Plans, Patterns and Move Categories Guiding a Highly Selective Search

Gerhard Trippen

The University of British Columbia
{Gerhard.Trippen}@sauder.ubc.ca.

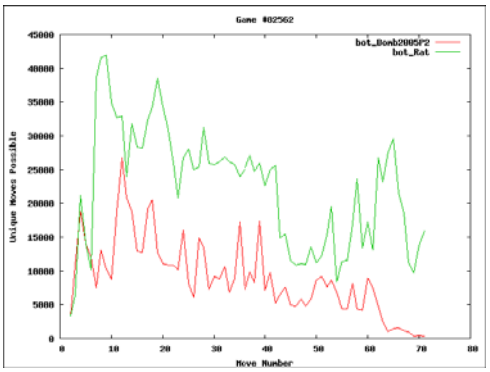
Abstract. In this paper we present our ideas for an Arimaa-playing program (also called a *bot*) that uses plans and pattern matching to guide a highly selective search. We restrict move generation to moves in certain move categories to reduce the number of moves considered by the bot significantly. Arimaa is a modern board game that can be played with a standard Chess set. However, the rules of the game are not at all like those of Chess. Furthermore, Arimaa was designed to be as simple and intuitive as possible for humans, yet challenging for computers. While all established Arimaa bots use alpha-beta search with a variety of pruning techniques and other heuristics ending in an extensive positional leaf node evaluation, our new bot, *Rat*, starts with a positional evaluation of the current position. Based on features found in the current position – supported by pattern matching using a *directed position graph* – our bot Rat decides which of a given set of plans to follow. The plan then dictates what types of moves can be chosen. This is another major difference from bots that generate "all" possible moves for a particular position. Rat is only allowed to generate moves that belong to certain categories. Leaf nodes are evaluated only by a very simple material evaluation to help to avoid moves that lose material. This highly selective search looks, on average, at only 5 moves out of 5,000 to over 40,000 possible moves in a middle game position.

1 Introduction

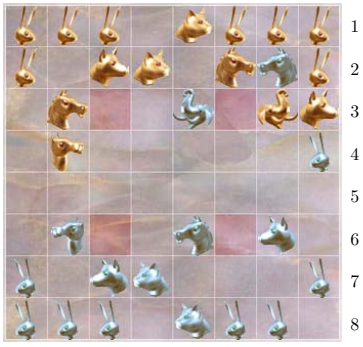
Arimaa is a modern board game designed to be difficult for computers [1]. It was invented by Omar Syed and Aamir Syed, and was motivated by the defeat of former world Chess champion Garry Kasparov by a Chess-playing computer developed by IBM called Deep Blue in 1997. Syed and Syed offer a prize of USD 10,000 until 2020 to the first person, company or organization to develop a program that can defeat three selected human players in an official Arimaa match [2].

Arimaa is a two-player partisan board game, i.e., Gold can only move gold pieces, and Silver can only move silver pieces. All information about a position is known to both players at any time. There are no chance moves involved, such as moves based on randomization generated by dice. The game can be played with a standard chess set. Each player has (in descending order of strength) an

elephant, a camel, two horses, two dogs, two cats and eight rabbits. The letters representing those pieces in the game notation are e, m, h, d, c and r, for the silver pieces. For the gold pieces we use capital letters. The game is won by moving a rabbit to the *goal* rank, which means to bring it to the opponent's base row, but it can also be won by immobilizing all of the opponent's pieces, or by capturing all his rabbits.



(a)



(b)

Fig. 1. (a) Graph of possible number of unique moves, generated with Brian Haskin's (aka Janzert) Game Grapher [3]. The peak moves possible for Silver was on turn 9 with 41939 moves possible. **(b)** Position on turn 9 for Silver in game #82562.

One major aspect that makes the game difficult for computers is that a player is allowed to use up to four steps in a single turn. A step consists of moving a piece to an adjacent square. It is also possible to push or pull an opponent's piece with a stronger piece. For example, in Fig. 1(b) the gold elephant could move from b3 to b4 pulling the silver horse from b2 to b3. Thus, a push or pull requires two steps. In a single turn players can move up to four different pieces, and this generates a huge number of possible moves. There are about 2,000 to 3,000 moves possible in the first turn depending on the way a player chooses to set up his pieces, and during the middle game the number of possible moves ranges from about 5,000 to over 40,000 (see also Fig. 1(a)) – compared to an average of about 30 for Chess.

The 4-step moves make it very difficult for computer programs to perform a deep search. Looking ahead only two moves for both players (= four plies) means to search to a depth of 16 steps. In the Arimaa Computer World Championship and in the Arimaa Challenge against humans, the time per move is restricted to two minutes. Simple alpha-beta bots are currently not able to reach this depth under the tournament settings nor in any reasonable amount of time. Even 12 steps might already be too deep. Only through extensive pruning, a variety of other heuristics, and quiescence search, are bots able to search deeper to find better moves.

To see whether bots perform better if they are given more time or are allowed to search deeper unrestricted by time, a P3 (= 3 plies = 12 steps) version of David Fotland's former Arimaa World Champion bot *Bomb* was available for a short while. However, the average move time in most of the games played was over 30 minutes, and at times the bot needed hours to find a move. Although it is basically impossible for a human to accurately predict the next 20 steps, it is still reasonable to look ahead three moves and get a good idea of the resulting position. Also the very simple bot ArimaaScore – which only needs about 1 second per move when playing at the P2 (= 2 plies) level – needs, on average, 5 minutes per move when playing at the P3 level.

The non-defined initial setup of the pieces is another problem for bots. While Chess programs can resort to a huge opening database, the initial setup of the 16 pieces is not fixed in Arimaa, which makes it very difficult to generate an opening database. Endgames of the kind found in Chess, in which only a few pieces remain, are also very rare in Arimaa. In many games a great number of pieces are still on the board when a rabbit reaches the goal.

A final problem area for computers are captures. Captures are performed not by moving onto an opponent's square like in Chess, or by jumping, as in Checkers. Instead, if a piece lands on a "trap" square (there are four of them: c3, f3, c6 and f6 (see Fig. 1(b)), and there are no pieces of the same color on any of the four squares adjacent to the trap, then the piece is captured and removed from the board. So, while in Chess a piece might be captured in a single move by any stronger or weaker piece, in Arimaa often a weaker piece must be pushed and pulled towards home traps (c3 and f3 for Gold) by a stronger piece, where it can finally be captured. More details of such a plan will be given in Section 2.

Several research papers presenting bots have been published. David Fotland presented his World Champion Arimaa Program at the Computers and Games Conference in 2004 [4]. Haizhi Zhong and Christ-Jan Cox both wrote a Master's Thesis on Arimaa several years ago [5, 6].

All the bots described above, and other well-established Arimaa bots, use an iterative deepening alpha-beta search with a variety of pruning techniques and other heuristics ending in an extensive positional leaf node evaluation. Quiescence search and other enhancements add to the strength of those programs.

Upper Confidence bounds applied to Trees (UCT) [7] and related research (e.g., [8, 9]) have shown great success in the domain of the classical board game Go. Several members of the Arimaa community have discussed the usefulness of some of these approaches for Arimaa [10]. Jeff Bacher, the programmer of the current Computer World Champion program *clueless*, is one of several people who also implemented or started to implement an UCT bot. Currently, it seems the traditional alpha-beta bots are still more successful. Pure random playouts seem not to be useful. In games where one player is missing an elephant (the strongest piece) and another player is missing a rabbit, the higher percentage is won by the party with the missing elephant. This contrasts with our intuition that the player with the elephant should win most of the games.

Our new bot Rat follows an approach different from UCT and from the traditional alpha-beta search. Rat follows a more human way of thinking by first analyzing the position, finding suitable plans, and then trying a certain highly selective number of moves. A tactical search, alpha-beta with some search extensions, ensures that the moves do not lead to too great a loss of material, by evaluating the leaf nodes with a very simple evaluation function considering only the material of both players. A conceptually similar approach, the Technology Chess Program, was presented by James Gillogly nearly 40 years ago [11]. Jonathan Schaeffer presented a related approach, Planner, that determines a long-range strategy based on an assessment of the current position, and makes moves in the short-term that are consistent with a long-term objective [12].

However, the details of our bot Rat are different and the main concepts are explained in this paper, which is organized as follows. Section 2 introduces an often-used strategy (especially against Bomb and other bots) known as the *elephant-horse attack* (EH attack). Bomb and some other bots, or earlier versions of them, try to take the horse hostage close to their own trap. However, this often leads to decentralization of the elephant, and if the bot plays very passively afterwards, then kidnapping and capturing of several of the bot's pieces might be possible. We will call this *flash-kidnapping*. Section 3 shows how our bot "thinks", i.e., how Rat makes a positional evaluation of the current position and prioritizes plans which result in an ordered list of moves. We also describe move categories that drastically limit the number of possible moves for any given position. Section 4 explains the use of *directed position graphs* for pattern matching by giving an example graph for the EH attack. Section 5 gives a brief overview of the general performance of our bot. The paper concludes with ideas for future research in Section 6.

2 Elephant-Horse Attack, Horse Hostage and Flash-Kidnapping

To motivate the idea of using plans we present a common strategy against bots in this section before we discuss details of the implementation of our bot in the next section. One strategy for attacking an opponent's trap is to advance the elephant together with a horse on the same wing. We illustrate the strategy from the view point of Gold. To attack the c6 trap the gold elephant should be positioned on d6, keeping it close to the center and able to switch quickly to the other wing if necessary. The gold horse should occupy b6 so that the trap will be enclosed from both sides. Often Silver's elephant returns to c5 to protect the trap, and even more often the elephant might push Gold's horse to a6 or b7 to take it hostage. This is exactly the position our bot Rat is waiting for to follow the flash-kidnapping strategy (explained below). However, before starting this strategy it is important for Gold to secure its own traps. Gold's Southeast (f3) trap will be guarded by the camel on g3. In this way Silver's smaller pieces will not advance on the East wing. Also the Silver's camel does not see a target on

this side and rarely starts an attack there. Gold's Southwest (c3) trap will be protected by a horse and a dog. (See also Fig. 1(b) with switched colors.)

After creating this horse hostage situation Rat tries to follow a strategy that we call *flash-kidnapping*. The whole sequence takes a maximum of 40 steps, i.e., 10 plies, given the "cooperation" of the opponent. With the great branching factor of Arimaa – on average there are over 17,000 possible moves in any given position [2] – it is not expected that a bot will be able to detect such a sequence without having knowledge of this plan.

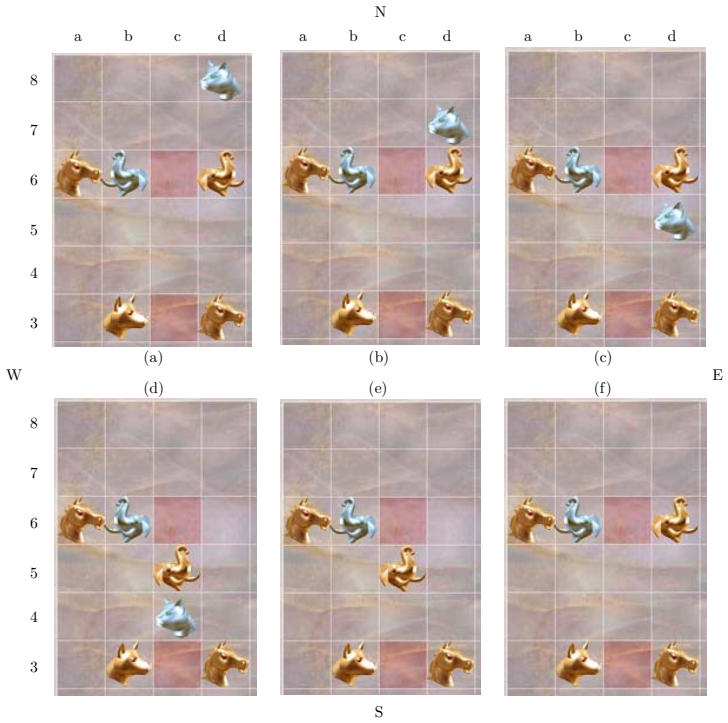


Fig. 2. Flash-kidnapping takes 10 plies (= 40 steps).

Figure 2 (a) shows the initial situation without the East wing. In the first move Gold pulls a Silver victim closer to her elephant (see Fig. 2 (b)). The intended capture of this victim lies still 6 plies ahead, so it is basically impossible for Silver to see. However, many bots try to avoid a situation where a weaker piece stands next to the opponent's elephant. If they do not, then in the next move Gold's elephant can flip the victim from d7 to d5 by first pulling it and then pushing it south (see Fig. 2 (c)). The advantage of this flip is that the elephant is back on d6 building a barrier against other Silver pieces that might want to help and free the victim, because now the intentions of Gold's elephant become quite clear. But the capture still lies four plies ahead, so it remains

difficult for Silver to detect. So the elephant might still hold the horse hostage. A double push brings the victim to c4 (see Fig. 2 (d)). Saving the victim with its elephant is often not desirable for Silver because it leaves the Northwest trap (c6) too weak. So in the next move Gold can capture the victim (see Fig. 2 (e)). And another move later, Gold's elephant can return to d6 to repeat the whole procedure (see Fig. 2 (f)).

3 Plans and Move Categories

Generally, alpha-beta bots generate a huge variety of possible moves in a given position and search recursively down to a certain depth at which the position will be evaluated. Our bot Rat imitates a more human approach. Before generating any moves, a positional evaluation of the current position is performed, which determines which plans Rat should follow next. Thus, rather than generating all possible moves in any given position, Rat uses a highly selective search. Before UCT became very popular in the recent years, many Go programs generated moves and evaluated them to find good candidate moves [13], because for Go, a brute force search approach seemed unpromising. We follow a similar approach. In the following we will give a detailed explanation of Fig. 3.

3.1 Positional Evaluation and Plans

Algorithm Positional Evaluation The positional evaluation starts with the location of certain pieces. Trap control is particularly important for Arimaa. Based on the location of pieces and the number of pieces on the board Rat also tries to identify the game phase (opening, middle game, endgame). The priority of the plans is given through the structure of the evaluation function. This means there is a hard-coded order given through our implementation. Moves belonging to certain plans will be appended to a singly-linked list. However, while going through the function and collecting additional information flags will be set that lower the priority of plans, and plans might be added later instead. While this is not done in our actual implementation, the list of plans could be implemented through a priority queue.

The first plan considered is a 4-step goal search, because if a player's rabbit reaches the goal he wins the game. However, this plan will not be added to the list if rabbits are not close enough to the opponent's home rank. Next, captures are considered, but if the opponent has the possibility of capturing a stronger piece than we can, defenses are tried first. Therefore, we first look at the opponent's possible captures to see which pieces or traps must be defended to avoid captures. Based on the outcome of both our possible captures and the opponent's, a decision is made which of the two plans (capture or defense) should be appended to the list first. After this, plans based on the outcome of the pattern matching (see Section 4) are added. Currently, only one strategy is considered, the flash-kidnapping strategy described in the previous section. Here a great number of additional strategies could be added by generating more directed

Algorithm Alpha-Beta Search
  Input: plans tried
    and usual parameters like position, $\alpha, \beta, \ldots$
  Output: move and score
  If depth ≥ 2 **and** quiet **then**
    score = simple eval; return score;
  Q = Positional Evaluation
  in the usual alpha-beta loop
    dequeue next plan;
    Generate Move (plan) and make it;
    recursive call;
    compare score to bounds and update;

Algorithm Generate Move
  Input: a plan
  Output: a move
  generate all moves in a move category
    corresponding to the plan;
  sort the list of moves;
  return best move;

Algorithm Positional Evaluation
  Input: plans tried
    and parameters like position, depth, ...
  Output: ordered list of plans, **Q**
  Q += Add Plan(goal, ...)
  Q += Add Plan(capture, ...)
  If depth < 2 **then**
    Q += Add Plan(pattern matching, ...)
    Q += Add Plan(frames, ...)
    continuing with frames, hostages, forks,
    camel hunt, retreats, trap defenses,
    goal preparations, flips, basic attacks,
    clearing of traps, elephant centralization

Algorithm Add Plan
  Input: a plan to consider and plans tried
    and position, its features, ...
  Output: a plan
  If all conditions for plan met
    then return plan; **else** return null;

Fig. 3. Main functionality of algorithms used to find a move.

position graphs that we use for pattern matching. In the order given below, the evaluation function adds the following plans to the priority list: frames, hostages, forks, hunt the camel, retreats, trap defenses (if not necessary earlier because of possible captures), goal preparations, flips, basic attacks, clearing of traps, and finally elephant centralization. This order is mostly fixed although some of the plans might not be added depending on the current position (see Algorithm Add Plan) or some flags might switch some of the priorities. Further refinement is certainly necessary here. However, this sequence of basic ideas will be used in particular if no other plan can be found through pattern matching. We believe that the pattern matching module will have the greatest impact on improving the playing strength of our bot. We have tested this only for the flash-kidnapping strategy so far.

Algorithm Alpha-Beta Search Rat performs an alpha-beta search to a depth of 2 plies. In the first call the function will generate a sorted list of plans for the player starting with an empty list. Similarly to the Chess program PARADISE presented by David Wilkins [14], Rat considers which plans have been tried (and possible refuted) earlier in the search. This helps further reduce the branching factor in the tree search. However, the nature of Arimaa with four steps per move requires the player to be able to work on different plans within the same move. For example, the first two steps might be used to defend a home trap while the next two steps might be used to attack an opponent's trap. Therefore, all plans that have not yet been tried in a higher level of the tree are considered in a recursive call. When a recursive call of the alpha-beta search reaches the opponent, Rat will use the same positional evaluation function to generate the opponent's list of plans. Quiescence search will be performed in case of captures

only. The alpha-beta search has a very simple leaf evaluation function. It checks whether a rabbit reached a goal, and it calculates the value of the material on the board. Only if a plan leads to higher material gain (or lower material loss) than a plan that appears earlier in the ordered list, is this plan chosen over the other plan. The search ends after a certain time limit is reached.

To summarize the generation of plans, we can say that Rat is much more limited than, for example, PARADISE and Jacques Pitrat's Chess combination program [15] that use production rules to produce plans. Rat knows only a certain set of plans embedded in the positional analysis. The order is mostly fixed although a number of flags and decisions based on features found in the position can lead to a certain reordering.

3.2 Move Categories

As described above, the plans are assigned different priorities and this yields an ordered list of plans, which itself translates into an ordered list of move categories for which actual moves must be generated.

Rat knows only a very limited number of move categories including goals, rabbit advancements, captures, retreats, kidnappings, attacks, and some trap protections corresponding to the plans mentioned above. Most of the trap protections and some of the trap attacks are implemented through patterns. This means Rat checks whether it can reach a certain local pattern from a given position. Rat knows a few hundred of these local patterns.

Algorithm Generate Move When Rat generates moves of a certain category, all moves of this type are listed and immediately evaluated. Generally speaking, the overall resulting position is not evaluated, but rather the location of the moved pieces. For example, in the category rabbit advancement, the closer a move brings a rabbit to its goal the higher the score this move will receive. Often, trap defense should use stronger pieces to prevent the opponent from surrounding the trap by simply pushing weaker pieces aside, and furthermore, to avoid exposing weaker pieces to the opponent. For example, a cat or a dog on the side of a trap could easily become a victim of a horse that pulls it to the opponent's half of the board. Captures also include the value of the piece captured to determine the ranking. Only the top three moves in each category are considered strong enough, and will therefore be used as possible candidates in the alpha-beta search. This can also be restricted to the best move.

3.3 Two Other Details of the Implementation

One of the key facts making Arimaa difficult for computers is the use of 4-step moves. It was mentioned in Section 3 that it might be necessary or useful to split the four steps to follow two different plans. In particular, a fourth step that is not necessary to follow a certain plan involving an attack of an opponent's trap might be used to help protect a home trap. This fourth step may not be

necessary, but is used as a fill-in step considering that the home trap might be under attack later in the game. Usually, there are several possibilities for such a step and this number increases considerably if there are two steps left for the trap protection plan. Including these fill-in steps in the regular alpha-beta search can make the tree much wider than necessary. If the trap is not under attack and in need of immediate defense we do not add the plan of trap protection into the alpha-beta search. Instead, we add a quick search after our main-line has been determined and fill in some trap protecting moves afterwards.

A note about time management. The alpha-beta search is used only to confirm that certain moves that follow a particular plan do not lead to loss of material, or that our bot Rat can indeed win material, for example. The priority of the plans is solely determined by the positional evaluation. Therefore, if the bot has tried at least one plan already, i.e., has searched the whole subtree with the opponent's responses, and none of them could refute the plan, then Rat will end the search earlier if that helps accumulate reserve time.

4 Pattern Matching with Directed Position Graphs

Within the positional evaluation of the current position, Rat additionally uses pattern matching to determine the most promising move ordering. To store the positions, we use a data structure very similar to a *directed acyclic word graph* (DAWG). A DAWG is a trie, i.e., prefix tree, but it not only eliminates prefix redundancy but also suffix redundancy. This data structure is very space efficient and lookup time is proportional to the length of the search string. We use a *directed position graph* (DPG), which is allowed to be cyclic. We further reduce the required space by eliminating infix redundancy.

When a human player studies an Arimaa position, the exact location of minor pieces like dogs, cats and rabbits is often not considered important if they are in a quadrant that is currently not under attack or involved in a plan. Those are the pieces that Mikhael Botvinnik might call Type III or Type IV pieces [16]. Therefore, when trying to match a position we focus on the most important features first. As an example of how to use DPGs for pattern matching we built a DPG for the horse hostage setup and the ensuing flash-kidnapping. It consists of approximately 250 nodes. A simplified version is shown in Fig. 4. The DPG will be searched using breath first search (BFS).

First, when playing Gold, the location of Silver's elephant is matched. Based on this, we learn whether we actually need to mirror all the following locations, because although we always refer to the c6 trap, the hostage situation could also take place at the f6 trap. Next, we try to match the location of Gold's elephant. Then we check whether Gold's horse is held hostage by Silver's elephant. If Silver's elephant stands on b6 (eb6), then possible locations for Gold's horse are a6 or b7 (Ha6 or Hb7). Also, Gold's elephant could stand on d6 or c5 (Ed6 or Ec5) (see also Fig. 2), and we would still recognize this position as horse hostage. This means that in our DPG directed edges can lead from eb6 to Ed6 and Ec5, and then from both of those to both Ha6 and Hb7 (see Fig. 4). Thus,

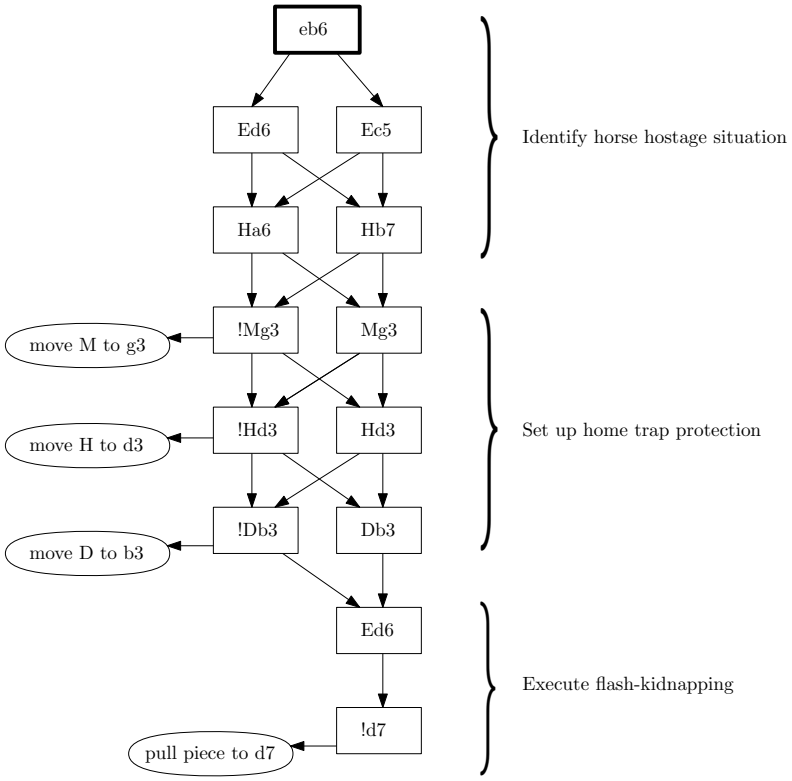


Fig. 4. Simplified DPG for flash-kidnapping strategy: When the BFS reaches a round node, this move will be added to the move list. Capital letters are used for Gold, small letters for Silver. Piece types are given using the capitalized bold letter in the words **E**lephant, ca**M**el, **H**orse, **D**og, **C**at, **R**abbit. A ! notation means no piece a of certain type if the type is specified, otherwise no piece at all.

we have four different combinations that we would identify as horse hostage situation. This is greatly simplified compared to the actual DPG that Rat is using. Before starting the flash-kidnapping, the home traps should be sufficiently secured by camel, horse and dog. So, from the nodes Ha6 and Hb7, edges are leading to a node Mg3. However, there are also edges leading to a node !Mg3, which indicates that Gold's camel is not on g3. A child of this node is then a leaf with a command to move the camel to g3. Both Mg3 and also !Mg3 lead to Hd3 and also !Hd3, while the latter has a child commanding a horse to d3. Because we search the DPG using BFS we can see that it is more important to bring the camel to g3 than to bring the horse to d3. We continue down with Db3 and !Db3. The positions of these three pieces form eight possible infixes. The logical sequence of first recognizing horse hostage, then setting up camel, horse and dog, and finally executing the flash-kidnapping suggested that we should match the location of Gold's elephant once more to determine which move should be

added to the list of candidates. This also greatly simplifies the task of editing the DPG. Furthermore, before actually suggesting to position camel, horse and dog as described above, we must ensure that Gold's elephant is protecting the c6 trap.

As we use BFS to traverse the DPG, the DPG does not need to be acyclic. Indeed in some situations we might prefer the camel to stand on e3 instead of g3. So in our full DPG we also have nodes Me3 and !Me3. An edge is leading from !Mg3 to !Me3 and vice versa. The camel might not be able to reach g3 in one move (or remaining steps thereof) because it is too far West. But it may be able to at least reach e3, moving it closer to its intended destination.

5 Experimental Results

The Arimaa website [2] allows human players as well as bots to choose from a huge number of bots to play against. Most participating bots from former Computer World Championships are available with a wide variety of settings; restricted in search depth as P1 or P2, or playing with a certain time limit, for example, Blitz and Fast; or also with the original Computer Championship (CC) setting (unrestricted search depth, 2 minutes per move). Rat has beaten several of them. Most of the P1 bots and several of the P2 bots can be beaten regularly. Rat can also beat generally stronger bots, even if they play in the CC setting, if these bots take the horse hostage and Rat can use the pattern matching to follow the flash-kidnapping plan. Currently, no UCT bots are regularly available as they are still under development or have been considered too weak by their developers to participate in the Championship. Therefore, our results are restricted to the common alpha-beta bots. Overall, Rat is still a weak bot and cannot play well in many situations because the knowledge of how to play or what plan to follow in these positions is simply missing. We hope that by adding more DPGs, i.e., "teaching" more strategies, Rat or any other bot could be transformed into a stronger bot.

6 Future Work

Bot Rat is the first Arimaa bot that strictly uses move categories. A positional evaluation of the current position prioritizes plans and leads to the ordering of move categories and determines which of those can be generated. Rat performs a highly selective alpha-beta search with only a material evaluation at the leaf node level. Traditional Arimaa bots generate a huge number of possible moves – many of which a human would never consider at all – and then evaluate the leaves with a complex evaluation function. Due to the high branching factor of Arimaa, even with a great amount of pruning, only a depth of at most 20 steps in search extensions seems reachable with current computers. Many strategies require looking much further ahead.

So far we have only implemented one single strategy to demonstrate our approach. Many more strategies could be added to create a stronger program.

Knowledge of various strategies can be added using space-efficient directed position graphs (DPG). A plain text DPG can be edited easily without actually modifying the source code of the program. While we used a DPG to follow a certain strategy in the middle game, DPGs could also be used in the opening and in the endgame.

This exploration delved forty years into the history of chess programming in the hope that these approaches might lead to more successful Arimaa bots than the popular alpha-beta searchers. However, further examination is needed to determine whether this approach or other useful techniques, like UCT for example, could help go beyond the strength of these.

Acknowledgments We would like to thank the anonymous referees for their careful review and incisive and encouraging comments. We greatly appreciate their constructive criticism that helped to improve the paper.

References

1. Syed, O., Syed, A.: Arimaa - a new game designed to be difficult for computers. International Computer Games Association Journal **26** (2003) 138–139
2. Syed, O.: Arimaa - the next challenge. http://arimaa.com/arimaa/ (2009)
3. Haskin, B.: Arimaa game graphs. http://arimaa.janzert.com/gamegraphs (2006)
4. Fotland, D.: Building a world champion Arimaa program. CG 2004, LNCS **3846** (2006) 175–186
5. Zhong, H.: Building a strong Arimaa-playing program. Master's thesis, University of Alberta, Dept. of Computing Science (September 2005)
6. Cox, C.J.: Analysis and implementation of the game Arimaa. Master's thesis, Universiteit Masstricht, Institute for Knowledge and Agent Technology (March 2006)
7. Kocsis, L., Szepesvari, C.: Bandit based Monte-Carlo planning. 15th European Conference on Machine Learning (2006) 282–293
8. Coulom, R.: Computing Elo ratings of move patterns in the game of Go. ICGA Journal **30**(4) (December 2007) 198–208
9. Chaslot, G., Winands, M., Bouzy, B., Uiterwijk, J.W.H.M., van den Herik, H.J.: Progressive strategies for monte-carlo tree search. In Wang, P., ed.: Proceedings of the 10th Joint Conference on Information Sciences, Salt Lake City, USA (2007) 655–661
10. Syed, O.: Arimaa forum. http://arimaa.com/arimaa/forum (2009)
11. Gillogly, J.J.: The technology Chess program. Artificial Intelligence **3** (1972) 145–163
12. Schaeffer, J.: Long-range planning in computer Chess. In: ACM 83: Proceedings of the 1983 annual conference on Computers : Extending the human resource, New York, NY, USA, ACM (1983) 170–179
13. Müller, M.: Computer Go. Artificial Intelligence **134**(1-2) (2002) 145–179
14. Wilkins, D.: Using patterns and plans in Chess. Artificial Intelligence **14** (1980) 165–203
15. Pitrat, J.: A Chess combination program which uses plans. Artificial Intelligence **8** (1977) 275–321
16. Botvinnik, M.M., Brown, A.: Computers, Chess and Long-Range Planning. Springer-Verlag New York, Inc., Secaucus, NJ (1970)