Tilburg centre for Creative Computing
Tilburg University
http://www.uvt.nl/ticc

P.O. Box 90153
5000 LE Tilburg, The Netherlands
Email: ticc@uvt.nl

June 15, 2009

TiCC TR 2009–004

Artificial Intelligence for the Game of Frank's Zoo

Loes Braun
MICC, Maastricht University

Pieter Spronck
TiCC, Tilburg University

Abstract

In this report we discuss the implementation of artificial intelligence to play the modern card game 'Frank's Zoo.' Frank's Zoo is a deterministic game with a high amount of imperfect information. In the present research we compare a heuristic AI, based on the authors' knowledge of the game, to several implementations of Monte-Carlo based AIs, and to randomly playing AIs. We used two versions of the game, namely the published version, and a much simplified version, which is less interesting to play but easier to analyse. From the results of our experiments we conclude that the heuristic AI is by far superior to the Monte Carlo AI, while the Monte Carlo AI is superior to the random AI. Furthermore, we conclude that the performance of the heuristic AI is positively influenced by an increasing amount of incomplete information. We also developed and tested a move predictor. From the results of the experiment performed with the move predictor, we conclude that it performs surprisingly well, and that it may be a powerful tool in creating opponent models.

Artificial Intelligence for the Game of Frank's Zoo

Loes Braun
MICC, Maastricht University

Pieter Spronck
TiCC, Tilburg University

1 Introduction

The present document describes the results of our research into artificial intelligence (AI) for the game of Frank's Zoo, performed in the period November 2006 to August 2007. The research is part of the project Rapid Online Learning for Entertainment Computing (ROLEC), funded by NWO under grant number 612.066.406.[1] The goal of the ROLEC project is to design methods that allow computer-controlled agents in commercial computer games to become more intelligent by online learning from earlier experiences.

The main track of the ROLEC project focuses on commercial video games, in particular Real-Time Strategy games [1, 2, 13, 14] and Role-Playing Games [16, 18]. The main obstacle encountered in the research is the large amount of imperfect information and non-determinism encountered in such games. Since two-player perfect-information deterministic board games do not share this obstacle, the decades of research invested in classic board games cannot be applied to the ROLEC research.

Or can it?

We noticed that modern board games, such as Settlers of Catan and Puerto Rico, share many features with both classic board games and modern video games. Incomplete information and non-determinism form essential elements of their gameplay, just as in modern video games, and they are often played with more than two players. However, tactics employed by humans to play modern board games have much in common with the tree-based searches that are used for classic board games. Would it be possible to use modern board games as a bridge between the bulk of research into artificial intelligence for classic board games and our ROLEC research into modern video games? We decided to spend a sidetrack of the ROLEC project to investigate this question. In particular, we chose the modern card game of Frank's Zoo, and aimed to answer the following question: to what extent is it possible to apply artificial intelligence techniques from classic computer games to design artificial intelligence for Frank's Zoo?

We compared two approaches. The first is a heuristic approach, which is used for many classic board games. The second is a Monte Carlo approach, which is also used for many classic board games [8], in particular those with some element of non-determinism. We found that the heuristic approach gives much better results than the four Monte Carlo approaches that we implemented. We believe this difference in results is because the Monte Carlo techniques suffer from the imperfect information inherent in Frank's Zoo. It is, however, possible to reduce the amount of imperfect information by opponent modelling techniques [3, 4, 5, 6, 7, 10, 11, 12, 15]. We added some preliminary work in this respect by implementing a move predictor.

The outline of this document is as follows. Section 2 explains the rules of Frank's Zoo. Section 3 describes the heuristic AI implemented for Frank's Zoo. Section 4 describes four different Monte Carlo approaches to the AI. Section 5 compares the results of all implemented artificial intelligences, both on the game of Frank's Zoo and on variants of the game with different card sets. Section 6 describes the

[1] At the time that this research started, both authors were affiliated to the Maastricht ICT Competence Centre (MICC) of Maastricht University. Since then, after receiving her PhD in Computer Science, Loes Braun left to study medicine, while Pieter Spronck moved to the Tilburg centre of Creative Computing of Tilburg University, where he continued and completed the ROLEC project.

Figure 1: All the different cards in the Frank's Zoo card set (including two card backs). Each card depicts an animal, and above it its predators. Pictures are by Doris Matthäus of 'Doris and Frank.' Photo by Francisco Rueda García.

move predictor, and the first experimental results achieved with it. Section 7 concludes, and Section 8 outlines directions for future research.

2 Gameplay

Frank's Zoo is a fairly simple card game for four to six players.[2] It uses a deck of 60 cards. Each card depicts an animal and its predators. In total, the deck contains 13 different types of animals (depicted in Figure 1). The animals, the numbers of cards per animal and their predators are summarised in Table 1.

Definition 1 (Card types) *In the standard Frank's Zoo deck, there are three types of cards, as described below.*

- Normal card*: A normal card is a card with no special privileges during the game.*
- Special card*: A special card is a card which may be used in two ways. First, it may be used as a normal card. Second, it may be used to replace a specific normal card.*
- Joker card*: A joker card is a card which may be used to replace any other card.*

[2]There are rules for three and seven players, but the game plays best with four to six.

Table 1: Types of animals, number of cards, and their predators in the Frank's Zoo deck.

			Predator												
		# cards	Whale	Elephant	Crocodile	Bear	Lion	Seal	Fox	Perch	Hedgehog	Fish	Mouse	Mosquito	Joker
Prey	Whale	5													
	Elephant	5											•		
	Crocodile	5		•											
	Bear	5	•	•											
	Lion	5		•											
	Seal	5	•			•									
	Fox	5		•	•	•	•								
	Perch	5	•		•	•		•							
	Hedgehog	5							•						
	Fish	5	•		•			•		•					
	Mouse	5			•	•	•	•	•		•				
	Mosquito	4									•	•	•		
	Joker	1													

The default deck of Frank's Zoo contains one special card type, namely the *mosquito*, of which there are four in the deck. The mosquito can be used to replace an *elephant*. The deck also contains exactly one *joker* card, represented by a chameleon.

Cards are dealt evenly among the players. The goal of the game is to empty one's hand as quickly as possible. The player left of the dealer opens, and players take turns clockwise around the table.

Definition 2 (Player turn) *A player turn consists of one of two possible actions.*

- Move*: Play any move that defeats the last move that is on the table. If the move is an opening move, there is no move on the table to defeat, and thus any legal move can be played.*
- Pass*: Play no cards; this is always allowed.*

When, after a move of one player, all other players pass, the player who played the last move gets to play a new opening move, thus he may play any legal move. If the player has already emptied his hand, the opening move goes to the player on his left.

In general, a legal move (see Definition 3) consists of a number of cards of *one* type. However, there are two exceptions. First, in a move containing *elephants*, exactly one *mosquito* may be added, which will then 'turn into an elephant' (e.g., three elephants and one mosquito represent four elephants). Second, the *joker* can be used to replace one of the cards in a move, as long as the move contains at least one 'normal' card (e.g., one perch and one joker represent two perches). This leads to the following definition of a legal move.

Definition 3 (Legal move) *A legal move is a combination of cards of one of the types below.*

- *Any number of cards of the same type.*
- *Any number of cards of the same type, combined with one special card of a type that may replace the first type.*
- *Any of the two options above, combined with a joker.*

The normal cards in the move indicate the main card type of the move. For instance, for a move consisting of two elephants, a mosquito, and a joker, the main card type is elephants.

As mentioned in Definition 2, a player may only make a move if it defeats the last move on the table. A defeating move is defined as follows.

Definition 4 (Defeating move) *A defeating move is a legal move of one of the two types below.*

- *Predator defeat: the card type in the move is a predator of the card type in the previous move and the number of cards in the move is equal to the number of cards played in the previous move.*
- *Similarity defeat: The card type is equal to the card type in the previous move and the number of cards in the move is exactly one more than the number of cards played in the previous move.*

When a player empties his hand, he is out of the game. The score of a player ($score_p$) is calculated as the number of players in the game (p_{total}) minus the number of players already out of the game at the time that he plays his final move (p_{out}). The exception to this calculation is the last player remaining, who gets 0 points. Consequently, the score of a player p is calculated by Equation 1.

$$score_p = \begin{cases} p_{total} - p_{out} & \text{if } p_{total} - p_{out} > 1 \\ 0 & \text{if } p_{total} - p_{out} = 1 \end{cases} \quad (1)$$

The game is usually repeated several times, and the players accumulate their points over the games, the player with the highest end total being the final winner.

The first AI created for Frank's Zoo was the so-called RandomAI. This AI randomly plays any legal move. RandomAI is the basis for all of the AIs described in the following sections. It also sets a lower bound on the results that can be achieved with an AI, since it is only possible to play weaker than RandomAI when serious mistakes are made.

3 Heuristics

One of the AIs we implemented, labelled HeuristicAI, is completely based on heuristics, which have been derived from the researchers' own considerable experience with the game. This AI generates a list of possible moves to be played in response to the previous move. If the list contains only one possible move, this move is played instantly and no further heuristics are employed. However, if the list contains multiple moves, four steps are executed, in sequence, to determine the best move to be played. The steps are as follows (see Table 2).

- exploiting winning moves (Subsection 3.1)
- avoiding losing moves (Subsection 3.2)
- checking winning sequences (Subsection 3.3)
- exploiting move weights (Subsection 3.4)

3.1 Exploit winning moves

The first and quite straightforward heuristic used is to exploit winning moves.

Definition 5 (Winning move) *A winning move is a defeating move that empties the hand at once, resulting in instant victory.*

Naturally, each hand has only one potential winning move, namely the move that plays all the cards currently in the hand. If playing all the cards from the hand constitutes a legal move, it is the winning move. If the list contains that winning move, it is played immediately.

3.2 Avoid losing moves

If no winning move was on the list, the second step in HeuristicAI is to check whether the list contains losing moves.

Definition 6 (Losing move) *A losing move is a legal move that leaves the player with only a joker, resulting in certain loss.*

Since a joker cannot be played on its own, a player stuck with only a joker will always lose the game. Naturally, with only one joker in the game, each hand has only one potential losing move, which can only be in the hand if it contains the joker. If, in such a case, playing all cards from the hand except for the joker constitutes a legal move, that move is a losing move. If the list contains that losing move, it is removed from the list.

3.3 Check winning sequences

The third step is to check whether the list contains a move that is the start of a winning sequence.

Definition 7 (Winning sequence) *A winning sequence is a sequence of undefeatable moves, possibly followed by at most* one *defeatable move.*

There are two ways in which a move may be the start of a winning sequence, which are checked recursively. First, if it is a winning move (see Definition 5). Second, if it is an undefeatable move (see Definition 8) which, after it is played, results in a winning sequence. If the current move is the start of a winning sequence, it is played immediately.

Definition 8 (Undefeatable move) *An undefeatable move is a defeating move that cannot be defeated by the shared collection of cards held by the opponents.*

Table 2: Pseudocode for heuristic AI

```
determine list l of possible moves
if size of l = 1
        play the move immediately
endif
for each possible move m in l
        if m is a winning move
                play m immediately
                exit
        endif
endfor
for each possible move m in l
        if m is a losing move
                remove m from l
        endif
endfor
for each possible move m in l
        if m is undefeatable
                if m starts a winning sequence
                        play m immediately
                        exit
                endif
        endif
endfor
for each possible move m in l
        determine whether it is the best move based on weights
endfor
play the best move
```

Table 3: List of weights indicating the likelihood that the AI will be left with the corresponding card type at the end of a game.

	Joker	Elephant	Whale	Bear	Crocodile	Seal	Fox	Mouse	Lion	Hedgehog	Perch	Mosquito	Fish
Card weights	0	2	8	14	18	30	40	46	48	62	70	76	84

We note that Definition 8 is quite strong, because it applies to the shared collection of cards held by all opponents together. Consequently, it is possible that a move is considered defeatable by our strong definition, when it is actually undefeatable with respect to each separate opponent. For instance, assume that our AI wants to play two whales. If it is known that the opponents share 3 whales, the move is defeatable according to Definition 8. However, if the three whales are distributed evenly over the opponents (each opponent holds one whale), the move would be undefeatable with respect to each separate opponent. Nevertheless, our definition will label the move defeatable.

Note that there are instances where the AI could know for certain that a move labelled as defeatable is undefeatable. For instance, in the example above, if the AI holds two whales and knows that three whales are left in the hands of its opponents, and there are still two opponents, one of which holds two cards and the other one holds one card, then the AI can easily deduce that "playing two whales" is undefeatable. This is not implemented in HeuristicAI, as it covers relatively rare occurrences.[3]

3.4 Move weights

If the list contains neither winning moves nor moves that start a winning sequence, the weights of the remaining moves on the list are calculated.

The weight of a move is based on (a) the main card type of the move, and (b) the likelihood that a player will get stuck with this type of card at the end of the game (see Table 3). Moves containing cards which are likely to remain in the last player's hand at the end of the game get a high weight (e.g., fishes). Moves containing cards which are not likely to remain in hand at the end of the game get a low weight (e.g., elephants).

The weights are determined by letting four RandomAIs play 2×10^5 games against each other. At the end of each game, the cards remaining in the hand of the last player (who scores 0 points) are counted for each separate type. The totals of all those card types at the end of the series of games are divided by $1,000$ and rounded to the nearest whole number, to get the weights in Table 3.

During play, the weight of a move is further influenced by the fact whether the move contains all the player's cards of the main card type. If it does not (i.e., the move leaves the player with one or more cards of the main card type) the sequence of cards is broken, which is undesirable. Consequently, the move's weight is decreased by dividing it by 4 (an empirically tuned number). For example, all things being equal, to defeat a perch the AI prefers to play a seal rather than a bear, as a seal has a higher weight. However, if the AI holds two seals and one bear, to defeat a perch the AI rather plays a bear than a seal. The idea is that if the AI plays a seal, it would still need at least two moves to get rid of the bear and the remaining seal, while if it plays a bear, one more move may suffice to get rid of the remaining seals.

Moves with a high weight should be played as soon as possible. Moves with a low weight should be played late in the game. To this end we compare the weight of each move with two reference numbers: U (upper bound) and L (lower bound). The values of the reference numbers are based on (a) predefined start values, (b) the number of players, and (c) the number of cards still in the game. The values decrease when the game progresses. L and U are calculated as follows:

$$L = 15 - (P \times N)/10 \tag{2}$$

[3]Actually, in this example the AI would play its two whales anyway, as it constitutes a winning move according to Definition 5.

$$U = 30 - (P \times N)/5 \tag{3}$$

where P is the total number of cards played, and N is the number of players. The 'magic numbers' which appear in this calculation have been tuned manually.

If the move weight is below L, the move is not a good candidate for play, so we remove it from the list of possible moves. The idea behind this tactic is that 'strong' moves, i.e., moves that are able to defeat many moves, can be used to get control of the opening move in late game stages. It is generally understood in the game of Frank's Zoo that when there are relatively few cards left in the game, players have few possible moves, and even moves with 'weak' cards, such as fishes, may be undefeatable. Therefore, being able to play an opening move in the late game (and thus being able to play 'weak' cards which have become undefeatable) considerably increases a player's chance to win. Thus, 'strong' moves should not be played early, and passing rather than playing such a move is preferred in the early game.

If the move weight is between L and U, the move is a candidate for play and remains in the list of possible moves. If the move weight is above U and the move weight is higher than the weights of all other moves evaluated so far, we label this move the *preferred move*.

When all moves have been evaluated, but none of them is played (i.e., there are no winning moves or winning sequences), we select a move based on their weights. If there is a preferred move, we play it. If there is no preferred move, we chose a move from the list of possible moves randomly and play it. If the list of possible moves is empty, we pass.

4 Monte Carlo

A search method often used in non-deterministic games with imperfect information is Monte Carlo simulation. In Monte Carlo simulation, moves are evaluated by simulating a number of game episodes for each move.

Definition 9 (Game episode) *A game episode is a sequence of moves starting at the current move (the move to be evaluated) and ending at the last move of the game.*

For each move, the scores obtained in the simulated game episodes are accumulated. The move which obtains the highest total score throughout the simulated game episodes is chosen as the move to be played.

In our research, we investigated three versions of the Monte Carlo approach.

- Standard Monte Carlo (Subsection 4.1)
- Monte Carlo with weights (Subsection 4.2)
- Monte Carlo with the highest weight (Subsection 4.3)

Each approach will be discussed separately below.[4]

4.1 Standard Monte Carlo

In the standard Monte Carlo approach, we generate all possible moves m that may be played in response to a previous move. For each move m, we simulate 100 game episodes (starting with move m). In each of these episodes, all players use RandomAI. Afterwards, we calculate, for each move m, the total score $score_m$ obtained in the game episodes. The move with the highest total score is selected as the move to be played.

[4]We wish to point out that these Monte Carlo techniques differ from Monte Carlo Tree Search (MCTS), a technique which has produced excellent results in, amongst others, the game of Go [8, 9]. MCTS combines tree search (wherein the tree is built progressively to exploit seemingly stronger moves) with Monte Carlo simulations for node evaluation. MCTS is a strong and interesting approach, but unsuitable for the game of Frank's Zoo. The reason is that, while MCTS is able to deal with some non-determinism, it is ill-suited to deal with a large amount of imperfect information, which is inherent in Frank's Zoo (namely in the form of the opponents' hands).

To enhance the standard Monte Carlo algorithm, we used two heuristics described earlier: exploiting winning moves (Subsection 3.1) and avoiding losing moves (Subsection 3.2). This was done to ensure that the algorithm wins if possible and avoids losing immediately. We considered adding the winning-sequence heuristic (Subsection 3.3), but decided not to do that, because it takes a significant amount of processing time to execute, and would substantially increase the already considerable amount of time needed to run the Monte Carlo process. Furthermore, in our experiments with HeuristicAI we found that the winning-sequence heuristic was responsible for less than 1 percent of HeuristicAI's victories.

4.2 Monte Carlo with Weights

A further enhancement to the Monte Carlo algorithm was the use of weights for each possible move sequence.

Definition 10 (Move sequence) *A move sequence $[a, b]$ is a situation in which move b is played in response to move a. Move sequences can be extended to any length, e.g., $[a, b, \ldots, y, z]$. Move sequences comprising X moves are denoted X-sequences.*

For instance, the move sequence $[a, b]$ is a 2-sequence.

We created a table of W weights, containing a weight $w_{[a,b]}$ for each 2-sequence $[a, b]$. The weight $w_{[a,b]}$ is calculated as

$$w_{[a,b]} = \frac{\text{\# games in which } [a, b] \text{ leads to a win}}{\text{\# games in which } [a, b] \text{ is played}}$$

Obviously, most of the 2-sequences in the table are impossible or illegal. Since impossible or illegal move sequences never occur during a game, these 2-sequences automatically keep their default weight of 0.

The weights are calculated based on 10^6 training games using RandomAI players. Subsequently, the weights are used in the Monte Carlo simulation. Instead of selecting a move at random, moves are selected using roulette-wheel selection based on the weights of the corresponding move sequences. For instance, suppose that the algorithm has to respond to move a with either move b or move c. In that case, the algorithm bases its choice on the weights of the move sequences $[a, b]$ and $[a, c]$. If $w_{[a,b]} = 0.5$ and $w_{[a,c]} = 0.25$, move b is twice as likely to be chosen as move c.

4.3 Monte Carlo with the Highest Weight

The final enhancement we employed uses the same weight table as discussed in the previous subsection. However, instead of using a roulette-wheel selection to pick a move, we simply select the move corresponding to the move sequence with the highest weight. For instance, if the algorithm has to respond to move a, it will chose move b were $w_{[a,b]} = max_x\{w_{[a,x]}\}$.

5 Results

In this section we discuss experiments we conducted to evaluate the performances of the AIs described in the previous sections. Besides using the original Frank's Zoo deck of cards, to determine to what extent an AI is susceptible to non-determinism we tested each AI on test decks with a different card types (in the different numbers and of different types). These test decks were designed in such a way that (a) each card type is a predator to all card types that are lower in the hierarchy, and (b) each card type is a prey to all card types that are higher in the hierarchy. The number of card types ranged from 2 to 13, and of each type the test decks contained 4 cards. The test decks did not contain special cards or joker cards. Thus, the number of cards in a test deck is four times the number of card types in the deck.

In the remainder of this section we discuss the evaluation of HeuristicAI (Subsection 5.1) and the various versions of the Monte Carlo AI (Subsection 5.2). Finally, we compare the results of HeuristicAI and the Monte Carlo AI to determine which AI is most suitable for the game of Frank's Zoo (Subsection 5.3).

Table 4: List of weights for each card type in a specific deck.

# card	Card ID												
types	1	2	3	4	5	6	7	8	9	10	11	12	13
2	85	54											
3	68	50	23										
4	64	40	33	9									
5	64	42	29	18	12								
6	62	35	35	12	6	3							
7	49	48	32	22	9	3	3						
8	52	44	39	23	9	2	3	1					
9	47	41	25	25	0	7	7	2	2				
10	55	43	32	15	8	7	8	1	1	0			
11	54	34	36	20	15	10	3	3	0	0	0		
12	52	42	34	20	15	14	2	3	2	1	0	0	
13	58	36	27	19	15	13	8	0	0	1	1	0	0

5.1 Heuristics

We tested HeuristicAI against three RandomAIs in 10,000 games. The weights used for the different card sets are shown in Table 4. To evaluate the results we calculated for each experiment (a) the percentage of wins of HeuristicAI (with respect to the total number of games played), and (b) the percentage of score of HeuristicAI (with respect to the maximum possible score). The results of the experiment are shown in Figure 2. The figure shows the percentage of wins and the percentage of score. With respect to the number of wins we make three observations.

- On average, HeuristicAI wins 44% of the games, consequently HeuristicAI outperforms RandomAI.
- HeuristicAI performs best (49% of the wins) when the deck contains 11 card types (which is the highest number of card types used in these experiments).
- HeuristicAI performs worst (32% of the wins) when the deck contains 2 card types.

With respect to the score, we make another three observations.

- On average, HeuristicAI obtains 79% of the maximum score.
- HeuristicAI performs best (84% of the maximum score) when the deck contains 11 card types.
- HeuristicAI performs worst (70% of the maximum score) when the deck contains 2 card types..

When testing HeuristicAI against three RandomAIs on the original Frank's Zoo card set, it wins 82% of the games and obtains 94% of the maximum score. Three suggested reasons for why the AI's performance on the Frank's Zoo deck is so much better than on the test decks, are:

- The performance increases with the number of cards and the number of card types, and the Frank's Zoo deck has more cards and more card types than all the test decks.
- The 'magic numbers' in HeuristicAI have been tuned for the Frank's Zoo deck.
- Because of the 'interesting' predator-prey relations in the Frank's Zoo deck, the existence of special cards, and the joker, the Frank's Zoo deck allows for intricate tactics, while the test decks are straightforward constructs that allow little in the name of tactics.

From these results we may conclude that HeuristicAI is quite strong in the game of Frank's Zoo. It is hardly influenced by the non-determinism. Its performance increases steadily as the number of card types in the deck increases. When tested on the original Frank's Zoo deck, the performance is surprisingly good, despite the complex dependencies among the cards.

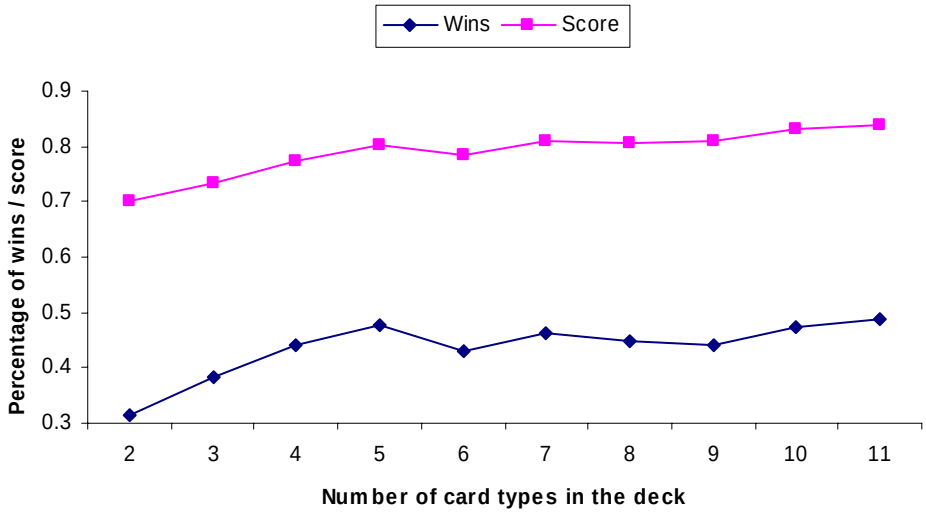

Figure 2: Percentage of wins and score of HeuristicAI when playing against RandomAI over 10,000 games.

5.2 Monte Carlo

With respect to the Monte Carlo AI we performed four experiments. In each experiment, we compared a variant of the Monte Carlo AI to RandomAI. To this end, we simulated a number of games in which our Monte Carlo AI plays against three RandomAIs. To evaluate the results we calculated for each experiment (a) the percentage of wins of the Monte Carlo AI (with respect to the total number of games), and (b) the percentage of score of the Monte Carlo AI (with respect to the maximum possible score).

Note that the number of games played in these experiments is much lower than in the experiments with HeuristicAI, as the Monte Carlo process is very slow compared to HeuristicAI.

Standard Monte Carlo vs. RandomAI

In the first experiment, we compared the standard Monte Carlo AI to RandomAI in a series of $1,000$ games. Within the Monte Carlo simulation we evaluated the moves according to two different criteria: (a) the number of wins, i.e., the move with the highest number of victories during the Monte Carlo simulation is selected, and (b) the score, i.e., the move that achieves the highest total score during the Monte Carlo simulation is selected. Figure 3 shows the difference in number of wins and score, when evaluating the *number of wins* within the Monte Carlo simulation. With respect to the number of wins, we make three observations.

- On average, the Monte Carlo AI wins 30% of the games, consequently the Monte Carlo AI outperforms RandomAI.
- The Monte Carlo AI performs best (34% of the wins) when the deck contains 7 card types.
- The Monte Carlo AI performs worst (26% of the wins) when the deck contains 5 card types.

With respect to the score, we make another three observations.

- On average, the Monte Carlo AI obtains 57% of the maximum score.
- The Monte Carlo AI performs best (62% of the maximum score) when the deck contains 7 card types.
- The Monte Carlo AI performs worst (54% of the maximum score) when the deck contains 5 card types..

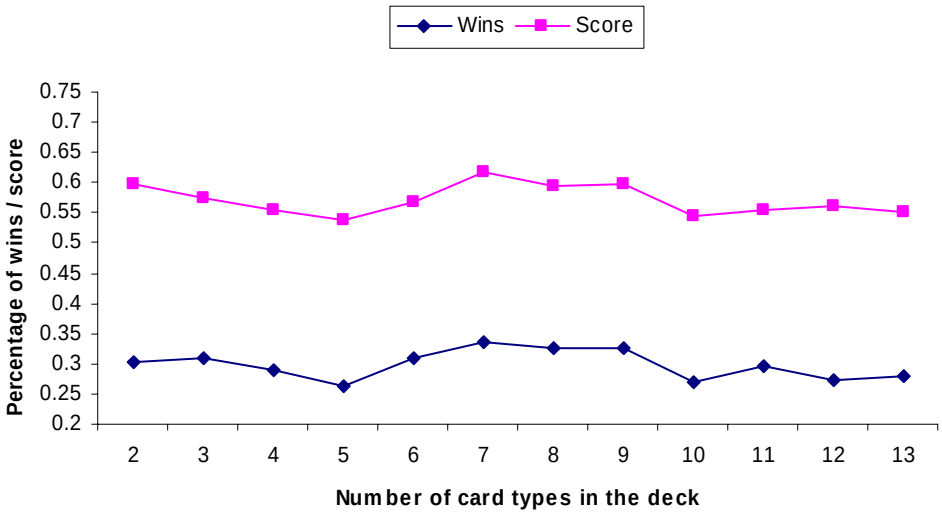

Figure 3: Percentage of wins and score of the standard Monte Carlo AI when playing against RandomAI over 1,000 games. In this experiment, the evaluation criterion for the Monte Carlo evaluation was the *number of wins*.

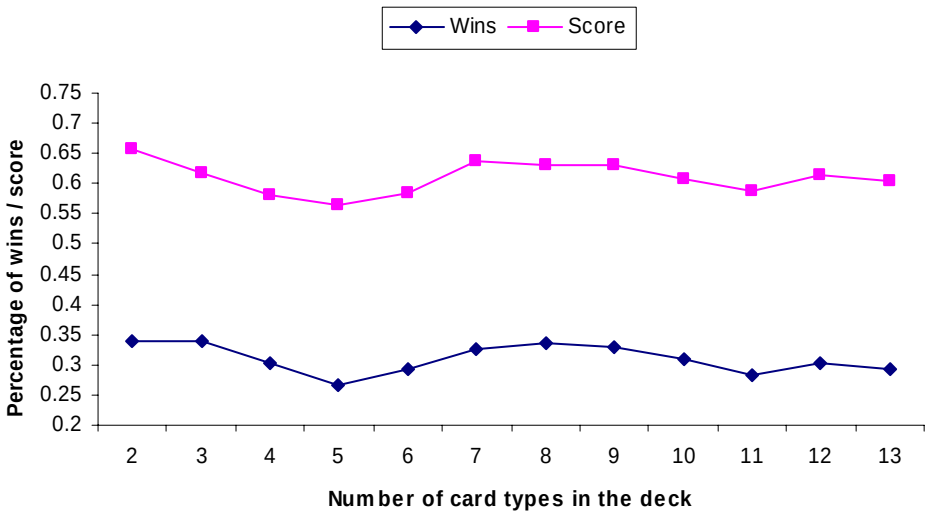

Figure 4: Percentage of wins and score of the standard Monte Carlo AI when playing against RandomAI over 1,000 games. In this experiment, the evaluation criterion for the Monte Carlo evaluation was the *score*.

These observations show that the performance of the Monte Carlo AI depends upon the number of card types in the deck. When the number of card types increases from 2 to 5, the performance of the Monte Carlo AI decreases. This is due to the increasing probability that the Monte Carlo AI plays suboptimal moves; RandomAI has an advantage here. When the number of card types in the deck increases from 6 tot 9, the performance of the Monte Carlo AI increases. This is due to the fact that with this numbers of card types, tactics plays an increasingly important role in the game. Since RandomAI uses no tactics at all, the Monte Carlo AI has an advantage here. When the number of card types increases from 10 to 13, the performance of the Monte Carlo AI decreases. This is due to the fact that, with higher numbers of cards and card types, the number of games played in the Monte Carlo simulation (namely 100) is not sufficient to reach the same performance level. If the number of simulation games would be increased, it is likely that the performance would also increase. Obviously, this is true for any Monte Carlo process.

Figure 4 shows the difference in number of wins and score, when the *score* is used as an evaluation criterion within the Monte Carlo simulation. When looking at the number of wins, we make three observations.

- On average, the Monte Carlo AI wins 31% of the games, consequently the Monte Carlo AI outperforms RandomAI.
- The Monte Carlo AI performs best (34% of the wins) when the deck contains 2 card types.
- The Monte Carlo AI performs worst (27% of the wins) when the deck contains 5 card types.

When looking at the score, we make another three observations.

- On average, the Monte Carlo AI obtains 61% of the maximum score.
- The Monte Carlo AI performs best (66% of the maximum score) when the deck contains 2 card types.
- The Monte Carlo AI performs worst (56% of the maximum score) when the deck contains 5 card types..

These observations show once more that the performance of the Monte Carlo AI depends upon the numbers of cards and card types in the deck. When the number of card types increases from 2 to 5, the performance of the Monte Carlo AI decreases. When the number of card types in the deck increases from 6 tot 9, the performance of the Monte Carlo AI increases. When the number of card types increases from 10 to 13, the performance of the Monte Carlo AI decreases. Furthermore, we observe that the Monte Carlo AI performs best when the score (instead of the number of wins) is used as the evaluation criterion within the Monte Carlo AI. Therefore, we use the score as the Monte Carlo evaluation criterion in the remainder of the experiments.

Monte Carlo with weights vs. RandomAI

In the second experiment, we compared the Monte Carlo AI with weights to RandomAI in a series of 200 games. Within the Monte Carlo simulation we evaluated the moves according to the score. Figure 5 shows the difference in number of wins and score, when evaluating the score within the Monte Carlo simulation. When looking at the number of wins, we make three observations.

- On average, the Monte Carlo AI wins 27% of the games, consequently the Monte Carlo AI outperforms RandomAI.
- The Monte Carlo AI performs best (36% of the wins) when the deck contains 5 card types.
- The Monte Carlo AI performs worst (18% of the wins) when the deck contains 13 card types.

When looking at the score, we make another three observations.

- On average, the Monte Carlo AI obtains 57% of the maximum score.
- The Monte Carlo AI performs best (67% of the total score) when the deck contains 3 card types.
- The Monte Carlo AI performs worst (45% of the total score) when the deck contains 13 card types.

These observations show some differences with the standard Monte Carlo AI. The Monte Carlo with weights performs better in the low range of card types. With these number of card types, the weights are quite accurate and enhance the performance. However, when the deck contains more types of cards, there are more possible moves. Consequently, to determine the weights with the same accuracy, more training games are needed with the increased number of card types. Since we used the same number of training games for each number of card types, the accuracy of the weights decreases when the number of card types in the deck increases. Therefore, the performance of the Monte Carlo AI decreases when the deck contains more card types. The local fluctuations in performance might also be due to the relatively low number of test runs (200 games).

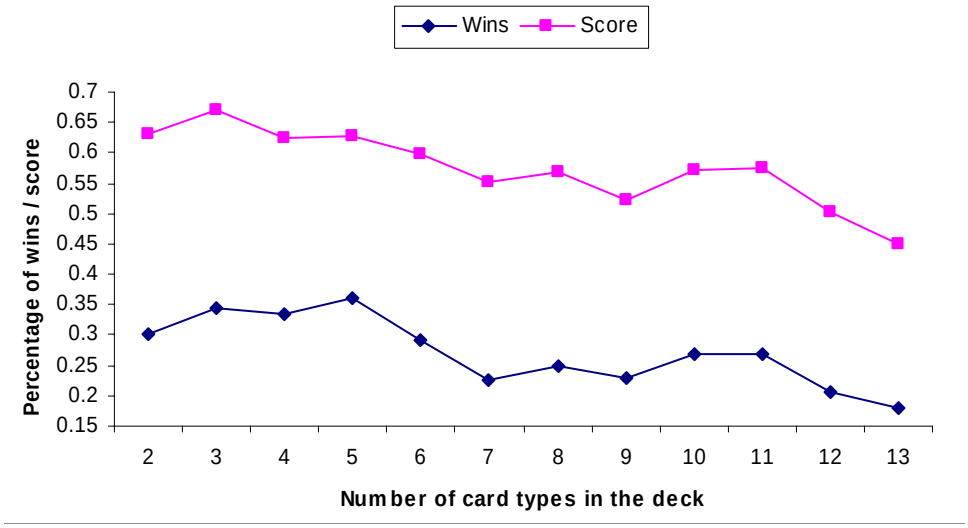

Figure 5: Percentage of wins and score of the Monte Carlo AI with weights when playing against RandomAI over 200 games.

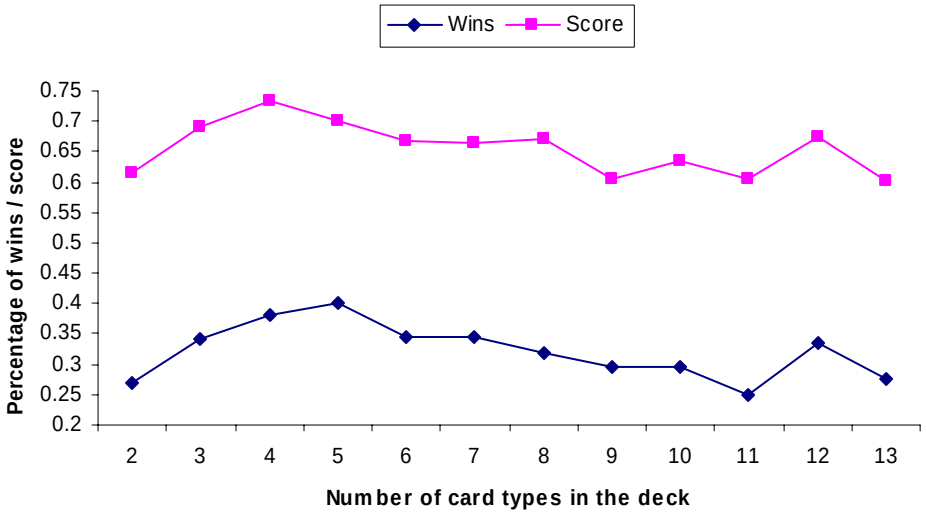

Figure 6: Percentage of wins and score of the Monte Carlo AI with the highest weight when playing against RandomAI over 200 games.

Monte Carlo with highest weight vs. RandomAI

In the third experiment, we compared the Monte Carlo AI with the highest weight to RandomAI in a series of 200 games. Within the Monte Carlo simulation we evaluated the moves according to the score. Figure 6 shows the difference in number of wins and score, when evaluating the score within the Monte Carlo simulation. When looking at the number of wins, we make three observations.

- On average, the Monte Carlo AI wins 32% of the games, consequently the Monte Carlo AI outperforms RandomAI.
- The Monte Carlo AI performs best (40% of the wins) when the deck contains 5 card types.
- The Monte Carlo AI performs worst (25% of the wins) when the deck contains 11 card types.

When looking at the score, we make another three observations.

- On average, the Monte Carlo AI obtains 66% of the maximum score.

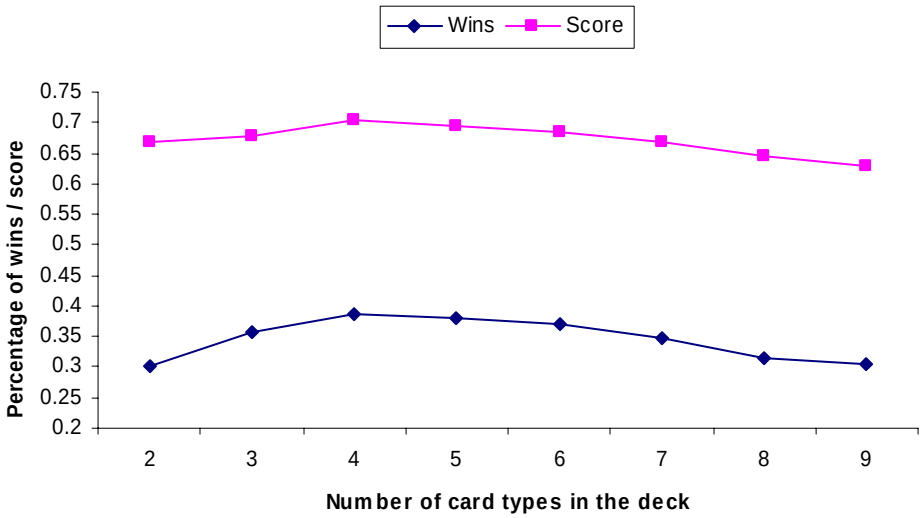

Figure 7: Percentage of wins and score of the Monte Carlo AI with weights when playing against RandomAI over 10,000 games.

- The Monte Carlo AI performs best (73% of the maximum score) when the deck contains 4 card types.
- The Monte Carlo AI performs worst (60% of the maximum score) when the deck contains 13 card types.

These observations show an improvement with respect to the results obtains with the Monte Carlo AI with weights (not the highest weight). The improvement is most obvious in the middle range of card types (i.e., 7–9). The use of the highest weight enables the Monte Carlo algorithm to deal with the high level of non-determinism in the game. However, in the high ranges of card types (10–13), the performance is still inadequate. As in the previous experiment, this might be due to the small number of test runs (200 games)

Monte Carlo with highest weight vs. RandomAI (bis)

In the fourth experiment, we compared the Monte Carlo AI with the highest weight to RandomAI in a series of 10,000 games. This is a repetition of the third experiment, but now with an increased number of games to take care of statistical fluctuations. Because of the enormous amount of time needed to run the games, which increases exponentially when the number of card types is increased, we ran this experiment only up to 9 card types (and even this took us more than one month to run).

Figure 7 shows the difference in number of wins and score, when evaluating the score within the Monte Carlo simulation. When looking at the number of wins, we make three observations.

- On average, the Monte Carlo AI wins 34% of the games, consequently the Monte Carlo AI outperforms RandomAI.
- The Monte Carlo AI performs best (39% of the wins) when the deck contains 4 card types.
- The Monte Carlo AI performs worst (30% of the wins) when the deck contains 9 card types (with the footnote that we only ran this experiment up to 9 card types).

When looking at the score, we make another three observations.

- On average, the Monte Carlo AI obtains 67% of the maximum score.
- The Monte Carlo AI performs best (70% of the maximum score)when the deck contains 4 card types.

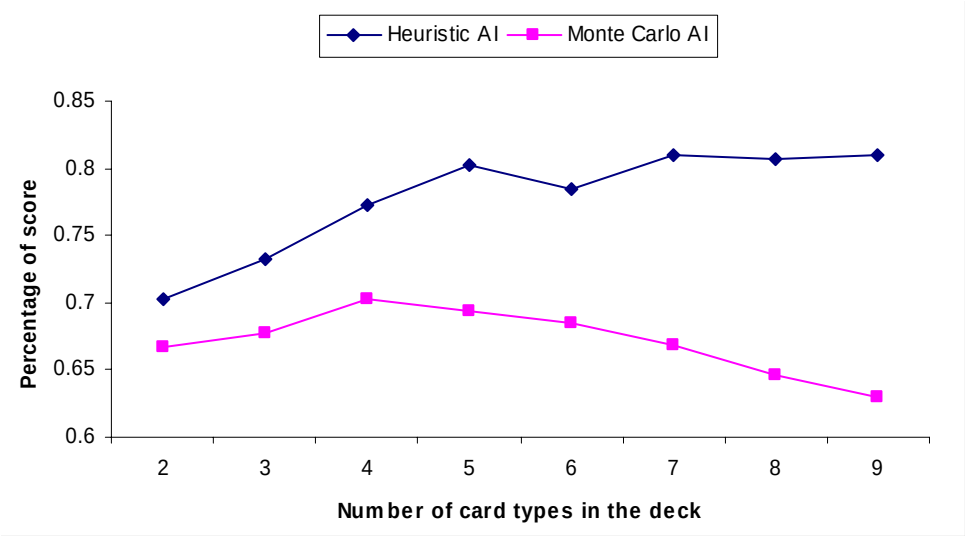

Figure 8: Percentage of score of HeuristicAI and the Monte Carlo AI with the highest weight when playing against RandomAI over 10,000 games.

- The Monte Carlo AI performs worst (63% of the maximum score) when the deck contains 9 card types (again, noting that we only ran this experiment up to 9 card types).

These observations show that Monte Carlo with the highest weight is the best Monte Carlo AI. When tested with a sufficient number of test runs ($10,000$ games), we observe non-fluctuating results in which the AI performs best at a deck containing 4 card types. The decreasing performance of the AI with an increasing number of card types may be due to the decreasing accuracy of the weights used.

5.3 Comparing Monte Carlo and HeuristicAI

To compare the results of HeuristicAI and the Monte Carlo AI with the highest weight (which achieved the best results of all Monte Carlo AIs), we plotted the results for both AIs in one figure (Figure 8). This figure shows, for both AIs, the percentages of the maximum score obtained by the AI when playing 10,000 games against RandomAI. From these results we make two observations.

- The score of HeuristicAI is higher than the score of the Monte Carlo AI for all number of card types in the deck.
- The score of HeuristicAI increases as the number of card types in the deck increases, whereas the score of the Monte Carlo AI decreases as the number of card types in the deck increases.

From these observations we may conclude that HeuristicAI performs better in the game of Frank's Zoo, and that it is positively influenced by an increasing level of non-determinism. Consequently, HeuristicAI seems a better choice than a Monte Carlo AI in the game of Frank's Zoo.

For lack of time, we did not yet run a competition of HeuristicAI against the Monte Carlo AIs. However, since on the Frank's Zoo deck, or decks with a high number of card types, the performance of the Monte Carlo AIs is not much better than the performance of RandomAI, the results of such a competition are clear: HeuristicAI would convincingly defeat any Monte Carlo AI.

6 Move Predictor

As Frank's Zoo is an incomplete-information game, we assumed that it would be easier for an AI to win if it had access to complete information, i.e., if it had complete knowledge of its opponents' hands. In such a case, we could also apply Monte Carlo Tree Search instead of the Monte Carlo techniques that we employ now. We considered the construction of opponent models to bridge the gap between

incomplete and complete information. A vital component of an opponent model is a move predictor, which predicts the next move that an opponent will make. If the next moves of opponents are known beforehand, we can, for instance, determine winning sequences (Subsection 3.3) more accurately.

The goal of our Move Predictor is to calculate a prediction for each possible move combination $[a, b]$, where a is the last move played, and b is the move to be predicted. The prediction $p_{[a,b]}$ is based on two parameters: (a) the likelihood $l_{[a,b]}$ of the move combination $[a, b]$ and (b) the probability p_b that the move b can be played by the player we are predicting.

The likelihood $l_{[a,b]}$ that a move b will be played in response to a move a is calculated by Equation 4.

$$l_{[a,b]} = \frac{\text{\# times that } [a, b] \text{ is played}}{\text{\# times that move } a \text{ is played}} \tag{4}$$

The probability that a certain move b can be played by the player we are predicting is calculated by Equation 5.

$$p_b = \frac{\binom{n}{m}\binom{x}{y}\binom{w}{z}\binom{d-n-x-w}{h-m-y-z}}{\binom{d}{h}} \tag{5}$$

where

- d = number of cards left in the deck $[1, 60]$
- h = number of cards in the hand of the player $[1, 15]$
- n = number of cards of type t left in the deck $[1, 5]$
- m = number of cards of type t in the move $[1, 5]$
- x = number of special cards left in the deck $[0, 1]$
- y = number of special cards in the move $[0, 1]$
- w = number of joker cards left in the deck $[0, 1]$
- z = number of joker cards in the move $[0, 1]$

The prediction $p_{[a,b]}$ is the product of the likelihood $l_{[a,b]}$ and the probability p_b (see Equation 6).

$$p_{[a,b]} = l_{[a,b]} \times p_b \tag{6}$$

Once the prediction $p_{[a,b]}$ is calculated for each possible move combination $[a, b]$, the move to be predicted is move b where $p_{[a,b]} = max_x\{p_{[a,x]}\}$.

6.1 Results

To test the Move Predictor we trained it by letting HeuristicAI (for which the moves should be fairly predictable) play 10,000 training games. We tested the Move Predictor by letting it predict all moves in 10,000 games. In total, the Move Predictor predicted 68% of the moves correctly. This is, indeed, a surprisingly high number, from which we may conclude that a Move Predictor may be a powerful tool in opponent modelling.

Note that a move predictor in itself does not accurately predict an opponent's hand. Therefore, it does not really reduce the amount of imperfect information in the game.

7 Conclusions

We developed and tested a completely heuristic AI incorporating four distinct steps in the move-selection process, viz. (a) exploiting winning moves, (b) avoiding losing moves, (c) checking winning sequences, and (d) using move weights. From the results of the experiments performed with HeuristicAI, we may conclude that HeuristicAI performs quite well. On average, the AI obtains 79% of the maximum score and 44% of the maximum number of wins. Furthermore, the performance increases as the number of card types in the deck increases. The optimal performance is reached with the original Frank's Zoo deck. With the original deck, the AI obtains 94% of the maximum score and 82% of the maximum number of wins.

We developed and tested four distinct versions of an AI incorporating Monte Carlo simulation, viz. (a) standard Monte Carlo, (b) Monte Carlo with weights, and (c) Monte Carlo with the highest weight (twice, with different numbers of episodes). From the experiments performed on these versions of the Monte Carlo AI, we may conclude that the Monte Carlo AI with the highest weight performs best. However, the optimal performance with this AI is reached for a deck containing only four distinct card types. As the number of card types increases, the performance of the Monte Carlo AI with the highest weight decreases. As mentioned before, this may be due to the decreasing accuracy of the weights used.

From the results of the experiments comparing HeuristicAI and the Monte Carlo AI we may conclude that HeuristicAI is by far superior to the Monte Carlo AI. Furthermore, we may conclude that the performance of HeuristicAI is positively influenced by an increasing amount of incomplete information.

We developed and tested a move predictor. From the results of the experiment performed with the move predictor, we observed that it predicts 68% of the moves of HeuristicAI correctly. Consequently, we may conclude that the move predictor performs quite well and that it may be a powerful tool in creating opponent models.

We posed the specific research question to what extent it is possible to apply artificial intelligence techniques from classic computer games to design artificial intelligence for Frank's Zoo. The heuristic approach, which is common in classic computer games, worked well. The Monte Carlo approaches, often used in classic board games to deal with non-determinism or small amounts of imperfect information, did not give good results. In particular, the performance of the Monte Carlo approaches decreased with an increase in complexity of the game (i.e., an increase in card types in the deck). This stands in sharp contrast to the fact that the performance of the heuristic approach increased with an increase in complexity of the game. Our current explanation for the inferior performance of the Monte Carlo techniques is the large amount of imperfect information in the game. To deal with that, a Monte Carlo approach needs to run so many simulations that it becomes impractical to use it for implementing AI. Therefore, we conclude that Monte Carlo techniques are unlikely to be a suitable approach for dealing with large amounts of imperfect information in games, in contrast to heuristics.

8 Future Research

The approaches used and the lessons learned in this research may be applied to other types of modern board games. Examples of such games are Settlers of Catan, Puerto Rico, and Tigris and Euphratis. The present line of research was indeed pursued for the ROLEC project by István Szita, Guillaume Chaslot, and Pieter Spronck, by applying Monte Carlo Tree Search (MCTS) to artificial intelligence for the game Settlers of Catan [17].

There are many other directions for future research, seven of which are described below.

1. HeuristicAI can be further extended. Although the current heuristic approach performs quite well, it would be worth investigating whether it may be improved by enhancing the approach with additional techniques.

2. The development of a generalised version of HeuristicAI would be an interesting direction for future research. Currently, HeuristicAI (more specifically, the weights it employs) have to be trained in advance. The weights should be trained and tuned by online learning.

3. It is worth investigating how to tune the magic numbers employed by HeuristicAI. Currently, these number are tuned manually. However, it would be interesting to tune the numbers automatically, for instance, by means of evolutionary learning.

4. It would be beneficial to develop a more efficient implementation of the Monte Carlo AI. Although the current version is technically correct, it runs rather slowly. The development of a more efficient implementation would decrease the evaluation time needed.

5. The weight table that is used by the Monte Carlo AI does not take into account the phase of the game. From HeuristicAI we know that moves that might be strong late in the game, might be weak early in the game. An extended weight table that takes the phase of the game into account might improve the performance of the Monte Carlo AI. It is also worthwhile to investigate whether less resource-intensive alternatives for the huge weight table can be designed.

6. The move predictor has not yet been integrated in the developed AIs. Currently, it is a 'stand-alone' component, but it could be used to improve the performance of the Monte Carlo AIs, and also of HeuristicAI.

7. A valuable addition to our current version of the Frank's Zoo engine would be a web interface. The possibility to let people play the game via the Internet has two main advantages. Firstly, it provides us with an excellent way of gathering player data. Secondly, it provides entertainment to the users.

All in all, the research described in this document provides many paths for further work. This work is worthwhile on its own, to develop strong or entertaining AI for modern board games. However, in our vision it is also a way to bridge the gap between the AI for classic board games and the AI for modern video games.

References

[1] S. Bakkes, P. Kerbush, P. Spronck, and H.J. van den Herik. Predicting success in an imperfect information game. *Proceedings of the Computer Games Workshop 2007*, pages 219–230, 2007. Maastricht University, The Netherlands.

[2] S. Bakkes and P. Spronck. Gathering and utilising domain knowledge in commercial computer games. *Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence*, pages 35–42, 2006. University of Namur, Belgium.

[3] M. Bergsma. Opponent modeling in machiavelli. B.Sc. thesis, Universiteit Maastricht, 2006.

[4] D. Billings, D. Papp, J. Schaeffer, and D. Szafron. Opponent modeling in poker. *Proceedings of 15th National Conference of the American Association on Artificial Intelligence*, pages 493–498, 1998. AAAI Press.

[5] D. Carmel and S. Markovitch. Learning models of opponent's strategies in game playing. *Proceedings of the AAAI Fall Symposium on Games:Planning and Learning*, pages 140–147, 1993.

[6] D. Carmel and S. Markovitch. Incorporating opponent models into adversary search. *AAAI*, pages 120–125, 1995.

[7] D. Carmel and S. Markovitch. Learning and using opponent models in adversary search. Technical Report CIS9609, Technion, Haifa, Israel, 1996.

[8] G. Chaslot, J-T. Saito, B. Bouzy, J. Uiterwijk, and H.J. van den Herik. Monte-Carlo Strategies for Computer Go. *Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence*, pages 83–91, 2006.

[9] G. Chaslot, M. Winands, H.J. van den Herik, J. Uiterwijk, and B. Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(3):343, 2008.

[10] A. Davidson. Using artificial neural networks to model opponents in Texas Hold 'em. http://spaz.ca/aaron/poker/nnpoker.pdf, 1999.

[11] A. Davidson, D. Billings, J. Schaeffer, and D. Szafron. Improved opponent modeling in poker. *Proceedings of the 2000 International Conference on Artificial Intelligence (ICAI'2000)*, pages 1467–1473, 2000.

[12] A. Lockett, C. Chen, and R. Miikulainen. Evolving explicit opponent models in game playing. *Proceedings of the Genetic and Evolutionary Computation Conference*, 2007.

[13] M. Ponsen, H. Muñoz-Avila, P. Spronck, and D. Aha. Automatically generating game tactics via evolutionary learning. *AI Magazine*, 27(3):75–84, 2006. AAAI Press.

[14] M. Ponsen, P. Spronck, and K. Tuyls. Hierarchical reinforcement learning with deictic representation in a computer game. *Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence*, pages 251–258, 2006. University of Namur, Belgium.

[15] F. Southey, M. Bowling, B. Larson, C. Piccione, N. Burch, and D. Billings. Bayes' bluff: Opponent modeling in poker. *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence (UAI-05)*, pages 550–558, 2005.

[16] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, and E. Postma. Adaptive game AI with dynamic scripting. *Machine Learning*, 63:217–248, 2005.

[17] I. Szita, G. Chaslot, and P. Spronck. Monte carlo tree search in settlers of catan. *Proceedings of 12th Advances in Computer Games Conference (ACG12)*, 2009. Will appear in the Lecture Notes in Computer Science series in the second half of 2009.

[18] T. Timuri, P. Spronck, and H.J. van den Herik. Automatic rule ordering for dynamic scripting. *Proceedings, The Third Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 49–54, 2007. AAAI Press.