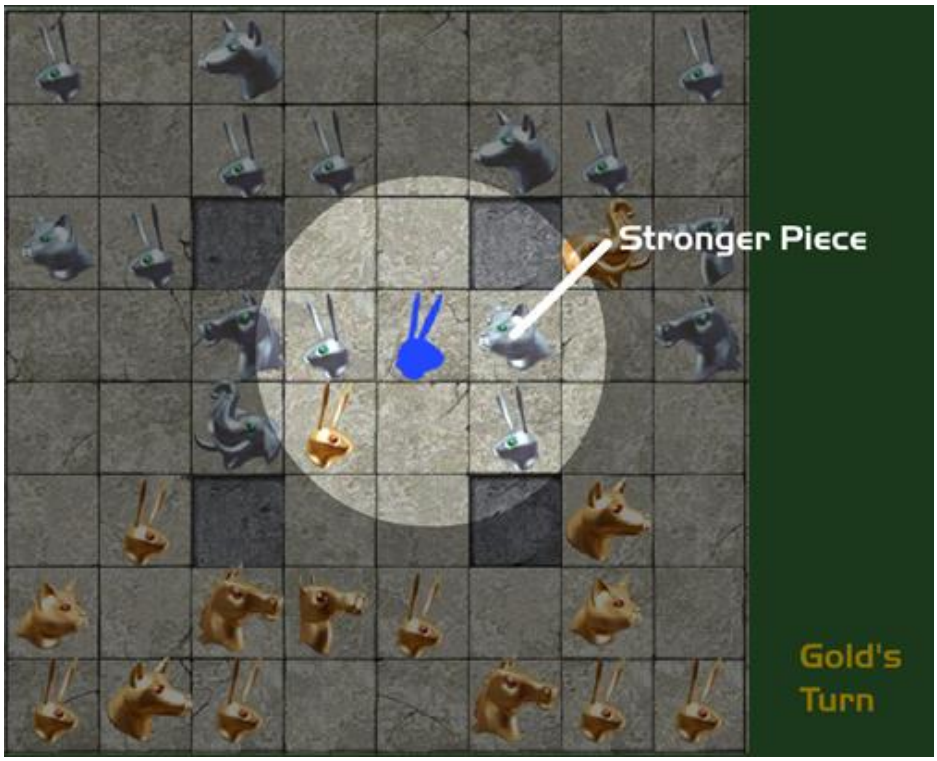


Figure 2: An example of a frozen rabbit. The gold rabbit on e5 (highlighted in blue) is frozen by silver's cat. Coincidentally, the rabbit on d4 is also frozen (by silver's elephant). No other pieces on the board are frozen. [4]

The Arimaa board also designates four capture squares, at c3, c6, f3, and f6. (These squares are shaded and extruded in Figures 1 and 2.) Any *isolated* piece standing on a capture square at the end of a *step* is removed from play. The prospects of *push*ing and *pull*ing allow players to threaten capture of enemy pieces. In Figure 2, the silver rabbit on f4 can be captured in 3 steps by the gold dog on g3: the dog can step north and execute a 2-step push to move the rabbit to the trap square on f3. (The dog ends up on f4.)

For the rules in some very special situations, please consult the Arimaa website [5].

[3] A direction is considered valid if no other piece is currently at that location and the direction is not opposite the direction of the push (i.e., swapping piece locations is not a valid push).

3. Understanding game phase

3.1 Defining Game Phase

Our work aims to improve AI in Arimaa by drawing inspiration from human gameplay. In particular, we aim to capture the changes in the strategy of a human player as the game progresses.

Intuitively, game phase captures the idea of progression (from beginning to completion) of a game. More precisely, game phase is defined using the following three features:

1. $\min(N_g, N_s)$, where N_g and N_s are the number of gold and silver pieces on the board, respectively.
2. $\max(A_g, A_s)$, where
    a. A_g and A_s are the number of advanced gold and silver non-elephant pieces on the board, respectively, and
    b. an advanced gold piece is defined as any piece on ranks 4 or above, and an advanced silver piece is defined as any silver piece on ranks 5 or below.
3. I_{threat}, an indicator variable for whether there is a goal threat from either player on the board.

3.2 Discriminating game phase

Depending on the value of a board's feature vector, the board can be grouped into different categories—called game phases. Two particular methods of categorization are expounded below: (1) heuristic game phase discrimination with hard coded categories, and (2) cluster-based game phase discrimination to find "natural" categories in existing data.

3.2.1 Heuristic game phase discrimination

Each board was assigned a real-valued score by taking the dot product of the board's feature vector and a hand-coded feature-weights vector. Given the score of a board, we classified it into an appropriate game phase based on hand-coded cutoffs. Hand-coded cutoffs and feature-weights were determined through trial and error.

3.2.2 Game phase discrimination using clustering

In the clustering approach, we used a data set of archived games in order to find groups of similar boards. We extracted the features (details in Section 3.1) from each board in each game, thereby converting our data to a set of points in feature space. These points were clustered using the X-Means algorithm [6,7], where each of the X resulting clusters represented a distinct game phase and X is the optimal number of clusters (according to a statistical criterion) within a user-specified range.

3.3 Application of each approach

We used the heuristic approach to encode game phase into the evaluation function, and we used the clustering approach to apply game phase to move ordering.

The strength of the heuristic approach is that it allows us to define intuitive game phases (i.e., beginning, middle, and end) and tune our discriminator to align to these game phases. Since we have a clear understanding of what these game phases mean in terms of gameplay, it is natural to encode phase-based nuances into a bot's evaluation function.

On the other hand, the strength of the clustering approach is that it leverages the power of statistics to find groups of similar boards. This approach seems best suited for the application of game phase to move ordering, with the hypothesis that experts play similar moves on similar boards and that statistical methods can capture these subtle gameplay patterns best.

4. Combining game phase with Fairy's evaluation function

4.1 Application to Fairy

We applied our heuristic game phase discriminator to the evaluation function of a pre-existing open-sourced bot, Fairy[4] [8], in order to see if we could improve the scores it gave to boards. Fairy was written in 2006 and is currently rated at an Elo of 1754. Fairy's evaluation function calculates board statistics such as the position of pieces (special emphasis is given to rabbits), the number and type of each piece on the board, and the number and type of pieces surrounding the four traps. These statistics are weighted and summed together to generate a score for a board position.

Given the statistics that Fairy uses to evaluate a board, we heuristically determined which phase(s) related to which statistic used by Fairy. Once we determined this mapping, we modified the weights assigned to certain statistics based on the game phase of the board being evaluated. Below is a list of the changes we made to Fairy's board evaluation:

- We penalized the piece values of cats and dogs that are advanced in the beginning game because cats and dogs are weak in isolation against the full force of the enemy's pieces.
- If a piece is framed[5] by three opponent pieces in the end game, we did not penalize its value (whereas Fairy does penalize its value) because a framed piece ties up the enemy pieces required to maintain the frame.
- In the beginning and middle game, we favored material value and trap value slightly more than the advancement of rabbits. In the end game, we heavily favored the advancement of rabbits because rabbit advancement allows us to win the game.

Our changes to Fairy's evaluation function resulted in a modified version of Fairy. We call our modified bot "Tweaked Fairy."

4.2 Results and discussion of tweaks to Fairy's evaluation

4.2.1 Evaluation of heuristic discriminator

We evaluated the heuristic discriminator by running it on a set of test games and qualitatively assessing the results. We stepped through these test games and verified that the heuristic discriminator agreed with our intuition in assigning game phases to boards. Figure 3 below shows how game phase changes as the game progresses in four example games.

[4] Different sources refer to Fairy with different names, such as Fairy, bot_Fairy and botFairy.

[5] A framed piece is a piece that is surrounded in the cardinal directions by 3 stronger opponent pieces and 1 allied piece.

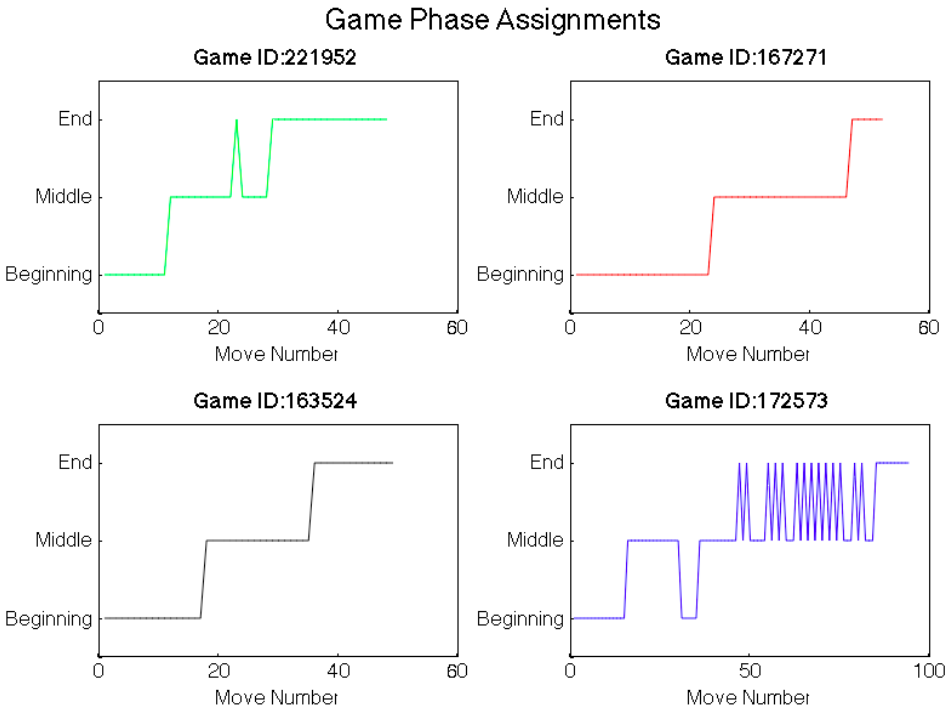


Figure 3: Game phase assignments for four of the games in our test set. These games, labeled by Game ID, were pulled from archives on the Arimaa website. At each move in each game, we assigned the board a game phase (plotted on the y-axis) according to our heuristic game phase discriminator. In our qualitative assessment, we noticed that many games oscillated between middle- and end-game, as in the bottom right plot. This oscillation results from the appearance and disappearance of goal threats.

4.2.2 Evaluation of Tweaked Fairy

We evaluated our work by playing Fairy against Tweaked Fairy in a series of test games, using a pre-existing bot-playing framework [9]. Both bots used alpha-beta pruning in traversing the minimax tree. We played one series of games with both bots using 1-ply lookahead and another series of games with both bots using 2-ply lookahead. We did not use lookahead at 3-ply or greater because of computation time constraints.

One disadvantage of this test game methodology was that a slight time advantage was given to Tweaked Fairy. In the test games, there was no time limit per move, and both bots searched the game-tree to the same depth. Since Tweaked Fairy's evaluation function involved more computation than Fairy's (because Tweaked Fairy calculated game phase), Tweaked Fairy took more time per evaluation. In particular, extracting the goal threats feature (indicator I_{threat} defined in Section 3.1) was the most computation-time-intensive component of Tweaked Fairy's game phase discrimination. In order to minimize the extra evaluation time that Tweaked Fairy requires, we created a new version of Tweaked Fairy that did not use goal threats at all. As a result, we tested two versions of Tweaked Fairy:

- Tweaked Fairy GT (**G**oal **T**hreats), which used all three features described in Section 3.1
- Tweaked Fairy NGT (**N**o **G**oal **T**hreats), which used all features except for I_{threat}

Version of Tweaked Fairy	Version of Fairy	Win percentage of Tweaked Fairy	Total games played
1-ply Tweaked Fairy GT	1-ply Fairy	88.35%	1914
1-ply Tweaked Fairy NGT	1-ply Fairy	67.12%	1834
2-ply Tweaked Fairy GT	2-ply Fairy	34.84%	155
2-ply Tweaked Fairy NGT	2-ply Fairy	33.33%	132

Table 1: Win percentages for Tweaked Fairy in games played against Fairy.

Our results, summarized in Table 1, show that Tweaked Fairy outperformed Fairy at 1-ply lookahead, but underperformed against Fairy at 2-ply lookahead. Also, while Tweaked Fairy GT greatly outperformed Tweaked Fairy NGT at 1-ply lookahead, it only slightly outperformed Tweaked Fairy NGT at 2-ply lookahead; this could be because goal threat detection is especially important when using 1-ply lookahead, since 1-ply lookahead does not recognize moves that would allow the enemy to score. Since play at 2-ply lookahead gives a better indication as to the strength of a bot, these results show that Tweaked Fairy is not an improvement to Fairy.

4.2.3 Error analysis

Our experimental methodology had shortcomings:

1. Fairy does not use a competitive evaluation function. Any results that we could have achieved by modifying Fairy may not generalize to other bots, especially to bots with the most competitive and highly-tuned evaluation functions.
2. As mentioned earlier, a slight time advantage was given to Tweaked Fairy, since Tweaked Fairy required more computation per board evaluation than Fairy but was still allowed to search the game-tree to the same depth and breadth.

Our results showed that Tweaked Fairy performed worse than Fairy at 2-ply lookahead. We propose these hypotheses as to why this could be the case:

1. We did not have a chance to tune or validate the hand-coded values that we used in our modifications to Fairy's evaluation function. These values could have been poorly calibrated. Therefore, even in light of negative results, the usefulness of game-phase-based evaluation cannot be discounted.
2. Since different weighting schemes to evaluate boards were used at different game phases, moves that transition between phases could be wrongly favored or disfavored.

5. Application of game phase to move ordering

5.1 Move ordering with Naive Bayes

In our past research into move ordering in Arimaa [3], we implemented a Naive Bayes classifier to model moves based on "expertness." We used a multivariate Bernoulli event model coupled with Laplace smoothing. Below, we describe this model and the problem of move ordering in more detail. Starting in Section 5.2, we describe how we applied game phase to build upon this past work.

The first major documented effort in applying move ordering to the game of Arimaa was presented by researcher David Wu. Move ordering applies machine learning to rank all possible moves given a game state

[2]. Game-playing bots can utilize the move-ordering ranking to consider moves in a smart order or to discard all but the top ranking moves. We implemented the following features—all inspired by and well-detailed in Wu's paper [2]—to capture qualities of moves played by high-ranking Arimaa players:

1. Position and Movement
2. Trap Status
3. Freezing Status
4. Stepping on Traps

Our implementation of the Naive Bayes model for move ordering slightly modified the traditional Naive Bayes classifier because we were dealing with move *ordering* rather than move *classification*. Instead of classifying a given move (denoted by feature vector x) as expert or non-expert (y = 1 or 0, respectively), our Naive Bayes model output the log of the posterior odds of y, as shown below. We used this value as a "score" for a move and ordered all possible moves in descending order according to this score. The ordering function h was given by:

$$h(x) = \log\left(\frac{P(y=1|x)}{P(y=0|x)}\right), x \in \{0,1\}^{1176}$$

In order to train our model, we performed the following algorithm (written below in pseudocode):

for each game in the training data**:**
 for each board position in the game**:**
 1. *Generate* all possible legal moves, including the "expert" move actually played in the game
 2. *Extract* feature vectors for all of these moves
 3. *Update* a table that holds the frequency of occurrences for each feature (for y = 1 and y= 0)

In order to evaluate our model, we used cross-validation by training on 70% of the data and testing on 30%. In order to train only on expert human moves, we only considered games in which both players' Elo ratings were above 2100. (This left 5,020 of the 278,000 total archived games as potential training candidates.)

5.2 Methodology of the clusterer

Building upon this past work, we decided that we could improve our singleton Naive Bayes model by splitting it into smaller models, each of which predicted expertness for a given phase of the game. More precisely, given some pre-computed notion of game-phase, these smaller models—one per game-phase—were trained only on moves corresponding to that particular game phase.

As for computing game phase for use in our move ordering scheme, we used the Weka library's implementation of X-Means [6,7]—an extension of the K-Means algorithm that efficiently and statistically optimizes the number of clusters in a dataset.

The X-Means clusterer was trained on a random sampling of games from the Arimaa database for which both players were slightly above average (Elo > 1600). For each board in each game, the board position was abstracted into a feature vector according to the same features described in Section 3.1. The resulting set of feature vectors was clustered using the X-Means algorithm.

5.3 Multiple Naive Bayes models

For each cluster generated by the X-Means algorithm, we created a corresponding Naive Bayes model. Our

slightly updated training algorithm is written below in pseudocode:

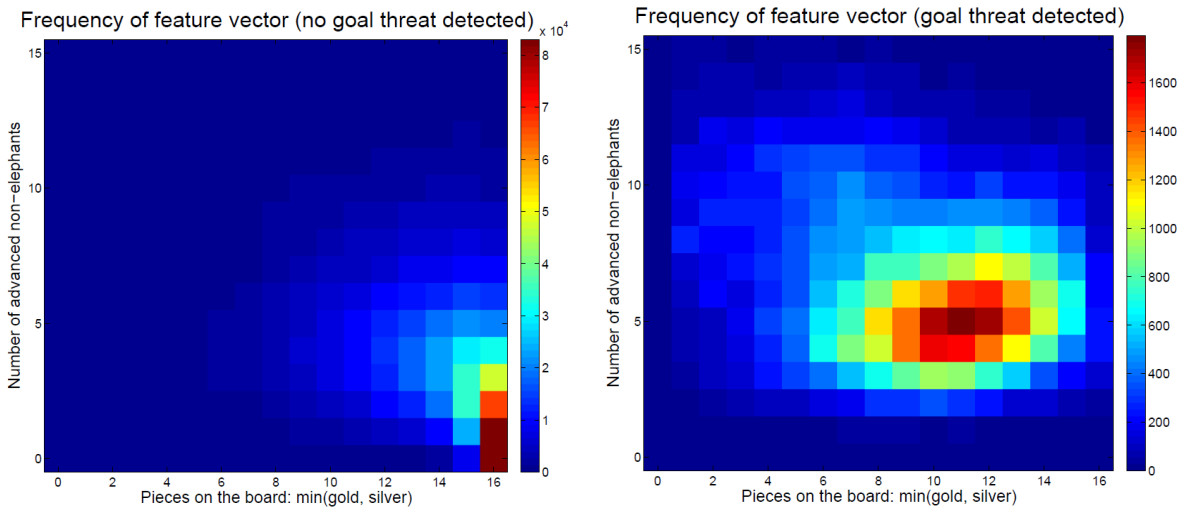
Train the X-Means clusterer using any games where players were rated above an Elo of 1600.
Train the Multiple Naive Bayes model with a slightly modified method as follows:

for each game in the training data set**:**
  for each board position in the game**:**
    1. *Generate* all possible legal moves, including the "expert" move actually played in the game
    2. *Extract* feature vectors for all of these moves
    3. ***Classify*** **the board position using X-Means**
    4. *Update* **the one table corresponding to the cluster assignment of the board** that holds the frequency of occurrences for each feature (for $y = 1$ and $y = 0$)

In comparing the singleton and the Multiple Naive Bayes models, it is worth noting that for a fixed amount of data, there is a trade-off between the two. In the Multiple Naive Bayes model, we have trained multiple models, each of which has been given a fraction of the total data. Therefore, to attain convergence, the Multiple Naive Bayes model requires training on more games than the original Naive Bayes model did.

5.4 Results and discussion of move ordering

5.4.1 Evaluation of X-Means



Figures 4 and 5: These figures show a pseudo-heat-map of the feature vectors used to train the X-Means clusterer. Because the features were integer-valued, the plots appear as pixelated mosaics, with each pixel's color representing the frequency of that particular vector. Furthermore, the third dimension of the three-dimensional feature space consists of an indicator variable, so Figure 4 represents $I_{threat} = 0$ (no goal threat on the board) and Figure 5 represents $I_{threat} = 1$ (goal threat present on the board). Note the large discrepancy in scale between the two figures (80,000 versus 1,800 boards).

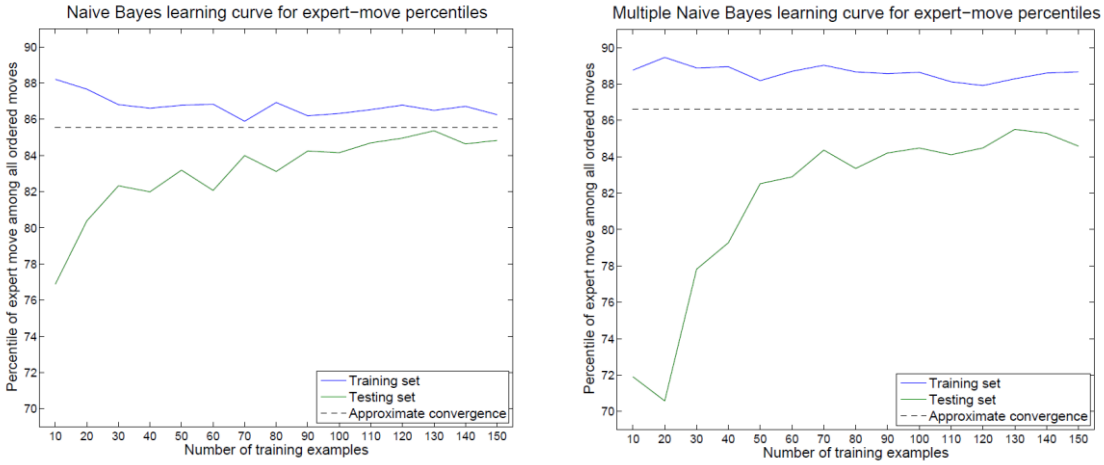
Figures 4 and 5 above show the frequency distribution of the game-phase feature vector across 15,000 games drawn from data where both players were rated above 1600 Elo (slightly above average and better).

As Figures 4 and 5 seem to corroborate, X-Means found two clusters in the data: one for starting boards, and one for boards with a goal threat. In practice, far more moves are played that do not establish a goal threat

than do (this is clear from the scale discrepancy between the two plots). In order to further distinguish between moves without a goal threat, we forced a minimum of three clusters for the X-Means algorithm. (Indeed, X-Means returned the lowest number of clusters possible: three.) Although forcing more clusters biases the X-means algorithm, an added benefit of increasing the number of clusters was that training data was more evenly split between clusters and, hence, models.

5.4.2 Evaluation of move ordering with clustering

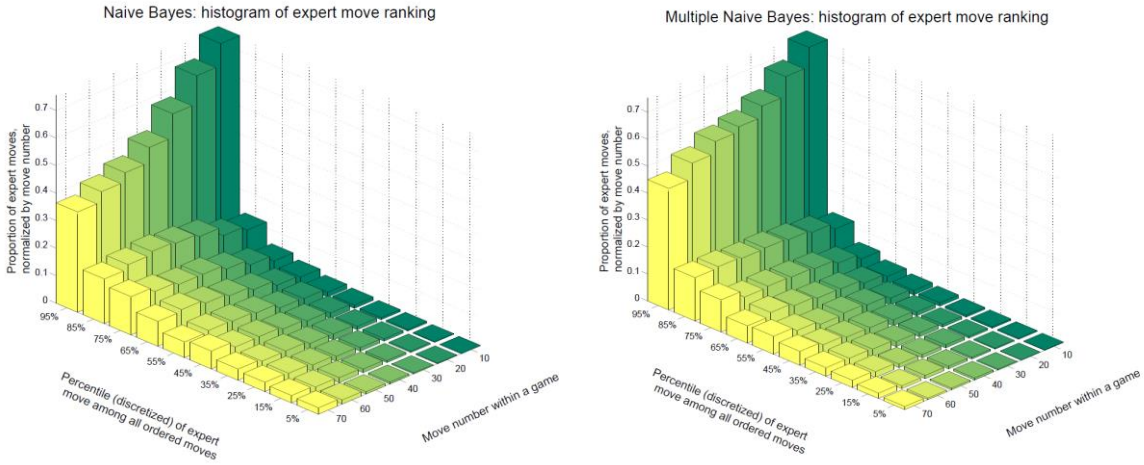
We evaluated the Multiple Naive Bayes model in the same way that we evaluated the singleton Naive Bayes model in our past work. This enables a side-by-side comparison of the two methods. Below are two learning curves. We tested our models four times each on {10, 20, …, 150} games to produce the trial averages shown below:



Figures 6 and 7: These curves follow the expected behavior for high bias models. We see that these plots seem to converge to around 86%, with slightly higher performance from Multiple Naive Bayes. We will need to implement more features (and perhaps select different models altogether) to decrease bias.

We see that the learning curves are similar for the Naive Bayes and Multiple Naive Bayes models. Multiple Naive Bayes converged more slowly to a slightly larger expert-move percentile. Not shown is that with 1000 training examples, Multiple Naive Bayes converged (quite nicely) to 87.5%: the training set percentile decreased only slightly (with increasing numbers of examples) while the testing set percentile increased a few percent to meet it.

In order to measure not only the average percentile but also the consistency with which the models rank expert moves highly, we plotted the histograms shown in Figures 8 and 9.



Figures 8 and 9: These plots compare the number of moves played within a game and the percentile of the expert moves vs. the proportion of expert moves. We discretized the number of moves played into buckets of 10 moves. We also discretized the percentile into buckets of 10%, where the displayed percentile is the average value of the bucket.

The histograms show expert move percentiles along with an added dimension: move number within the game. The plots were generated by running two Naive Bayes trials for Figure 8 and two Multiple Naive Bayes trials for Figure 9. Each model was trained on 700 games (to attain convergence of training/testing) and tested on 300 games. We see that both models rank the expert moves more highly in the beginning of the game (for about the first 10-20 moves, marked in dark green) than for the middle- and end-game, where the distribution flattens. However, the Multiple Naive Bayes model fares better than the singleton Naive Bayes model in the middle- and end-game. The proportion of expert moves ordered in the top 10% of the move ordering decreases monotonically (as move numbers increase) to a final value of .44 compared to .36 for the singleton Naive Bayes model. (The value of .44 means that 44 out of every 100 expert moves were within the top 10% of the move-ranking, across moves made by players 61-70 moves into the game.)

5.4.3 Error analysis/discussion for move ordering

We hypothesize that the features implemented tend to capture the qualities of a good move more for the beginning-game than for the end-game. Our feature set has not been extended to consider other important features of moves such as capture threats and goal threats. Additionally, even with multiple models for different game phases, the X-Means clustering plots show that there is an imbalance in the availability of training data for some of the clusters. States near the starting state are overrepresented in training because every game starts in that state. The result is that model training data has a proportional preponderance to train the "beginning game" model. Furthermore, many games end even before the 50th move, so the data for the histogram plots simply contains more data instances for earlier moves.

6. Conclusions and future work

6.1 Conclusions

Preliminary results of applying game phase to board evaluation were negative. Notably, when playing Fairy against Tweaked Fairy—where both bots used a 2-ply minimax search with alpha-beta pruning—Tweaked Fairy won less than ⅜ of its games. Still, it is hard to entirely discount the application of game phase to the evaluation function since the performance of this approach may improve when applied to different bots or with more tuning of parameters.

Applying game phase to move ordering produced slightly positive results. Across a large test set of games, the Multiple Naive Bayes model converged to an expert percentile 1.5% above that of the singleton Naive Bayes model. Furthermore, past move twenty, the Multiple Naive Bayes model on average placed expert moves at or above the 90th percentile with higher frequency than did Naive Bayes. This suggests that the Multiple Naive Bayes model captured some phase dependency in expert play that the singleton Naive Bayes model did not.

6.2 Future work

6.2.1 Strict vs. probabilistic cluster representation

In our game phase discrimination, we assigned boards to discrete game phases. However, one shortcoming of this strict assignment of game phase is that it assumes that all boards belong to a single, predefined game phase; in reality, boards are not so black-and-white. One could instead interpret game phase probabilistically, assigning boards probabilities of belonging to different game phases. Under a probabilistic interpretation, a board that is "in between" beginning and middle game would be represented as such, possibly resulting in better and more nuanced models.

6.2.2 Game phase pruning

Another interesting application of game phase would be to predict whether, at a given board state in the minimax tree, it is worth searching further down the tree. For example, if a path down the tree is leading a bot closer to the start of the game, it would be able to stop looking down that path and spend computation time else where.

6.2.3 Speeding up move ordering

One major caveat of our move ordering is that it is significantly slower than evaluating leaf nodes. With our current implementation, it is only feasible to prune the first ply of the minimax tree. By decreasing the computation time of move ordering, we would able to apply move ordering to all plies of the minimax tree, thus allowing us to look deeper down the tree.

6.2.4 Improving the quality of move ordering

In our previous work on move ordering, in addition to evaluating a Naive Bayes model, we tested linear support vector machine and logistic regression models. Those models performed better than Multiple Naive Bayes did here, suggesting that "Multiple SVM" or "Multiple logistic regression" models could yield further improvement.

Secondly, the feature set that we used for move ordering was quite limited—we implemented only a fraction of what was implemented in Wu's thesis [2]. Based on Wu's thesis and our comparatively poor results, we have reason to believe that additions to the feature set will improve the performance of any models used for move ordering.

6.3 Concluding remarks

Overall, we believe that game phase is a useful concept, especially in the context of Arimaa. It surfaces different opportunities to prune the minimax tree and thus combat Arimaa's large branching factor. Hopefully, with continued improvements in how we examine the minimax tree, we will be able to achieve "expert-level" performance in Arimaa and other games in the near future.

Acknowledgements

We would like to thank Arzav Jain, who was a collaborator for the entirety of our work on Arimaa minus the writing of this paper. Furthermore, we extend our gratitude to Professor Michael Genesereth of the Stanford Logic Group for mentoring us over the course of our research. We also recognize David Wu and the Fall 2013 CS229 teaching staff at Stanford University for supporting us throughout our previous work. We built upon both bot_Clueless and botFairy in our research, and we thank Jeff Bacher and Ola Mikael Hansson, respectively, for implementing and open-sourcing these bots. Lastly, we are grateful for Stanford's Farmshare computing resources, which were used to train and test our models.

Works cited

[1] O. Syed. (2013). *Please say more about the design decisions* [Online]. Available: http://arimaa.com/arimaa/forum/cgi/YaBB.cgi?board=talk;action=display;num=1367476894#7

[2] D. J. Wu, "Move Ranking and Evaluation in the Game of Arimaa," Undergraduate honors thesis, Dept. Computer Science, Harvard Univ., Cambridge, MA, 2011.

[3] A. Jain *et al.*, "Move Ranking in Arimaa," Stanford University, Stanford, CA, 2013. Available: http://cs229.stanford.edu/proj2013/JainEbrahim-ZadehMohanChoksi-MoveRankingInArimaa.pdf

[4] *Learn Arimaa* [Online]. Available: http://arimaa.com/arimaa/learn/flash/jc/new/Arimaa4.swf

[5] *Arimaa Game Rules* [Online]. Available: http://arimaa.com/arimaa/learn/rules.pdf

[6] *Downloading and Installing Weka* [Software]. Available: http://www.cs.waikato.ac.nz/ml/weka/downloading.html

[7] D. Pelleg and A. W. Moore, "Extending K-means with Efficient Estimation of the Number of Clusters," in Seventeenth International Conference on Machine Learning (ICML), 2000, 727-734.

[8] O. Hansson. (2006, May 26). *bot_Fairy: A sample (non-bitboard) Arimaa playing program written in C* [Software]. Available: http://arimaa.com/arimaa/download/

[9] B. Haskin. (2009). *AEI - Arimaa Engine Interface* [Software]. Available: http://arimaa.janzert.com/aei/

[10] J. Bacher. (2006, March 6). *bot_Clueless: A sample bot written in Java* [Software]. Available: http://arimaa.com/arimaa/bots/