

Rowan University

Rowan Digital Works

Theses and Dissertations

2-14-2017

Exploring algorithms to recognize similar board states in Arimaa

Malik Khaleeqe Ahmed
Rowan University

Follow this and additional works at: <https://rdw.rowan.edu/etd>



Part of the [Artificial Intelligence and Robotics Commons](#)

**Let us know how access to this document benefits you -
share your thoughts on our feedback form.**

Recommended Citation

Ahmed, Malik Khaleeqe, "Exploring algorithms to recognize similar board states in Arimaa" (2017).
Theses and Dissertations. 2360.
<https://rdw.rowan.edu/etd/2360>

This Thesis is brought to you for free and open access by Rowan Digital Works. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Rowan Digital Works. For more information, please contact LibraryTheses@rowan.edu.

**EXPLORING ALGORITHMS TO RECOGNIZE SIMILAR BOARD STATES IN
ARIMAA**

by

Malik Khaleeqe Ahmed

A Thesis

Submitted to the
Department of Computer Science
College of Science and Mathematics
In partial fulfillment of the requirement
For the degree of
Master of Science in Computer Science
at
Rowan University
December 16, 2016

Thesis Chair: Dr. Nancy Lynn Tinkham

Dedication

For my daughter Haya, who was born almost three years after I started work on this thesis. Although pushing through and finishing this thesis has been difficult with the number of responsibilities I had taken upon myself after starting this work, I wanted to make sure I finished what I started so that it can be an example for her to turn to whenever life seems overwhelming - work hard, discipline yourself, and surround yourself with positive people who will help you accomplish everything you start, and there will be nothing beyond your grasp.

Acknowledgments

I would like to thank Dr. Tinkham for her support, her input, her time, and her patience throughout the work done on this thesis. All of those things, along with the many extensions filed, have helped me immensely. I would also like to acknowledge Patrick McKee, with whom I set out on this research journey, and all of the friends, coworkers, and faculty that encouraged, supported, prodded, and pushed me to finish. Out of those many friends and coworkers, I would especially like to thank Jim Wise for his continuous badgering and business card threats, as well as Shahid Akhter for his peer review and constant encouragement and support.

Finally, I would like to thank my family - my parents for the love, support, and encouragement has always helped me through every phase of my life; my siblings for the distractions; and my wife Khadija and daughter Haya, both of whom came into my life after this adventure started, and with their love, understanding, patience, and encouragement helped me complete it.

All of you have helped me accomplish this goal, and I am eternally grateful to have you all in my life.

Abstract

Malik Khaleeqe Ahmed

EXPLORING ALGORITHMS TO DETERMINE SIMILAR BOARD STATES IN ARIMAA

2016

Dr. Nancy Lynn Tinkham

Master of Science in Computer Science

The game of Arimaa was invented as a challenge to the field of game-playing artificial intelligence, which had grown somewhat haughty after IBM's supercomputer Deep Blue trounced world champion Kasparov at chess. Although Arimaa is simple enough for a child to learn and can be played with an ordinary chess set, existing game-playing algorithms and techniques have had a difficult time rising up to the challenge of defeating the world's best human Arimaa players, mainly due to the game's impressive branching factor. This thesis introduces and analyzes new algorithms and techniques that attempt to recognize similar board states based on relative piece strength in a concentrated area of the board. Using this data, game-playing programs would be able to recognize patterns in order to discern tactics and moves that could lead to victory or defeat in similar situations based on prior experience.

Table of Contents

Abstract	v
List of Figures	viii
List of Tables	x
Chapter 1: Arimaa Rules.....	1
1.1 Background	1
1.2 Gameplay	2
1.3 Current Methods and Literature Review	6
1.4 Alpha Beta Search.....	6
1.5 Monte Carlo Tree Search	7
1.6 Other Unique Techniques.....	8
Chapter 2: Our Approach.....	10
2.1 Piece Data Format	11
2.2 Partial Hash Matches.....	15
2.3 Recognizing Similar Moves	17
2.4 Examples of Transitions.....	17
2.5 Recognizing Features Using Piece Data	19
2.6 Examples of Known Patterns	19
2.6.1 Frozen pieces	19
2.6.2 Immobilized pieces.....	22
2.6.3 “Capturable” pieces.	24
2.6.4 Flash kidnapping.....	25
Chapter 3: Implementation	30

Table of Contents (Continued)

3.1 Game Database.....	30
3.2 Knowledge Database.....	30
3.3 Generating Best Move.....	35
Chapter 4: Results and Analysis	38
4.1 Trends Per Radius	38
4.2 Trends During Gameplay	41
4.3 Notable Examples	46
4.4 Average Move Grouping by Radius	51
Chapter 5: Improvements and Future Work	55
5.1 Scoring Heuristics	55
5.2 Recognize Tactics	56
5.3 Opening Moves	56
5.4 Piece Data Format Adjustments.....	58
5.5 Optimal Radius Selection.....	60
5.6 Integration with Alpha-Beta Pruning	61
5.7 Performance Optimizations.....	62
Chapter 6: Conclusion.....	63
References	64
Appendix: Move Groups Per Turn Per Radius	66

List of Figures

Figure	Page
Figure 1. Arimaa pieces listed in order from strongest (elephant) to weakest (rabbit) with their movement directions and equivalent chess pieces.	3
Figure 2. Algebraic notation example.....	4
Figure 3. Position with spaces labeled with distance from the rabbit on e5	13
Figure 4. Example of two board states from the Arimaa games database where two entirely different pieces have the same piece data hash up to radius 4.....	16
Figure 5. Board state before win-in-two move.	17
Figure 6. Board state after win-in-two move.	18
Figure 7. Board state with frozen silver horse.	20
Figure 8. Example of elephant blockade.....	22
Figure 9. Example of the “flash kidnapping” strategy over multiple turns.	26
Figure 10. Directed acyclic word graph describing the flash kidnapping strategy.....	28
Figure 11. Knowledge base recursive trie tree.....	31
Figure 12. Pseudocode to persist the knowledge base's recursive trie tree.....	33
Figure 13. Percentage of wins and losses for each radius.....	39
Figure 14. Wins broken down by the side that the bot played as.	39
Figure 15. Average number of times the knowledge database was used in a game for a given radius.	40
Figure 16. "Forced" and "voluntary" fallback usage	41
Figure 17. Average number of results from the knowledge database on specific turns.....	43

List of Figures (Continued)

Figure	Page
Figure 18. Average value of moves recalled from the database per turn.	44
Figure 19. Total distribution of types of moves performed.	45
Figure 20. Case where bot_rucsmat recalled a move from its knowledge base (right) that was better than bot_hippo's alpha-beta calculated move (left).....	47
Figure 21. Case where bot_rucsmat's best recalled move (right) was worse than bot_Hippo's alpha-beta generated move (left).....	49
Figure 22. An initial board state from the Arimaa database (game 1, turn 2w).	52
Figure 23. Two initial moves that are similar up to radius 4.	53
Figure 24. Aggressive opening move sending the elephant into enemy territory.	57

List of Tables

Table	Page
Table 1. Calculated Piece Data Values.	13
Table 2. Knowledge Base learned_moves Table Schema.	34

Chapter 1

Arimaa Rules

1.1 Background

When World Chess Champion Garry Kasparov was defeated by IBM's DeepBlue in 1997, many lauded the achievement as a sign that artificial intelligence was catching up to the power of human intelligence. While this indeed was the first instance of a computer defeating one of the best human chess players in the world under standard chess tournament time rules, many people, including computer scientist Omar Syed, saw this as a victory for hardware and brute force rather than software and artificial intelligence. As part of his mission to advance the field of artificial intelligence following DeepBlue's victory, Omar Syed developed Arimaa with the specific intent of creating a game that would be simple for humans to learn but difficult for computers to play [9]. The game's impressive branching factor makes searching through the expansive game trees via standard game-tree search methods impractical without massive computational resources. For comparison, the average branching factor in chess is approximately 35 [2, 12] whereas the average branching factor in Arimaa is over 17,000 [3].

To that end, the Arimaa Challenge was designed with rules that constrain all computer participants of the challenge to specific hardware specifications and move time limits to make brute force searches impractical and forcing exploration of other heuristics and algorithms. The challenge began in 2004 and was to remain open until 2020 or until a computer challenger could defeat each of the top three human Arimaa players in the world in an official best-of-three match, with a prize of at least \$10,000 to the humans

who create the winning program [2]. Until very recently, humans trounced opposing computer challengers with relative ease, with the computer opponent often losing most of the games even against humans giving the bot a handicap [2]. Then, in 2015, David Wu's bot Sharp astounded the Arimaa world with a resounding victory against the top three human players, finally ending the challenge after eleven years [2, 13]. Although the official challenge is over, there are still many avenues of research and many opportunities to advance the field of game-playing artificial intelligence with Arimaa. The game presents challenges to traditional artificial intelligence techniques that will continue to encourage the introduction of new ideas and algorithms in the future.

1.2 Gameplay

Although Arimaa has its own rules and pieces, the game can be played with a normal chess set, as described in Figure 1. There are two sides - gold and silver - with the gold player moving first. The game starts with the gold player putting his or her pieces in any configuration on the first two rows of the board. The silver player then puts his or her pieces in any configuration on the first two rows of the board in front of the player, allowing the silver player the ability to set up pieces in a way that responds to gold's setup. After the setup phase is complete, the gold player takes his or her first turn, with turns alternating between both players.





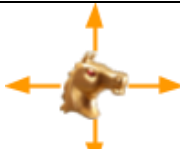







<u>Piece and Range of Movement</u>	<u>Chess Equivalent</u>
 Elephant	 King
 Camel	 Queen
 Horse	 Rook
 Dog	 Bishop
 Cat	 Knight
 Rabbit	 Pawn

Figure 1. Arimaa pieces listed in order from strongest (elephant) to weakest (rabbit) with their movement directions and equivalent chess pieces.

During a player's turn, the player can move any of his or her pieces to an adjacent empty square in any of the four cardinal directions (except rabbits, which cannot move backwards toward the player). This single space movement is called a "step" and a player is allowed up to four steps in a turn. Note that the player may move any of his or her pieces with each step, so four different pieces can be moved in a single turn. In official games and in the remainder of this thesis, these moves are recorded in algebraic notation similar to the notation used in chess, as described in Figure 2. This format describes the piece (upper-case letters representing gold pieces and lower-case letters representing silver pieces), followed by the position (e.g. b2), followed by the cardinal direction that the piece is being moved (north, south, west, east, and x for a captured piece).



Figure 2. Algebraic notation example.

Besides simply moving his or her own pieces, a player may also "push" or "pull" a weaker opposing piece into an adjacent empty square. As noted in Figure 1, each piece

in Arimaa has an inherent strength that makes it weaker or stronger than a different piece. The elephant, for example, is the strongest piece and can thus push or pull any opposing piece that is weaker than it (pieces of equal strength cannot move each other, so an elephant cannot push or pull an opposing elephant). Since rabbits are the weakest piece, they cannot push or pull any piece. A “push” is initiated by moving an opposing piece that is next to a stronger friendly piece to an empty adjacent square, and then finished by moving the friendly piece to the square that the opposing piece just vacated. Similarly, a “pull” starts by moving a friendly piece to an empty adjacent square and then finishes by moving the weaker opposing piece to the square that the friendly piece just vacated. Each step of a push or pull counts as one of the four steps a player can make in his or her turn, and the two steps comprising a push or pull must be done one after another.

Pieces can be captured when they land on one of the four trap squares on the board with no adjacent friendly pieces. As long as there is a friendly piece on an adjacent square of the trap, that trap is considered “safe” for that player’s pieces. If both players have pieces next to a trap, then the trap is safe for both players’ pieces until the defending pieces are pushed, pulled, or moved away from the trap.

Freezing pieces is another tactical mechanic in Arimaa. A piece is immediately “frozen” if it is adjacent to a stronger enemy piece and there is no friendly piece (regardless of the friendly piece’s strength) in any adjacent square. Frozen pieces cannot move until a friendly piece is in an adjacent square in any cardinal direction.

There are three ways to win a game of Arimaa. The most common goal, and the one that drives the other two win conditions, is to get a rabbit to the final row at the

opponent's side of the field. Because a rabbit's advancement is so important, another way to win the game is to capture all eight of the opponent's rabbits. And finally, a player can win if the opponent cannot move any of his or her pieces. As soon as any of these three conditions is true, the winner of the game is decided and the game is over.

1.3 Current Methods and Literature Review

All of the top-tier Arimaa programs use iterative-deepening depth-limited alpha-beta searches to find their best moves [12], which is at the core of most chess AI programs as well. As previously expressed, however, the massive branching factor of the game means the search depth is very limited and cannot be conducted exactly the same way it is conducted in chess. Instead, these programs must heavily rely on heuristic evaluation functions, depth limits, and move-ordering functions to be effective [12]. Most of the programs that have faced human challengers in the Arimaa challenge have used these augmented alpha-beta searches with varying levels of success.

1.4 Alpha Beta Search

David Wu's 2011 thesis titled "Moved Ranking and Evaluation in the Game of Arimaa" [12] describes the core implementation of bot Sharp, which uses iterative-deepening depth-limited alpha-beta search as mentioned above. An *alpha-beta search* examines each node in a game tree of possible moves depth-first and calculates the minimax value of each node [12, 13]. Because of Arimaa's immense branching factor, a depth-limit is typically imposed upon this search to prevent programs from exceeding the time limits provided for each move and so that other branches can be explored. *Iterative-deepening* increases the depth limit as promising nodes are examined and as time permits,

so that the search can be performed for a certain amount of time rather than a static depth. Additionally, the lower and upper bounds for the subtree starting at a specific node are also tracked, allowing entire subtrees to be efficiently pruned if the algorithm determines further searching within the subtree will not provide a better result than the nodes already examined. This *alpha-beta pruning* relies heavily on the evaluation function used to value each node, which is where there is the most variance and exploration in Arimaa bots.

In both his 2011 thesis and his 2015 paper discussing the final version of his championship-winning bot, Wu stresses the importance of developing a good move ordering and function dependent on positional features, among other improvements [12, 13]. If “good” moves are examined first in an alpha-beta search, then the search will complete much quicker, because the upper and lower bounds that are set during the search will prune more nodes quickly. The positional features that these functions rely on involve Arimaa-specific concepts, such as trap control and goal threats, and tactical configurations such as blockades, hostages, and frames [12]. The innovation that differentiates Wu’s bot Sharp from other alpha-beta based bots is that its move ordering function was a result of using machine learning to train a Bradley-Terry model over thousands of expert Arimaa games in order to better “predict” expert moves [12,13].

1.5 Monte Carlo Tree Search

Although augmented alpha-beta based search algorithms adapted from chess have shown to be the most promising, other unique approaches have also been explored with interesting results, as discussed in Kozelek (2009) [6] and Jakl (2011) [4]. One such

approach is Monte Carlo Tree Search, which has found some measure of success in the bots playing the game of Go [4, 6]. Monte Carlo methods have been useful in applications where calculating the full range of possibilities would be unreasonable, and since Go and Arimaa have extremely large branching factors [4], they are prime candidates to benefit from their usage.

A Monte Carlo tree search (MCTS) simulates pseudo-randomly playing through the game tree up to a certain point in the tree [4, 6]. Each playout is recorded, and the final results are analyzed in order to choose the moves that result in the most positive playouts. Arimaa-specific domain knowledge is imbued in each step of the playout and its evaluation in order to more efficiently choose the best results. For example, Kozelek gave pushing/pulling pieces, capturing pieces, moving elephants, and moving rabbits later in the game higher precedence [4]. Although both Jakl and Kozelek found that their MCTS-based bots were able to play reasonably well with weak evaluation functions, their bots could not match the level of the top alpha-beta-based bots.

1.6 Other Unique Techniques

Although alpha-beta searching is the most promising algorithm discussed so far, there have been many techniques explored that would enhance or supplement the typical alpha-beta search. For example, Gerhard Trippen in his 2009 paper [10] implemented a bot that enhances the evaluation function to recognize, detect, and perform plans such as trap defense, trap attacks, and multi-turn strategies like flash kidnapping using a directed acyclic word graph. Trippen's bot only implemented flash-kidnapping, but stresses the ability to make the bot stronger by adding additional strategies to the graph [10]. The

flash-kidnapping strategy revolves around “kidnapping” and capturing a piece before the opponent is able to effectively respond without weakening its position. As such, the graph is flexible enough to not require specific pieces that would be captured, but is focused on certain positions on the board.

Another paper by Choksi, et al (2013) explores the impact of game phase on evaluation [1]. Game phase is something that human players of the game may consider (for example, shifting strategic focus to getting a rabbit to the other side of the board after many critical pieces are captured or incapacitated), but is not very clearly defined in a game like Arimaa. Choksi et al attempt to break games into three phases based on the advancement and number of pieces on the board for each player and augment the board evaluation and move ordering functions in an alpha-beta bot based off of the open-sourced bot Fairy. They found that taking game phase into account in the board evaluation function did not improve the bot’s win rate (it won less than $\frac{3}{8}$ of its games), but did have a slightly positive effect when used in the move ordering function [1].

Chapter 2

Our Approach

In general, the human ability to improve over time can be partially attributed to our ability to analyze and learn from our past experiences. The idiom “practice makes perfect” encourages improvement over simple and mindless repetition by learning what works and what fails. Our approach attempts to capture this for Arimaa by generalizing the board state so that patterns within the board can be recognized and reconciled against prior experiences. As more games are played (and either won or lost), we can attempt to draw a correlation between the moves made in the game and whether it led to a victory or a loss. Over time, as more games are played, we would expect to see that moves that led to victory more often than not can be considered “good moves” whereas moves that led to losses more often than not can be considered “bad moves.”

While this sounds simple, the complexity of moves in Arimaa makes this relatively non-trivial. It is quite rare that the *exact* same move will be made in any two games of Arimaa (barring initial moves from identical opening setups). It is also quite possible that determining the effectiveness of a move is highly dependent upon the situation in which it is used; a move that led to a victory in one game can lead to a critical loss in another. Therefore, the current board state and the resulting state must be taken into consideration when gauging such moves.

With these basic assumptions, we developed a format for generalizing the state of the board (or a specific region of a board state) in a way that does not require the exact

same pieces to be on the board in the exact same spots. The latter is where our approach is different from existing approaches that we know of; transposition tables and Zobrist hashing is used often in chess and Arimaa alpha-beta searches to make sure that the exact same board state is not encountered again [4, 6], but our approach can detect states that are relatively similar but not exact. For each piece on the board, we capture what we shall henceforth call the piece’s *piece data* in a single “hash” string.

2.1 Piece Data Format

This piece data string by itself does nothing more than capture the unbiased information about a piece’s position on the board, including distances of stronger and weaker friends, stronger and weaker enemies, as well as traps and boundaries. Note that the specific mechanics of this game - the ability to freeze and unfreeze pieces by moving certain pieces next to them, for example - drove the decisions of what information to capture. This information is captured for each possible *radius* (more accurately, a taxicab distance) on the board. The largest possible distance between any two pieces on the board is 14 (i.e. the taxicab distance between a1 and h8), so the aforementioned information is captured in 14 *chunks*, where each *chunk* represents the data at that distance. Each chunk can be seen as a 8-character hexadecimal string describing how many of the following elements can be found at that specific distance:

1	0	0	1	1	2	0	1
Number of stronger friends	Number of equal friends	Number of weaker friends	Number of stronger enemies	Number of equal enemies	Number of weaker enemies	Number of boundaries at this distance in a straight line	Number of traps

A full piece data hash would contain 14 of these chunks with each subsequent chunk representing the data at that radius (so the first chunk describes radius 1, the next describes radius 2, and so on). Grouping this data by radius allows us the ability to compare a localized area of the board rather than the full board in order to spot common patterns that only involve a few pieces in a certain area of the board. Additionally, data for radius 0 is prepended to the final hash in order to capture data about the spot it is currently occupying and the piece itself:

1	2	0
Owned by current player	On Number of Boundaries	On Trap

As an example, consider the silver rabbit at e5 in the position shown in Figure 3. Table 1 describes calculates the data needed to generate the piece data hash for that rabbit



Figure 3. Position with spaces labeled with distance from the rabbit on e5

Table 1

Calculated Piece Data Values

Radius	Friends			Enemies			Boundaries	Traps
	stronger	equal	weaker	stronger	equal	weaker		
1	1	0	0	0	0	0	0	0
2	1	0	0	1	1	0	0	1
3	2	1	0	0	0	0	2	2
4	0	0	0	1	0	0	2	1
5	0	1	0	0	2	0	0	0
6	0	1	0	1	1	0	0	0
7	0	1	0	0	1	0	0	0
8	0	0	0	0	1	0	0	0

This table calculates the data that make up the piece data hash for the rabbit at e5 in the win-in-two state shown in Figure 3.

Table 1 only calculates up to radius 8 because there is no interesting data for this piece beyond that radius due to the rabbit's position on the board, as shown in Figure 3. Although the complete hash would contain all the data up to radius 14, the full hash is almost never used during actual gameplay as we will explore later. Also note that the "boundaries" value only counts the boundaries that can be optimally reached at the specified radius. For example, looking at the board above where each spot is labelled with the number of steps it would take, the rabbit at e5 can be thought to be three to seven steps away from the eastern boundary since all of the spaces in column H are part of the boundary. If we included those spaces, we can see that the number of boundaries for radii 4 - 7 in the table would all equal 4, since the rabbit can reach any border on the board within 4 - 7 steps. Tracking this in the piece data hash is cumbersome and of little worth, since eventually we would see 4s in the later radii of the hash. Instead, these calculations only consider the minimal distance since the piece data for two pieces in similar spots with respect to the boundaries would still be equal. This approach also simplifies and optimizes implementation by only needing to look at the piece's current row and column to determine boundary distances.

The full piece data hash would consist of the three header values described above followed by a concatenation of the values in Table 1 in radius order. If the board state shown is silver's turn, the header value would be "100," whereas it would be "000" if it was gold's turn. Putting all of this data together, the final piece data hash for re5 would be the following when it is silver's turn to move (note that spaces have been added between radii segments for clarity, but are not needed in the final hash string):

```
100 10000000 100110001 21000022 00010021 01002000 01011000
01001000 00001000 00000000 00000000 00000000 00000000
00000000 00000000
```

2.2 Partial Hash Matches

More often than not, however, a partial hash string will prove to be more useful. Using the full hash limits the ability to compare two states since the hashes will only match when the positions are exact rotations of each other. A partial hash string will truncate this full hash to the desired radius to focus on a specific region of interest on the board. For example, since pieces can only move a maximum of 4 steps on the board, we could truncate the hash generated above for re5 to 100 10000000 100110001 21000022 00010021, allowing comparison between other 4-step regions in other board states.

As an example, consider the highlighted pieces in the two board states in Figure 4.

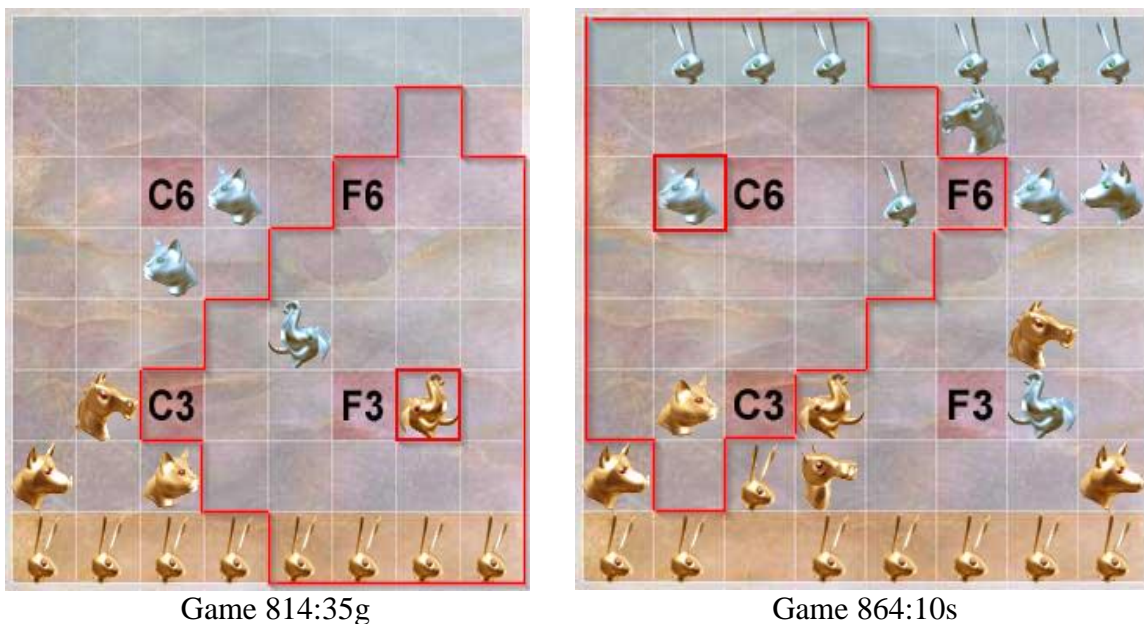


Figure 4. Example of two board states from the Arimaa games database where two entirely different pieces have the same piece data hash up to radius 4.

The hash for the highlighted pieces at radius 4 (outlined by the red borders) in both images would be 100 00000011 00100010 00201000 00100002. Even though the pieces themselves and the board positions are different, this hash value shows us that both the gold elephant and the silver cat are in very similar positions relative to other pieces and spaces on the board within a 4-space radius. Because a piece can only take up to a maximum of 4 steps in a given turn, this makes movement for one of those pieces an inherently interesting move to consider for the other piece. This ability to generalize a board state in terms of relative distances and strengths is the core value gained by calculating these piece data hashes.

2.3 Recognizing Similar Moves

Piece data hashes on their own only describe the pieces on a board in a static board state. In order to extend this concept to find “similar moves,” we must generalize the change that these pieces go through in terms of the piece data being collected in the hash, which we call a move transition. A move transition consists of a collection of piece data hash pairs, each pair representing a piece transition that describes the piece data before and after the move for each piece that was moved as an effect of the move performed for that turn.

2.4 Examples of Transitions

Consider the board state in Figure 5 with silver’s turn to move.

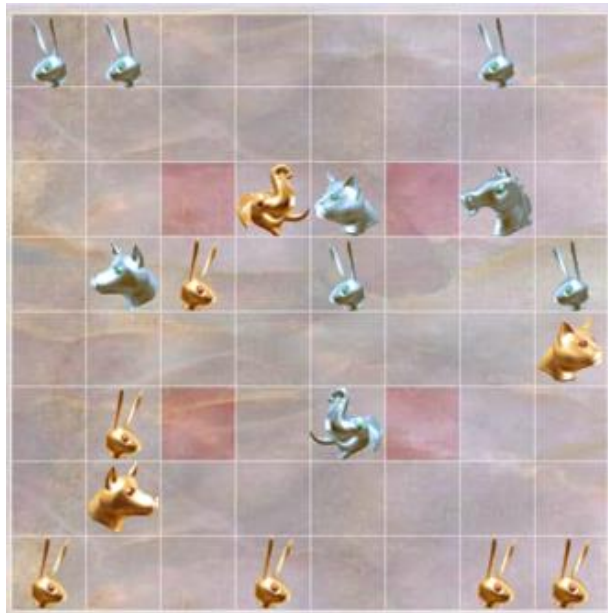


Figure 5. Board state before win-in-two move.

The silver rabbit on e5 would have a piece data hash (up to radius 4 for simplicity) of 10010000000100110012100002200010021. If the rabbit moves to f2 via the move re5s re4e rf4s rf3s, then the rabbit's piece data hash given the resultant board state below would be 10000000011100010100000200000020002.

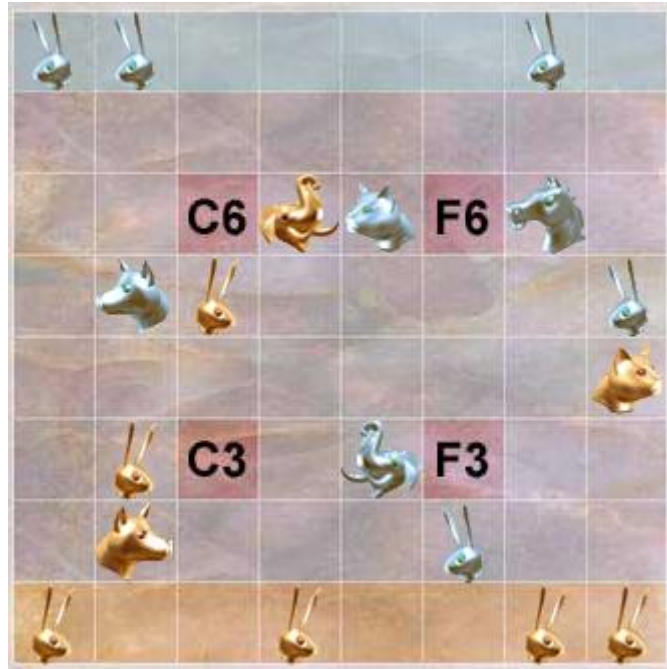


Figure 6. Board state after win-in-two move.

This transition can be stored in a map data structure mapping the piece data before the move to the piece data after the move in order to describe how a piece's piece data changed as a result of a given move. In this case, the map would conceptually look like this if we only stored radius-5 hash strings in the map:

```
{ '10010000000100110012100002200010021': '10000000011100010100000200000020002' }
```

This map represents a “*move transition*” and contains one “*piece transition*.” In our implementation, only pieces moved during the turn are stored in the map. If the silver elephant on e3 was also moved, for example, then its “before” and “after” piece data would also be stored in the map. If a piece was captured by the end of the move, the “null” piece data value is used for the ending value in the map, which is a piece data hash with all values set to 0 in our implementation.

2.5 Recognizing Features Using Piece Data

One key benefit of this piece data format is the ability to recognize tactical patterns, or features, on a given board in a query-able format. The components of the hash as described above intentionally capture the kind of information that can fuel informed decisions about a piece’s state on the board in a sequence of bytes. The following section outlines a few of the features that can be easily recognized from these hashes (or variations of these hashes) using a format akin to bitmasks to express the general pattern. None of these features have been implemented in `bot_rucsmat`, but are explained here for use in future work and to illustrate the possible uses of this format.

2.6 Examples of Known Patterns

2.6.1 Frozen pieces. As described in 1.2 above, a core tactical mechanic in Arimaa is the ability to “freeze” a piece by having a stronger enemy piece immediately next to it with no friendly pieces in adjacent squares. For example, the golden elephant on e4 has frozen the silver horse on d4 in the following board (Game ID 864, Turn 9g):

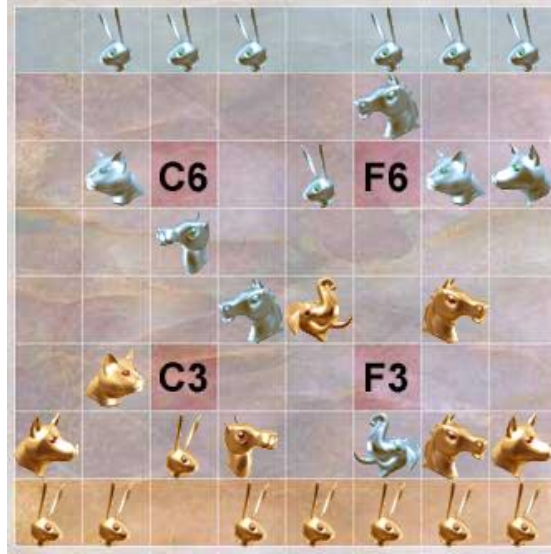


Figure 7. Board state with frozen silver horse.

If we express this logic in terms of the data captured in the piece data hashes described earlier, we can say that a piece is “frozen” if one or more stronger enemies are exactly 1 square away and exactly 0 friendly pieces are exactly 1 square away. The piece data hash up to radius 1 for the silver horse in this example would be 000 00010000 (expressing that there are no friendly pieces and one stronger enemy piece exactly one square away) and the hash for the gold elephant would be 100 00000100 (expressing that there are no friendly pieces and one weaker enemy piece one square away), which adequately expresses the aforementioned condition for freezing a piece.

This, however, is a relatively trivial case where there are no other pieces or factors in the radius-1 hash. To generalize this, we would need to account for the other variable factors in that hash (traps, boundaries, additional pieces, etc). If we were to replace the portions of the hash that do not play a part in determining whether or not a given piece is

frozen with an X and set the required portions of the hash to the appropriate values, this template hash (excluding the header for now) would look something like this for radius 1:

0	0	0	1	X	X	X	X
Number of stronger friends	Number of equal friends	Number of weaker friends	Number of stronger enemies	Number of equal enemies	Number of weaker enemies	Number of boundaries at this distance in a straight line	Number of traps

Given the template above, we know that if the piece data hash at radius 1 (excluding the header) is of the format 0001XXXX, then the piece this hash is describing is frozen. Depending on the format that these hashes are stored, normal pattern-matching techniques (such as regular expressions if stored as a string or bitmasks if stored as raw bytes) can be used to determine whether or not a piece is frozen.

Note that we have ignored the three hexadecimal digits in the header for the above example. The header for this feature can be ignored unless we are specifically looking for the current player's pieces that are frozen or the opponent's pieces that are frozen. In those cases, the "owned by current player" digit can be examined separate from the IS_FROZEN matching above to determine whether or not the frozen piece is owned by the current player.

2.6.2 Immobilized pieces. Pieces can be immobilized without necessarily being frozen. Since the elephant is the strongest piece in the game, there is no way for an elephant to be frozen by a stronger piece. However, as the gold elephant in the board below [14] illustrates, elephants can be immobilized with great effort:

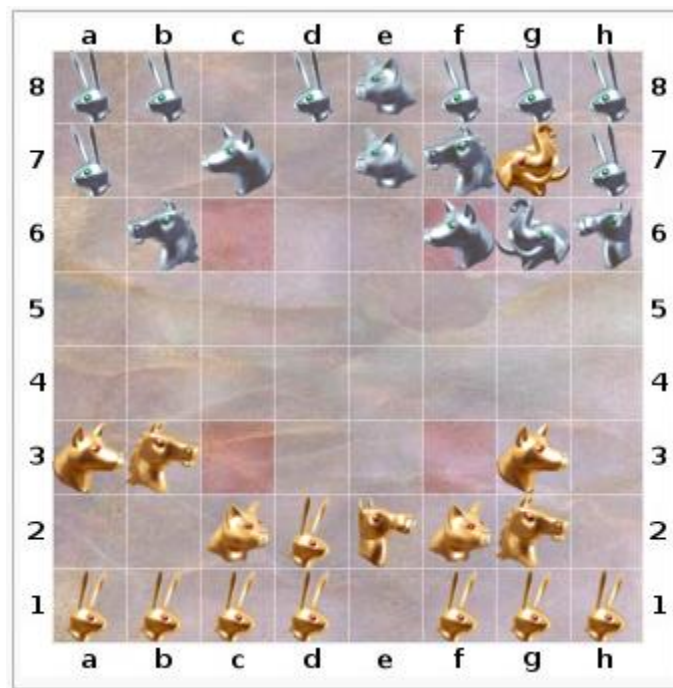


Figure 8. Example of elephant blockade.

The gold elephant in this position can make no legal move and is thus immobilized.¹ This can also be expressed in piece data hashes, albeit with more complications than the

¹ The silver player has created an *elephant blockade*, a strategy discussed in more detail in [14].

frozen pattern described above. There are many cases to be considered that all would lead to a piece being immobilized like this:

1. Have 4 stronger or equal strength pieces at radius 1,
2. Have 4 weaker pieces at radius 1, and 8 pieces of any strength at radius 2 (so those weaker pieces cannot be pushed or pulled),
3. Have two borders within 1-2 spaces and fill the remaining spaces at radii 1-2 in any combination of the above so that the piece cannot push/pull its way out,
4. Any combination of the factors above such that the piece is unable to move a single step in any direction.

Although each of these cases can be expressed in terms of different piece data hashes, the number of possibilities are numerous. There is also the possibility that a friendly piece can attempt to rescue an immobilized piece by pushing and pulling the surrounding pieces out of the way across multiple turns, which would require more computation and analysis. We would need to decide whether we want to consider these additional cases and the complexity they add in order to detect “total” immobilization, or whether simply knowing that a piece is currently immobilized is sufficient. Although immobilizing pieces so that the opponent cannot make any legal moves is a valid win condition in Arimaa, the chance that such a win condition would arise in high-level play is very rare.

2.6.3 “Capturable” pieces. Capturing pieces is an important core mechanic in Arimaa, so being able to detect capture opportunities can be useful. However, this pattern is a bit more involved than detecting freezing pieces because of the interaction with other pieces that may or may not be defending a trap. In the simplest case, having a piece 1 space away from a trap is an indicator that the piece could potentially be captured. Similar to the way we look at the radius-1 hash for frozen pieces, we can check that the Number of Traps portion of the hash in the radius-1 segment equals 1.

X	X	X	X	X	X	X	1
Number of stronger friends	Number of equal friends	Number of weaker friends	Number of stronger enemies	Number of equal enemies	Number of weaker enemies	Number of boundaries at this distance in a straight line	Number of traps

However, if the trap is safe (i.e. there is another friendly piece adjacent to the trap), then relying only on the total number of traps like this hash can be misleading, since stepping onto the trap would not result in a capture. In this case, the piece data format described earlier may not suffice. Because a maximum of two traps may be the same distance away from a piece on any given spot on the board, it may be better to split the “Number of Traps” field into “safe traps” and “unsafe traps,” where a “safe trap” is one where there is at least one other friendly piece besides the current piece next to the trap, and an “unsafe trap” is the opposite. Then, simply looking at the “unsafe traps” field in the hash would easily provide insight into the pieces that are in danger of being captured at the cost of additional computation per piece.

2.6.4 Flash kidnapping. Although these piece data hashes only attempt to express a generalized view of the relationships on a static board state, they can be used in tandem with other techniques that benefit from the generalized format to provide more functionality. For example, consider the following figure from Trippen (2009) [10] detailing the steps taken to perform the multi-turn “flash-kidnapping” strategy (note only the relevant portion of the board is displayed in each step):

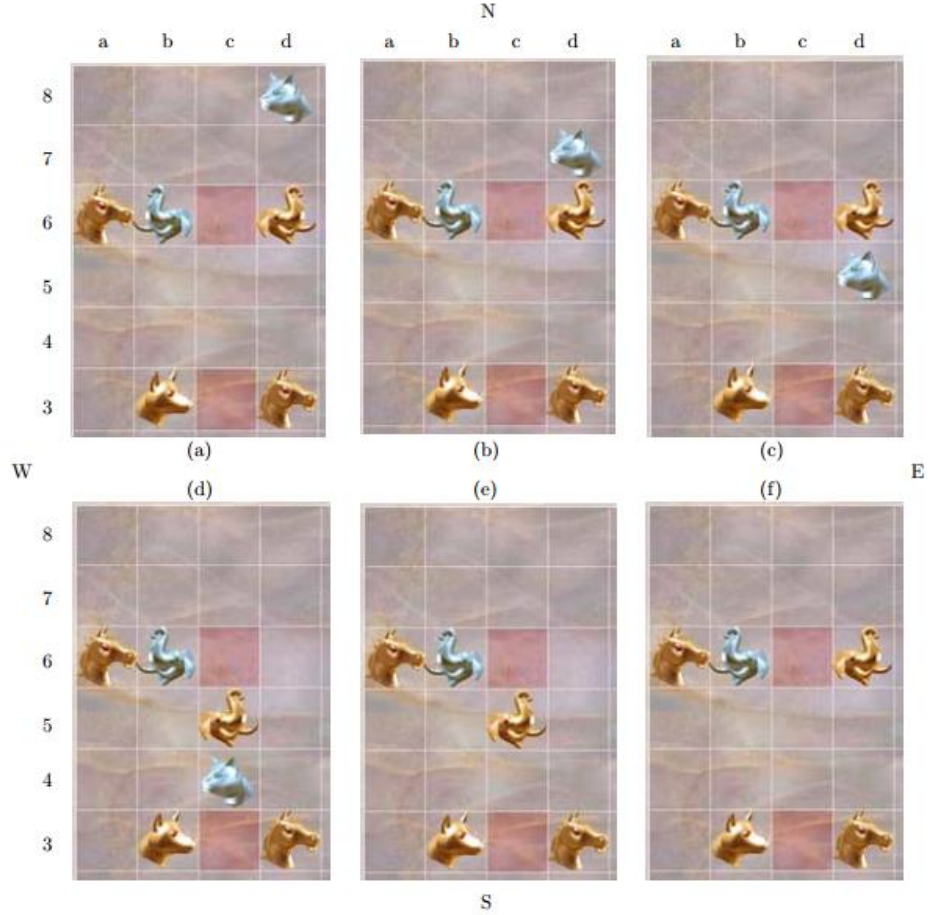


Figure 9. Example of the “flash kidnapping” strategy over multiple turns.

Starting at (a), we see that the silver elephant has the gold horse frozen along the boundary of the board. This is a “horse hostage” strategy that an Arimaa player may employ, since the horse cannot be freed easily without focusing the effort of many gold pieces across many turns in an attempt to free it. In the turns after (a), the gold elephant plans to use the hostage situation to its advantage by capturing other pieces while the silver elephant is occupied (a strategy called “flash kidnapping” in [10]). Steps (b) - (f) show the states at the end of each of gold’s turns as it ideally tries to implement its flash

kidnapping strategy. In (b), the gold elephant has pulled the silver cat towards its position (Ed6n Ed7s cd8s), and in (c) - (e) pushes and pulls the cat until it is captured in the c3² trap, all the while keeping the c6 trap safe (by keeping the gold elephant adjacent to the trap at the end of each turn, the silver elephant cannot capture its horse hostage in that trap). Finally, in (f), the gold elephant returns to its starting position so that it can perform the same strategy again if any piece comes within push/pull distance of it on the board.

The basic premise here is that the gold player (in this case, a bot that is implementing this strategy) is banking on the fact that opposing silver bot will not expend resources to save the silver cat or obstruct its path to the unprotected trap. This assumption relies on the fact that the bot would not be able to perform an alpha-beta search deep enough to look three to four moves in the future, and by the time it realizes gold's true intent, it would be too late to efficiently stop its demise and would not be inclined to free a strong gold horse just to save a weak silver cat.

Trippen also explains in his paper [10] how a directed acyclic word graph could be used to detect the patterns that would trigger the use or detection of this strategy:

² The moves to reach each state would be as follows:

(c) - Ed6e cd7s cd6s Ee6w

(d) - cd5w Ed6s cc5s Ed5w

(e) - cc4s cc3x Ec5s Ec4n

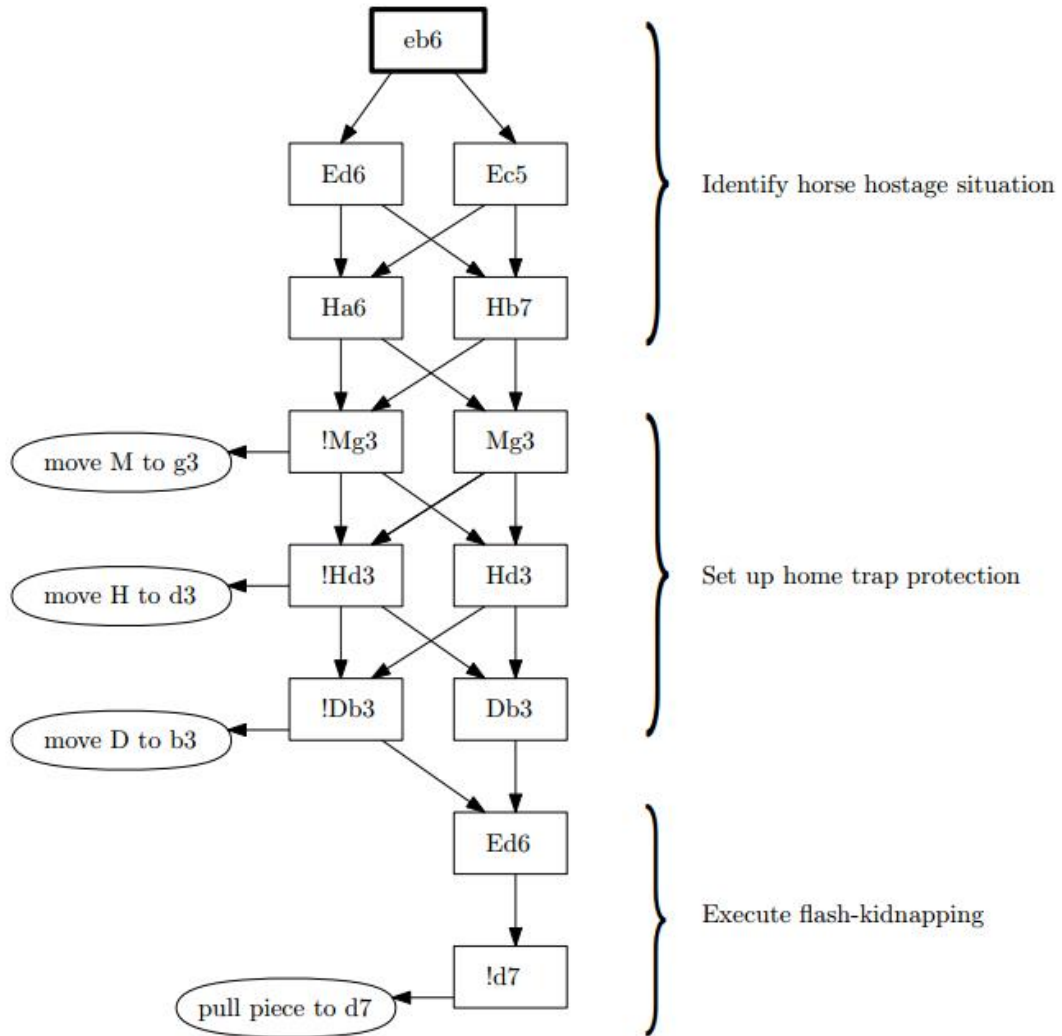


Figure 10. Directed acyclic word graph describing the flash kidnapping strategy.

Once the bot has identified that a horse hostage situation exists (as is the case with the gold elephant and silver horse above), the home trap (at c3 in the above example) must be protected with additional pieces so that the hostage cannot be captured there without removing those additional pieces first. Finally, the flash-kidnapping strategy can be executed as the above example describes with the sequence of moves in (a) - (f).

Note that each phase in the graph above specifically spells out exact positions to recognize and the type of pieces that can be a part of this strategy. Generalizing this using the piece data format allows the ability to express the phases and actions in this graph in a succinct way that can still be used, even if the elephant, horse, and dog in the graph are instead a camel, dog, and cat, respectively. Additionally, by using distances to the boundaries and the traps, the same technique can be applied in any orientation on the board, instead of a very specific case with specific positions on the board. This flexibility allows us the ability to specify strategies and patterns in very general terms and have them recognized and applied during gameplay without exact matches.

Chapter 3

Implementation

In order to test the usefulness of piece data hashes, we attempted to create a bot that would save these piece data hashes in a database and attempt to “recall” similar moves previously performed to guide the selection of the bot’s moves. This chapter outlines our implementation of the bot, affectionately named bot_rucsmat³.

3.1 Game Database

In order for the bot to have a decent baseline to start with of “good moves”, we used the game data that is freely available for download on the official Arimaa website. This game data came with information about the game and its outcome, the ranking of both players, and the list moves performed during the game. All of this data was collected into an SQLite database as described in [8] that could be used to replay games and have bot_rucsmat learn what moves “good players” (as defined by a higher ranking) made that lead to victory. This baseline would make up the initial core of the bot’s knowledge database.

3.2 Knowledge Database

The “knowledge database” (or “knowledge base,” as it will be called often throughout this paper) is where bot_rucsmat stores transition data that it can reference when deciding to make a move. Conceptually, the transition hashes are stored in a

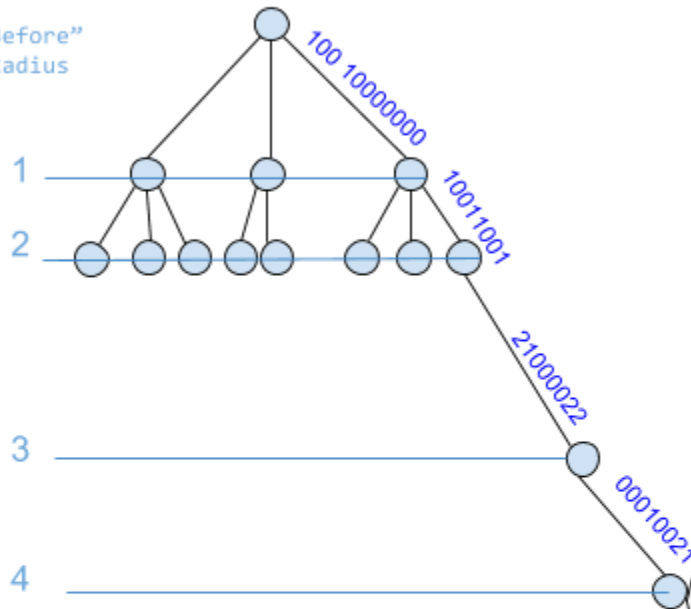
³ Rowan University Computer Science – McKee, Ahmed, Tinkham.

recursive trie tree as portrayed in Figure 11, which allows us to easily and efficiently scope queries to a specific radius of interest.

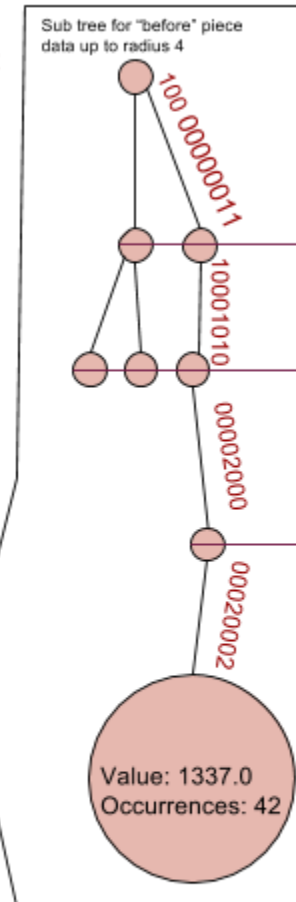
Transition:

- Before: 100 10000000 10011001 21000022 00010021
- After: 100 00000011 10001010 00002000 00020002

"Before"
Radius



Given a transition like the one above (which is truncated to radius 4 for brevity), each segment of the before-hash links the nodes in the outer trie tree. Each node contains a trie tree that similarly uses the after-hash, with each node containing the calculated value and the number of times the transition has occurred up to the before and after radius for the node.



"After"
Radius

Figure 11. Knowledge base recursive trie tree.

A given node at depth d_1 represents a partial piece data hash (the before_hash) up to radius d_1 (the parents of the node make up the earlier segments of the string). Note that the header is included as part of the hash for radius 1 for efficiency, since the minimum radius we would want to examine would be radius 1. Each of those nodes stores a trie tree that contains all of the possible resulting states of the transition (the after_hashes) learned for the given before_hash. Each node at depth d_2 in the subtree stores a value for the transition up to that radius, which is calculated based on the following formula:

$$\frac{sMR}{t(14 - d_1)(14 - d_2)}$$

- M is a multiplier applied to the move based on factors of the game. For our implementation, M is simply 1 if the player making the move won the game or -1 if the player lost.
- R is the player's ranking on the Arimaa site.
- t is the total number of piece transitions in the move. A move may consist of one or more pieces moved, so a move can be thought of as a piece's *transition* from one state to another and dividing by the number of transitions in the move gives each transition equal weight toward the move's total value. For example, if 4 pieces were moved on the board, then there would be 4 piece transitions in that move, and we are giving each of those 4 piece transitions equal worth in the final value.
- d_1 is the desired radius of the before-hash (the node's depth in the outer trie tree).

- d_2 is the desired radius of the after-hash (the node's depth in the inner trie tree).
- s is a “scaling factor” applied to the move based on the stage in the game when the move was performed (i.e. moves performed later in the game have more weight than moves performed earlier in the game). For our implementation, this scaling factor was simply the turn number divided by the total number of turns in the game, giving each move a fractional weight based on how close it brought the player to the game's end.
- The constant 14 is the maximum distance possible on the board between the two furthest positions on the board (i.e. opposite corners like a1 and h8).

Our implementation uses a relational database to store this tree using materialized paths [5, 11] for both hashes in the transition. The pseudocode in Figure 12 describes how these entries are added into the database, whose schema is described in *Table 2*.

```
# Depth-first processing of the outer trie tree.
def persist(root_node, before_hash = ''):
    for node in root_node.children:
        materialized_before_hash = before_hash + node.hash_str
        process_learned_moves(materialized_before_hash, node.learned_moves.root_node)
        persist(node, materialized_before_hash)

# Depth-first processing of the inner trie tree.
def process_learned_moves(materialized_before_hash, root_node, after_hash = ''):
    for node in root_node.children:
        materialized_after_hash = after_hash + node.hash_str
        $db.insert({
            'before_hash': materialized_before_hash,
            'after_hash': materialized_after_hash,
            'value': node.value,
            'occurrences': node.occurrences,
            'normalized_value': node.value / node.occurrences
        })
        process_learned_moves(materialized_before_hash, node, materialized_after_hash)
```

Figure 12. Pseudocode to persist the knowledge base's recursive trie tree.

Table 2

Knowledge Base learned_moves Table Schema

Field Name	SQLite3 Field Type	Description
before_hash	VARCHAR(255) NOT NULL	The piece data hash up to radius d_1 for a piece before the move was made.
after_hash	VARCHAR(255) NOT NULL	The piece data hash up to radius d_2 for a piece after the move was made.
value	REAL NOT NULL	The total value for this before-after transition. If this transition is seen again, the value of that transition will be added to this stored value.
occurrences	BIGINT NOT NULL	The number of times this transition has been seen.
normalized_value	REAL NOT NULL	The “average” value of this before-after transition, calculated as value/occurrences.

Note. Each row in this table represents a node in the inner trie trees described in Figure 11.

As games are played, existing entries will be updated positively or negatively depending on the outcome of the match or new entries will be created if one for the specific before-after combination does not exist. Because this data is stored as a trie tree where each level of the tree represents a specific radius, it is possible that only a few portions of the hash would warrant a new entry.

In order to jumpstart bot_rucsmat's education with an appropriate baseline of relatively "expert" moves, we had bot_rucsmat run through a subset of 1000 games from the games database. The Arimaa site also has a collection of "win-in-two" puzzles, which consists of a list of game states from the game database where making a specific move will guarantee the player's victory in his or her following turn. We had bot_rucsmat also learn all of these states while assigning an artificially high value to the "win-in-two" transition described as the solution to each puzzle, which should allow bot_rucsmat the ability to recognize a similar win-in-two condition if one were ever to present itself.

3.3 Generating Best Move

With a collection of learned transitions available to bot_rucsmat, the next and arguably most important task was to use this data to actually pick the "best" move during gameplay. In order to do this, bot_rucsmat first generates ALL possible unique moves for a given board state (which can be done rather efficiently despite the large number of moves possible) and generates a transition hash for each move. We then must search the knowledge database for entries that match the transition up to a specified radius, and order the moves based on the `normalized_value` stored for that transition. Note that `normalized_value` is used so that the moves that have consistently performed well over

time are chosen over the moves that simply have been encountered often. In the latter case, the value field would have a significantly larger amount stored as the “value” for that transition, even if the average value for both moves being considered are equal.

The ordered collection of transitions can further be filtered by a desired range for the move’s value. For example, our implementation only considers moves that have a value greater than 0.0, which simply means only moves that have had a net positive impact over the games observed thus far are considered and moves that have led to losses more often than victories are not considered. Other implementations may consider moves within a specific positive range or may even include some negative moves in case one of those moves performs better in a specific instance.

The “best” move would be the move with the maximum value from the collection of transitions searched. Our implementation considers all moves with a value within a delta of 1.0 to be equal in order to explore further options. If there is a tie for the maximum value (or multiple moves are within 1.0 of the maximum value), then additional heuristics are employed to break the tie and pick the move that works best for the current board state. Our prototype implements a simple heuristic that computes a score favoring states where the player has control of a greater area of the board, has fewer pieces captured or at risk of being captured, has fewer pieces frozen, and has elephants towards the center of the board, brings our rabbits closer to the goal boundary and opposing rabbits further from their goal boundary.

In the case that the knowledge database query returns no results or returns moves that have a value outside of the acceptable range, bot_rucsmat relies on its fallback to

generate a move. Our implementation falls back on an alpha-beta bot provided on the Arimaa site called bot_Hippo to generate a move in such cases.

Note that our current implementation does not search further into the game tree. Over time, this approach should reduce the need to perform a deep search into the game tree since the proper heuristics used to score transitions as they are learned should take the move's effect on the game into account.

Chapter 4

Results and Analysis

In an attempt to compare the effect of limiting knowledge database matches to specific radii, we had bot_rucsmat play at least one hundred games against bot_Hippo where it matched transition data from the knowledge database up to a specific radius. Because bot_rucsmat falls back on bot_Hippo's move generator if a move cannot be generated from the knowledge base, we can better compare the impact that using the knowledge base has on a bot's win rate. In this chapter, we will present the data collected and the conclusions that we derived from these results.

4.1 Trends Per Radius

Figure 13 shows the percentage of wins and losses that bot_rucsmat had for each radius over a hundred or more games, with Figure 14 breaking down the wins by color. For each radius, bot_rucsmat played at least 100 games with the before_radius and after_radius set to the same value. In general, we noticed that choosing a mid-range radius usually resulted in better performance. Also, bot_rucsmat's won marginally more while playing as gold, but the difference is not drastic enough to be significant. A radius of 6 resulting in an interesting advantage with an approximate 54% win rate over bot_Hippo, while a radius of 5 came close with a win rate of approximately 45%.

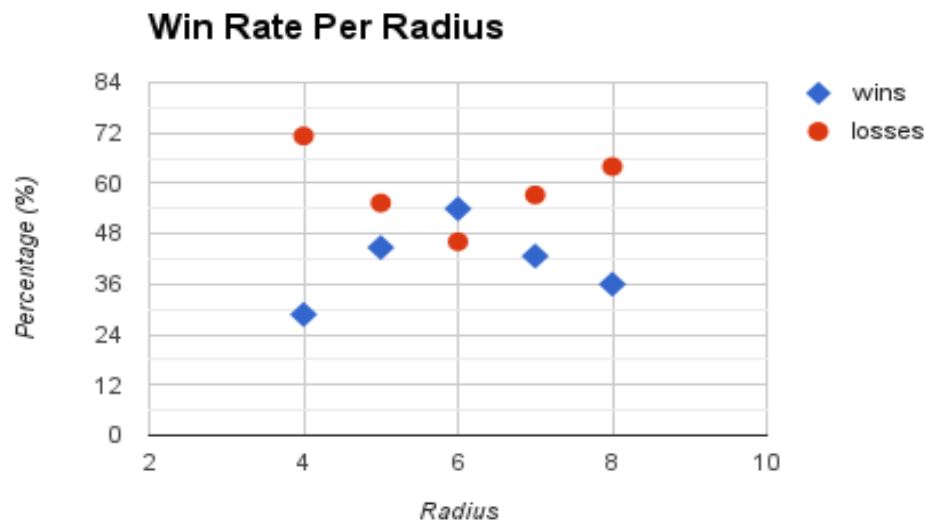


Figure 13. Percentage of wins and losses for each radius.

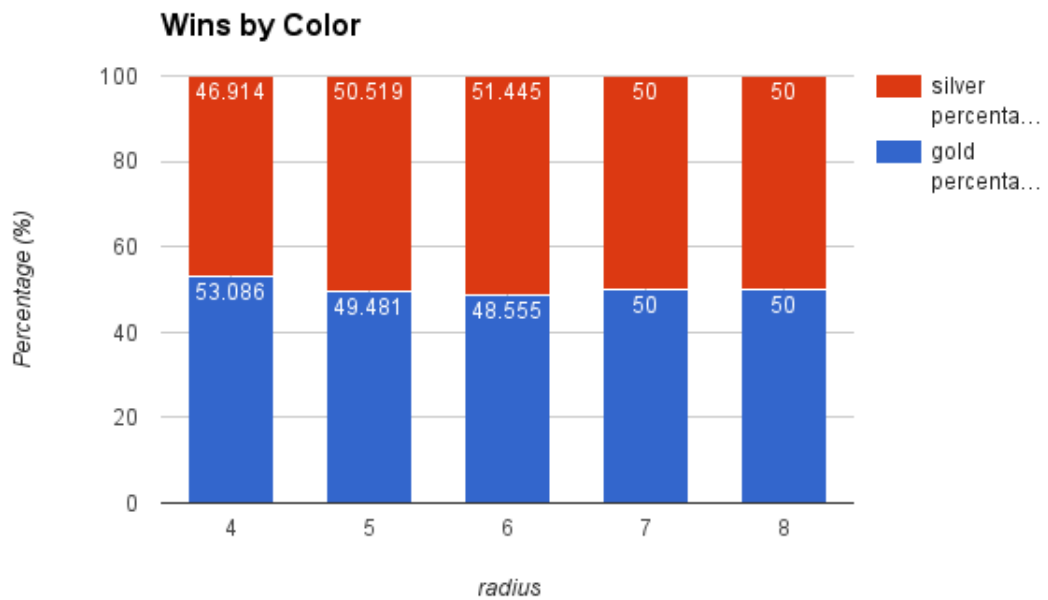


Figure 14. Wins broken down by the side that the bot played as.

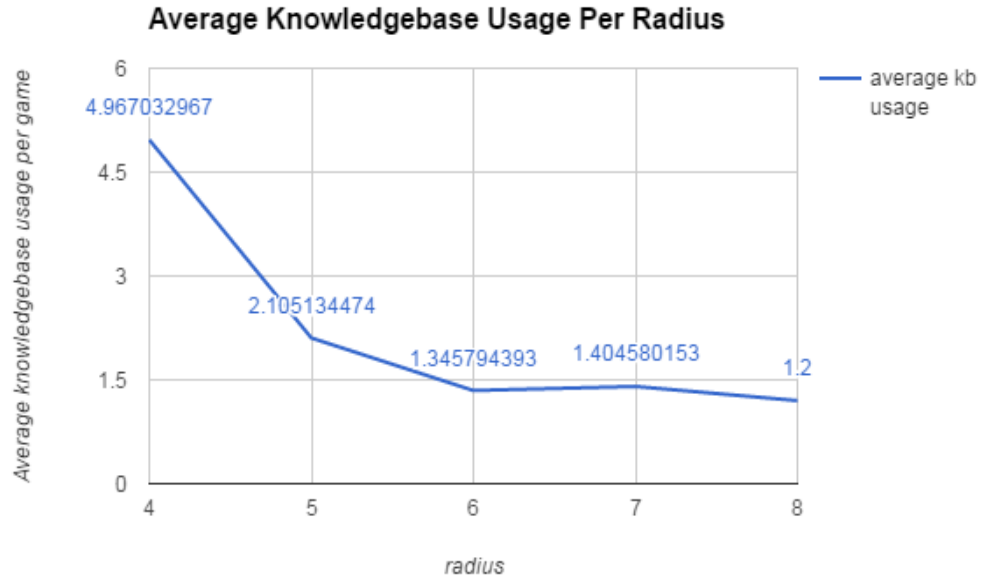


Figure 15. Average number of times the knowledge database was used in a game for a given radius.

Additionally, Figure 15 shows that knowledge database usage is much lower for higher radii. While it is expected for higher radii to return fewer results since it inherently requires more of the board to match, the lower win rate for low radii seems to result from the weakness in our tie-breaking heuristic, as we will see in the data in the next few sections.

This data led us to our conclusion that a radius of 6 is the ideal radius to use when matching moves in a knowledge database storing piece data transitions in the format described in this paper. Radii lower than 6 did not contain enough relevant board information to make proper decisions among the many moves matched. Radii higher than

6 performed similarly, but for a different reason; rather than not having enough information, the matches would be too stringent, resulting in even fewer matches.

4.2 Trends During Gameplay

As hinted by Figure 15 above, actual usage of the knowledge base tended to be very infrequent for most of the radii we tested. Figure 16 plots how often the fallback method was used, and shows that the knowledge base was used most often during the first few turns of the game and almost not at all for the rest of the game, with occasional usage in the last few turns of the game.

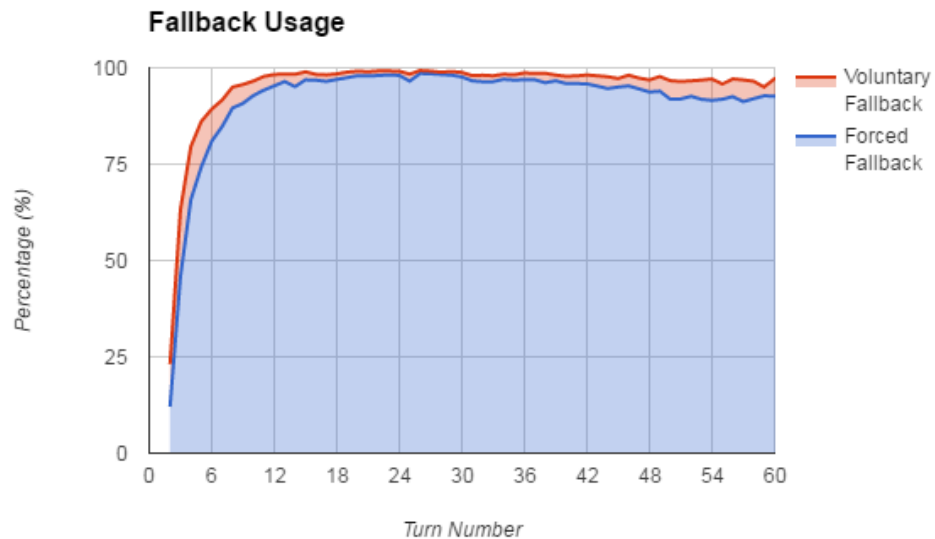


Figure 16. "Forced" and "voluntary" fallback usage

A “forced” fallback here represents the percentage of moves on this turn across all games where there were no results from the knowledge base and the fallback algorithm had to be used. We “voluntarily” use the fallback when the value of the knowledge base move(s) are below an acceptable threshold value. Combining the “forced” and “voluntary” percentages in this graph yields the total percentage of fallback usage on that turn across all games.

The strong knowledge base usage at the start of the game and occasional usage at the end of the game seems to suggest that our approach is most useful when the board state is most “stable” in terms of piece interaction. During the initial turns of the game, especially the first turn after setup, pieces are in relatively predictable positions on the board and generally grouped closely together. As gameplay continues and tactics are employed, pieces are maneuvered around the board in a way that diverges very quickly for different games even if they had similar openings. As a result, many different pieces will be interacting with different pieces within a localized radius range, resulting in very few similar states in the middle portion of the game. Finally, In the late stages of the game, many pieces have been captured and one or both players are nearing their goals, resulting in similar states for localized areas of the board when there are comparatively few pieces left in play. This logically follows from the piece data format, which attempts to describe the interaction between pieces within a specific region of the board by noting the distances and strengths of the pieces within that region. Figure 17 and Figure 18 show the average number of moves recalled from the knowledge base and the average value of

those moves over the course of a game, which also show that there is a bit more consistency towards the start and end of the games than in the middle.

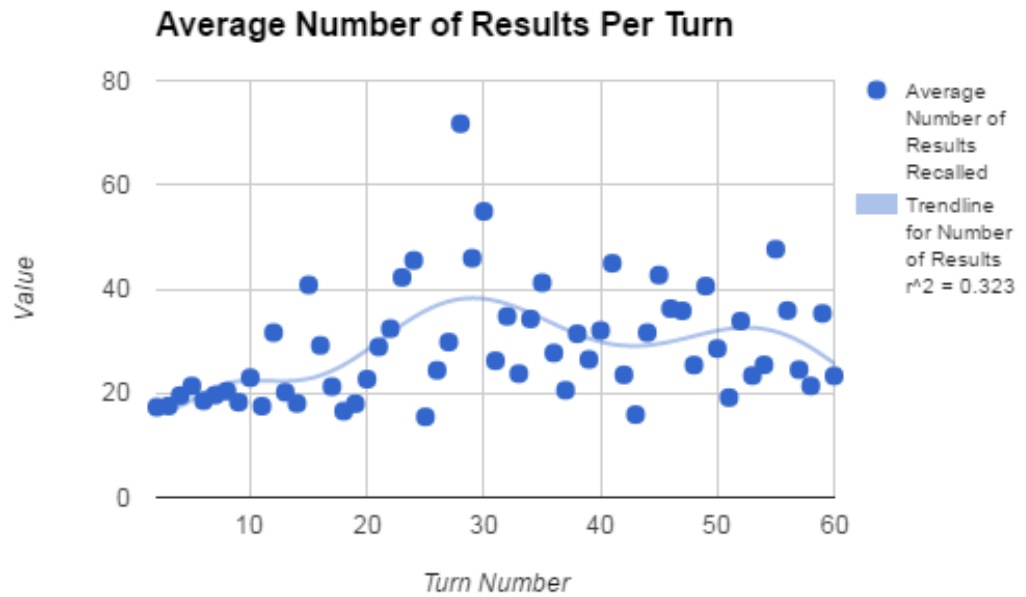


Figure 17. Average number of results from the knowledge database on specific turns.

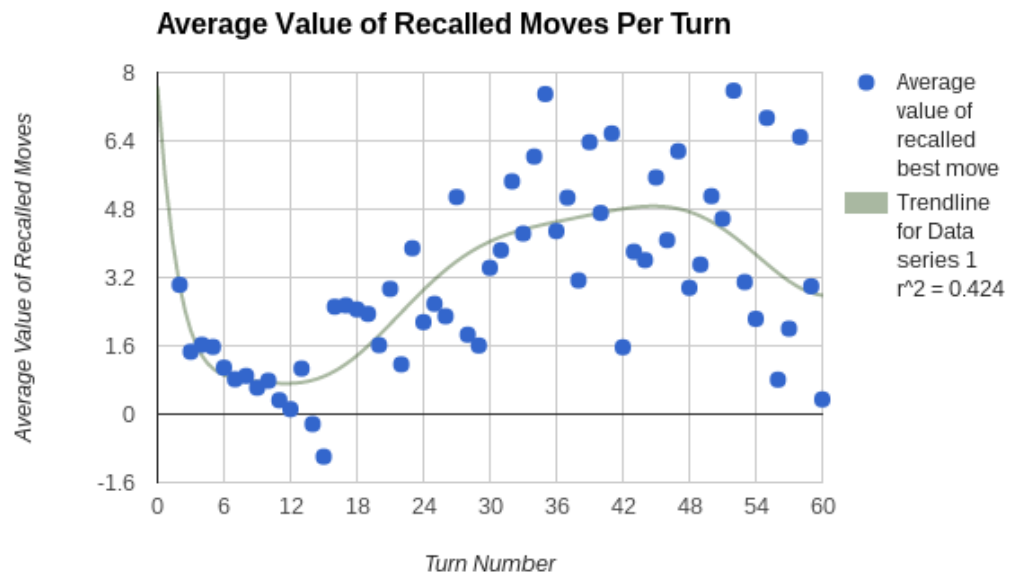


Figure 18. Average value of moves recalled from the database per turn.

Figure 19 summarizes the overall distribution of the types of moves that bot_rucsmat performed and how often the knowledge base was used versus the alpha-beta fallback.

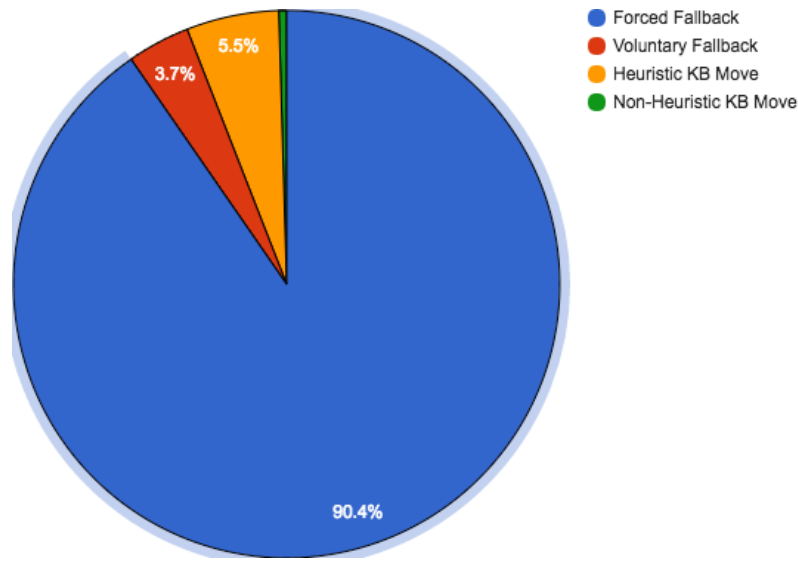


Figure 19. Total distribution of types of moves performed.

Unfortunately, the knowledge base was very rarely used. About 90.4% of the time, bot_rucsmat could not find an appropriate matching move in the knowledge base, and another 3.7% of the time the moves returned would have such a low value that we would resort to using the fallback anyway in an attempt to make a better move. Even though bot_rucsmat was using bot_Hippo's natural alpha-beta move generator 94% of the time, it still was not able to achieve a consistent 50-50 win-rate against bot_Hippo due to our weak heuristic tie-breaker, which was used 5.5% of the time versus the approximately 0.5% of the time we found a clear winner in the knowledge base.

4.3 Notable Examples

As mentioned earlier, the knowledge base found the most usage during the initial turns of the game. As more games are played and similar moves are encountered and their outcomes observed, the more confident bot_rucsmat would be that it is making the best move. For example, in the setup depicted in Figure 20, bot_rucsmat chose a move that seemed better than bot_Hippo's potential move.

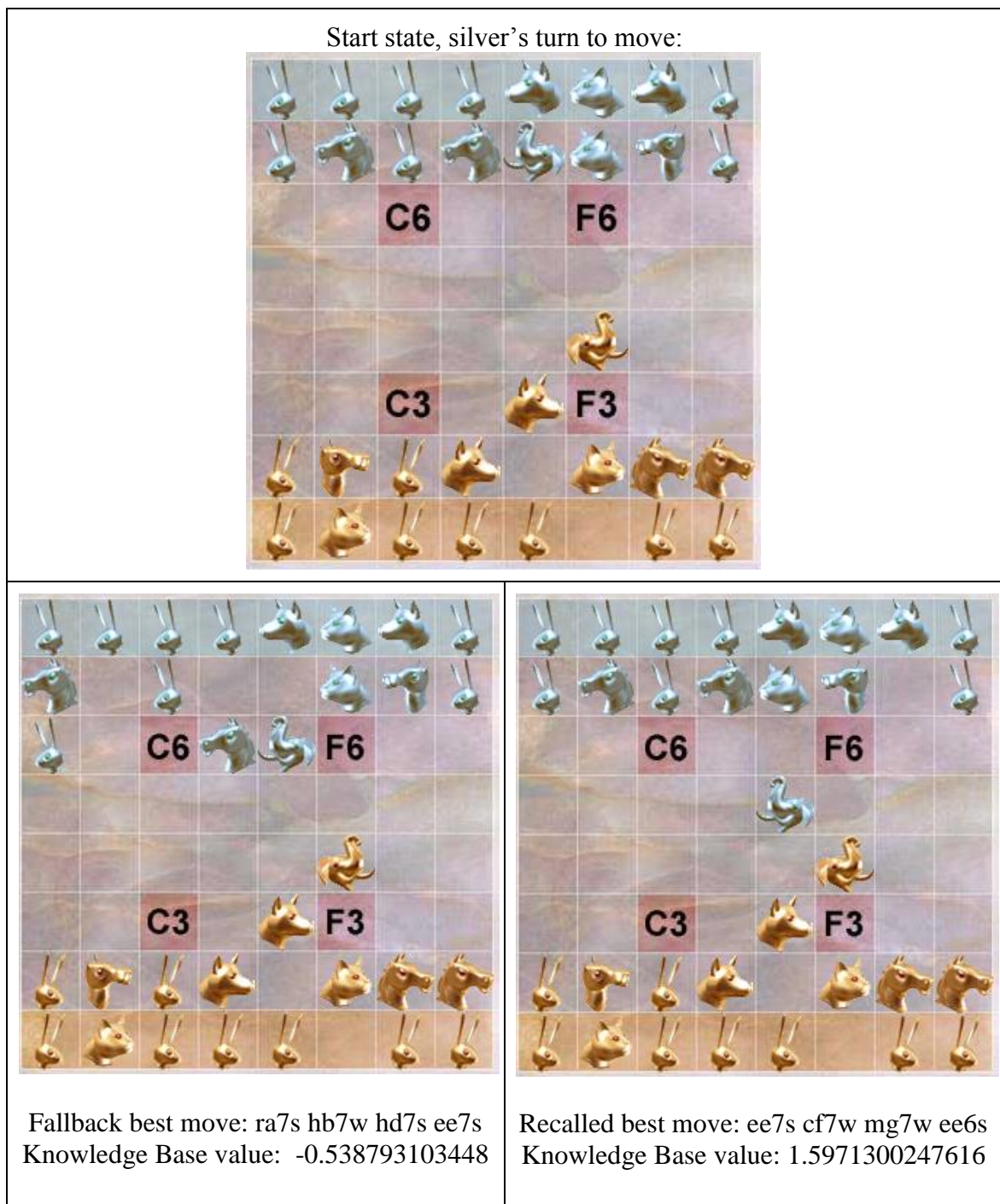
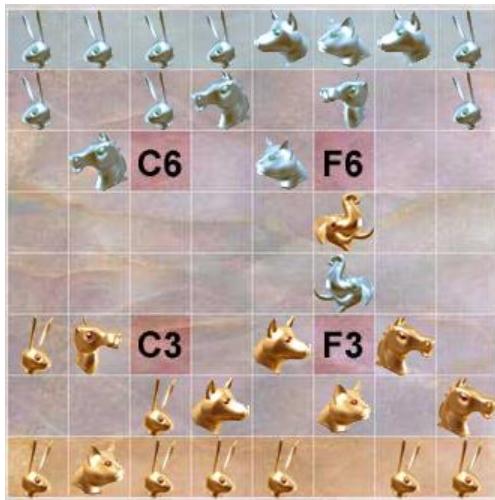
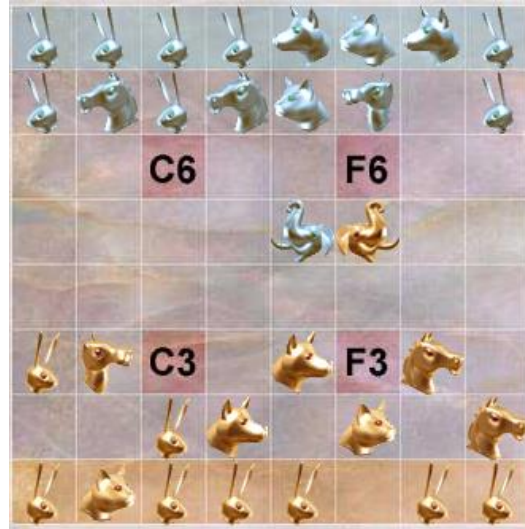


Figure 20. Case where bot_rucsmat recalled a move from its knowledge base (right) that was better than bot_hippo's alpha-beta calculated move (left).

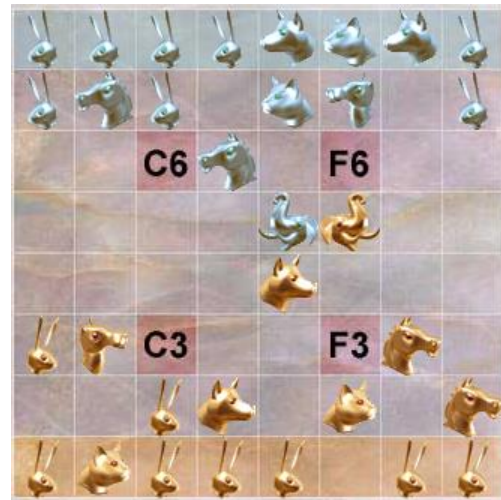
Here we see the primary objective of bot_rucsmat at work - bot_Hippo's alpha-beta search yielded ra7s hb7w hd7s ee7s as the best move given the position on the board. While this may seem to be the best move in terms of bot_Hippo's evaluation function, bot_rucsmat has the benefit of hundreds of prior games and similar examples to form the conclusion that the move may not be the best move, since most of the games that this move (or more importantly, a move *similar* to this move) has been used in has resulted in a loss. This prior history, along with the fact that moves are weighted based on when during the span of the game the move was employed, imbues a valuable forward-looking worth into the knowledge base's score. Instead of representing the value of a move given only the current board state and then recursively exploring the values for possible future states in the game tree as alpha-beta would do, this knowledge base score essentially takes full playouts of games using similar moves into account, providing a score that already takes possible future states into account.

After this move, the opposing bot's move resulted in the state depicted in Figure 21, where bot_rucsmat decided to rely on its fallback method rather than a move found in its knowledge base due to the move's low score.

Start state, silver's turn to move:



Fallback best move: ee5s ee4e hb7s ce7s
Knowledge Base value: N/A



Recalled best move: hd7s ee5s ee4n De3n
Knowledge Base value: -0.82708333333

Figure 21. Case where bot_rucsmat's best recalled move (right) was worse than bot_Hippo's alpha-beta generated move (left).

In this case, we see that the move that bot_Hippo would have chosen results in silver's powerful control over three of the four traps on the board, and the silver elephant's position makes gold's plays more awkward, since none of the weaker gold pieces near the F3 trap that the silver elephant is guarding can move past the elephant without coordinated effort to unfreeze the pieces, which wastes steps and slows gold down. In contrast, the move that bot_rucsmat found in the knowledge base is relatively weak; it manages to pull and freeze the gold dog, but it will not be able to capture the dog next turn and the gold elephant can easily unfreeze the dog in one step and still has 3 more steps in the turn to fortify its position, essentially undoing silver's efforts the previous turn. The important fact to note here is that bot_rucsmat did not do any of this analysis; it simply saw that moves like the one it was considering have seen more failures than successes in the past, so it can discard that move as a weak play based on previous experience, without examining why it was a weak play.

Much to our delight, bot_rucsmat went on to win this particular game. Whether it was due to these two smart decisions at the start of the game (the knowledge base returned no results for the rest of the game, so it was essentially bot_Hippo playing against itself for the remaining turns), is subjective, but these kinds of intelligent judgement calls are the kind of behavior we hoped to see when using the knowledge base.

We did, however, notice a negative aspect of partial data hash matches. During its first turn, bot_rucsmat actually found many best moves (a few hundred, in fact) from the knowledge base that had a value close to, albeit lesser than, the value of the move that it eventually chose. While choices are not a bad thing and our tie-breaking heuristics chose

the best move out of the lot, we noticed that many of the moves it was choosing from involved a sacrifice of its own pieces. In this particular example, the majority of the matching moves called for the sacrifice of the rabbit on c7. This is a trend we noticed in many games and one that warrants stronger and smarter heuristics around the move chosen. The reason that these seemingly detrimental moves were even considered is due to the fact that piece data hashes count the number of pieces at certain distances. When there is one piece more at a specific radius than expected, the hashes will be different. So, although the original move learned may not have performed such a barbaric sacrifice, the fact that the number and relative strength of pieces at each radius match makes this move seem similar to the move in the knowledge base. In this particular case, the number of friendly pieces may have been 1 more than needed to match a potential move from the knowledge base, so the move that kills its own rabbit results in a state that more closely resembles the state represented in the knowledge base. This is a limitation of the piece data format as currently defined, which we attempt to mitigate using a heuristic tie-breaker to weed out detrimental moves. Enhancements to the format itself in the future may be an interesting prospect for further research.

4.4 Average Move Grouping by Radius

An additional application of the piece data hashes that we explored was its ability to reduce the state space of the possible moves in a given turn. As mentioned previously, the branching factor in Arimaa is quite large - some of the games bot_rucsmat examined had over 30,000 possible unique moves in one turn for a given board state - which is why

any optimizations that improve the ordering and evaluation of these moves typically results in better bots.

The appendix contains a table that lays out the total number of unique board states possible at each turn of a game from the Arimaa database, as well as how many groups can be made by grouping together states that result in the same hash for each piece after the move. For example, Figure 22 below depicts the board state at the start of turn 2w in game id 1 of the game database.

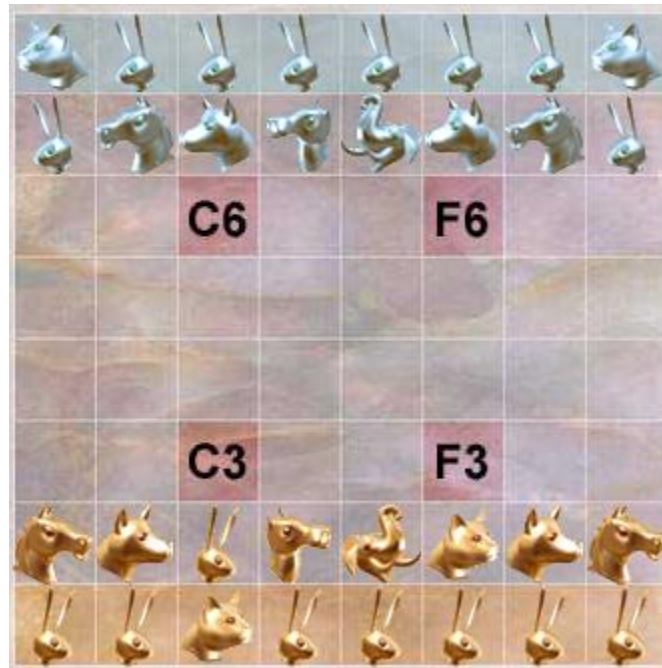


Figure 22. An initial board state from the Arimaa database (game 1, turn 2w).

We can generate all of the possible moves for this state and examine the piece data hashes of the pieces after the move has been completed. We can then group cases where the resulting hashes are equal (meaning the resulting board states are “similar”) for each radius. At radius 4, for example, the piece data transition as described in 2.3 for both Db2n Db3w Ha2e Hh2n and Dg2n Dg3e Hh2w Ha2n would look like this:

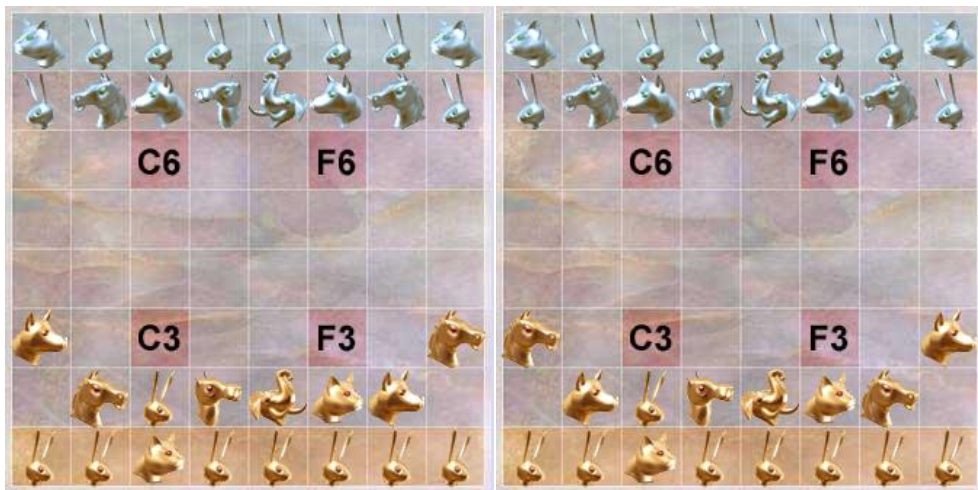


Figure 23. Two initial moves that are similar up to radius 4.

Transitions:

```
{
1001020002010200001101000000020000001110002: 1100000000010100011002000001010010010110112,
1100020001000200000101000011010000000200100: 1100000000000200011002000001010010010110112
}
```


Note that the two moves result in states that are essentially mirrored horizontally, which further supports our claim that the two states are similar. Both of these moves can thus be grouped together at radius 4, despite being unique moves in their own right.

The table in the appendix lists the total number of unique moves, as well as the number of these “move groups” that can be made at each radius. We can see that overall, the number of move groups is not significantly lower than the total number of moves for many turns, with the majority of turns and radii seeing less than 1% reduction from the total number of moves. In a similar vein as the data presented above, the most reductions were observed in the first few turns of the game, with extremely low radii seeing the largest reductions (as high as 58% for radius 1 on gold’s third turn of the game) and most radii above 3 resulting in 0 reductions for most turns beyond the first few.

Using this data, we can determine that our current piece data format will not provide a significant benefit when attempting to reduce the search space of the game tree. However, Appendix A also contains data where we consider the results piece data for all pieces on the board, not just the ones moved in the transition format above. We can see from the data that there are some cases where one may reduce more than the other or perform better at certain radii, although the general trend of being strongest at the start of the game and generally weaker in the late game still hold true. This is only one variation that we explored, and we believe that further research with tweaked variations of the format and how we determine “similar” board states may yield more interesting results.

Chapter 5

Improvements and Future Work

5.1 Scoring Heuristics

One major area of improvement for bot_rucsmat are the various scoring heuristics used to rank and choose moves. As described in Chapter 3, our implementation currently employs very basic heuristics for scoring transitions saved in the knowledge database and for choosing a move in the case of ties for “best move.” These heuristics are essential to the bot’s performance, since this is what the bot relies on to make its decisions.

When scoring transitions, for example, our implementation only takes into account the number of total moves in the game, the player’s rank, and the number of transitions in the move. Because we set the rankings for both bots to be the same value, bot_rucsmat began to stagnate over time, either choosing inconsequential moves simply because they happened to repeat often and be part of winning games a bit more than average. Improving this to include game features such as frozen pieces, material advantage, captures, and goal threats into account would add a dimension of game knowledge to these simple heuristics that may enable the bot to learn quicker and select better moves. Determining which factors are important to consider and how to appropriately incorporate those factors into an accurate move score is an area of exploration that would greatly benefit this bot.

5.2 Recognize Tactics

As described in Chapter 2.2, piece data hashes allow us the ability to recognize common patterns and tactics. However, these are all tactics that must be taught to the bot and hard-coded in order to be of use, which was not done for this initial iteration of bot_rucsmat. Through machine learning techniques, we may be able to recognize patterns in the piece data hashes, especially patterns that occur often and lead to beneficial board states, that may help uncover additional tactical techniques that human players may not have discovered yet. Deep analysis to find recurring patterns and automated learning of the effects that learned moves have on the board state would significantly improve the bot's ability to improve over time without additional human intervention.

5.3 Opening Moves

As discussed in Chapter 4.2, the knowledge base often yielded interesting results during the first few turns of the game. Leveraging this, an analysis of opening moves using piece data hashes to group “similar” openings together would be an interesting area of focus. Unlike chess, Arimaa has no database of opening moves or endgame positions due to the sheer number of states that are possible - each player may set up his or her pieces in 64,864,800 different ways, which means an entire initial board has 4.2 1015 possible initial states [2]. However, generalizing the board state using piece data hashes can simplify this greatly.

For example, consider a common opening move that moves the gold player's elephant four spaces forward towards the silver player's ranks.

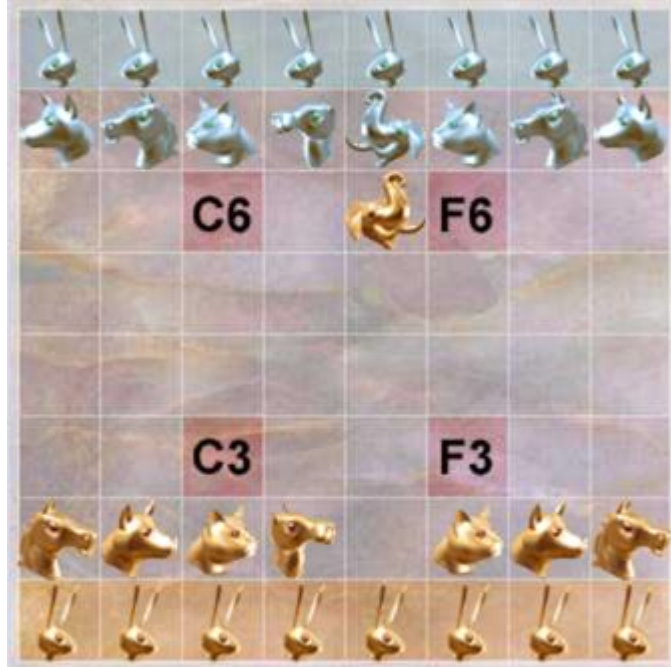


Figure 24. Aggressive opening move sending the elephant into enemy territory.

Because the piece data hashes only compare relative strengths of pieces, the piece data hash will be different only based on the opponent elephant's position. No matter where gold's pieces are positioned in the initial ranks, the distances to all of those weaker pieces will remain the same. There are 16 spaces that the enemy elephant can start on, but the distance to any of those spots will be between 1 - 6. In other words, there are a total of 6 unique transition hashes that could represent this opening move across all 4.2×10^{15} possible initial board states. Of course, this is a trivially simple case because of the unique power that elephants hold in this game, but this example powerfully illustrates the potential for analyzing starting states using this piece data format.

5.4 Piece Data Format Adjustments

There are a number of variations in the piece data format that we have considered whose ramifications on the bot's abilities would be an interesting avenue of research to pursue. Earlier variations of this format did not include counts for pieces of equal strength and just grouped equal pieces in with weaker pieces. This was then changed to the current format when we realized that it would be difficult to recognize "frozen" pieces and pushable pieces without differentiating pieces of equal strength, which would be immune to both of those mechanics. The header portion of the hash was also added later in our experimentation when we found that `bot_rucsmat` could not tell the difference between a move that would be good for itself or for its opponent without it, but it further limited the number of matches we would find in the knowledge base.

There are still many other possible variations that can be considered. For example, by including the distance to boundaries and traps in the hash, partial hash matches for pieces that are near those spaces are limited to only cases near a trap or boundary. In such cases, we see that one state is essentially a mirror of the other in a different corner of the board. If we exclude the boundary and trap counts from the hash completely, a matching piece could be anywhere else on the board as long as the other dependent pieces that make up the hash are the same distance away. Whether or not this is desirable is something that would require additional experimentation and research.

Another variation along the same lines that can be explored involves additional details about the boundaries or traps. For example, the trap count could be split between safe and dangerous traps for the current piece, where safe is defined as traps that the

piece could safely step on without being captured and dangerous is defined as traps that would get the piece captured if stepped on. This exposes more details about the game state and may lead to more accurate correlations to the moves performed on such pieces, since a trap's state may influence a player's decision to move a certain piece. Pieces could be moved towards a dangerous trap to make it safe or moved away from it to avoid capture. Additionally, boundaries may be split between "home," "enemy," and "side" boundaries. Although such a change will make matches even more rigid than the example described earlier in this section, it may also expose a stronger correlation between the piece data and the type of move made.

The current implementation for describing moves involves tracking the change in piece data for each piece involved in a move. Initially, however, this was implemented by generating a hash for the entire board before and after the move, grouping each piece's piece data together by radius in the following format:

$$p_{1_1}p_{2_1}p_{3_1} \dots p_{1_2}p_{2_2}p_{3_2} \dots p_{1_{14}}p_{2_{14}}p_{3_{14}} \dots$$

where p_{x_y} is the piece data hash for an arbitrary piece x at radius y . Although this has the potential to find more accurate matches where every piece on the board has the same piece data up to a certain radius, in practice we found that the matches found this way were too strict when pieces are within the range of the radius chosen. This approach may not have been fruitful for us, but other ways of expressing and storing transition data for all pieces on the board and assigning proper heuristic values to that data may help improve the bot's ability to find appropriate matches.

Another potential variation might entail “classifying” transitions based on core game mechanics. David Wu mentions many move ordering and move evaluation features in his 2011 paper titled “Move Ranking and Evaluation in the Game of Arimaa.” Those features could be explicitly added to transitions as they are being learned by the bot in order to find patterns in the type of moves players tend to make given a board state. For example, we may find that 80% of the winning moves made in similar situations for a given board state caused a capture or secured a trap or brought a rabbit in range of the goal. Given this information, the bot can potentially score moves that cause those board features higher than those that do not, even if we do not find an exact match in our knowledge database. This would also help avoid situations like the one described in Chapter 4.3, where matches from the knowledge base often included moves that would end up in killing the bot’s own pieces, simply to make the hashes match. If the format were adjusted to track things like captures, these moves may no longer be considered if the original learned move did not also include such a sacrifice. This is a highly valuable capability that would be ideal for future enhancements and further research.

5.5 Optimal Radius Selection

The optimal radius for finding matches is another area of improvement for bot_rucsmat. In Chapter 4, we looked at how often bot_rucsmat won matches when searching based on a specific radius. Our simple implementation sets static values for the before_radius and after_radius used in a search, but an improved implementation may choose to vary the radii based on the optimal values. A radius of 4, for example, attempts to account for all of the pieces in the range of spaces that a single piece can reach within

a turn. Increasing that radius for a before or after hash will take pieces further away into account, allowing only more stringent and more limited matches. Decreasing the radius would allow for more matches since it would only take very close pieces into account and would place a heavier burden on scoring heuristics to make up for the loss in “precision” caused by the reduced range. Experimenting with variable radii is also an area of research that may yield interesting results upon further exploration.

5.6 Integration with Alpha-Beta Pruning

One of the most interesting and potentially promising applications of piece data hashes and transitions is its potential to improve alpha-beta pruning by using the value stored in the knowledge database as cutoff values. As described in Chapter 1.3, most top-tier Arimaa programs utilize some variant of alpha-beta pruning to search the game tree for the optimal move. Along with the existing enhancements that are made today specific to Arimaa, these searching algorithms can leverage the data learned to better order the nodes searched and prune more nodes sooner.

This goal of better pruning and move ordering was the focus of David Wu’s enhancements to bot Sharp, and those enhancements helped the bot succeed as well as it did [13]. Many of the move ordering and move ranking features that bot Sharp recognizes [12] can also be expressed in terms of piece data hashes and transitions, some of which have been described earlier in Chapter 2.3. The fact that the knowledge base score takes into account future states based on prior games and that similar moves in one level of the game tree can be grouped together results in significant trimming of the game state space that the alpha-beta search needs to consider, with theoretically better lower- and upper-

bounds on each node. Even if the majority of the moves being explored in one level of the search have no hits in the knowledge base, the ability to group similar moves together reduces the number of nodes to search, and the few moves that *do* have knowledge base hits will provide a baseline score to compare against the alpha-beta bounds during its search. We see this as the most promising improvement to explore, as Wu's bot Sharp has shown how important move ordering and efficient pruning can be for a powerful Arimaa bot [13].

5.7 Performance Optimizations

Lastly, our current implementation is highly taxing in terms of performance. For ease of use and debugging, the hashes are indexed and stored in the SQLite database as strings instead of integers, and every radius combination is stored. These factors contribute to an extremely large database (after about 1000 games, the database would be close to 14GB in size), and querying the database for an average of 1400 moves per turn can take quite a while. Our implementation distributes the queries across 8 parallel processes, which allows searches to complete in 30 to 35 seconds on average. Optimizing the storage and retrieval of data from the knowledge database would greatly improve the practicality of this approach and allow more time in official games to explore other options or search further into the game tree.

Chapter 6

Conclusion

Besides fine-tuned optimizations to the alpha-beta implementation, David Wu’s bot Sharp’s unique innovations came from recognizing Arimaa-specific tactical patterns and logic that would improve the move ordering and evaluation done while performing a search [13]. In a game like Arimaa, the human ability to grasp and enhance abstract concepts becomes vital to success. Our research and the “piece data” concept offer the blueprints for such functionality in bots by granting the ability to use core game concepts to generalize board states and claim similarity between various states. We implemented, as proof-of-concept, a bot that was designed to use this functionality to “learn” from the games it plays, generalizing states it sees in a way that can be referenced in future games with the hopes of removing a common bot weakness of falling for the same tricks repeatedly. While this idea needs more research and fine-tuning to be useful in practice, we have shown that we can indeed find comparable board states using this method, which we have not previously seen in our research, and hope that this can be expanded and used as building blocks for further investigation into this unique area of study.

References

- [1] Choksi, V., Ebrahim-Zadeh, N., & Mohan, V. (2013). Leveraging Game Phase in Arimaa. Stanford University, Stanford, CA. Retrieved October 22, 2016, from http://www.vivekchoksi.com/assets/game_phase_arimaa.pdf.
- [2] Computer Arimaa. (2016, September 14). In Wikipedia, The Free Encyclopedia. Retrieved October 22, 2016, from https://en.wikipedia.org/w/index.php?title=Computer_Arimaa&oldid=739425208.
- [3] Haskin, Brian "Janzert." A Look at the Arimaa Branching Factor. Retrieved April 23, 2016, from http://arimaa.janzert.com/bf_study/.
- [4] Jakl, Tomas. (2011, October). Arimaa challenge - comparisson study of MCTS versus alpha-beta methods [sic]. (Unpublished Bachelor's thesis). Charles University, Prague. Retrieved October 22, 2016, from <http://arimaa.com/arimaa/papers/ThomasJakl/bc-thesis.pdf>.
- [5] Karwin, Bill (2008, December). "How do you store a trie in a relational database?". StackOverflow. Retrieved October 22, 2016, from <http://stackoverflow.com/a/355064>
- [6] Kozelek, Tomas. (2009, December). Methods of MCTS and the game Arimaa (Unpublished Master's dissertation). Charles University, Prague. Retrieved October 22, 2016, from <http://arimaa.com/arimaa/papers/TomasKozelekThesis/mt.pdf>.
- [7] Lewis, Andy. (2015, July). Arimaa: Game Over? *Kingpin Chess Magazine*. Retrieved October 22, 2016, from <http://www.kingpinchess.net/2015/07/arimaa-game-over/>.
- [8] McKee, Patrick. (2014, August). Arimaa: Developing a Higher Ranked Fall Back Move Generator Using a Relational Database. Rowan University. Retrieved October 22, 2016, from <http://arimaa.com/arimaa/papers/PatrickMcKee/mckee-t.pdf>.
- [9] Syed, A., & Syed, O. (1999, January 1). The creation of Arimaa. Retrieved August 12, 2014, from <http://arimaa.com/arimaa/about/>
- [10] Trippen, Gerhard. (2009, May). Plans, Patterns and Move Categories Guiding a Highly Selective Search. The University of British Columbia. Retrieved October 22, 2016, from <http://arimaa.com/arimaa/papers/0905Trippen/Contribution118.pdf>.

- [11] Tropashko, Vadim. (2005, April). Trees in SQL: Nested Sets and Materialized Path. DBAzone.com. Retrieved October 22, 2016, from <http://www.dbazine.com/oracle/or-articles/tropashko4/>.
- [12] Wu, David Jian. (2011, May). Move Ranking and Evaluation in the Game of Arimaa (Unpublished undergraduate dissertation). Harvard College, Cambridge, Massachussets. Retrieved October 22, 2016, from <http://arimaa.com/arimaa/papers/DavidWu/djwuthesis.pdf>.
- [13] Wu, David Jian. (2015). Designing a Winning Arimaa Program. ICGA Journal, Vol. 38, No. 1, pp. 19-41. Retrieved October 22, 2016, from http://icosahedral.net/downloads/djwu2015arimaa_color.pdf
- [14] Arimaa/Introduction to Strategy/Elephant Blockade. (2016, August 22). *Wikibooks, The Free Textbook Project*. Retrieved October 22, 2016, from https://en.wikibooks.org/w/index.php?title=Arimaa/Introduction_to_Strategy/Elephant_Blockade&oldid=3108306.

Appendix

Move Groups Per Turn Per Radius

The following table describes how many groupings of moves can be made for a game from the game database (id: 1) at each specific radius using the piece data format described in this paper. The Board Groups column describes how many groups can be made by comparing the piece data hashes of all pieces on the board after a move, while the Move Groups column describes the number of groups that can be created by only examining the piece data transitions of the pieces moved. The percent reduction column describes the reduction in the number of unique moves that result from grouping moves either way.

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
1w	1	0	0	0	0	0
1w	2	0	0	0	0	0
1w	3	0	0	0	0	0
1w	4	0	0	0	0	0
1w	5	0	0	0	0	0
1w	6	0	0	0	0	0
1w	7	0	0	0	0	0
1w	8	0	0	0	0	0
1w	9	0	0	0	0	0
1w	10	0	0	0	0	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
1w	11	0	0	0	0	0
1w	12	0	0	0	0	0
1w	13	0	0	0	0	0
1w	14	0	0	0	0	0
1b	1	0	0	0	0	0
1b	2	0	0	0	0	0
1b	3	0	0	0	0	0
1b	4	0	0	0	0	0
1b	5	0	0	0	0	0
1b	6	0	0	0	0	0
1b	7	0	0	0	0	0
1b	8	0	0	0	0	0
1b	9	0	0	0	0	0
1b	10	0	0	0	0	0
1b	11	0	0	0	0	0
1b	12	0	0	0	0	0
1b	13	0	0	0	0	0
1b	14	0	0	0	0	0
2w	1	3359	2767	17.62429294	2244	33.1944031
2w	2	3359	3288	2.113724323	2796	16.76094076
2w	3	3359	3334	0.7442691277	2987	11.07472462
2w	4	3359	3338	0.6251860673	3054	9.080083358

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
2w	5	3359	3352	0.2083953558	3109	7.442691277
2w	6	3359	3354	0.1488538255	3150	6.222089908
2w	7	3359	3354	0.1488538255	3166	5.745757666
2w	8	3359	3354	0.1488538255	3168	5.686216136
2w	9	3359	3354	0.1488538255	3168	5.686216136
2w	10	3359	3354	0.1488538255	3168	5.686216136
2w	11	3359	3354	0.1488538255	3168	5.686216136
2w	12	3359	3354	0.1488538255	3168	5.686216136
2w	13	3359	3354	0.1488538255	3168	5.686216136
2w	14	3359	3354	0.1488538255	3168	5.686216136
2b	1	3341	1782	46.66267585	1996	40.25740796
2b	2	3341	2856	14.51661179	3006	10.02693804
2b	3	3341	3281	1.7958695	3265	2.274768034
2b	4	3341	3336	0.1496557917	3325	0.4788985334
2b	5	3341	3341	0	3340	0.02993115834
2b	6	3341	3341	0	3341	0
2b	7	3341	3341	0	3341	0
2b	8	3341	3341	0	3341	0
2b	9	3341	3341	0	3341	0
2b	10	3341	3341	0	3341	0
2b	11	3341	3341	0	3341	0
2b	12	3341	3341	0	3341	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
2b	13	3341	3341	0	3341	0
2b	14	3341	3341	0	3341	0
3w	1	13435	10036	25.29959062	12110	9.862299963
3w	2	13435	13259	1.310011165	13413	0.1637513956
3w	3	13435	13377	0.4317082248	13435	0
3w	4	13435	13410	0.1860811314	13435	0
3w	5	13435	13431	0.02977298102	13435	0
3w	6	13435	13434	0.007443245255	13435	0
3w	7	13435	13434	0.007443245255	13435	0
3w	8	13435	13434	0.007443245255	13435	0
3w	9	13435	13434	0.007443245255	13435	0
3w	10	13435	13434	0.007443245255	13435	0
3w	11	13435	13434	0.007443245255	13435	0
3w	12	13435	13434	0.007443245255	13435	0
3w	13	13435	13434	0.007443245255	13435	0
3w	14	13435	13434	0.007443245255	13435	0
3b	1	9277	4412	52.44152204	3906	57.89587151
3b	2	9277	8768	5.486687507	8210	11.50156301
3b	3	9277	9236	0.4419532176	9215	0.6683194998
3b	4	9277	9276	0.01077934677	9276	0.01077934677
3b	5	9277	9277	0	9277	0
3b	6	9277	9277	0	9277	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
3b	7	9277	9277	0	9277	0
3b	8	9277	9277	0	9277	0
3b	9	9277	9277	0	9277	0
3b	10	9277	9277	0	9277	0
3b	11	9277	9277	0	9277	0
3b	12	9277	9277	0	9277	0
3b	13	9277	9277	0	9277	0
3b	14	9277	9277	0	9277	0
4w	1	19008	13216	30.47138047	17104	10.01683502
4w	2	19008	18837	0.8996212121	18999	0.04734848485
4w	3	19008	18936	0.3787878788	19008	0
4w	4	19008	18980	0.1473063973	19008	0
4w	5	19008	19005	0.01578282828	19008	0
4w	6	19008	19008	0	19008	0
4w	7	19008	19008	0	19008	0
4w	8	19008	19008	0	19008	0
4w	9	19008	19008	0	19008	0
4w	10	19008	19008	0	19008	0
4w	11	19008	19008	0	19008	0
4w	12	19008	19008	0	19008	0
4w	13	19008	19008	0	19008	0
4w	14	19008	19008	0	19008	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
4b	1	5777	5218	9.676302579	5214	9.745542669
4b	2	5777	5745	0.5539207201	5776	0.0173100225
4b	3	5777	5775	0.03462004501	5777	0
4b	4	5777	5777	0	5777	0
4b	5	5777	5777	0	5777	0
4b	6	5777	5777	0	5777	0
4b	7	5777	5777	0	5777	0
4b	8	5777	5777	0	5777	0
4b	9	5777	5777	0	5777	0
4b	10	5777	5777	0	5777	0
4b	11	5777	5777	0	5777	0
4b	12	5777	5777	0	5777	0
4b	13	5777	5777	0	5777	0
4b	14	5777	5777	0	5777	0
5w	1	31393	24438	22.15462046	29573	5.797470774
5w	2	31393	30980	1.315579906	31393	0
5w	3	31393	31039	1.12763992	31393	0
5w	4	31393	31071	1.025706368	31393	0
5w	5	31393	31367	0.08282101105	31393	0
5w	6	31393	31393	0	31393	0
5w	7	31393	31393	0	31393	0
5w	8	31393	31393	0	31393	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
5w	9	31393	31393	0	31393	0
5w	10	31393	31393	0	31393	0
5w	11	31393	31393	0	31393	0
5w	12	31393	31393	0	31393	0
5w	13	31393	31393	0	31393	0
5w	14	31393	31393	0	31393	0
5b	1	8415	7502	10.8496732	8039	4.468211527
5b	2	8415	8401	0.1663695781	8413	0.02376708259
5b	3	8415	8414	0.0118835413	8415	0
5b	4	8415	8415	0	8415	0
5b	5	8415	8415	0	8415	0
5b	6	8415	8415	0	8415	0
5b	7	8415	8415	0	8415	0
5b	8	8415	8415	0	8415	0
5b	9	8415	8415	0	8415	0
5b	10	8415	8415	0	8415	0
5b	11	8415	8415	0	8415	0
5b	12	8415	8415	0	8415	0
5b	13	8415	8415	0	8415	0
5b	14	8415	8415	0	8415	0
6w	1	26200	22248	15.08396947	25571	2.400763359
6w	2	26200	25853	1.324427481	26200	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
6w	3	26200	25903	1.133587786	26200	0
6w	4	26200	25930	1.030534351	26200	0
6w	5	26200	26177	0.08778625954	26200	0
6w	6	26200	26200	0	26200	0
6w	7	26200	26200	0	26200	0
6w	8	26200	26200	0	26200	0
6w	9	26200	26200	0	26200	0
6w	10	26200	26200	0	26200	0
6w	11	26200	26200	0	26200	0
6w	12	26200	26200	0	26200	0
6w	13	26200	26200	0	26200	0
6w	14	26200	26200	0	26200	0
6b	1	13766	11571	15.94508209	13308	3.327037629
6b	2	13766	13760	0.04358564579	13765	0.007264274299
6b	3	13766	13766	0	13766	0
6b	4	13766	13766	0	13766	0
6b	5	13766	13766	0	13766	0
6b	6	13766	13766	0	13766	0
6b	7	13766	13766	0	13766	0
6b	8	13766	13766	0	13766	0
6b	9	13766	13766	0	13766	0
6b	10	13766	13766	0	13766	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
6b	11	13766	13766	0	13766	0
6b	12	13766	13766	0	13766	0
6b	13	13766	13766	0	13766	0
6b	14	13766	13766	0	13766	0
7w	1	29883	25567	14.44299434	29542	1.141117023
7w	2	29883	29479	1.35193923	29883	0
7w	3	29883	29533	1.171234481	29883	0
7w	4	29883	29607	0.923602048	29883	0
7w	5	29883	29860	0.07696683733	29883	0
7w	6	29883	29883	0	29883	0
7w	7	29883	29883	0	29883	0
7w	8	29883	29883	0	29883	0
7w	9	29883	29883	0	29883	0
7w	10	29883	29883	0	29883	0
7w	11	29883	29883	0	29883	0
7w	12	29883	29883	0	29883	0
7w	13	29883	29883	0	29883	0
7w	14	29883	29883	0	29883	0
7b	1	14284	12851	10.03220386	13733	3.857462896
7b	2	14284	14251	0.2310277233	14284	0
7b	3	14284	14283	0.007000840101	14284	0
7b	4	14284	14283	0.007000840101	14284	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
7b	5	14284	14284	0	14284	0
7b	6	14284	14284	0	14284	0
7b	7	14284	14284	0	14284	0
7b	8	14284	14284	0	14284	0
7b	9	14284	14284	0	14284	0
7b	10	14284	14284	0	14284	0
7b	11	14284	14284	0	14284	0
7b	12	14284	14284	0	14284	0
7b	13	14284	14284	0	14284	0
7b	14	14284	14284	0	14284	0
8w	1	20160	16831	16.51289683	19124	5.138888889
8w	2	20160	19888	1.349206349	20155	0.0248015873
8w	3	20160	19934	1.121031746	20160	0
8w	4	20160	19955	1.016865079	20160	0
8w	5	20160	20142	0.08928571429	20160	0
8w	6	20160	20160	0	20160	0
8w	7	20160	20160	0	20160	0
8w	8	20160	20160	0	20160	0
8w	9	20160	20160	0	20160	0
8w	10	20160	20160	0	20160	0
8w	11	20160	20160	0	20160	0
8w	12	20160	20160	0	20160	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
8w	13	20160	20160	0	20160	0
8w	14	20160	20160	0	20160	0
8b	1	10751	9567	11.01292903	10191	5.208817784
8b	2	10751	10677	0.6883080644	10723	0.2604408892
8b	3	10751	10749	0.01860292066	10751	0
8b	4	10751	10750	0.009301460329	10751	0
8b	5	10751	10751	0	10751	0
8b	6	10751	10751	0	10751	0
8b	7	10751	10751	0	10751	0
8b	8	10751	10751	0	10751	0
8b	9	10751	10751	0	10751	0
8b	10	10751	10751	0	10751	0
8b	11	10751	10751	0	10751	0
8b	12	10751	10751	0	10751	0
8b	13	10751	10751	0	10751	0
8b	14	10751	10751	0	10751	0
9w	1	30321	25141	17.08386927	28504	5.99254642
9w	2	30321	29936	1.26974704	30317	0.01319217704
9w	3	30321	29989	1.094950694	30321	0
9w	4	30321	30042	0.9201543485	30321	0
9w	5	30321	30298	0.07585501797	30321	0
9w	6	30321	30321	0	30321	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
9w	7	30321	30321	0	30321	0
9w	8	30321	30321	0	30321	0
9w	9	30321	30321	0	30321	0
9w	10	30321	30321	0	30321	0
9w	11	30321	30321	0	30321	0
9w	12	30321	30321	0	30321	0
9w	13	30321	30321	0	30321	0
9w	14	30321	30321	0	30321	0
9b	1	8624	7661	11.16651206	8214	4.754174397
9b	2	8624	8616	0.09276437848	8623	0.01159554731
9b	3	8624	8623	0.01159554731	8624	0
9b	4	8624	8624	0	8624	0
9b	5	8624	8624	0	8624	0
9b	6	8624	8624	0	8624	0
9b	7	8624	8624	0	8624	0
9b	8	8624	8624	0	8624	0
9b	9	8624	8624	0	8624	0
9b	10	8624	8624	0	8624	0
9b	11	8624	8624	0	8624	0
9b	12	8624	8624	0	8624	0
9b	13	8624	8624	0	8624	0
9b	14	8624	8624	0	8624	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
10w	1	23498	19473	17.12911737	22673	3.510937101
10w	2	23498	23191	1.30649417	23498	0
10w	3	23498	23238	1.106477147	23498	0
10w	4	23498	23264	0.9958294323	23498	0
10w	5	23498	23477	0.08936930803	23498	0
10w	6	23498	23498	0	23498	0
10w	7	23498	23498	0	23498	0
10w	8	23498	23498	0	23498	0
10w	9	23498	23498	0	23498	0
10w	10	23498	23498	0	23498	0
10w	11	23498	23498	0	23498	0
10w	12	23498	23498	0	23498	0
10w	13	23498	23498	0	23498	0
10w	14	23498	23498	0	23498	0
10b	1	9820	8384	14.62321792	8941	8.951120163
10b	2	9820	9721	1.00814664	9759	0.6211812627
10b	3	9820	9818	0.02036659878	9820	0
10b	4	9820	9819	0.01018329939	9820	0
10b	5	9820	9820	0	9820	0
10b	6	9820	9820	0	9820	0
10b	7	9820	9820	0	9820	0
10b	8	9820	9820	0	9820	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
10b	9	9820	9820	0	9820	0
10b	10	9820	9820	0	9820	0
10b	11	9820	9820	0	9820	0
10b	12	9820	9820	0	9820	0
10b	13	9820	9820	0	9820	0
10b	14	9820	9820	0	9820	0
11w	1	12161	11537	5.131156977	12027	1.101883069
11w	2	12161	11940	1.817284763	12161	0
11w	3	12161	11946	1.767946715	12161	0
11w	4	12161	11963	1.628155579	12161	0
11w	5	12161	12126	0.2878052792	12161	0
11w	6	12161	12161	0	12161	0
11w	7	12161	12161	0	12161	0
11w	8	12161	12161	0	12161	0
11w	9	12161	12161	0	12161	0
11w	10	12161	12161	0	12161	0
11w	11	12161	12161	0	12161	0
11w	12	12161	12161	0	12161	0
11w	13	12161	12161	0	12161	0
11w	14	12161	12161	0	12161	0
11b	1	10479	9237	11.85227598	9521	9.142093711
11b	2	10479	10448	0.2958297547	10404	0.7157171486

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
11b	3	10479	10479	0	10479	0
11b	4	10479	10479	0	10479	0
11b	5	10479	10479	0	10479	0
11b	6	10479	10479	0	10479	0
11b	7	10479	10479	0	10479	0
11b	8	10479	10479	0	10479	0
11b	9	10479	10479	0	10479	0
11b	10	10479	10479	0	10479	0
11b	11	10479	10479	0	10479	0
11b	12	10479	10479	0	10479	0
11b	13	10479	10479	0	10479	0
11b	14	10479	10479	0	10479	0
12w	1	16110	14990	6.9522036	15584	3.265052762
12w	2	16110	15813	1.843575419	16110	0
12w	3	16110	15837	1.694599628	16110	0
12w	4	16110	15877	1.446306642	16110	0
12w	5	16110	16070	0.2482929857	16110	0
12w	6	16110	16110	0	16110	0
12w	7	16110	16110	0	16110	0
12w	8	16110	16110	0	16110	0
12w	9	16110	16110	0	16110	0
12w	10	16110	16110	0	16110	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
12w	11	16110	16110	0	16110	0
12w	12	16110	16110	0	16110	0
12w	13	16110	16110	0	16110	0
12w	14	16110	16110	0	16110	0
12b	1	8997	7935	11.80393464	8124	9.703234411
12b	2	8997	8975	0.2445259531	8977	0.222296321
12b	3	8997	8997	0	8997	0
12b	4	8997	8997	0	8997	0
12b	5	8997	8997	0	8997	0
12b	6	8997	8997	0	8997	0
12b	7	8997	8997	0	8997	0
12b	8	8997	8997	0	8997	0
12b	9	8997	8997	0	8997	0
12b	10	8997	8997	0	8997	0
12b	11	8997	8997	0	8997	0
12b	12	8997	8997	0	8997	0
12b	13	8997	8997	0	8997	0
12b	14	8997	8997	0	8997	0
13w	1	10912	10280	5.791788856	10455	4.188049853
13w	2	10912	10682	2.107771261	10908	0.0366568915
13w	3	10912	10702	1.924486804	10912	0
13w	4	10912	10734	1.631231672	10912	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
13w	5	10912	10878	0.3115835777	10912	0
13w	6	10912	10912	0	10912	0
13w	7	10912	10912	0	10912	0
13w	8	10912	10912	0	10912	0
13w	9	10912	10912	0	10912	0
13w	10	10912	10912	0	10912	0
13w	11	10912	10912	0	10912	0
13w	12	10912	10912	0	10912	0
13w	13	10912	10912	0	10912	0
13w	14	10912	10912	0	10912	0
13b	1	7640	7292	4.554973822	7242	5.209424084
13b	2	7640	7627	0.1701570681	7640	0
13b	3	7640	7640	0	7640	0
13b	4	7640	7640	0	7640	0
13b	5	7640	7640	0	7640	0
13b	6	7640	7640	0	7640	0
13b	7	7640	7640	0	7640	0
13b	8	7640	7640	0	7640	0
13b	9	7640	7640	0	7640	0
13b	10	7640	7640	0	7640	0
13b	11	7640	7640	0	7640	0
13b	12	7640	7640	0	7640	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
13b	13	7640	7640	0	7640	0
13b	14	7640	7640	0	7640	0
14w	1	20943	19557	6.617963043	19237	8.145919878
14w	2	20943	20595	1.661653058	20902	0.1957694695
14w	3	20943	20636	1.465883589	20943	0
14w	4	20943	20693	1.193716278	20943	0
14w	5	20943	20903	0.1909946044	20943	0
14w	6	20943	20943	0	20943	0
14w	7	20943	20943	0	20943	0
14w	8	20943	20943	0	20943	0
14w	9	20943	20943	0	20943	0
14w	10	20943	20943	0	20943	0
14w	11	20943	20943	0	20943	0
14w	12	20943	20943	0	20943	0
14w	13	20943	20943	0	20943	0
14w	14	20943	20943	0	20943	0
14b	1	8128	7804	3.986220472	7710	5.142716535
14b	2	8128	8113	0.1845472441	8127	0.01230314961
14b	3	8128	8128	0	8128	0
14b	4	8128	8128	0	8128	0
14b	5	8128	8128	0	8128	0
14b	6	8128	8128	0	8128	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
14b	7	8128	8128	0	8128	0
14b	8	8128	8128	0	8128	0
14b	9	8128	8128	0	8128	0
14b	10	8128	8128	0	8128	0
14b	11	8128	8128	0	8128	0
14b	12	8128	8128	0	8128	0
14b	13	8128	8128	0	8128	0
14b	14	8128	8128	0	8128	0
15w	1	30847	25692	16.71151165	29530	4.269458943
15w	2	30847	30481	1.186501118	30847	0
15w	3	30847	30554	0.949849256	30847	0
15w	4	30847	30597	0.8104515836	30847	0
15w	5	30847	30807	0.1296722534	30847	0
15w	6	30847	30847	0	30847	0
15w	7	30847	30847	0	30847	0
15w	8	30847	30847	0	30847	0
15w	9	30847	30847	0	30847	0
15w	10	30847	30847	0	30847	0
15w	11	30847	30847	0	30847	0
15w	12	30847	30847	0	30847	0
15w	13	30847	30847	0	30847	0
15w	14	30847	30847	0	30847	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
15b	1	8485	7381	13.01119623	8239	2.899233942
15b	2	8485	8470	0.1767825575	8469	0.1885680613
15b	3	8485	8485	0	8485	0
15b	4	8485	8485	0	8485	0
15b	5	8485	8485	0	8485	0
15b	6	8485	8485	0	8485	0
15b	7	8485	8485	0	8485	0
15b	8	8485	8485	0	8485	0
15b	9	8485	8485	0	8485	0
15b	10	8485	8485	0	8485	0
15b	11	8485	8485	0	8485	0
15b	12	8485	8485	0	8485	0
15b	13	8485	8485	0	8485	0
15b	14	8485	8485	0	8485	0
16w	1	27733	24669	11.04820971	26606	4.063750766
16w	2	27733	27409	1.168283273	27732	0.00360581257
16w	3	27733	27461	0.980781019	27732	0.00360581257
16w	4	27733	27502	0.8329427036	27733	0
16w	5	27733	27695	0.1370208777	27733	0
16w	6	27733	27733	0	27733	0
16w	7	27733	27733	0	27733	0
16w	8	27733	27733	0	27733	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
16w	9	27733	27733	0	27733	0
16w	10	27733	27733	0	27733	0
16w	11	27733	27733	0	27733	0
16w	12	27733	27733	0	27733	0
16w	13	27733	27733	0	27733	0
16w	14	27733	27733	0	27733	0
16b	1	2573	2470	4.003109211	2552	0.8161678974
16b	2	2573	2573	0	2573	0
16b	3	2573	2573	0	2573	0
16b	4	2573	2573	0	2573	0
16b	5	2573	2573	0	2573	0
16b	6	2573	2573	0	2573	0
16b	7	2573	2573	0	2573	0
16b	8	2573	2573	0	2573	0
16b	9	2573	2573	0	2573	0
16b	10	2573	2573	0	2573	0
16b	11	2573	2573	0	2573	0
16b	12	2573	2573	0	2573	0
16b	13	2573	2573	0	2573	0
16b	14	2573	2573	0	2573	0
17w	1	31430	28399	9.643652561	29741	5.373846643
17w	2	31430	31074	1.132675787	31429	0.00318167356

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
17w	3	31430	31136	0.9354120267	31429	0.00318167356
17w	4	31430	31178	0.8017817372	31430	0
17w	5	31430	31390	0.1272669424	31430	0
17w	6	31430	31430	0	31430	0
17w	7	31430	31430	0	31430	0
17w	8	31430	31430	0	31430	0
17w	9	31430	31430	0	31430	0
17w	10	31430	31430	0	31430	0
17w	11	31430	31430	0	31430	0
17w	12	31430	31430	0	31430	0
17w	13	31430	31430	0	31430	0
17w	14	31430	31430	0	31430	0
17b	1	4922	4308	12.47460382	4837	1.726940268
17b	2	4922	4922	0	4922	0
17b	3	4922	4922	0	4922	0
17b	4	4922	4922	0	4922	0
17b	5	4922	4922	0	4922	0
17b	6	4922	4922	0	4922	0
17b	7	4922	4922	0	4922	0
17b	8	4922	4922	0	4922	0
17b	9	4922	4922	0	4922	0
17b	10	4922	4922	0	4922	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
17b	11	4922	4922	0	4922	0
17b	12	4922	4922	0	4922	0
17b	13	4922	4922	0	4922	0
17b	14	4922	4922	0	4922	0
18w	1	19782	17980	9.109291275	18242	7.784854919
18w	2	19782	19509	1.380042463	19756	0.1314326155
18w	3	19782	19564	1.10201193	19782	0
18w	4	19782	19587	0.9857446163	19782	0
18w	5	19782	19749	0.1668183197	19782	0
18w	6	19782	19782	0	19782	0
18w	7	19782	19782	0	19782	0
18w	8	19782	19782	0	19782	0
18w	9	19782	19782	0	19782	0
18w	10	19782	19782	0	19782	0
18w	11	19782	19782	0	19782	0
18w	12	19782	19782	0	19782	0
18w	13	19782	19782	0	19782	0
18w	14	19782	19782	0	19782	0
18b	1	2825	2775	1.769911504	2800	0.8849557522
18b	2	2825	2824	0.03539823009	2814	0.389380531
18b	3	2825	2825	0	2825	0
18b	4	2825	2825	0	2825	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
18b	5	2825	2825	0	2825	0
18b	6	2825	2825	0	2825	0
18b	7	2825	2825	0	2825	0
18b	8	2825	2825	0	2825	0
18b	9	2825	2825	0	2825	0
18b	10	2825	2825	0	2825	0
18b	11	2825	2825	0	2825	0
18b	12	2825	2825	0	2825	0
18b	13	2825	2825	0	2825	0
18b	14	2825	2825	0	2825	0
19w	1	22307	20116	9.82202896	20753	6.966423096
19w	2	22307	21953	1.586945802	22241	0.2958712512
19w	3	22307	22061	1.102792845	22307	0
19w	4	22307	22084	0.9996861972	22307	0
19w	5	22307	22270	0.1658672166	22307	0
19w	6	22307	22307	0	22307	0
19w	7	22307	22307	0	22307	0
19w	8	22307	22307	0	22307	0
19w	9	22307	22307	0	22307	0
19w	10	22307	22307	0	22307	0
19w	11	22307	22307	0	22307	0
19w	12	22307	22307	0	22307	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
19w	13	22307	22307	0	22307	0
19w	14	22307	22307	0	22307	0
19b	1	2169	2062	4.933148917	2082	4.011065007
19b	2	2169	2169	0	2090	3.642231443
19b	3	2169	2169	0	2169	0
19b	4	2169	2169	0	2169	0
19b	5	2169	2169	0	2169	0
19b	6	2169	2169	0	2169	0
19b	7	2169	2169	0	2169	0
19b	8	2169	2169	0	2169	0
19b	9	2169	2169	0	2169	0
19b	10	2169	2169	0	2169	0
19b	11	2169	2169	0	2169	0
19b	12	2169	2169	0	2169	0
19b	13	2169	2169	0	2169	0
19b	14	2169	2169	0	2169	0
20w	1	29656	27231	8.177097383	28029	5.486242244
20w	2	29656	28936	2.427839223	29290	1.234151605
20w	3	29656	29330	1.099271648	29656	0
20w	4	29656	29357	1.008227677	29656	0
20w	5	29656	29611	0.1517399514	29656	0
20w	6	29656	29656	0	29656	0

Turn ID	Radius	Total Moves	Board Groups	Board Groups Percent Reduction (%)	Move Groups	Move Groups Percent Reduction (%)
20w	7	29656	29656	0	29656	0
20w	8	29656	29656	0	29656	0
20w	9	29656	29656	0	29656	0
20w	10	29656	29656	0	29656	0
20w	11	29656	29656	0	29656	0
20w	12	29656	29656	0	29656	0
20w	13	29656	29656	0	29656	0
20w	14	29656	29656	0	29656	0