9-18-2014

ARIMAA: developing a higher ranked fall back move generator using a relational database

Patrick McKee

ARIMAA: DEVELOPING A HIGHER RANKED FALL BACK MOVE GENERATOR USING A RELATIONAL DATABASE

by
Patrick McKee

A Masters 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
at
Rowan University
August 13, 2014

Thesis Chair: Dr. Nancy Tinkham

Dedication

For my Father, because even when he found out he had cancer and did not know how long he had to live, he still told me to not bother visiting him and to work on my thesis. Without him I would have never finished because there is nothing I would have rather have done than sit with him watching FOX News.

Acknowledgments

I would like to thank Dr. Tinkham, Dr. Hartley and Dr. VH whose attention and support turned this concept from an afternoon discussion into a fruit bearing research effort. Receiving E-mails at 3AM by Dr. Tinkham always made laugh, and the memory of it will continue to do so.

I would also like to thank everyone who sat with me in the CS offices in Robinson studying and making this journey one whose memories I will carry with me forever. Finally, I would like to specifically thank Malik Ahmed for introducing me to Arimaa and being available for questions at all hours of the day.

Abstract

Patrick McKee

ARIMAA: DEVELOPING A HIGHER RANKED FALL BACK MOVE GENERATOR USING A RELATIONAL DATABASE

Dr. Nancy Tinkham, Ph.D.

Master of Science in Computer Science

Our approach to playing the game of Arimaa understands that the game was created to showcase the limits of brute force computing power. Using a relational database, we will be able to view similar situations that have already been played out, inventory a number of suitable reactions and make the best move given a number of attributes. This results in us making a decision that can theoretically result in a win, and hopefully will.

Measurable positive results have been procured specifically from the area of concentration component of the research. In the area of concentration component we target a specific square on the board based on prior moves. This square becomes the focal point of our research that we develop an attribute index for. After querying the database to see if we can find a previous game that contained this exact same area of concentration, we either make a move or fall back. If we fall back, we take into account our developed shrinking method where we target specific pieces whose strengths essentially do not matter in terms of making a move. It is with this action that we have developed measurable positive results that, with further research, may amount to a permanent fixture as a better fall back move generator.

Table of Contents

Table of Contents (Continued)

List of Figures

List of Tables

Chapter 1

Introduction and Rules

1.1 – Background

With the advent of the chess playing computer Deep Blue, who won the iconic chess match against Gary Kasparov in 1997, IBM proved that their artificial intelligence systems could overtake human opponents. This defeat fueled the curiosity and ingenuity of a man named Omar Syed, an Indian American computer engineer. Syed believed that Kasparov (1999)"had just been out calculated, not really out smarted[sic]" in his defeat. He believed that Kasparov was not able to showcase his true brilliance in a game of Chess, and felt compelled to develop a game that needed (1999) "the kind of real intelligence that humans possess and computers have not even begun to acquire". This game would come to be known as Arimaa.

Using a standard chess board, Syed created a game whose rule set was a purposeful hindrance to the current artificial intelligence and game playing theories. In game playing, a branching factor is a way of describing the difficulty of a game. A game is represented by a tree data structure, and the branching factor is the number of children generated by a node. The branching factor of a game of chess is about 35, while the branching factor of a game of Arimaa is around 17,281 (Haskin, 2006). This means that the particular strategies of tree traversal that are used for chess are severely limited in the game of Arimaa.

1.2 – Setup and Movement

The game of Arimaa can be played on a standard chess board with standard chess pieces. Each piece in Chess corresponds to a piece in Arimaa. The table below gives the represented equalities.

Table 1 – Chess and Arimaa Equivalences

Chess	Arimaa	Arimaa Shorthand
King	Elephant	E or e
Queen	Camel	M or m
Rook	Horse	H or h
Bishop	Dog	D or d
Knight	Cat	C or c
Pawn	Rabbit	R or r

The pieces carry an inherent weight that is used to represent their strength. The weights correspond to the general weight of the animal the piece represents. The elephant is the strongest piece, followed by the camel, followed by the horse and so on with the rabbit being the weakest piece. The goal is for a player to move one of his rabbit pieces to the other end of the board on any square in the back row of the opponent's side of the board. The game is divided into two teams, a gold team and a silver team (formerly a white team and a black team). The game has a start state of an empty board. Letters a through h designate the horizontal axes from left to right, while numbers 1 through 8 designate the

vertical axes from the bottom to the top. Coins, or another token, are to be placed on squares c3, f3, c6, and f6. These squares represent traps, the significance of which will be explained later.

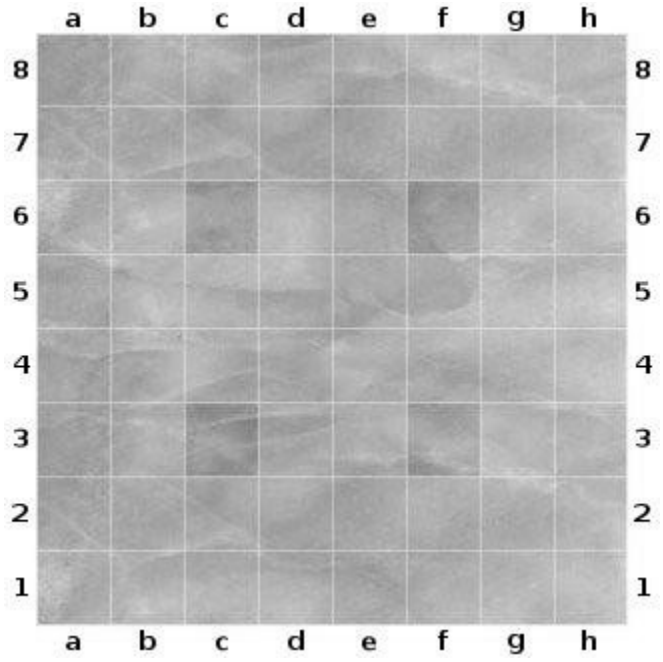

Figure 1 – An Arimaa Start State Board (The darker colored squares represent traps.)

The player representing the gold team gets the honor of placing his pieces on the board first. The gold player may place his pieces in any order that he wants, filling the back two rows on his side of the game board. Once he is finished, the silver player then does the same. After the pieces are arranged, the players may begin their turns moving pieces, with the gold player going first. Each player gets a maximum of 4 moves per turn, with a minimum of 1 move that must be taken. Each piece, except the rabbit, may move to

occupy any unoccupied square adjacent to it (that is forward, backward, left, or right). There is no diagonal movement. This movement constitutes a single move. A rabbit, however, may not move backward, and therefore can only move forward, left, and right.

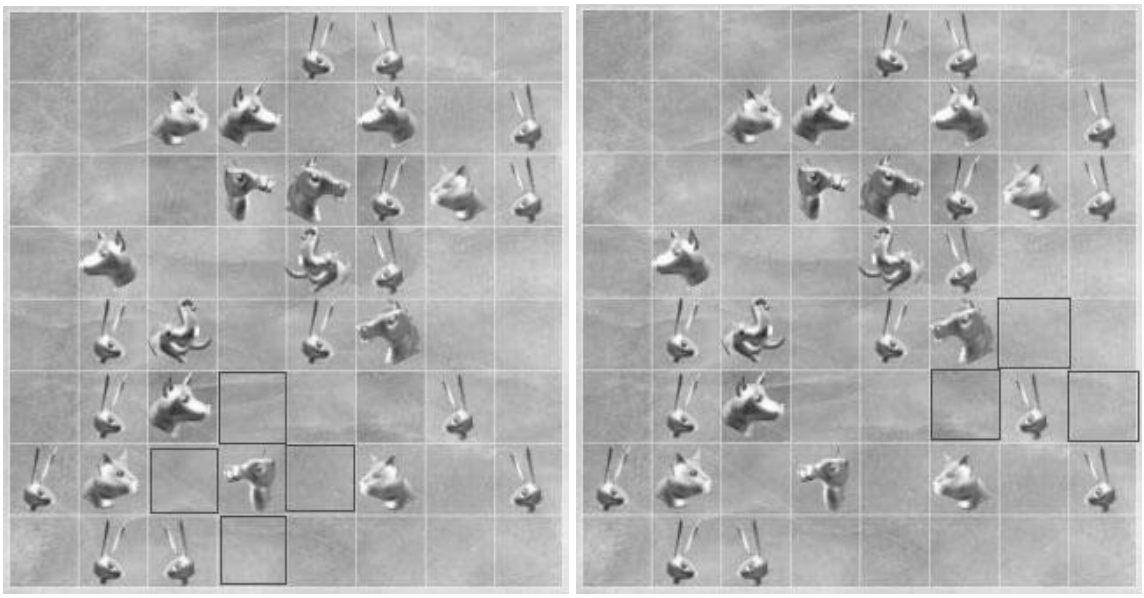

Figure 2 – Example Moves for a Camel and a Rabbit (The black squares on the left indicate possible moves for the silver camel. The black squares on the right indicate possible moves for the rabbit, though moving into a trap square is not always wise!)

If a piece happens to move into a position where it is adjacent to an opponent's piece of a stronger weight it is at risk of becoming frozen. If the weaker piece is not adjacent to a friendly piece, that weaker piece is now frozen and immediately loses its ability to move. There are two ways to have a piece unfrozen. The first way is to wait for the piece of

stronger weight to move from the adjacent square. The second way is for the frozen piece to have an ally of any strength move into an adjacent square. In both situations the unfreezing is immediate rather than occurring at the end of the turn.

1.3 – Pushing / Pulling, Elimination and Win States

In a turn, a player may also push or pull an opponent's piece. Pushing and pulling may be done to an enemy piece by a piece only of greater weight. In a push, a piece is moved to an adjacent square as if it was making the move itself, but following this move the piece doing the pushing then occupies the square of the weaker piece. A pull is done by moving the stronger piece to an open square where it may possibly move to as if it were taking a turn, but then replacing the square that the stronger piece inhabited with the weaker piece. A push and a pull take two moves; therefore only two can be accomplished in a given turn. Pushes and pulls can be used with regular moves as well, so a player may move two squares to a piece, and then follow through with a push or pull. Pushes and pulls are key in eliminating the opponent's pieces from the game.

To eliminate a piece from the game, a player may push or pull an enemy's piece into one of the trap squares. Once the movement is over, the piece in the trap square is removed from the game if the weaker piece does not have any friendly pieces adjacent to that trap square. If a piece in a trap square has an ally piece of any strength adjacent to that square, it is said to be protected and is not removed from the game. If a piece is on a trap square and is protected by a friendly piece, the piece on the trap square is captured immediately if that friendly piece moves either of its own accord or by being pushed or

pulled.

A win state is accomplished in one of three ways. A player may win by having one of his rabbits occupy the back row of the opponent's game board. Secondly, a player may win if he renders the other player without any moves, either by trap situations or by freezing. Lastly, a player may win if he manages to capture all of the opponent's pieces in traps.

1.4 – Notation

To our benefit, each game that is played online using the Arimaa game room is recorded. The first two steps, the board placement steps, use a set of three character strings separated by a space. The first character is the Arimaa shorthand notation for the piece that is being placed (uppercase for gold, lowercase for silver). The second character is the column (a lowercase letter from a to h). The third character is the row (a number from 1 to 8). A sample starting position would be written as Ra1 Rb1 Rc1 Cd1 Ce1 Rf1 Rg1 Rh1 Ra2 Hb2 Dc2 Md2 Ee2 Df2 Hg2 Rh2. After the turn is finished an end line character (\n) is used. Once the starting positions are finished the notation changes. The three character strings become 4 character strings with the new character designating the movement of the piece (n, e, w, s). For example, Ra1n means that the gold rabbit in position a1 is moving north, therefore his new position will be Ra2. The new position is not set up in the database, but it can be discovered by an accurate trace. The intention and invention of that program is where our research begins. Before we address this, however, there is one caveat of Arimaa that makes brute force even more difficult, and

that is the concept of time constraints.

1.5 – Time Constraints

Arimaa has a set of time rules that can change from match to match. Users can agree not to restrict themselves to a time limit, but for the most part games are played with an inherent time factor. This time factor is specified as M/R/P/L/G/T. M is the number of minutes and seconds allowed per move. If a user finishes his move before M elapses, then the remaining time is placed into R based on a percentage P. R can increase throughout the game for each move until a max amount, L, is reached. G is the total time allowed for a game and T is the total time that a user has to make a move in a turn. If a user fails to make a move in the allotted time designated by M+R, then the user loses. If G elapses, then the game is scored by captured pieces and the winner is determined that way. M and R are the only pieces of information that are required. P defaults to 100, L, G and T default to 0 which means no limit. For example, if M is two minutes and R is 0 on the first move, if a user makes a move in 30 seconds, then a minute and a half is moved to R. If the user exhausts the two minutes in M on his next turn, then he has another 1 minute and 30 seconds to finish his move before he is declared in violation and loses the game. These time limit rules are important because they are strictly enforced in tournament style games and in the Arimaa game room where most of our test games will take place. It is therefore imperative that our program function in a capacity where performance and speed is important, because our program is going to compete directly with an approach that is built for speed.

1.6 – Literature Review

Arimaa, being a relatively new field of interest, has not been blessed with the benefit of time, therefore the literature surrounding the research is limited. One of the first papers that I read that gave me an understanding of how others approached the idea of move-making was "Move Ranking and Evaluation in the Game of Arimaa" by David Jian Wu of Harvard College (Wu, 2011). In his paper, David Jian Wu discusses separating moves from each other and viewing them as a competition against each other with only one move able to be the expert. He expresses this competition using the Bradley-Terry Model, a model first used by a man named Coulom with respect to the game of Go (Coulom, 2007). In this model individual competitions are waged between move sets using a probability that is proportional to strengths assigned to the moves. The judgment call and definition of these strengths is based on a model developed over a set of game records. It was this paper that started the idea of using past game records to develop a new approach.

The new approach of using previously played games was kept on the back burner as we began researching traditional methods. Choksi, Ebrahim-Zadeh and Mohan of Stanford developed a method to determine game phase in the paper "Leveraging Game Phase in Arimaa" (Choksi, Ebrahim-Zadeh & Mohan, 2013). Game phase was an important subject piece because it was the first paper we read that acknowledged game moves in different portions of a game should not be judged the same way. Unfortunately, the efforts of this technique were for naught as the methods adapted and used did not end up faring well against the simple 2-ply mini-max search with alpha-beta pruning. The

technique, however, of separating portions of a game became of high interest as a technique that may not work with a tree based approach, but perhaps may work with an approach that uses past games.

The final paper that we reviewed was "Methods of MCTS and the game Arimaa" by Tomas Kozelek of Charles University in Prague (Kozelek, 2009). In this paper, a Monte Carlo approach was used; that is, simulated games were played until completion, the results were ranked, and those results were then used to compute the best move set for a given input state. The Monte Carlo approach, again, resulted in poor performance. It was not until a hybrid Monte Carlo method was developed involving heuristics that the research began to bear fruit. The Monte Carlo method was interesting because the bottleneck involved, the simulated playing out of the games, was something that we could somewhat estimate based on the fact that we had a database of simulated games already at our disposal. We then began to wonder if we could use this approach accompanied with the others to create a hybrid solver based on past games.

The idea being presented was to take the current research and adapt it for use in a database. We would use a set of simulated games, like the Monte Carlo approach, and then separate the game states into different areas based on characteristics of our database responses. These areas would then be assessed based on the historical significance of previously played games. This way we would be able to use and develop previously failed ideas in a new light, and hopefully develop a more enhanced game playing bot.

1.7 – The Current Method

As discussed on Page 21 in further detail, Arimaa brute forcing of a game tree can easily look two moves ahead. Many bots use this fact when developing their approach. A popular approach is to take some amount of allotted time to decide which move to follow through with. If time is becoming an issue, the bot starts a search tree that looks two moves ahead to find a fall back move in case the original move function did not happen fast enough. The amount of time taken varies, but a safe padding is around 30 seconds so that the two minute M mark is not reached. This current method, though helpful in not losing, basically amounts to a move that is no better than a random move. Therefore, as we run our tests on data retrieval, we will be looking at response times to make sure that we are not in violation of the standard 30 second buffer that is used for the two move search tree. It is important that our first set of moves occur in a quick fashion as well, so that we can develop a large bank in R, just in case we violate our time constraint. When we run our tests we will keep track of time allowed per move and we will be looking at scaling to see how this will improve based on the pool of moves that we need to evaluate at each step. With that said, we will now take a look at how we set up our database.

Chapter 2

Setting Up Our Database

This chapter includes information on where we received our data, what it means, how we process it, where we store it and how we store it.

2.1 – The Archives

Our approach to Arimaa is one where we will try to mimic the behavior of a previous player. To do so, we must take into account previous games that have been played. The Arimaa website contains a running archive of games that have been played starting from 2002 to current day. To begin, we downloaded all the archives starting from the beginning of 2002 to 3/23/2013 totaling 261,080 games from the Arimaa archives at http://arimaa.com/arimaa/download/gameData. The archives are in the form of tab delimited text files. They are structured like a database with the following columns

Table 2 – Database Setup for the Arimaa Game Archives

Column Name	Type	Description
id	INT	A unique ID used to identify the game in the Arimaa database
gplayerid	INT	The player ID from the Arimaa website of the gold player
splayerid	INT	The player ID from the Arimaa website of the silver player

gusername	STRING	The user name from the Arimaa website of the gold player
susername	STRING	The user name from the Arimaa website of the silver player
gtitle	STRING	Title of the gold player in ranked games
stitle	STRING	Title of the silver player in ranked games
gcountry	STRING	The country where the gold player resides
scountry	STRING	The country where the silver player resides
grating	INT	The player rating of the gold player
srating	INT	The player rating of the silver player
gtype	CHAR	The type of player for the gold player stored as a char where h is human and b is bot
stype	CHAR	The type of player for the silver player stored as a char where h is human and b is bot
event	STRING	The name of the event where the match took place such as Casual game
site	STRING	The location of the event
timecontrol	STRING	The type of time control for the game

postal	BOOLEAN	This field marks whether or not a game is a long form game. A long form game has looser time constraints defined in the time control.
startts	INT	Timestamp for the start of the game
endts	INT	Time stamp for the end of the game
result	CHAR	The winning player
termination	CHAR	How the player lost
plycount	INT	The move count
mode	STRING	The rule set of the game
rated	BOOLEAN	Whether or not the game is rated
movelist	STRING	The entire move list in Arimaa notation which is delimited with a \n to indicate an end of line
events	STRING	The event list of the game in Arimaa notation which is delimited with a \n to indicate an end of line

*Note – Some databases use the terms w and b rather than g and s since early iterations of the Arimaa game called the pieces white and black rather than gold and silver. This is taken into account when parsing the files.

2.2 – Our Machine and the Import of the Archives

Our data will be stored on a machine with the following specs: Intel i5 760 2.8 GHz Quad Core Processor, 6GB of DDR3 RAM and 2 x 500GB SATA Hard Drives in RAID 1. This machine is running Ubuntu 12.4, MySQL 5.5.31 and PHP 5.3. The data was successfully imported in three hours and twenty-three minutes run at normal priority. It is important that our machine be "consumer grade", as the rules for Arimaa tournaments strictly prohibit the participation of any machine constructed of parts not available either by stock or price by the average consumer.

2.3 – Translating the Archives to Our Database

The archives contain a large amount of data, most of which is unnecessary for our goal. Our goal is to take the archives and parse them to create our own database using select pieces of information that are important to us. The following is the database that was designed and the data we wish to achieve with our successful parse.

Table 3 – Database Layout for our Processed Games

Column Name	Type	Description
id	BIGINT	Unique row identifier (auto_increment, unique, primary key)
game_id	INT	Game number from the original database
turn	SMALLINT	What turn we are on
side	ENUM('b','w','g','s')	Which side is making the move
brating	SMALLINT	Black's rating after game is over
wrating	SMALLINT	White's rating after game is over
winner	ENUM('b','w','g','s')	Which side won
boardstatus	CHAR(64)	The board as it looked before the move took place to analyze as if move will be made by black. 64 characters, A8-H8, A7-H7...A1-H1. Character representing pieces, or '-' for no piece
next_move	VARCHAR(255)	Move notation for the move they made next
maxturns	SMALLINT	Total number of turns for the side in this game. For here we can get turns left for a game.

Table 4 – Indexes for our Database

Index Name	Index Type	Fields	Purpose
Primary	Primary Key	id	Primary key for row
move	index	boardstatus	Data search will be for a specific board (boardstatus). We'll be returning the next move and sorting by rating, number of moves left, and if they won or not, but this is not what you're searching for and is not in the index

The rationale for the design of the database is that we have available to us, thanks to the archives, a move list. We want to take this move list and trace it, storing each individual board state that was achieved during the game and the move that occurred after that state. By doing so we can develop a database of possible next moves given a board state. To do so, we developed a program, written in PHP, that created this inventory of board states from the move lists. The program is available in File 1. At the completion of the import, which took 86 hours and 3 minutes, we now had a database of useful information for our next step.

Chapter 3

Developing a Mimic Strategy

In this chapter we will discuss our strategy to reproduce past moves that have been made by former players. We will also address the algorithm used to decide what is the best move.

3.1 – Finding the Best Starting Position

Our game database contains 261,080 games, starting from 2002 and ending on 3/23/2013. We will discuss two ways to approach the situation of starting position that will yield the largest useable database subset. In the event that we are gold, we are given the opportunity of arranging our pieces first. In this event, we want to run a query that will return the most common starting position in the database. By receiving the most common starting position in the database we are increasing our set of useable games that have already been played, and thus will have a larger sample size of games to reference. We must therefore define a query to give all the unique gold starts and how many times they got used. By analyzing this we can determine what the best starting position is for us as gold, because a starting position that is used the most will give us the largest usable subset to work with. The query that was designed was

```
SELECT COUNT(*) AS number_repeats, next_move FROM moves WHERE turn=1
AND (side='w' OR side='g') GROUP BY next_move
```

This query returns a result in 656 seconds with a database size of 261080 games. This, of course, is too long a response time to conduct in game; therefore the favored starting position will be either hard coded in or stored in the database. The following table shows the top ten positions and calculates the percentage of total games that used these positions. As can be seen, there is a clear favorite starting position so our usable subset is still large.

Table 5 – Top Ten Most Popular Starting Positions

Rank	Number of Games	Position	% of Database
1	19398	Ra1 Rb1 Rc1 Cd1 Ce1 Rf1 Rg1 Rh1 Ra2 Hb2 Dc2 Md2 Ee2 Df2 Hg2 Rh2	7.43%
2	12784	Ra1 Rb1 Rc1 Rd1 Re1 Rf1 Rg1 Rh1 Da2 Hb2 Cc2 Md2 Ee2 Cf2 Hg2 Dh2	4.90%
3	10996	Ra1 Rb1 Rc1 Rd1 Re1 Rf1 Rg1 Rh1 Da2 Hb2 Cc2 Ed2 Me2 Cf2 Hg2 Dh2	4.21%
4	10308	Ra1 Rb1 Rc1 Dd1 De1 Rf1 Rg1 Rh1 Hb2 Ra2 Cc2 Md2 Ee2 Cf2 Rh2 Hg2	3.95%

5	9132	Ra1 Rb1 Rc1 Rd1 Re1 Rf1 Rg1 Rh1 Ha2 Db2 Cc2 Md2 Ee2 Cf2 Dg2 Hh2	3.5%
6	9013	Ra1 Rb1 Rc1 Dd1 De1 Rf1 Rg1 Rh1 Ra2 Hb2 Cc2 Ed2 Me2 Cf2 Hg2 Rh2	3.45%
7	8604	Ra1 Rb1 Rc1 Dd1 De1 Rf1 Rg1 Rh1 Ra2 Hb2 Cc2 Md2 Ee2 Cf2 Hg2 Rh2	3.3%
8	6295	Ra1 Rb1 Rc1 Cd1 Ce1 Rf1 Rg1 Rh1 Ra2 Hb2 Dc2 Ed2 Me2 Df2 Hg2 Rh2	2.41%
9	5717	Ra2 Hb2 Dc2 Ed2 Me2 Df2 Hg2 Rh2 Ra1 Rb1 Rc1 Cd1 Ce1 Rf1 Rg1 Rh1	2.19%
10	5138	Ra2 Db2 Hc2 Ed2 Me2 Hf2 Dg2 Rh2 Ra1 Rb1 Rc1 Cd1 Ce1 Rf1 Rg1 Rh1	1.97%

Therefore the start position that we will choose if we are gold is Ra1 Rb1 Rc1 Cd1 Ce1 Rf1 Rg1 Rh1 Ra2 Hb2 Dc2 Md2 Ee2 Df2 Hg2 Rh2. It may also be worthwhile to note that there are only 22737 different starts in a database of 261080 games.

The second situation we must discuss is when we are silver. When we are silver we move second. This means we must react, and are no longer interested in the most common starting position, as it may not be worthwhile to use it. Therefore, we must then search the database for games where gold had the same start position as the game we are

currently in, and then we must find the most common silver start position. To develop a query and test the performance, we used the gold starting position given above as the position we are reacting to. The following query was designed to return our best move set given gold's decision

```
SELECT next_move FROM moves WHERE turn=1 AND (side='b' OR side='s') AND
boardstatus='-----------------------------------------------RHDMEDHRRRRCCRRR '
```

The biggest problem with being silver is that we are in a position that needs to react and must therefore run this query in game. It can not be run out of the game and stored like if we are gold, therefore it is very important to view how well this method can scale. Initially it took around twenty-two minutes to run this search, which is not usable in game. However, subsequent runs of the same query returned much nicer results in a short amount of time. Initially we believed it was because of MySQL caching the file and data. We then tried to reproduce the long run by flushing the cache. This did not work, as our results still returned quickly. It was then hypothesized that the Linux file system we were using kept the database file in cache, and therefore could return results much more quickly than the initial run, therefore it must be noted that to achieve worthwhile results for this query the data must continuously be queried to stay in cache. To do so, a bot may simply do a query of any kind to keep the file in cache between games so that it does not fall out. This instruction mimics the responsibility of a NOP instruction. Since we were concerned about scale and how long to expect a response time, we ran the query on

subsequent 1/8ths of the data.

Table 6 – Query Times for the Silver Starting Position

Data Size	Response Time in Seconds
1993479	1
3986958	2.3
5980437	3.49
7973916	5.34
9967395	5.58
11960874	7.17
13954353	7.40
15947465	7.96

So even with our large data set, a result can be found fairly quickly. If we continue to increase our data size, and the response time becomes a legitimate concern, the query can be changed to only query a certain size of the data. An example of such a query is

SELECT next_move FROM moves WHERE turn=1 AND (side='b' OR side='s') AND wboardstatus='rrrccrrrrhdemdhr' AND id BETWEEN X and Y

where X and Y is the span of games that wish to be searched. Of course it is possible that the person choosing a start position for gold chooses a position that is not in our database.

In that event, we will choose the same position that gold has chosen, staying true in our attempt to mimicking behavior. Now that we have chosen how we will align our pieces, we must then consider how we will move.

3.2 – Win in Two Database

In Arimaa there is an area of concentration called the "win in two" puzzle. This area of study tries to determine how difficult it is for a human or a bot to realize that a given board state can be solved in two moves. This area of concentration was developed based on the brute force abilities of modern chess programs. To discount brute force as a viable option, we must develop a ceiling of search depth similar to what has been discovered for Chess. If the average branching factor of a game of Chess is 35 and Arimaa is 17,281, then a computer that can search a depth of 8 turns per player in chess can not even reach 3 turns per player in Arimaa.

$$\frac{8}{\log_{35} 17281} \approx 2.9$$

Figure 3 – Brute Force Depth Calculation for Arimaa (Why we love Arimaa, 2011)

With that known, seeing three moves ahead becomes too taxing of a service. Seeing two moves ahead, however, is feasible and relatively quick. Since this is an important area of concentration, a database was started where win in two puzzles were recorded. The database can be found at http://arimaa.com/arimaa/games/puzzles.txt. The format of the

database is gameNumber:moveNumber winningMove. Therefore, before we do any calculations, we can query our database to find out if our boardState matches any of those in the win in two database. If it does, then we know which move to make to secure a win. To begin, we took the data from the win in two database, parsed it, and put it into a database.

Table 7 – Win in Two Database Setup

Column Name	Type	Description
id	INT	Unique row identifier (auto_increment, unique, primary key)
game_id	INT	Game number from .txt file
turn	SMALLINT	What turn we are on
side	ENUM('b','w','g','s')	Which side is making the move
boardstatus	CHAR(64)	The board as it looked **before** the move took place to analyze. 64 characters, A8-H8,A7-H7...A1-H1. Character representing pieces, or hyphen for no piece
move	VARCHAR(255)	Move notation for the move they made

Now, before we even make a move, we query this database to see if our representation of a board state is one that matches a board state in the database. If it does, then we know we have a win in two state and know which move to make. This database

responds extremely fast. A match in the database returns a result in 0.01 seconds. A non-match returns a result in less than that. Given the size of the database, and the relatively quick response times, this portion of the program can scale to a much larger database size with very little effect on the overall response time of our bot.

3.3 – Finding the Best Move Using the Whole Board

Now that our pieces are on the board, we must then decide how we are going to find the best possible move given a situation. To get the best possible move, we will take our current board state, expressed as a 64 character string, and try to match that board state with one in our database. The algorithm for finding the best move is as follows:

```
function findBestMove(currentBoardState)
        query database to find a match to our currentBoardState
                record if the user making the next move won the match
                record the user making the next moves rank
                record how many moves are left in the game
        if results  == 1 then
                make nextMove
        if results > 0 then
                if results have wins
                        discard results that result in losses
                make move by user with the highest rank
                if the users have the same rank then
                        make move resulting in less moves left
                        if movesLeft are the same then
                        make any move left in the result set
        if results == 0 then
                halfBoard(currentBoardState)
```

Figure 4 – Pseudocode for Matching on an Entire Board

This algorithm will either result in a next move or result in the calling of another function. Our rationale is that if we can find a board match, then we want to make sure that the move we are making results in a possible win. To further make the best move we will make the move done by the user with the highest rank. We can further separate the moves by how many plays are left giving us a move that results in a quicker finish. If there is no match in the database we receive a response in less than one second, which is good if we need more time to determine a move. The main response time of this function is mostly dependent upon the query in the beginning. The query that is used to get a response is as follows:

```
SELECT IF(side=winner,1,0) AS iswinner, IF(side='w' OR side='g', wrating, brating) AS rating, next_move FROM moves WHERE BINARY boardstatus='BOARDSTATUSHERE' AND (side='GOLDORSILVER' OR side='WHITEORBLACK') ORDER BY iswinner DESC, rating DESC, (maxturns-turn) ASC
```

To test the response time of this function we queried the database to find a common board state (the start state) on data sets containing subsequent 1/8th sets of data. The term negligible is used when the time program returns a run-time of less than 1 second.

Table 8 – Query Times for a Full Board State Match

Data Size	Response Time in Seconds
1993479	Negligible
3986958	Negligible
5980437	Negligible
7973916	Negligible
9967395	Negligible
11960874	Negligible
13954353	1.20
15947465	1.28

The noticeable problem that we have with matching full board states is the amount of hits that we actually get. Our response times, even at the largest amount of data, places us in a spot where we still have time left over out of our 30 second allotment. To take advantage of this leftover time, we will try to query a piece of the currentBoardState that results in more hits. We then have to observe how we can shrink the currentBoardState to get more results.

3.4 -Finding the Best Move Using Half the Board

In a majority of the moves that we make in our game, the focus will be on the middle four rows of our game board. Since this still gives us a worthwhile amount of games to reference while cutting the amount of the board we need to match by half, we can improve the likelihood that we receive a match. Therefore our algorithm for the

halfBoard function is as follows:

```
function halfBoard(currentBoardState)
        query DB to find a boardState that matches currentBoardState's middle 4 rows.
                record if the user making the next move won the match
                record the user making the next moves rank
                record how many moves are left in the game
        if results  == 1 then
                make nextMove
        if results > 0 then
                if results have wins
                        discard results that result in losses
                make move by user with the highest rank
                if the users have the same rank then
                        make move resulting in less moves left
                        if movesLeft are the same then
                                make any move left in the result set
        if results == 0 then
                areaOfConcentration(currentBoardState)
```

Figure 5 – Pseudocode for Matching on Half a Board

Our basic algorithm stays the same, with the major change being in the query to get a board that matches half a board. That query

```
SELECT IF(side=winner,1,0) AS iswinner, IF(side='w' OR side='g', wrating, brating) AS rating, next_move FROM moves WHERE boardstatus LIKE BINARY %s AND (side= %s OR side=%s) AND turn <> 1 ORDER BY iswinner DESC, rating DESC, (maxturns-turn) ASC
```

The response time on subsequent 1/8th sets of data of this query is as follows:

Table 9 – Query Times for a Half Board State Match

Data Size	Response Time in Seconds
1993479	Negligible
3986958	Negligible
5980437	Negligible
7973916	Negligible
9967395	1.3
11960874	1.36
13954353	1.42
15947465	1.8

There was an increase, but still manageable, response time even in our heaviest sets of data. This is a direct relation to the amount of hits we are receiving. We wish to increase our result set and still not violate our given allotment of time. Between a full board state and a half board state we have enough room to make one more query. This query will be called the area of concentration.

3.5 -Finding the Best Move Based on the Area of Concentration

After playing many games of Arimaa, it is noticed that areas of concentration develop where all the action occurs. This action is usually around the trap squares. The following algorithm explains how to develop an area of concentration and how to decide which move to make.

```
function areaOfConcentration(currentBoardState)
        note location of lastMovedPiece by the opponent.
        find location of the nearest trap square to the last moved piece.
        take a board state using the 5x5 grid where the trap square is the center.
        all locations of the board not in the 5x5 grid mark as don't cares

        query DB to find a boardState that matches the area of concentration
                record if the user making the next move won the match
                record the user making the next moves rank
                record how many moves are left in the game
        try
        if results  == 1 then
                Make nextMove
        if results > 0 then
                if results have wins
                        discard results that result in losses
                make move by user with the highest rank
                if the users have the same rank then
                        make move resulting in less moves left
                        if movesLeft are the same then
                                make any move left in the result set
        catch IllegalMoveException
        if results == 0 || IllegalMoveException then
                areaOfConcentration(currentBoardState, pieceToMove)
```

Figure 6 – Pseudocode for Matching on an Area of Concentration

This function will query our database to find a smaller version of the board state, the 5x5 area around the trap square that we believe to be the concentrated area of movement. The 5x5 area takes into account six squares of the 3x3 grid surrounding the two trap squares adjacent to the one we are interested in, and one square to the one that isn't. Therefore, if we are wrong about the trap square being an area of influence, we are still taking into account 2/5th of the area on each of two other trap squares. If we are completely wrong about the area of concentration we still receive data that could pose useful in our situation. Once again the majority of the algorithm stays the same, with the changes occurring in the query portion. This time we take into account certain "don't cares" in our instruction. The "don't cares" are wild card characteristics, where our initial assessment of the board does not care what is in these spots because they are too far out of the area of concentration to have an effect on our initial determination. We mark them such that the query that we use will be the same as before, but boardstatus will be changed such that we replace the area outside our area of influence with the MySQL single wild-card character '_'. The response time on subsequent 1/8th sets of data of this query is as follows:

Table 10 – Query Times for an Area of Concentration

Data Size	Response Time in Seconds
1993479	Negligible
3986958	Negligible
5980437	Negligible
7973916	1.8
9967395	2.3
11960874	2.48
13954353	2.68
15947465	2.8

This is the first function that we have to be careful on what our move set will be as it is possible for the piece that we are interested in receives a moveSet that takes it out of the area of concentration, thus into an area that we do not have information on. If this is the case, we then run the areaOfConcentration function again where the piece that we moved was the lastMovedPiece. This then gives us an appropriate view of the new area of influence and thus we can react with even more information. However, in the worst case, we receive a move set that is illegal. That is, the piece that we are supposed to move is asked to move to a space outside the area of concentration. In this case we will adopt a new function using the piece we are supposed to move as the center of our area of concentration.

3.6 – Finding the Best Move Using Areas of Concentration With a Specific Piece

In Arimaa, pieces are weighted according to the size of the relative animal that they represent. Therefore if we plan on moving a camel, we are not concerned with what pieces are in our way since a camel can move most of the pieces on the board. We can then use the following pseudocode to find a move.

```
function areaOfConcentration(currentBoardState, pieceToMove)
        note location of pieceToMove
        take a board state using the 5x5 grid where the trap square is the center.
        all locations of the board not in the 5x5 grid mark as don't cares.
        all friendly pieces will have their strength shrunk to rabbits.
        all opponent's pieces of lesser strength will have their strength shrunk to rabbits
        all opponent's pieces of greater strength will have their strength grown to elephants

        query DB to find a boardState that matches the area of concentration
                record if the user making the next move won the match
                record the user making the next moves rank
                record how many moves are left in the game
        try
        if results  == 1 then
                make nextMove
        if results > 0 then
                if results have wins
                        discard results that result in losses
                make move by user with the highest rank
                if the users have the same rank then
                        make move resulting in less moves left
                        if movesLeft are the same then
                                make any move left in the result set
        catch IllegalMoveException
        if results == 0 || IllegalMoveException then
                        fallback()
```

Figure 7 – Pseudocode for areaOfConcentration with Regards to a Single Piece

In all of our functions we are basically trying to match a board state that is similar or exact to the one that we encountered. When we reach this function we are essentially saying that we are approaching a point where our board state does not match any of those in our database at any level. It is because of this that we have decided to alter the way the board is seen so that we can give us a more adequate response set. First of all, we rarely have to care about which pieces on our side are which. A rabbit, cat or dog in the way of our path generally does not matter because, if need be, we can usually move that piece out of the way. Since it is mostly under our control, we will pretend that all of our pieces are rabbits. When we query for a match, we are essentially saying "if a friendly piece is on the board, I do not care what the strength of that piece is." Therefore all we truly care about is whether or not a square is occupied by a friendly piece and strength does not matter. This will improve our result set. The second piece that we wish to add to the equation is the strength of the opponent's pieces. If we are moving a camel, and an opponent has a piece in the way, then it does not matter if this piece is a rabbit, cat, or dog since our stronger piece can still control it. In this part of the function we either reduce a piece to that of a rabbit meaning that we are stronger than it, or increase it to that of an elephant showing that it is beyond our strength and therefore we can not manipulate it. This will then give us a better chance at receiving a hit from our query. We can alter our board state such that matching will occur based on a regular expression. Our code will therefore develop a regular expression based on the strength of the single piece we are looking to relate to. In doing so, our response times increased because of the new overhead.

Table 11 – Query Times for an AOC in Regards to a Specific Piece

Data Size	Response Time in Seconds
1993479	8.03
3986958	9.89
5980437	12.89
7973916	13.95
9967395	14.17
11960874	15.96
13954353	18.54
15947465	20.03

This is the longest of our queries and, in turn, becomes our most problematic. In our large data set we are brushing with the 20 second mark. Since we use 2.8 seconds from the previous function and 1 second for the others we are looking at close to 25 seconds used of our allotted 30 second time limit.

3.7 – The Fall Back

The duration of the final piece of our query may result in an issue with time. It is because of this that we decided to input our own fall back. Our own fall back function simply finds a random move and processes it. It is a last resort function that is only to be called when we are sure we are going to run out of time or if our best efforts for finding a move have failed. We wish to keep the calls to this function to a minimum so it is important that we find out how often our other functions get hits and how to improve.

Chapter 4

Analysis and Statistics

In this section we discuss the hit rates on our various functions and whether or not we have devised a successful approach. These statistics were gathered through 1,000 successful play throughs between our Bot and the current fall back two move search bot.

4.1 – Matches on Full Board

The first function we wish to know the success of is the one that tries to match on the full board. The first thing that we noticed was that after the first ten moves, the matching on a full board state became non-existent.

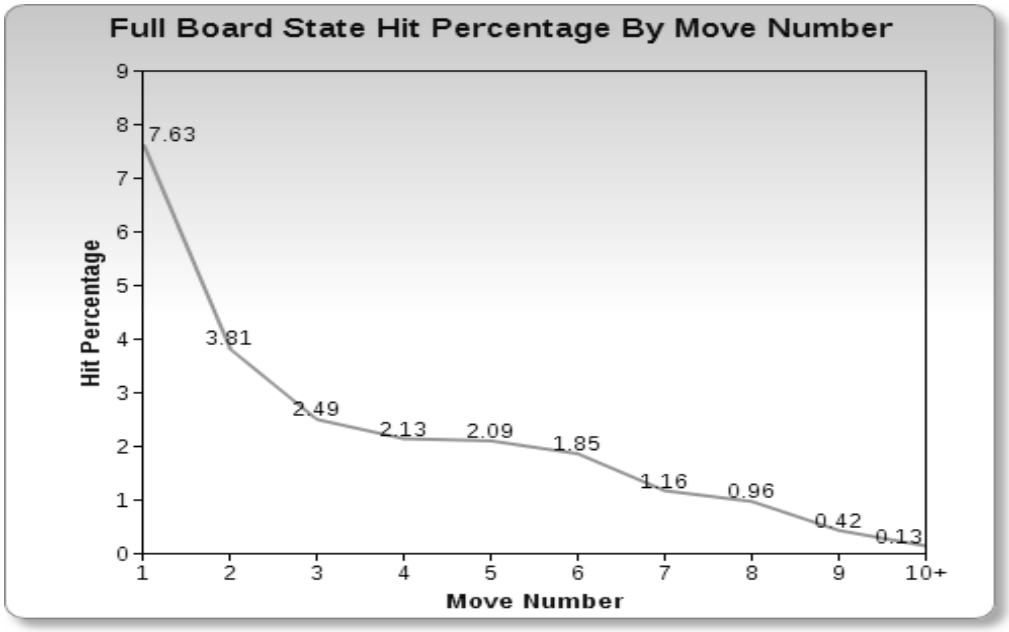

Figure 8 – Full Board State Hit Percentage By Move Number

The first move is the only time where a full board state match became a viable solution. Luckily, this function takes a negligible amount of time to run, especially in the case of not having a board state match.

4.2 – Matches on a Half Board

The second function, the half board approach, shows a bit of growth over the full board state function, but not enough growth as anticipated.

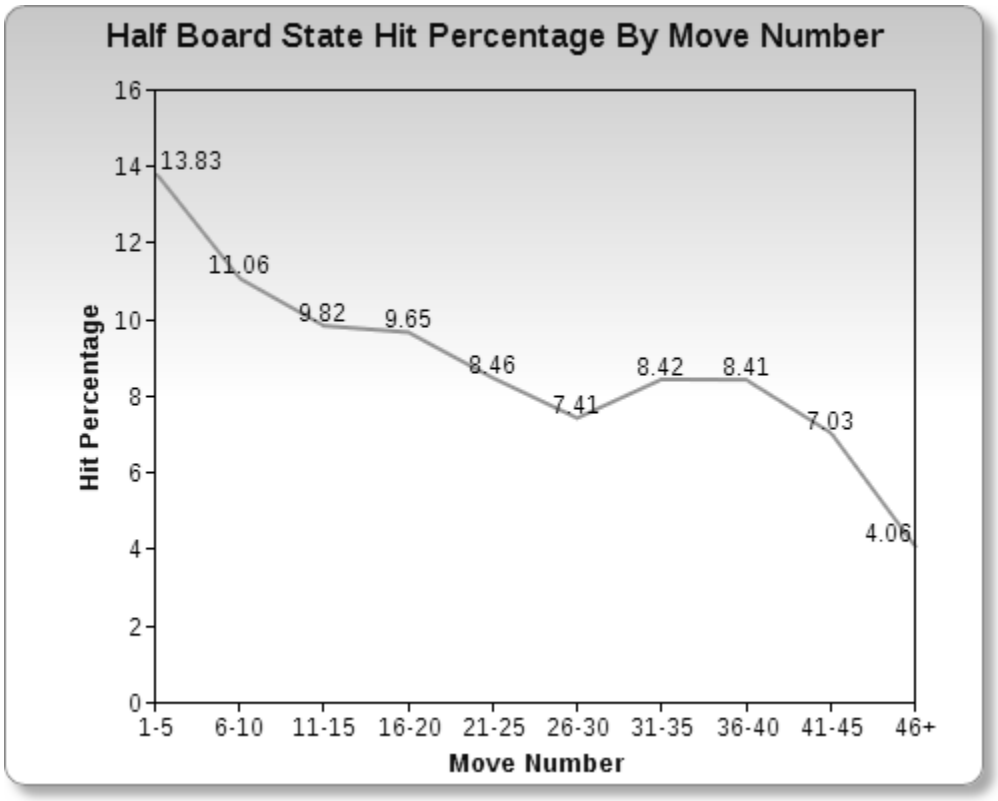

Figure 9 – Half Board State Hit Percentage By Move Number

We are still unable to achieve a higher than 15% match rate on any of our move numbers. The half board state does show a difference from the full board state in that there is not a full steady decline. From move number 26-30 to 31-35 there is a rise in the rate of hits. This may be due to the fact that as the game winds down more pieces become captured and therefore exact matches become easier to come by.

4.3 – Matches on an Area of Concentration

The area of concentration function ran around a relatively small area of the board where action generally takes place. By limiting the amount of space we need to match, our anticipation was that we would receive more hits in our database. We did receive more hits vs. the half board state, but still not up to our initial expectations.

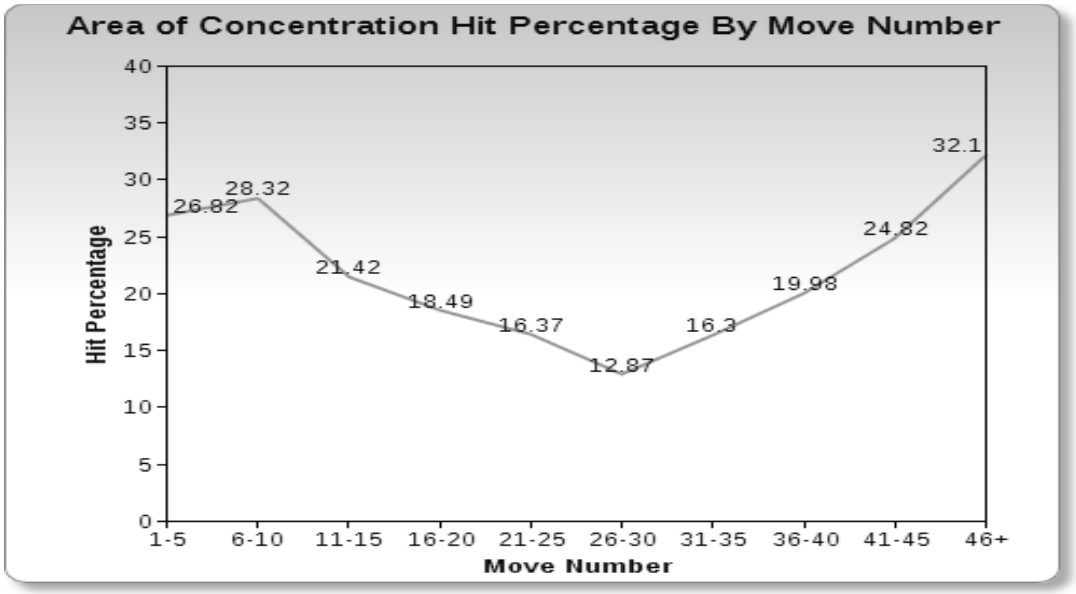

Figure 10 – Area of Concentration Hit Percentage By Move Number

The hit percentage starts out generally favorable but takes a sharp plunge in the middle of game resulting in a reduction of half the original rate. The good news is that by the end of the game the hit rate rockets up to 32.1%. The downside of this data is that most games do not reach this point in move totals, so the odds of reaping the benefits of this portion of the graph are dependent entirely on the ability to survive.

4.4 – Matches on an Area of Concentration With a Specified Piece

This is the last point that try to find a match on. In this instance we incorporate additional overhead due to not looking for exact matches in our query. The additional overhead should come with results, and in our example it did.

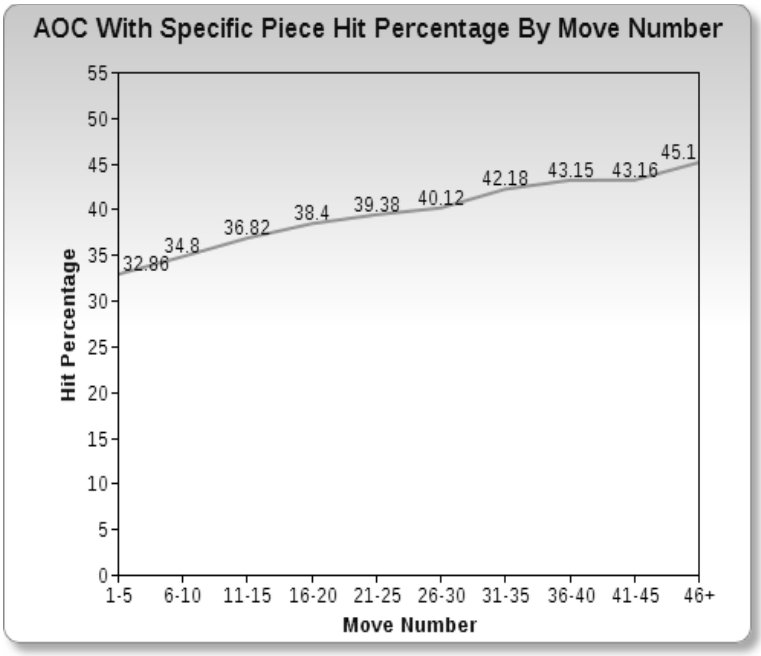

Figure 11 – AOC With Specific Piece Hit Percentage By Move Number

In this case we have a large improvement over our last graph. Our plot infers that as the game progresses we are more likely to find a match, though the hit ratio between the top and bottom is still fairly small (12.24%).

4.5 – Overall Odds of Finding a Hit

Given our data, we have developed the following table that incorporates the given data and shows the overall percentage of acquiring a hit on a board state if we run our entire function top to bottom. Our overall hit rate is the chance that we will get a hit in one of our functions. To calculate that we do the following:

A = Full Board

B = Half Board

C = Area of Concentration

D = Area of Concentration on a Single Piece

P(Just A) = P(A) * (1-P(B)) * (1-P(C)) - * (1-P(D))

P(Just B) = (1-P(A)) * (P(B)) * (1-P(C)) - * (1-P(D))

P(Just C) = (1-P(A)) * (1-P(B)) * (P(C)) - * (1-P(D))

P(Just D) = (1-P(A)) * (1-P(B)) * (1-P(C)) - * (P(D))

P(Only A, B, C, or D) = P(Just A) + P(Just B) + P(Just C) + P(Just D)

Overall Miss Rate – 1 – Overall Hit Rate

Therefore with that data we achieved the following results:

Table 12 – Hit Rate Calculations for Each Function Vs. Move Number

Move Number	Full Board	Half Board	Area of Concentration	Area of Concentration Single Piece	Overall Hit Rate	Overall Miss Rate
1-5	3.63	13.83	26.82	32.86	43	57
6-10	0.904	11.06	28.32	34.8	21.99	78.01
11-15	0.13	9.82	24.12	36.82	25.16	74.84
16-20	0.13	9.65	18.49	38.4	28.24	71.76
21-25	0.13	8.46	16.37	39.38	30.11	69.89
26-30	0.13	7.41	12.87	40.12	32.32	67.68
31-35	0.13	8.42	16.3	42.18	32.29	67.71
36-40	0.13	8.41	19.98	43.15	31.58	68.42
41-45	0.13	7.03	24.82	43.16	30.13	69.88
46+	0.13	4.06	32.1	45.1	29.34	70.66

Therefore in a given move number we are achieving a 30.416% average hit rate. This number is not as high as anticipated, though still high enough to hopefully prove the benefit of this solution against the old style solution. It is also noteworthy that the win in two puzzle function is absent from the table. This is because there were never any hits for the win in two puzzle function in our 1,000 games.

4.6 – Win Rate

Our data was collected in 1,000 games played. The following graph shows the win rate as we progressed through our 1,000 games.

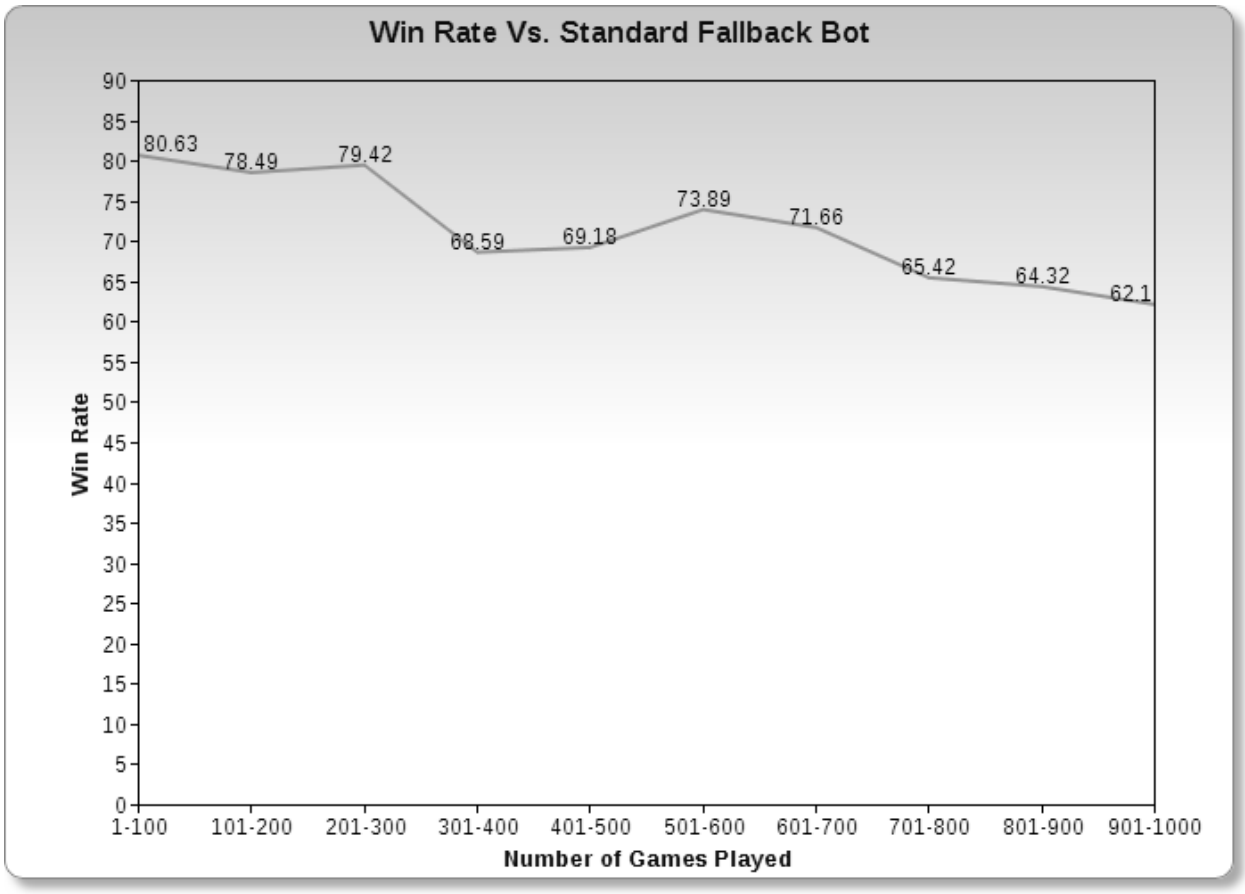

Figure 12 – Win Rate Vs. Standard Fallback Bot

As we played our games we mostly hovered above the 65% mark. This result is positive in that in a regular game of Arimaa, our bot has proven that it has the ability to defeat the standard fallback bot more than half the time.

Chapter 5

Improvements and Future Research

In this section we will discuss the improvements that can be made as well as goals in future research related to the ideas presented in this research.

5.1 – Better Win in Two Research

In our 1,000 games we did not use any puzzles in the win in two database. The amount of time it takes to find a win in two puzzle is negligible, but since we did not hit any of these it is possible that it is wasted space. What might perhaps be a better fit for the win in two database portion is a simple brute force win in two attack. That is, instead of finding current win in two matches, find out if the board state you are facing is a win in two puzzle at all, and then decide which move to make. This would deter from the idea of a 100% relational database approach, but it would result in a better move set than the current approach, as the current approach did not result in any moves.

5.2 – Full Board State Matches

The biggest detriment to our matching is the full board state match. It has a quick response time, but it mostly only results in a response in the first ten moves. After the first ten moves it becomes useless. Incorporating a move check to see what move number we are on to eliminate the full board function for moves 10 and on might be beneficial and might free up some more clock time for other functions.

5.3 – Better Area of Concentration Algorithm

The current area of concentration algorithm is fairly primitive and could stand for some improvement. Eliminating the possibility of returning a move set that can not be accomplished is imperative as it is wasted cycles since we must call the function over again. It is beneficial to consider determining if a move set is possible before it is returned, therefore we can continue querying the database inside the function rather than restarting it entirely.

5.4 – More Games!

The benefit of using the relational database is that games can be added very easily. We stopped with games almost a year before this research was handed in, therefore there is a whole years worth of games we could add, perhaps even adding games to the database as we play. To do so we would have to do some future research on the scaling of adding more games.

Chapter 6

Conclusion

The relational database fallback move generator that we developed based on this research shows a small improvement over the current fallback move generator with no detriment to the time period constraints of the game. In 1,000 games our bot did not once run out of its 30 second allotted time period, nor did it crash. This speaks to the robustness of our solution and the readiness of the implementation. With future research we may be looking at a viable alternative to be implemented in bots everywhere who have the resources to accommodate the database size.

List of References

[1] – Choksi, V., Ebrahim-Zadeh, N., & Mohan, V. (2013). Leveraging Game Phase in Arimaa. Stanford University, Stanford, CA. Retrieved August 12, 2014, from http://www.vivekchoksi.com/assets/game_phase_arimaa.pdf

[2] – Coulom, R. (2007). Computing Elo Ratings of Move Patterns in the Game of Go. Amsterdam, The Netherlands: ICGA Computer Games Workshop. Retrieved August 12, 2014, from http://remi.coulom.free.fr/Amsterdam2007/MMGoPatterns.pdf

[3] – Haskin, B. (2006, January 1). A Look at the Arimaa Branching Factor. Retrieved August 12, 2014, from http://arimaa.janzert.com/bf_study/

[4] – Kozelek, T. (2009). Methods of MCTS and the game Arimaa (Unpublished Master's dissertation). Charles University, Prague.

[5] – Syed, A., & Syed, O. (1999, January 1). The creation of Arimaa. Retrieved August 12, 2014, from http://arimaa.com/arimaa/about/

[6] – Why we love Arimaa. (December 21, 2011). In Why we love Arimaa. Retrieved August 12, 2014, from http://arimaa.com/arimaa/mwiki/index.php/Why_we_love_Arimaa

[7] – Wu, D. J. (2011). Move Ranking and Evaluation in the Game of Arimaa (Unpublished undergraduate dissertation). Harvard College, Cambridge Massachussets.