Combining Monte-Carlo Tree Search and Opponent Modelling in Poker

G.E.H. Gerritsen

Master Thesis DKE 10-01

Thesis committee
Dr. J.W.H.M. Uiterwijk
M. Ponsen M.Sc.
G.M.J-B. Chaslot M.Sc.
M.P.D. Schadd M.Sc.

Master Artificial Intelligence

Department of Knowledge Engineering
Faculty of Humanities and Sciences
Maastricht University

Maastricht, January 2010

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science of Artificial Intelligence at the Department of Knowledge Engineering of Maastricht University

Preface

When I started my studies, back in 2003, I already knew the stories about writing a thesis and especially about spending a lot of time finishing it. Back then, I thought I would not fall for that and finish my thesis nice and clean in a few months. However, that's not what happened. I started in October 2008 and since then, I have been postponing deadlines more than I ever have. Eventually, although I already started my regular day job, I finished it during some spare evenings and days during the weekend. I am very thankful for finishing it and for everyone who supported me along the way. Some people I would like to thank by name:

First of all, I would like to thank my supervisors Jos Uiterwijk, Marc Ponsen and Guillaume Chaslot. Their insightful comments and explanations of various techniques aided me during different phases of my thesis. I learned a lot from you all about poker, MCTS and scientific research in general.

Secondly, I would like to thank my parents, who saw many a deadline be postponed and still stayed patient. Furthermore, I'd also like to thank my girlfriend for her many proof-readings and her ongoing support.

Geert Gerritsen,
Amsterdam, 15 February 2010

Abstract

Games have always been an area to receive much attention from researchers in artificial intelligence. During the last decennia, research was focused on games like chess, resulting in a computer player (Deep Blue) that was able to beat the world champion (Garri Kasparov). In the past years, researchers became increasingly interested in imperfect-information games like poker. In these domains, a different approach should be used to be able to develop a computer player that can compete against human professionals. The majority of previous research focuses on limit poker or on poker with only one opponent (heads-up), while the most popular poker variant nowadays is a no-limit variant with several players.

This research introduces a new search technique for games in the domain of poker: Monte-Carlo Tree Search (MCTS). This technique has already proved valuable in the field of Go [11]. MCTS is able to deal with the larger state space associated with no-limit poker against multiple opponents. Furthermore, if a computer poker player should be able to compete against poker professionals, it should use some sort of opponent modelling [5]. Therefore, we wanted to show that the Bayesian approach to opponent modelling proposed by Ponsen et al. [16] could be a valuable component of a computer poker player.

We tested both MCTS and Bayesian opponent modelling in 2 experiments: in the first experiment, our poker bot played against a relatively simple rule-based computer player (ACE1). In the second experiment, our computer poker player faced Poki, a more sophisticated computer player [5].

Results from the experiments show that: (1) MCTS is a valuable search technique which can yield good results against a relatively simple opponent. However, unless the technique improves, MCTS itself will not be able to compete against experienced human poker players. Our second conclusion is that (2) a poker AI incorporating the Bayesian approach to opponent modelling is more effective than a poker AI who does not use this form of opponent modelling. This suggests that the Bayesian approach to opponent modelling proposed by Ponsen et al. [16] can be a valuable contribution to a poker program. However, there are some limitations to our research including the relatively low number of MCTS iterations (1000) and the relatively low number of games played (10,000) during each of the experiments. Future research is necessary to gain more evidence for our conclusions.

Contents

Preface iii

Abstract v

Contents vii

List of Figures ix

List of Tables xi

1 Introduction 1

2 Background 5
2.1 Texas Hold'Em Poker 5
2.2 Related work 6

3 Monte-Carlo Tree Search 9
3.1 Monte-Carlo Tree Search in general 9
3.2 Monte-Carlo Tree Search in poker 12
3.2.1 Selection 12
3.2.2 Expansion 14
3.2.3 Simulation 14
3.2.4 Backpropagation 15

4 Adding domain knowledge to MCTS 17
4.1 Opponent model 17
4.2 Model tests used in MCTS 19

5 Experiments 25
5.1 Experimental setup 25
5.2 Experiment 1 27
5.3 Experiment 2 30

6 Discussion **33**
6.1 Conclusions 33
6.2 Limitations to our research 34
6.3 Future research 35

A Appendix: Output of experiment 1 **39**

B Appendix: Output of experiment 2 **43**

List of Figures

3.1 An iteration of MCTS. 12

4.1 MCTS with opponent modelling. 21

5.1 Thesisbot against ACE1, using MCTS. 28
5.2 Thesisbot against ACE1, using MCTS with opponent modelling. 29

A.1 Observed actions without opponent model. 39
A.2 Observed actions with opponent model. 40
A.3 Bankroll without opponent model. 41
A.4 Bankroll with opponent model. 41

B.1 Observed actions without opponent model. 43
B.2 Observed actions with opponent model. 44
B.3 Bankroll without opponent model. 45
B.4 Bankroll with opponent model. 45

List of Tables

5.1 Rule-based poker bot ACE1. 27
5.2 Results of experiment 1. 29
5.3 Results of experiment 2. 32

Chapter 1

Introduction

The past decennia, poker has been gaining much popularity, both in casinos and in recreative play. Together with gaining popularity by the crowd an increase in scientific focus has been perceived. The grown scientific interest can be explained by looking at how poker is different from other (more traditional) games like chess or checkers. These differences include:

- Poker contains imperfect information. This means that a player does not know his opponent's cards. Therefore, he has to guess whether his own cards have a good chance of beating his opponent's. This situation is a major cause for bluffing. Bluffing is the act in which a player plays as to show that his cards are quite good, while in fact his cards are weak and he therefore has a low chance of winning.

- Stochastic events take place in poker. Since the order of cards is not known, players do not know what cards will be dealt. This shows that the outcome of a poker game depends on chance for some part.

- The game is partially observable, which constitutes of the fact that if every opponent folds, a player can muck. Mucking is the act of not showing one's cards, even when winning the pot. This makes it much more difficult to examine another player's strategy.

- Poker can be played with more than 2 players. On the Internet, the common variant of poker is played at 9-player tables. Playing against multiple opponents increases game the complexity, which in turn increases the processing time of a poker program significantly.

Because of the differences between poker and more classical games like chess, poker is a very interesting but very challenging research subject [8]. A more extensive explanation about poker research and its scientific challenge can be found in Chapter 2 and in [5].

Chapter 2 also contains an explanation of the rules for Texas Hold 'Em, the most popular variant of poker played nowaydays. This variant is normally played with 2 (which is called heads-up) to 9 players. Every player gets two cards and has to bet in an effective way, during multiple betting rounds, to optimize his profit. The exact rules of Texas Hold 'Em are explained in Chapter 2.

Because of the fact that Texas Hold 'Em contains multiple betting rounds, one can imagine that a player should make many decisions during a game. To be able to come up with a playing strategy, a poker program should implement a so-called search algorithm. Such an algorithm should evaluate every possible action at a certain moment and then select the course of action most probable to yield the best result. The search algorithm we use in our research is Monte-Carlo Tree Search (MCTS). MCTS is relatively new and has not been applied to poker yet. In Chapter 3 MCTS will be discussed in detail.

As will be explained in Chapter 3, general MCTS treats every opponent the same (i.e., as a rational decision taker). Since this does obviously resemble real poker, we wanted to import specific knowledge about an opponent into the program. This is referred to as opponent modelling. In fact, early poker research by Billings et al. [4] constituted of designing an online poker bot, which could play online against human players. Initially, the poker bot played well, however, after a number of games, human players tended to adapt their playing style to exploit the poker bot's weaknesses. This leaded the researchers to conclude that opponent modelling is very important in designing a good poker program. Therefore, we use the Bayesian opponent-modelling technique proposed in [16]. We will explain this technique further in Chapter 4.

As described in the previous two paragraphs, we chose to combine two techniques in this research: MCTS and our approach to opponent modelling. While MCTS alone could prove valuable in poker, it has an assumption that is not in line with real-life poker: the assumption that every player plays according to a theoretical optimal strategy (Nash equilibrium). However, poker is just too complex to expect a player to play according to such a strategy. Therefore, players will take suboptimal actions, which can be exploited. This is why incorporating only MCTS in a poker program will not yield optimal results. In order to take advantage of the opponent's behaviour, we should add some domain knowledge to our program. Domain knowledge is added in the form of an opponent model, with which our program can predict (to a certain extent) the cards and actions of its opponents.

We conducted this research with the following problem statement in mind:

"Will a poker bot incorporating the Bayesian approach to opponent modelling be more effective than a poker bot who does not use this form of opponent modelling?"

Our research questions were:

1. How can we incorporate poker domain knowledge in Monte-Carlo Tree Search?

2. Does using the MCTS algorithm yield good results in poker?

3. Does a poker bot yield better results if it incorporates the Bayesian opponent model?

The outline of this thesis is as follows: in Chapter 2, we will take a closer look at the details of Texas Hold 'Em poker while we also look at related research concerning this topic. This research tries to combine two elements in a poker program: (1) the search-tree approach we use and (2) our opponent-model approach. In Chapter 3 we will explain the first of these two elements while the second element will be discussed in Chapter 4. In Chapter 5 we will discuss the experiments we conducted and their results and in Chapter 6 we will present a conclusion and some recommendations for future research.

Chapter 2

Background

In this chapter we will provide some background on the subject of poker research. In the first section we will explain the rules of the most popular poker variant, Texas Hold 'Em. The second section will contain a discussion of related work done in the area of poker research.

2.1 Texas Hold'Em Poker

Poker is a game of cards that is played between several opponents. Each player gets dealt two cards, which are only visible to the player himself. Together, these cards are called a 'hand'. Players invest money in a game by putting the correct amount of money (or the representation thereof: the chips) in the pot. The correct amount of money a player has to put in the pot is expressed in terms of a big blind. The big blind is the minimum amount of chips that should be bet. When a bet is placed, other players have to step out of the game (fold), match the bet amount (call) or invest more (raise). When an equal amount is put in the pot by every player that is still active (i.e., has not folded), a number of cards is dealt to the table (board cards). This process is called a betting round.

In Texas Hold 'Em, the poker variant this research focuses on and which is played much on the Internet, there are several betting rounds: preflop, flop, turn and river. Each round, one or more board cards are dealt, except for the preflop round, in which no cards are dealt to the table. When the flop round begins, 3 cards are dealt and during both turn and river rounds, 1 card is dealt. Eventually, there are 5 board cards. When the last betting round is finished, each player's combination of cards (hand and board cards) are compared. The player with the highest combination of cards wins the pot. For a more extensive explanation of poker and its Texas Hold 'Em variant, see [8].

Texas Hold 'Em poker can be played in two variants: Limit and No-Limit. The Limit variant received more interest from the scientific world, since there is a limit on how much one can bet, which makes the game less complex. A player can only bet a fixed amount, which therefore restricts possible actions to three: fold, check/call, raise. Therefore, the state space is relatively limited, making this variant an easily accessible start domain for poker research. On the other hand, No-Limit poker does not limit the amount a player can bet. The maximum bet, of course, is the amount of money a player possesses in the game. Since there is no limit on the amount to bet, there are numerous raise actions to take, which increases the state space significantly. Therefore, designing a poker program for this poker variant poses a larger challenge.

2.2 Related work

In this section, we will discuss related research. Early poker research was conducted by Nash and Shapley [14], Newman [15] and Cutler [12]. This research focused on mathematical models of simplified poker environments. Later, in line with the upcoming popularity of poker, research on popular poker games (such as Texas Hold 'Em) emerged [8].

Nash and Shapley [14] discovered that there exist optimal strategies in simplified poker variants. An optimal strategy in this case is called a Nash equilibrium. This means that if this strategy is used by a player, it cannot result in a lower pay-off when faced with a perfect opponent. Furthermore, finding an equilibrium solution is only possible in relatively small state spaces, which therefore is not practical for the domain of full-scale poker. This lead Billings to investigate abstraction in poker, in order to reduce the state space and complexity. Abstractions investigated were: suit equivalence, rank equivalence, deck reduction, betting round reduction, game truncation, betting rounds merging and bucketing of hands (i.e., grouping equivalent hands together). Especially the use of bucketing yielded good results in decreasing the state space while still preserving the possibility to find a Nash equilibrium strategy [6].

Recent research shows the underlying assumptions with abstraction are not always correct: Waugh et al. [17] discovered that when trying to keep the abstraction as close as possible to the real domain, a weak strategy emerges in comparison to the emergent strategy using more rigid abstractions. This indicates that abstractions should not be generally accepted to use in research on games.

Through the research conducted by Billings et al., for the first time ever, computer programs were not completely outclassed by strong human opposition in 2-player Limit Texas Hold'em. Still, two challenges remain:

- It is difficult to develop a poker program that can play reasonably against more than 2 players. Furthermore, a program should eventually also be able to play No-Limit Hold 'Em at a reasonable level. Designing a poker program to play No-Limit at a fair level is quite a challenge, due to the increased complexity of the game. To develop a poker program that can handle multiple players and the No-Limit variant, one has to deal with a large state space (i.e., a larger complexity).

- Another aspect of the previous poker research that could be improved is that of the optimal strategy. Billings et al. found that approximations of Nash equilibria yielded good results. A Nash equilibrium is not optimal in maximizing pay-off, it is only optimal in minimizing loss. However, if an opponent shows weak play, a strategy minimizing loss would not react to this, since weak opponent play is not a danger to the program's loss. In contrast, a strategy trying to maximize pay-off could profit from these weaknesses. The challenge here is therefore to find an optimal strategy which can exploit weak play.

We came up with possible solutions for both challenges:

- First, when playing against more than 2 players and/or playing No-Limit instead of Limit Hold 'Em, a significant increase in state space is established. To cope with this problem, we introduce a sampling technique (MCTS). MCTS does not look at the whole state space, but only inspects the parts of it which could prove useful. This is done through sampling. MCTS is discussed further in Chapter 3.
- Secondly, in order to exploit weak opponent play we implemented a Bayesian opponent model proposed by Ponsen et al. [16]. This model is initialized using a general prediction of each opponent's cards and action at a certain moment in the game. The general prediction is further adapted to individual opponents over time. In Chapter 4, the opponent model will be discussed more extensively.

During the past several years, there has been an increase in scientific interest in the poker domain. A number of articles will be discussed here.

The Computer Poker Research Group at the University of Alberta conducted a vast body of research. The Research Group designed various poker programs which can compete at several levels, some can even challenge professional poker players. In one of their bots (VexBot), they implemented a tree-based search technique called Adaptive Imperfect Information Search Tree Search [7]. This technique consists of implementing a search tree in a poker bot, which adapts to the decisions of the opponent. At first, the actions taken on basis of this technique will not be very accurate, but in time, the tree adapts to the opponent's playing style, resulting in intelligent poker decisions. VexBot was tested in the domain of Limit poker, in which it yielded good results.

Further, Gilpin et al. [13] focused their research on designing a bot which would be able to play No-Limit poker heads-up (1 vs. 1). The three techniques they incorporated in their program were:

- A discretized betting model to deal efficiently with the most important strategic choice points in the game.

- Abstraction algorithms to identify strategically similar situations in the game, used to decrease the size of the state space.
- A technique used to automatically generate the source code for an equilibrium-finding algorithm. This proved to be more efficient than a general-purpose equilibrium-finding program.

The program designed using these techniques yielded good results in No-Limit heads-up games, ending second in the 2007 AAAI Computer Poker Competition[1]. However, this research still focuses on playing according to a Nash equilibrium strategy and did not implement any opponent modelling.

Andersson [1] used the research done by Billings et al. and implemented some abstractions to design a poker program for heads-up No-Limit Hold 'Em. This program performed well with small stack sizes (i.e., players possess small amounts of money). Still, this program uses an optimal strategy (Nash equilibrium) rather than a maximal strategy (which can exploit weak opponent play). If opponent modelling and a search technique that can handle a larger state space would be added, the program could perhaps play a maximal strategy against multiple opponents.

A first step in opponent modelling was taken by Billings et al. [3]. They designed a poker program, Loki, which would compute the probability (weight) of the opponent possessing certain cards. After each betting action by the opponent, this probability was recomputed. This resulted in an opponent model which could predict an opponent's hand cards (to a certain extent). The results from the conducted experiments with Loki showed that its play was improved using this model. Still, according to the researchers, the model is not very sophisticated. Better results could be reached by importing more domain knowledge, such as combinations of actions within the same betting round [3]. Furthermore, Loki was designed for heads-up Limit Hold 'Em, so it was not ready to play No-Limit poker against multiple opponents.

In conclusion, we present in this research two techniques that can improve a poker program to be able to play against multiple opponents in a No-Limit Hold 'Em game. These techniques are a sampling search technique (MCTS) and a Bayesian opponent model. In Chapter 5 we will take a look at the results of our experiments and see whether these techniques prove valuable in designing a poker program.

[1] Results are available at http://www.cs.ualberta.ca/ pokert/2007/index.php.

Chapter 3

Monte-Carlo Tree Search

This chapter focuses on the search technique used in our research, MCTS. First we will provide an overview of the basics of MCTS, after which we will discuss the way in which we implemented MCTS in poker.

3.1 Monte-Carlo Tree Search in general

The game-theoretic tree approach we use in this research is Monte-Carlo Tree Search (MCTS) [11]. We will first explain why we chose this technique for our research and then we will explain its workings in general (not applied to poker).

As described in Chapter 2, poker research has come a long way. Research started in the domain of Limit Hold 'Em, which is less complex than No-Limit Hold 'Em. This difference in complexity is due to the use of a fixed raise amount in Limit poker. Because of the difference in complexity, techniques that resulted in good performance in Limit Hold 'Em are not guaranteed to come up with the same performance in No-Limit research. For instance, the Adaptive Imperfect Information Search Tree Search used by Billings et al. [7] would not yield the same performance in No-Limit poker as it does in Limit poker.

In Billings' research, the expected value (the evaluation function) of a node in the tree is based on what the program knows about the opponent's actions in earlier games. This knowledge is limited, since the amount an opponent raises is known beforehand, because of the Limit Hold 'Em rules. If Billings' evaluation technique would be applied in No-Limit poker, the expected value would not be accurate at all. Therefore, we decided to take a different approach. According to [10], designing a normal evaluation function for a tree-search method can be quite hard, depending on the domain. Therefore a different approach was proposed: Monte-Carlo Tree Search. This method basically performs a large number of simulations, of which the results are processed in order to select the optimal action. These simulations are not done randomly, they are executed according to a selection mechanism.

MCTS originates from the Monte-Carlo simulation method, which does a number of random simulations from a certain point in the game and then chooses the action which eventually (through the simulations) reaches the highest expected value. One simulation in the original Monte-Carlo technique consists of searching the whole state space for actions which are taken in a game to transform it from the starting point to an end situation. A way to represent this is with a search tree, in which every possible game situation is represented by a node. Nodes are interconnected by actions; if executing action 1 in game situation A results in game situation B, then B is considered a child of its parent A, and is interconnected through action 1. The starting point is represented by the root node and an end situation is represented by a leaf node (i.e., a node without children).

In order to traverse the whole tree, as is done in original Monte Carlo, the whole search tree should be known beforehand. However, the more complex a game, the more extensive the search tree will be. This can put a strain on computer memory. Furthermore, the more complex a game, the more simulations need to be run for original Monte Carlo to be accurate. The number of simulations the original Monte-Carlo approach needs could easily be 100,000, depending on the domain. Because of these two limitations, the original Monte-Carlo search technique is too slow to use in poker, which is a highly complex game. When we show how Monte-Carlo Tree Search surpasses the limitations of original Monte Carlo, we see that it can be much more effective in poker:

1. Limitation 1: the more complex a game, the bigger the impediment on computer resources. MCTS solves this problem by not taking into account the whole search tree. As we will discuss hereafter, MCTS starts with only the root node and gradually builds the search tree, while simulating games. The building of the tree is done selectively, which means building nodes only if they seem valuable (i.e., have a relatively high expected value). This leads to only keeping the most important parts of a search tree in memory, therefore decreasing the load on computer resources.

2. Limitation 2: the more complex a game, the more simulations are needed to be accurate. Original Monte Carlo runs simulations randomly, therefore dividing attention equally between all the paths from the root node to a leaf node. MCTS, however, uses a selection mechanism to decide what actions to take (i.e., what nodes to choose) during simulation. Depending on the parameters in the selection mechanism, a balance between exploration (building the search tree randomly) and exploitation (choosing only the nodes with highest expected value) is found. We will discuss this more extensively in the rest of this chapter.

Because of the way it handles the limitations of original Monte-Carlo simulations, we chose MCTS to use in our program. We will now discuss the workings of MCTS in general.

In MCTS, every node represents a state of the game. Properties of a game state which are important for MCTS are:

- the value of the game state. This normally is the average of the reward of all simulated games that visited this node.

- the visit count of this node. Every time this node is visited, this number increases by one.

One important property of MCTS is that the tree is built up along the way. This means that normally, MCTS starts with a tree containing only the root node. We will now dive into the workings of MCTS and meanwhile try to give an impression of how the tree is built up gradually. While MCTS runs, it repeats 4 steps, as illustrated in Figure 3.1, taken from [11]. The 4 MCTS steps are:

1. Selection: starting from the root node, the tree is traversed to a leaf node (L). When MCTS is started, it starts with a tree containing only the root node (see the top node in figure 3.1). Therefore, selection is quite straightforward in this situation: the root node is selected, since it does not have any children and therefore is a leaf node.

2. Expansion: upon reaching L, children of L are added to the tree. This only happens if L is not a terminal node (i.e., there are still some actions that can be performed by a player). The actions that can follow directly from the game state represented by the node selected during the selection process are added to the tree as child nodes of the root node. It is also possible that the node selected during selection is a terminal node, which means it is an end situation. In this case, no children are added to the tree. In Figure 3.1, the search tree consists of 4 layers and a fifth one is added by expanding the selected node in the fourth layer.

3. Simulation: if L is a terminal node, the game outcome is clear and no simulation takes place. If L is not a terminal node, a game is simulated from one of the children of L. This can take up many different forms, depending on the domain, varying from playing a whole game (with multiple actions of each player) to simply evaluating the current game state. When we look at a situation where there is a root node and a number of child nodes in the tree, one game is simulated from the game state represented by one of the child nodes. Simulation is implemented depending on the domain and is represented by the third part of Figure 3.1.

4. Backpropagation: depending on the outcome of the simulation, a positive or negative reward is returned. This reward is then backpropagated through the tree, in a way that the value of every node visited prior to this simulation is updated with the reward of the simulation. In our example, a game was simulated from one of the child nodes of the root node. The simulation returns a reward, with which the game state value of the child node at hand is updated. Normally, a reward is -1 (loss), 0 (draw) or +1

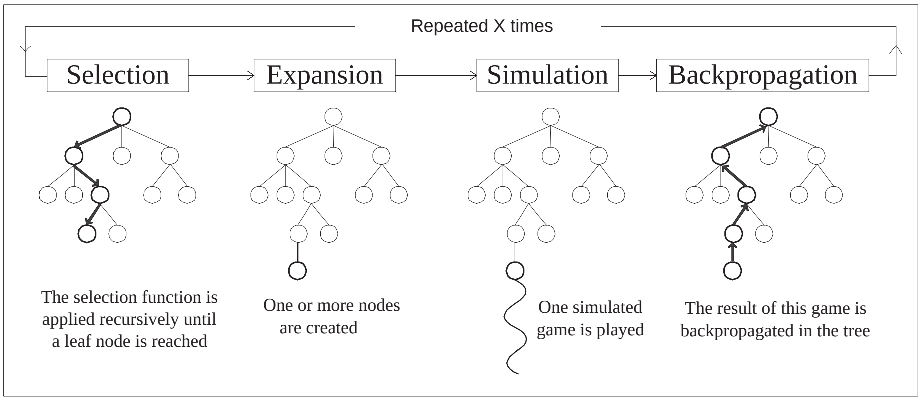

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 3.1: An iteration of MCTS.

(win). An example: if a game is simulated from a child node with game state value -1 (since a loss occurred in the last MCTS iteration), and the simulation returns a win, the node's value is updated with +1 returning an average value of 0. Now, the value of the child node's parent is updated with the reward. This is repeated until the root node is reached (see Figure 3.1).

The second property (visit count) of a node is incremented once in all nodes that are traversed during one MCTS iteration. The combination of properties (a node's value and visit count) can be used in the process of selection and expansion. An important issue in MCTS is to have a good balance between exploration and exploitation. At first, when the tree only contains the root node, there should be a high level of exploration. This will cause the tree to grow fast. After this exploration phase, exploitation should be more important in order to select the nodes which prove the most valuable (i.e., return the highest reward) [11]. We will come back to this in Section 3.2.

3.2 Monte-Carlo Tree Search in poker

In Section 3.1 the technique of MCTS is explained. However, MCTS should be adapted to a certain domain in order to be effective. We therefore describe the way we implemented MCTS in a poker bot in this chapter. This is done in four steps: selection, expansion, simulation and backpropagation.

3.2.1 Selection

Selection is the process of traversing the search tree in order to reach a leaf node. A node in the search tree represents a game situation. In poker, players can take a number of actions: fold, check or call, raise or bet. Checking is the act of calling when the outstanding bet is equal to 0 and raising is the act of betting when the outstanding bet is higher than 0. In the tree, check and call are combined into a ‘c’-action and bet and raise are combined into an ‘r’-action. One can imagine a various number of raise amounts to be implemented. However, for simplicity and balance of the tree, only one raise node per choice point is implemented. The raise amount depends on the game phase.

Selection starts at the root node and depending on the policy used, it repeatedly chooses a node one level deeper, until a leaf is reached. In our program, we use three different selection policies:

Random

In this policy, an action is randomly picked. The three action possibilities are uniformly randomized, which therefore maximizes exploration.

UCT

Here we use a policy named Upper Confidence bound applied to Trees (UCT). This method is easily implemented and used in many programs, among which MCTS applied to the game of Go [11]. In UCT, I is the set of nodes (possible actions) reachable from the current node, p. Using the following equation, UCT selects the child node k of parent node p which has the highest expected value. The equation for UCT is

$$k \in argmax_{i \in I}(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}}) \tag{3.1}$$

where v_i is the expected value of the node i, n_i is the visit count of i, and n_p is the visit count of p. C is a coeffcient, which is optimized experimentally.

Opponent Model

If we want to simulate an action for our opponent, we look at the opponent model to see the probabilities of all the opponent’s actions at a certain moment. From this distribution we sample an action. In Chapter 4 this topic is explained in more detail.

As is clear, some boundaries are needed for the program to know what policy to use at what time during selection. First, the opponent model policy is always used if the program needs to sample an action for an opponent on which there is such a model. Second, the random policy and the UCT policy are used to sample our own actions. At first, the random policy is used to increase exploration. When a node’s visit count crosses a certain threshold T, UCT is used. This threshold consists of the minimum visits a node should have had. Furthermore, when we execute an experiment without use of the opponent model, we also use the random as well as the UCT policy for the actions of the opponent.

3.2.2 Expansion

In this process, the actual search tree is built. When a leaf node is selected, the possible next actions are used to add the child nodes to the tree. So, in practice, this comes down to a simple process of checking whether a leaf node is a terminal node (i.e., there are no further actions possible) or whether it can be expanded. If the latter is the case, the possible actions at that node are used to add new game situations to the tree as children of the leaf node.

3.2.3 Simulation

Simulation is a very important process in MCTS, since this process functions as the evaluation method used to rate actions by game outcomes. We chose to start the simulation as soon as a leaf node is visited. In poker, the choices one makes about simulation can be difficult: on the one hand, it is intuitively better to 'play out' a whole game. On the other, this consumes many resources and would endanger the accuracy of MCTS reached by a large number of iterations. Therefore, we chose to simulate an end-game situation: deal board cards and compare the hand of every player to see who will win the pot, which consists of the bets done until that point in the game. Of course, one problem exists with this approach: if we simulate a game from a point where one player just raised, he has more money at stake than (some of) his opponents. Therefore we implemented a process that simulates a call or a fold for each opponent that does not reach the outstanding bet.

The rewards that are the result of a simulation consist naturally of the amount of money the player loses or wins. The process of simulation is executed twice every MCTS iteration: once to get the reward for the bot itself and once to get the rewards for the opponents. The former simulation uses the cards the bot got dealt in the real game and the latter uses sample cards. The reason for this is best explained with an example: suppose the bot plays against one opponent. Two Aces are dealt to our bot. If we then do one iteration of MCTS and the raise action is selected for the opponent, he probably loses the game in simulation, since the bot's hand is very strong. This leads to the opponent's rewards (and therefore value) of the chosen leaf node being lower than if the bot for instance had a 10 and a 2 in his hand. Therefore, in the tree, the opponent will probably fold sooner, leading to less exploration and exploitation of the branch of the tree that starts with a raise action of the opponent. This causes the bot's value of that branch to be underestimated and so we chose to do two different simulations per MCTS iteration.

3.2.4 Backpropagation

The normal backpropagation strategy consists of computing the average of the reward of all the games that are simulated after visiting the node at hand. According to [11], the best results are obtained by using this backpropagation strategy, which we therefore also use in our research.

Chapter 4

Adding domain knowledge to MCTS

This chapter discusses the way in which we implement domain knowledge in the poker program. In section 4.1 we will explain the workings of the opponent model we used for this purpose. Subsequently, in section 4.2, we will look at the way in which our program uses the opponent model to optimize its play.

4.1 Opponent model

In this section, we will focus on the workings of the Bayesian opponent model proposed by Ponsen et al. [16]. First we will discuss why we chose Prolog as our language of choice to implement this model.

Prolog is a logical programming language, in which facts and relations can be declared. These are stored in a simple database and can be queried. In our opponent model, a lot of relations should be stored. For example, the model can learn that opponent A will fold preflop if he does not have a Queen or higher in his hand. This, of course, is a simple rule, but the model should be able to learn more complicated rules. The more complex a rule, the more work is done when implementing it in an object-oriented language such as Java. This is not the case in rule-based languages such as Prolog.

An example rule could deal with the notion of pot odds. Pot odds reflect the ratio between the minimum amount a player has to call and the current size of the pot. If the pot consists of $ 100 and the amount to call is $ 10, the pot odds for a player are 0.1. An example rule concerning pot odds could be the following: opponent B has a probability of 0.7 to raise on the flop if his pot odds are higher than 0.5 and maximally 2 opponents have raised already in this betting round and maximally 4 raises were done previously in the game. To be able to efficiently query a database containing these rules, an object-oriented language would not be suited. In Prolog, however, a programmer does not have to clarify how the program should execute an assignment (as should be done in an object-oriented language). The programmer only has to input the query and then ask for every combination of data in the database that fulfills the constraints of the query. Another advantage of using Prolog in our research is that it can easily deal with multiple opponents without adding any code.

Now we will dicuss the workings of the opponent model in detail. As mentioned in Chapter 1, in poker it can prove very valuable to adapt one's playing style to the playing style of the opponent(s). Knowledge of the opponent consists of predicting two things: (1) the opponent's cards at a certain time in the game and (2) the action of an opponent at a certain moment in the game. Ponsen et al. [16] developed a model in which both these elements are computed from a limited amount of experience using a general prior, which after time adapts to each individual opponent. The way in which adaptation to an individual opponent is implemented is that the model starts with a general prior distribution over possible action choices and outcomes. Using a corrective function for this prior the model will adapt according to observed experience (i.e., observed actions and cards). An added difficulty to this model is that there is a lot of hidden information in poker. For example, if a certain opponent raises a large amount of money so that every other player folds, the opponent wins without showing his hand. This could mean that the opponent has a very strong hand but he could also have a weak hand. The background of the opponent model is explained here:

Consider a player p performing the i-th action a_i in a game. The player will take into account his hand cards, the board B_i at time point i and the game history H_i at time point i. The board B_i specifies both the identity of each card on the table (i.e., the community cards that apply to all players) and when they appeared, and H_i is the betting history of all players in the game. The player can *fold*, *call* or *bet*. Although MCTS can easily deal with multiple different actions (like check and call), the benefit for distinguishing these actions from each other in the opponent model is not significant. Therefore, we consider *check* and *call* to be in the same class, as well as *bet* and *raise* and we do not consider the difference between small and large calls or bets at this point.

Ponsen et al. proposed a two-step learning approach. First, functions are learned predicting outcomes and actions for poker players in general. These functions are then used as a prior, and we learn a corrective function to model the behavior and statistics of a particular player. The key motivations for this are first that learning the difference between two distributions is an elegant way to learn a multi-class classifier (e.g., predicting distributions over 2 + (52 × 51 / 2) possible outcomes) by generalizing over many one-against-all learning tasks, and second that even with only a few training examples from a particular player already accurate predictions are possible. In the following description, the term example references a tuple (i, p, a_i, r_p, H_{i-1}, B_i) of the action a_i performed at step i by a player p, together with the outcome r_p of the game, the board B_i and the betting history H_{i-1}.

Ponsen et al. applied TILDE [9], a relational probability tree learner to learn the corrective function (i.e., the function that learns the difference between the prior and the modelled player). A decision tree that separated instances from the prior distribution with observed instances from the player was learned for each phase of the game. The language bias used by TILDE (i.e., all possible tests for learning the decision tree) includes tests to describe the game history H_i at time i (e.g., game phase, number of remaining players, pot odds, previously executed actions etc.), board history B_i at time i, as well as tests that check for certain types of opponents that are still active in the game.

We will consider the difference between small and large calls or bets as features in the learned corrective function, which is used to adapt to individual opponents. We limit the possible outcomes of a game r_p for a player p to: 1) p folds before the end of the game, 2) p wins without showing his cards and 3) p shows his cards (there is a showdown). This set of outcome values also allows us to learn from examples where we did not see the opponents cards, registering these cases as *win* or *lose* without requiring the identities of the cards held by the player. The learning task now is to predict the outcome for an opponent $P(r_p|B_i, H_i)$ and the opponent action (given a guess about his hand cards) $P(a_i|B_i, H_{i-1}, r_p)$ [16].

Because the model Ponsen et al. proposed handles hidden information very well, it is very interesting for our research.

4.2 Model tests used in MCTS

In this section, we will discuss the way in which MCTS is influenced by the opponent model. MCTS without opponent model treats every opponent as rational and therefore does not look at possible mistakes that can be exploited. Therefore, the opponent model was implemented (as explained in the previous section).

As mentioned before, the opponent model we use computes a probability distribution for two things: (1) the cards an opponent possesses and (2) the action an opponent can take at a certain moment. First, we look at card prediction.

Normally, using only MCTS, the opponent's cards would be sampled randomly and one would be dependent on the numerous iterations of MCTS to give a good uniform card distribution. However, using the opponent model, card sampling can be done much more accurately. For instance, if the opponent model 'knows' that the opponent will always raise when he has a King or higher, it will attach a higher probability to the opponent having a King or an Ace if he has raised a lot. Attaching a higher probability to a card leads to more MCTS iterations executed with the opponent having a King or an Ace. This improves the computed expected value and in turn improves the playing style of our program.

In MCTS, opponent's actions are sampled using UCT [11]. This technique computes the optimal action using the visit count and the expected value of a node. UCT is based upon the notion that the opponent is a game-theoretic player. This assumption can yield good results in games where there is little or no opponent modelling needed, but in poker this does not relate well to human play. In Chapter 1 we already discussed why opponent modelling can be an important feature of a poker program. Human players are known to switch strategies, bluff or lose focus and make mistakes. In other words, their play can be very dynamic. If a poker program is able to foresee mistakes and bluff moves, it can increase its winnings considerably.

We will now look at how opponent modelling increases the accuracy of MCTS by computing a probability distribution for the action of an opponent. In our program, when the tree has an opponent node under consideration (i.e., an opponent should take an action), first two opponent cards are sampled from the card distribution based on the current game state. Then, the opponent model is queried for the probability of every possible opponent action based on the game state and the sampled opponent cards. From this, using a prior, a probability for each action is computed. See Section 4.1 for details. In addition, the UCT value of every possible opponent action is computed (see Section 3.2 for details). Here an important program parameter (balance parameter B) is implemented, which, depending on its value, increases emphasis on one value while decreasing emphasis on the other. The equation using this parameter is:

$$P_a = PO_a \times B + PU_a \times (1 - B) \tag{4.1}$$

where P_a is the final probability for action a, PO_a is the probability for action a computed using the opponent model, PU_a is the probability for action a computed using UCT and B is the balance parameter. B can be used to experimentally find the optimal balance in opponent modelling between using MCTS or the Bayesian opponent model. In case that $B = 0$, the opponent is treated as a rational perfect game-theoretic player (i.e., full use of MCTS), while in case that $B = 1$, the program tries to maximally exploit patterns in the opponent's strategy (i.e., full use of the opponent model). MCTS runs typically for a large number of iterations, which leads to the opponent model being queried very often. This results in expected values for every action the program can take that are influenced by the opponent model.

In conclusion, we improved MCTS to be able to adapt to an opponent's playing style in two ways:

1. We compute a probability distribution for the opponent's hand.
2. We compute a probability distribution for the opponent's actions with two techniques: using the opponent model and using UCT (as in standard MCTS). With the use of balance parameter B, we compute the final probability distribution for the opponent's actions.

Both these computations are executed multiple times during every MCTS iteration, resulting in a poker program combining a powerful search technique (MCTS) with domain knowledge (the Bayesian opponent model). This process is displayed in Figure 4.1 and is explained as follows:

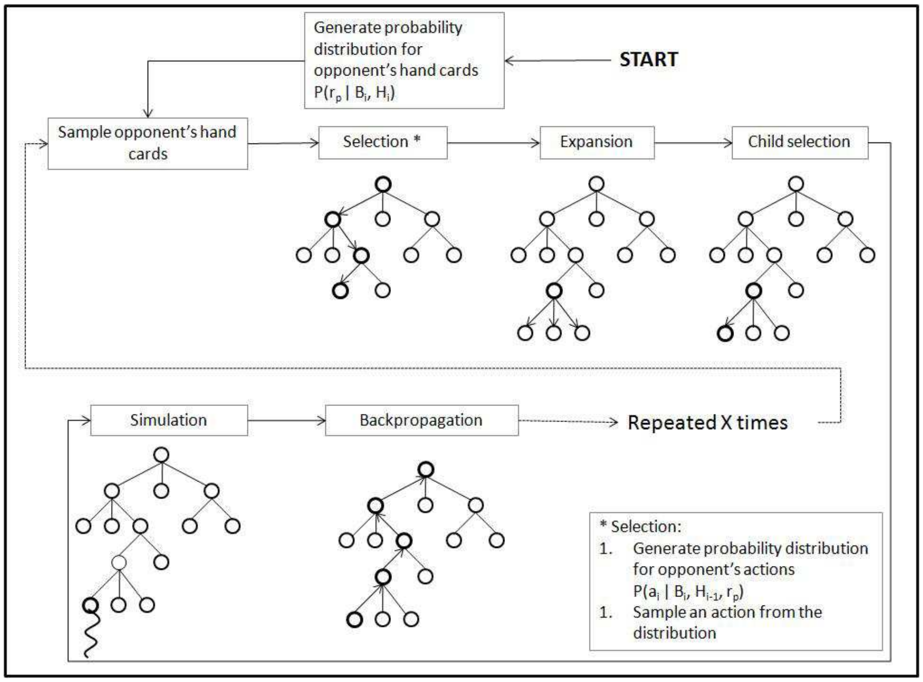

Generate probability distribution for opponent's hand cards
P(r_p | B_i, H_i)
START
Sample opponent's hand cards
Selection *
Expansion
Child selection
Simulation
Backpropagation
Repeated X times
* Selection:
1. Generate probability distribution for opponent's actions
P(a_i | B_i, H_{i-1}, r_p)
1. Sample an action from the distribution

Figure 4.1: MCTS with opponent modelling.

The first thing executed when starting an MCTS iteration with opponent modelling is generating a probability distribution of the opponent's hand cards. This is done only once for a set of MCTS iterations. The distribution is based upon the current situation in the game, the previous actions of the opponent in the game and the opponent's actions in previous games (along with their outcomes).

When the opponent's cards distribution is generated, the second step is sampling an opponent's hand cards from the distribution. This is done randomly, since the knowledge about the opponent is already contained in the distribution. The sampled opponent hand will be used throughout the current MCTS iteration. In a next iteration, a new opponent hand will be sampled.

As is done in regular MCTS, now the process of selection will start. This process always starts at the root node and traverses the search tree until a suitable leaf node is reached. During the selection process, for nodes for which an action should be taken by the program itself, the regular selection techniques explained in Chapter 3 are used, which are Random selection and UCT selection.

However, if the selection process traverses a node of the opponent, a different approach is used to select an action. This is done as follows:

1. First, a probability distribution for the opponent's actions at that node is generated. This distribution is based upon the game situation in the current node, the opponent's possible actions, the sampled opponent hand cards, the previous actions of the opponent in the game and the opponent's actions in previous games (along with their outcomes).
2. Secondly, the same probability distribution is generated using UCT.
3. After two probability distributions for the opponent's actions have been generated, they are combined using balance parameter B. Depending on this parameter is the influence of both probability distributions and therefore of both techniques (opponent model and UCT).

Now, still in the process of selection, there is a probability distribution for the opponent's actions from which an action is sampled. The sampled action will be the next node under consideration in the selection process. Selection is done iteratively until a leaf node is reached.

When a leaf node is reached, the program will switch from selection to expansion. It will try to expand the leaf node to add child nodes to the tree. An important part of the MCTS technique is that the search tree is built iteratively. Therefore, when doing an MCTS iteration for the first time, the tree consists of only the root node. Gradually, as more MCTS iterations take place, the tree is built up through the process of expansion. When a new cycle of MCTS iterations is started, the game situation changed, so a new tree should be built up from scratch.

After finishing the expansion process, again the process of selection takes place to select one of the newly added child nodes. This is only done if child nodes were added. Otherwise, the node under consideration does not change.

At this time, a simulation game is played from the selected node. This process works the same as MCTS without opponent model. The only difference is that the simulation game is executed with the opponent hand cards sampled by the opponent model, instead of just randomly sampling the opponent's cards.

As a last step in MCTS with opponent modelling, the backpropagation process is initiated, which backpropagates the result from the simulated games up through the tree. There is no difference in backpropagation in MCTS with or MCTS without opponent modelling.

Here is presented the pseudocode representing MCTS with opponent modelling:

```
getProbDistrHandCards(Opponent);
while(count < MAX#_MCTS_ITERATIONS)
{
    sampleOpponentHand(Opponent);
    Node selectedNode = rootNode;
    while (selectedNode.numberOfChildren > 0)
        selectedNode = select(selectedNode);
    boolean nodesAdded = expandNode(selectedNode);
    if(nodesAdded)
        selectedNode = select(selectedNode);
    simulateGame(selectedNode);
    backpropagate(selectedNode);
}

function getProbDistrHandCards(Opponent)
{
    assertDummyAction();
    foreach (possibleCardCombination :  Opponent)
        getCardCombinationProbability();
    retractAssertions();
    return cardProbabilities;
}
function select(Node n)
{
    Node returnNode = n;
    if (n.player == thisPokerBot)
        returnNode = doNormal(); //use random and UCT selection
    else
    {
        distrActionsModel = getProbDistrActionsFromModel(Opponent);
        distrActionsUCT = getProbDistrActionsFromUCT(Opponent);
        combineProbDistr(distrActionsModel, distrActionsUCT);
        returnNode = sampleOpponentAction(Opponent);
    }
    return returnNode;
}
function getProbDistrActionsFromModel(Opponent)
{
    assertBoardHistory();
    assertDummyAction();
    foreach (possibleAction :  Opponent)
        getActionProbability();
    retractAssertions();
    return actionProbabilities;
}
```

In Chapter 5 we will investigate whether this program is able to defeat several opponents.

Chapter 5

Experiments

In this chapter, we describe the experiments performed. We start by discussing the parameters used in the experiments and the ways in which their results are interpreted. Then we will discuss experimental results, starting with experiment 1, in which our program plays against a relatively simple rule-based poker bot (ACE1). After that, the results of experiment 2 will be covered. In experiment 2, our program faces a more complicated poker bot, Poki [8]. Output from Poker Academy Pro, the software used in the experiments, can be found in Appendix A and B.

5.1 Experimental setup

In our experiments, a number of parameters are used. We will discuss them here:

1. The number of MCTS iterations. As discussed in Chapter 3, MCTS is a technique of which the accuracy improves with more iterations. Therefore, the higher this number, the better results MCTS will yield. Unfortunately, this parameter is the most limiting factor in terms of time. For instance, running experiment 1 without model took more than 5 hours, using 1000 iterations every time an action of our program was requested. One can imagine that when the opponent model (which is written in Prolog and therefore needs the aid of a Java-Prolog interface) is used, experiment times increase drastically. Therefore, we chose to set this parameter to 1000 iterations in each experiment. Using a value of 1000 for this parameter is relatively low compared to the research with MCTS applied to Go [11]. There, the researchers worked with 20,000 MCTS simulations per move.

2. The UCT coefficient C [11]. This coefficient is used during the MCTS process of selection. C influences the balance between exploration (selecting nodes through which few simulations have been conducted) and

exploitation (selecting nodes that have a high expected value). We found that a value of 2.0 for C results in a good amount of exploration before exploitation takes over. This value was used in each experiment.

3. The UCT threshold T. The value of this parameter is the minimum visit count a node should have before another selection strategy is used. In our experiments, T was set to 100, which leads to a random selection strategy being performed for nodes with visit count $<$ T. UCT is used when a node's visit count $>=$ T.

4. The balance parameter B. This parameter finds a balance between the influence of UCT and the influence of the opponent model on the selection of nodes during MCTS. When $B = 1$, an experiment is executed with the full use of the opponent model in MCTS, while when $B = 0$, an experiment using only standard MCTS is conducted. When doing an experiment with a combination between MCTS and the opponent model, B should have a value of $0 <= B <= 1$. For the experiments described in this thesis, we used a value of 1.0 for B, which results in maximal influence of the opponent model and minimal influence of UCT in selection. We chose a value of 1.0 in our experiments to maximize the difference between results of MCTS alone and MCTS combined with opponent modelling.

Results of the experiments are displayed using four characteristics: (1) seen flops, (2) preflop aggression, (3) postflop aggression and (4) the profit expressed in big blinds per hand. These characteristics are derived from the player characteristics sheets that the poker software (Poker Academy Pro 2.5.7) provides. We will now discuss these characteristics:

- Seen flops: this number shows the amount of games a bot participated in a game. Most of the time, when a poker player has a weak hand, he will fold preflop. Therefore, if a player sees a flop, he must be fairly sure of its chances. This is a measure for the probability of winning perceived by the poker program.

- Preflop aggression: this measure is computed using the following equation:

$$(numberOfRaises + numberOfBets)/numberOfCalls \tag{5.1}$$

  We can see that if a player bets a lot, it will have a higher aggression value than a player that calls a lot.

- Postflop aggression: this statistic is computed in the same way preflop aggression is computed. The difference is that postflop aggression considers betting and calling on three betting rounds, instead of one. Furthermore, poker players tend to play a different strategy postflop in comparison with preflop, due to more information being available (e.g., board cards are dealt).

- Big blind per hand: this is a measure of income rate. For every hand (i.e., every game), the total return is expressed in big blinds. If a large number of hands (games) is played, this provides a reliable measure of the strength of a certain strategy. For example, if 30 games are played in an hour and the income rate is 0.10 bb/game, it means that the program wins 3 times the big blind in an hour. Some human players prefer the measure of bb/hour to dollars/hour, since it does not depend on the speed of the games [5].

5.2 Experiment 1

In the first experiment, we wanted to simply show that the opponent model can be a contribution to a poker bot. We therefore let our bot play two games against a relatively simple opponent: ACE1. This bot is rule-based, which is shown in Figure 5.1.

Table 5.1: Rule-based poker bot ACE1.

Preflop	card group < 4	bet
	card group < 9	call
	otherwise	fold
Flop	sum cards >= 16	bet
	sum cards >= 12	call
	otherwise	fold
Turn	sum cards >= 20	bet
	sum cards >= 16	call
	otherwise	fold
River	sum cards >= 22	bet
	sum cards >= 20	call
	otherwise	fold

In Table 5.1, one can see that ACE1 decides what actions to take based only on game stage (preflop, flop, turn, river) and on one property of his hand, either the card group or the sum of his cards. First we will explain card groups. As described in [5] when discussing pre-flop behavior, "the best known and most widely respected expert opinion on preflop play is that of David Sklansky, a professional poker player and author of the most important books on the game". Sklansky came up with the idea of card groups, which are a "hand classification scheme" with "a strong correlation between [card group] rankings and the results of roll-out simulations". For a more detailed description of card groups, see [5]. Sklansky proposed 8 groups, of which group 1 contains the strongest combination of cards. In our research, we used 9 groups to make the difference between hand strengths a bit more clear. As shown in Table 5.1, ACE1 will bet if his hand cards fall into a group better than 4, as does an example hand of Ten Jack suited. If, on the other hand, its cards qualify for a group stronger than group 9 (which is for example a 3 and a 5 suited), ACE1 will call. Thus, ACE1 will only fold if his hand strength is very weak. This rule is already quite a strong one, separating the best hands from the rest. Therefore, our opponent model should be able to notice, after a number of games, ACE1's hand difference in raising or calling preflop.

For the second property of ACE1's hand, the sum of the cards in his hand, we should explain that each card's value is downshifted by 2 in our model. This means that a 2 has a value of 0 and an Ace is mapped to a value of 12. Therefore, the sum of the highest card combination (Ace and Ace) is 24. As shown in Table 5.1, once ACE1 is still active after the preflop stage, it completely bases its actions on the sum of its cards. Like ACE1's preflop decision making, this behaviour should be no problem for the opponent model. As can be concluded from the above, ACE1 clearly is not a game-theoretic perfect player, especially since it is oblivious to postflop hand strength. It can be expected that a combination of MCTS and the opponent model will exploit the play of this opponent.

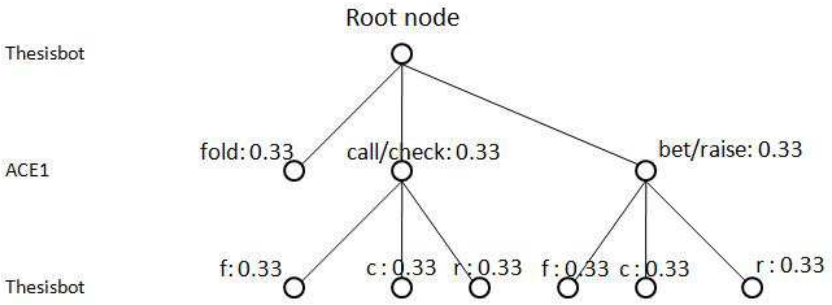


Figure 5.1: Thesisbot against ACE1, using MCTS.

In Figure 5.1 a simple search tree is pictured from a game between our program (Thesisbot) and ACE1, during the flop phase. As can be seen, in this figure, MCTS does not give a very accurate probability distribution for ACE1's actions. This is caused by the fact that not many MCTS iterations have occurred yet. From this, one can see that an opponent model could be a valuable addition: because there is information about the opponent, the probabilities for its actions can be estimated more accurately. Figure 5.2 shows an example of this. Here we see that, although the tree has not been fully built yet, the estimates for ACE1's actions are more accurate than in Figure 5.1. If we look at Table 5.1, we can see that during the flop, ACE1 has about 50% chance of folding. This is represented in the tree as a 0.5 chance of ACE1 choosing f (fold). Furthermore, we can see that ACE1 does not take into account the actions of its opponents: whether our program calls/checks or bets/raises, ACE1 has the same probabilities for its actions.

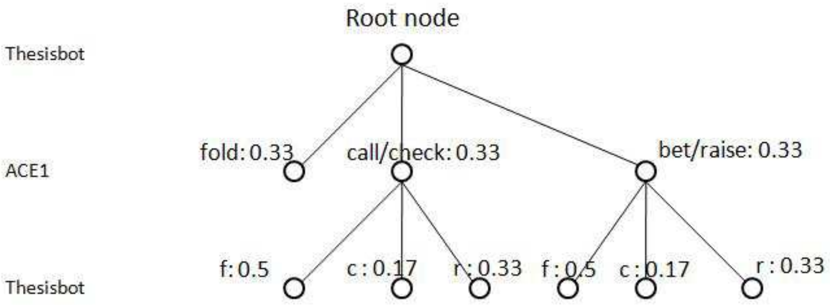


Figure 5.2: Thesisbot against ACE1, using MCTS with opponent modelling.

We will now discuss the results of experiment 1. We divided experiment 1 in two parts: (1) using only MCTS (without knowledge supplied by the opponent model) and (2) using MCTS in combination with the opponent model. In both cases, our program played around 10,000 games against ACE1, which should be enough to eliminate (to a certain extent) the uncertainty with which one deals in poker due to random card dealing. Table 5.2 shows the results of experiment 1, which are extracted from the output in Appendix A.

Table 5.2: Results of experiment 1.

	seen flops	preflop aggression	postflop aggression	bb/hand	#games
no model	6435	0.5	4.7	1.09	10010
model	4710	7311	8.3	1.67	10010

As shown in Table 5.2, an important observed difference between the experiment with and without the opponent model is the difference in preflop aggression: 0.5 against 7311. We can explain this difference by considering the technique used in the first condition (only MCTS, no opponent model). In this condition only MCTS was used, which does not consider opponent mistakes.

MCTS works under the assumption that an opponent is rational and will take the actions with the highest expected value. However, this does not include typical poker-playing strategies like slowplaying (not raising with a strong hand) or bluffing (raising with a weak hand). Because of this assumption, the actions selected by MCTS are not aggressive: it mostly raises when having a very strong hand. When we look at the preflop aggression in the second condition (MCTS with opponent model), MCTS's assumption that its opponent is rational is released: through the opponent model, the program 'knows' that ACE1 will bet or raise only if his hand is of card group 1, 2 or 3. Therefore, if our program is dealt a stronger hand than the hand of ACE1 (of which he has an estimate through ACE1's actions), he will raise. However, if our program is the first to act in a game, it does not have an estimate of the hand strength of ACE1. Therefore, it will take a more conservative strategy.

The same is true, though less explicitly, for postflop aggression. Because of the fact that ACE1 has a rule base with static boundaries, our program can get a good estimate of its cards by looking at its actions. He 'knows' that if ACE1 bets or raises on the flop and later calls on the turn, that the sum of its cards is 16, 17, 18 or 19. From this, he can conclude that ACE1's highest possible pocket pair (two cards with equal value) is JJ (2 Jacks). Furthermore, he can conclude that if ACE1 possesses an Ace, its kicker (the other card) would not be higher than a 9. All these conclusions and inferences are (implicitly) stored in the opponent model, and are queried when necessary.

The third characteristic (big blind per hand) gives information about the strength of the program's strategy against ACE1. It shows that using only MCTS, our program will win easily (1.09 big blind per game on average). If we add particular information about the opponent to MCTS, it will perform even better: it wins 1.67 big blind per game on average.

Furthermore, the fact that our program can get a good estimate of ACE1's hand strength by reviewing its actions causes the number of seen flops to drop when using the opponent model. This can be understood by imagining why our program would not see a flop: when it suspects that its opponent has got a stronger hand. From this, the significant increase in revenue (from 1.09 to 1.67 big blind per game) is even more important: when using opponent modelling, our program can make a quite accurate estimate whether he will win the game. If not, he will not see the flop and therefore minimize its loss.

5.3 Experiment 2

In the second experiment, we wanted to see how strong our program is against a more complicated bot: Pokibot [5]. Therefore, we randomly chose a variant of Pokibot, Anders, to play against our bot two times. The first time without the use of the opponent model (only using MCTS) and the second time using MCTS combined with the opponent model. We will now discuss the workings of Pokibot.

The way in which the Poker Academy Pro software describes Anders is: “A standard opponent with a moderate pre-flop hand selection. Likes to check-raise and slowplay big hands”. This means he is not likely to fold preflop and he will probably not raise if he has a very strong hand. We will now look at how Pokibot is built up:

> In addition to storing public game context like round number, bets to call, betting history, number of active players, position, etc., Poki stores private information: its current hand and a collection of statistical opponent models. The first-round betting decisions are made with a simple rule-based system. The opponent model (essentially a probability distribution over all possible hands) is maintained for each player participating in the game, including Pokibot itself. The Opponent Modeler uses the Hand Evaluator, a simplified rule-based Betting Strategy, and learned parameters about each player to update the current model after each opponent action. After the flop, the Hand Evaluator in turn uses the opponent model and the game-state information to assess the value of Pokibot’s hand in the current context. Thus, there is a certain amount of cyclic feedback among the core components of the system. The evaluation is used by a more sophisticated rule-based Betting Strategy to determine a plan (how often to fold, call, or raise in the current situation) and a specific action is chosen. The entire process is repeated each time it is our turn to act. For a more advanced decision procedure, the Simulator iterates this process using different instantiations of opponent hands. [5]

As one can see, Pokibot is a fairly complicated poker playing program. Furthermore, the results from experiments conducted with Pokibot show that it is a fairly tough opponent: it yielded a result between +0.10 and +0.20 bb/hand in the lowest-level games (with beginning human poker players as opponent). In games with stronger opposition its play resulted in an income between +0.07 and +0.10 bb/hand. For a more detailed explanation of Pokibot and its experiments, see [5].

We will now discuss the results of experiment 2. As we did with experiment 1, we divided experiment 2 in two parts: (1) using only MCTS (without knowledge supplied by the opponent model) and (2) using MCTS in combination with the opponent model. In both cases, our goal was to let our program play around 10,000 games against Pokibot in both conditions. Table 5.3 shows the results, which are extracted from the Poker Academy Pro output (see Appendix B).

As shown in Table 5.3, the first observed difference between the experiment with and without the opponent model is the difference in preflop and postflop aggression: 0.7 and 1.0 against 0.2 and 0.8. We can explain this difference by considering what we know of Anders. We know that he likes to slowplay and checkraise when he has a relatively strong hand. The first condition of this

Table 5.3: Results of experiment 2.

	seen flops	preflop aggression	postflop aggression	bb/hand	#games
no model	9738	0.7	1.9	-0.50	10304
model	9444	0.2	0.8	0.06	10310

experiment only uses MCTS. As discussed earlier, MCTS treats an opponent as rational and therefore predicts that if Anders has a relatively strong hand, he will play accordingly (bet or raise). In the same way, it will probably predict that Anders will check or call with a moderate hand and therefore, our MCTS program does exactly what Anders wants it to do, namely wrongly estimating Anders' hand strength. This leads to a significant loss for our poker program (0.50 big blind per game, on average). When we look at the results of condition 2 (MCTS combined with the opponent model), we see that the introduction of the opponent model aids our program in not falling for the trick of Anders' slowplaying. Our bot's medium aggressive stance decreases to a more passive one.

This brings us to the fourth column, the income rate. As shown in Table 5.3, combining MCTS with the opponent model improves the income rate considerately. Where, in condition 1, our program consistently lost against Anders by 0.50 big blind per game, our program is able to beat Anders in condition 2 (generating an income of 0.06 big blind per game).

Remarkably, the difference in seen flops between our program with and without the opponent model is negligible (9738 vs. 9444). We can see that our program plays much more efficiently using the opponent model. It will almost never fold before the flop, but it can defeat a bot like Anders, which has a preference for slow-playing, albeit with minimal profit.

Chapter 6

Discussion

In this chapter, we will first draw conclusions from the results of our experiments, after which we will discuss the limitations of our research. Finally, we will present some recommendations for future research.

6.1 Conclusions

First, we can conclude that MCTS alone can be a valuable search technique to be used in poker bots. It can easily defeat a simple rule-based opponent by winning 1.09 big blind per game on average (see experiment 1). However, when we look at the results of experiment 2, MCTS alone cannot beat a more complicated poker program (Poki). Using only MCTS against Poki translates in losing -0.50 big blind per game on average, therefore, we conclude that MCTS alone will not be able to compete against experienced human poker players. We think that this is caused by the inherent nature of the game of poker, which consists of incomplete information and partially observable states. In games such as Go, where only MCTS did yield good results [11], there is no incomplete information and the states are fully observable. When it comes to designing a poker program which can compete against experienced human poker players, MCTS can only lay the groundwork.

Secondly, we conclude that the Bayesian opponent model proposed by Ponsen et al. [16] improves the playing style which results from MCTS by adapting to a specific opponent. In experiment 1, the opponent was relatively simple, which caused our program to play with a very aggressive style using the opponent model. This aggressive style lead to a higher income rate (1.67 big blind per game on average) than the original (more passive) playing style which originated from using only MCTS. In experiment 2, our program played against a much more complicated opponent (Poki), which yielded good results against human players in the past [5]. The opponent model changed our program's medium aggressive style to a very passive one, in order to not be tricked by the opponent, with its preference for slowplay. This resulted in a positive income rate of 0.06 big blind per game on average.

Based on these conclusions, we can answer our research questions as follows:

1. How can we incorporate poker domain knowledge in Monte-Carlo Tree Search?
   Answer: Poker domain knowledge was incorporated in MCTS using the Bayesian opponent model proposed by Ponsen et al. [16].

2. Does using the MCTS algorithm yield good results in poker?
   Answer: The MCTS algorithm yields good results in poker when it is used for playing against a relatively simple rule-based bot in No-Limit Hold 'Em. A second experiment showed that using only MCTS does not yield good results against a more sophisticated opponent. This outcome could also be due to the relatively low number of MCTS iterations (1000) used in the experiments because of time limitations.

3. Does a poker bot yield better results if it incorporates the Bayesian opponent model?
   Answer: In our experiments a significant increase in income rate occurred when using the opponent model. So, in comparison with using only MCTS, a poker bot yields better results if it incorporates the Bayesian opponent model as well.

We conducted this research with the following problem statement in mind:

"Will a poker bot incorporating the Bayesian approach to opponent modelling be more effective than a poker bot who does not use this form of opponent modelling?"

We can answer our problem statement as follows:

Answer: By running our experiments we obtained results which indicate that a poker AI incorporating the Bayesian approach to opponent modelling is more effective than a poker bot who does not use this form of opponent modelling. This suggests that the Bayesian approach to opponent modelling proposed by Ponsen et al. [16] can be a valuable contribution to a poker program.

6.2 Limitations to our research

Of course our research has several limitations. We will discuss them here:

- The number of MCTS iterations which we used (1000) in our experiments is not very high. It is possible that by executing 10,000 or even 20,000 iterations, MCTS would come up with a strategy that would produce a higher income rate, even when playing against a more sophisticated opponent (like Poki).

- The MCTS parameters we used in our experiments were not tested thoroughly. It is therefore possible that optimizing these parameters will yield better results in time and/or strategy. There are two parameters. The first parameter is the UCT coefficient C, which determines the balance between exploration and exploitation. The second parameter consists of a switch parameter, which, upon starting the MCTS process of simulation, determines whether players who did not call yet are included or excluded from the simulation.

- As is clear, more experiments should be conducted with MCTS and/or the opponent model. In that case, more evidence can be found that the approach we used in this research can be valuable to include in the design of a competitive poker AI.

6.3 Future research

We have multiple recommendations for future research that will be covered in this section. Our experiments were executed using only one opponent in order to minimize the effort on computer resources, since this proved to be a serious obstacle. Nevertheless, we stated in Chapter 3 that MCTS is able to work with multiple opponents. Therefore, a future research direction could be to implement MCTS in a poker bot and let it play against multiple opponents. Doing this, one could find out whether MCTS is a relatively quick technique to play poker on a moderate level against multiple opponents as well. Furthermore, MCTS could perhaps be improved by increasing the number of iterations and optimizing its parameters, as discussed in Section 6.2. Especially parameter B, the balance parameter can prove valuable in future research. For instance, we could investigate whether a poker program would yield optimal results assuming its opponent(s) play(s) according to a game-theoretic rational strategy or whether results would be optimized with an assumption of imperfect play (which resembles human play).

Another area that future research could focus on is the improvement of the opponent model. In our research, the model was learned beforehand, after which it was implemented in our bot to play a large number of games against the modelled opponent. It likely would be more efficient (and faster) if the model was able to learn on-the-fly, during play. This would perhaps also be a solution for competing with opponents who tend to switch their playing style several times over the course of one poker game. Furthermore, another feature of the opponent model that could be improved is the interface between Prolog (with which the model was implemented) and Java (with which the bot was designed). For this, we used PrologCafe [2]. This interface translates Prolog files into Java files, which seemed to be efficient. However, when using the opponent model in experiment 1, the processing time for 1000 MCTS iterations changed from 5 seconds to 50 seconds (using an average modern pc). One could see the benefits of researching what Prolog-Java interface would be optimal in this case.

This research has contributed to future research leading to the design of a poker program able to compete against and eventually beat a professional poker player. That program could perhaps even implement the techniques of MCTS and/or Bayesian opponent modelling. However, there is still a long way to go.

Bibliography

[1] Andersson, R. (2006). Pseudo-optimal strategies in no-limit poker. M.Sc. thesis, Umea University, Sweden.

[2] Banbara, M. and Tamura, N. (2009). PrologCafe: a Prolog to Java translator system. http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/.

[3] Billings, D., Papp, D., Schaeffer, J., and Szafron, D. (1998a). Opponent modeling in poker. *In Proceedings of the 15th National AAAI Conference (AAAI-98).*

[4] Billings, D., Papp, D., Schaeffer, J., and Szafron, D. (1998b). Poker as a testbed for AI research. *Advances in Artificial Intelligence*, Vol. 1418, pp. 228–238.

[5] Billings, D., Davidson, A., Schaeffer, J., and Szafron, D. (2002). The challenge of poker. *Artificial Intelligence*, Vol. 134, p. 201240.

[6] Billings, D., Burch, N., Davidson, A., Schauenberg, T., Holte, R., Schaeffer, J., and Szafron, D. (2003). Approximating game-theoretic optimal strategies for full-scale poker. *The Proceedings of the Eighteenth International Joint Conference on Artifcial Intelligence*, pp. 661 – 668.

[7] Billings, D., Davidson, A., Schauenberg, T., Burch, N., Bowling, M., Holte, R., Schaeffer, J., and Szafron, D. (2004). Game-tree search with adaptation in stochastic imperfect-information games. *Proceedings of Computers and Games.*

[8] Billings, D. (2006). *Algorithms and assessment in computer poker.* Ph.D. thesis, University of Alberta.

[9] Blockeel, H. and Raedt, L. De (1998). Top-down induction of first-order logical decision trees. *Artificial Intelligence*, Vol. 101(1-2), pp. 285–297.

[10] Chaslot, G., Jong, S. De, Saito, J.-T., and Uiterwijk, J.W.H.M. (2006). Monte-Carlo tree search in production management problems. *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium* (eds. Pierre-Yves Schobbens, Wim Vanhoof, and Gabriel Schwanen), pp. 91–98.

[11] Chaslot, G.M.J-B., Winands, M.H.M., Uiterwijk, J.W.H.M., Herik, H.J. van den, and Bouzy, B. (2008). Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation*, Vol. 4, No. 3.

[12] Cutler, W.H. (1975). An optimal strategy for pot-limit poker. *American Math Monthly*, Vol. 82, pp. 386–376.

[13] Gilpin, A., Sandholm, T., and Srensen, T. B. (2008). A heads-up no-limit Texas Hold'Em poker player: Discretized betting models and automatically generated equilibrium-finding programs. *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal.*

[14] Nash, J.F. and Shapley, I.S. (1950). A simple three-person poker game. *Contributions to the Theory of Games*, Vol. 1, pp. 105–116.

[15] Newman, D.J. (1959). A model for 'real' poker. *Operations Research*, Vol. 7, pp. 557–560.

[16] Ponsen, M., Ramon, J., Croonenborghs, T., Driessens, K., and Tuyls, K. (2008). Bayes-relational learning of opponent models from incomplete information in no-limit poker. *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI-08)*, pp. 1485–1487.

[17] Waugh, K., Schnizlein, D., Bowling, M., and Szafron, D. (2009). Abstraction pathologies in extensive games. *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009).*

Appendix A

Appendix: Output of experiment 1

Here, the actions of our poker program during 10,010 games against ACE1 are shown, for every game phase (preflop, flop, turn and river). Figure A.1 shows the actions of our program without using the opponent model and Figure A.2 shows the actions our program decided upon with using the opponent model. Furthermore, the aggression (as computed through Equation 5.1) is shown, during preflop as well as during postflop.

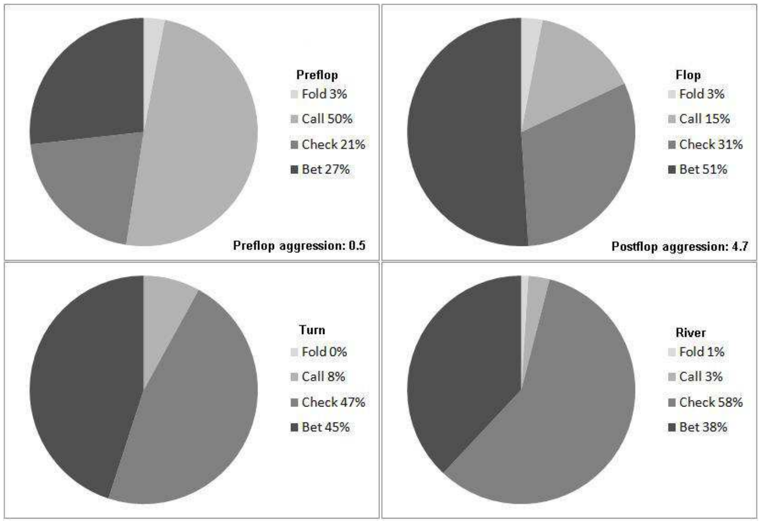


Figure A.1: Observed actions without opponent model.

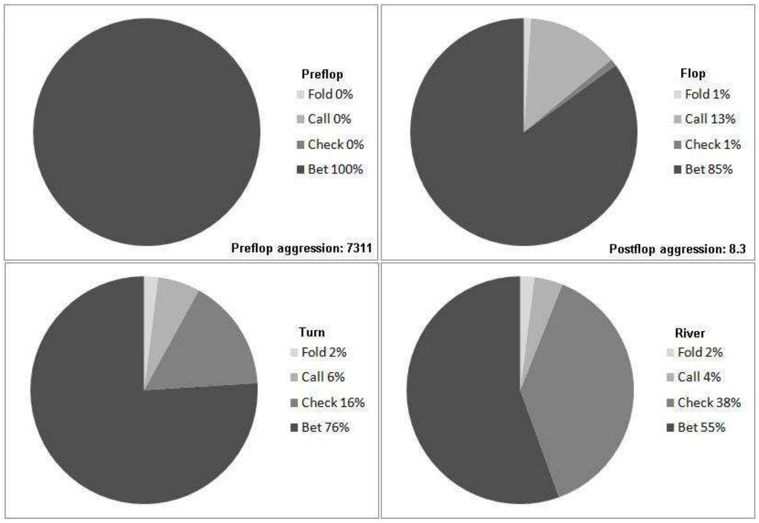
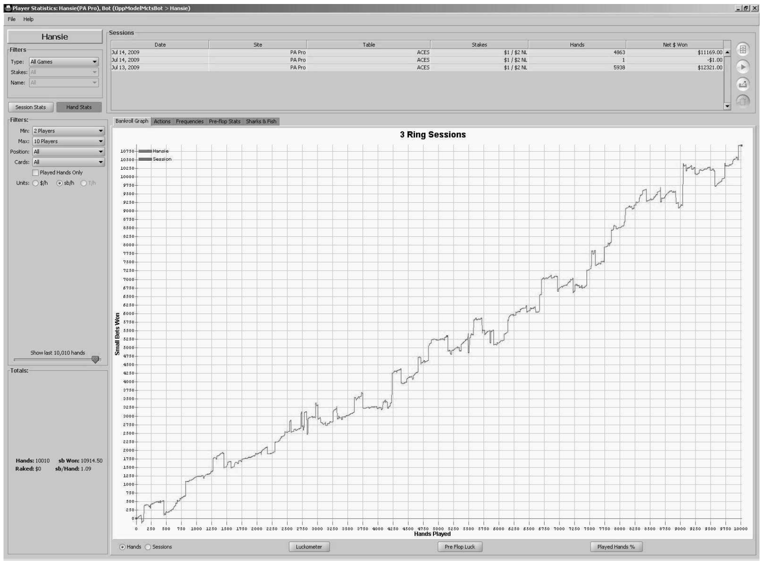


Figure A.2: Observed actions with opponent model.

In Figures A.3 and A.4, the amount of money (bankroll) of our poker program is shown over the course of 10,010 games. The x-axis represents the number of hands played and the y-axis shows the number of small bets won (the income rate). In Figure A.3, only MCTS is used in the program, while in Figure A.4, the opponent model is incorporated as well. With the numbers shown in the graph, the average income can be computed. In Figure A.3, the total amount of small bets won is 10,914.50, so the average income is 1.09 bb./hand. And likewise, in Figure A.4, the total amount of small bets won is 16,724.50, which relates to an average income of 1.67 bb./hand.



Player Statistics: Hansie(PA Pro), Bot (OppModelMctsBot > Hansie)

File Help

Hansie

Filters

Type: All Games

Stakes: All

Name: All

Session Stats | Hand Stats

Sessions

Date	Site	Table	Stakes	Hands	Net $ Won
Jul 14, 2009	PA Pro	ACES	$1 / $2 NL	4863	$11169.00
Jul 14, 2009	PA Pro	ACES	$1 / $2 NL	1	-$1.00
Jul 13, 2009	PA Pro	ACES	$1 / $2 NL	5938	$12321.00

Filters:

Min: 2 Players

Max: 10 Players

Position: All

Cards: All

□ Played Hands Only

Units: ○ $/h ● sb/h ○ T/h

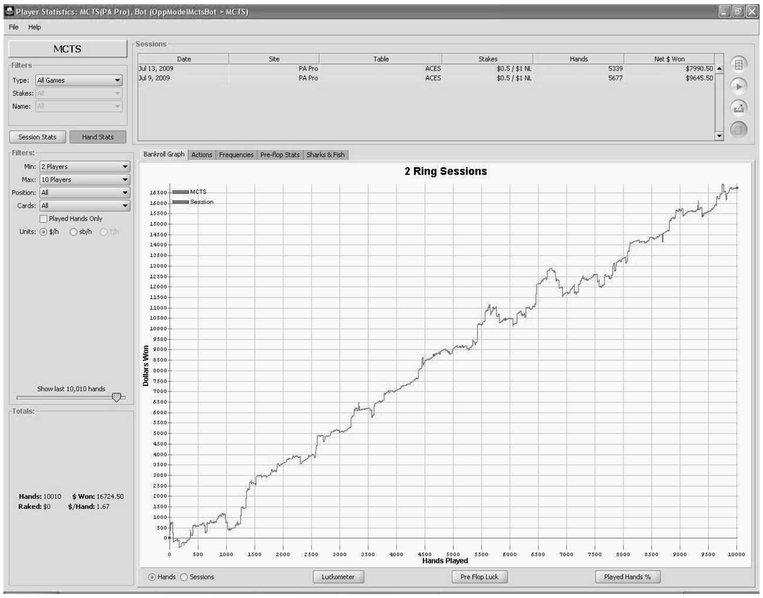
Show last 10,010 hands

Totals:

Hands: 10010 **sb Won:** 10914.50

Raked: $0 **sb/Hand:** 1.09

Figure A.3: Bankroll without opponent model.



Player Statistics: MCTS(PA Pro), Bot (OppModelMctsBot > MCTS)

File Help

MCTS

Filters

Type: All Games

Stakes: All

Name: All

Session Stats | Hand Stats

Sessions

Date	Site	Table	Stakes	Hands	Net $ Won
Jul 13, 2009	PA Pro	ACES	$0.5 / $1 NL	5339	$7990.50
Jul 9, 2009	PA Pro	ACES	$0.5 / $1 NL	5677	$9645.50

Filters:

Min: 2 Players

Max: 10 Players

Position: All

Cards: All

□ Played Hands Only

Units: ● $/h ○ sb/h ○ T/h

Show last 10,010 hands

Totals:

Hands: 10010 **$ Won:** 16724.50

Raked: $0 **$/Hand:** 1.67

Figure A.4: Bankroll with opponent model.

Appendix B

Appendix: Output of experiment 2

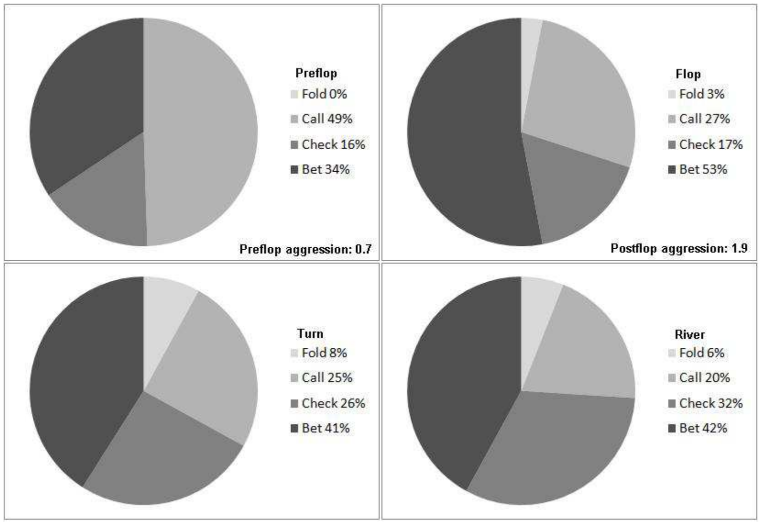


Figure B.1: Observed actions without opponent model.

Here, the actions of our poker program during 10,304 (Figure B.1) and 10,310 (Figure B.2) games against Poki (Anders) are shown, for every game phase (preflop, flop, turn and river). Figure B.1 shows the actions of our program without using the opponent model and Figure B.2 shows the actions of the program with the opponent model. Furthermore, the aggression (as computed through Equation 5.1) is shown, during preflop as well as during postflop.

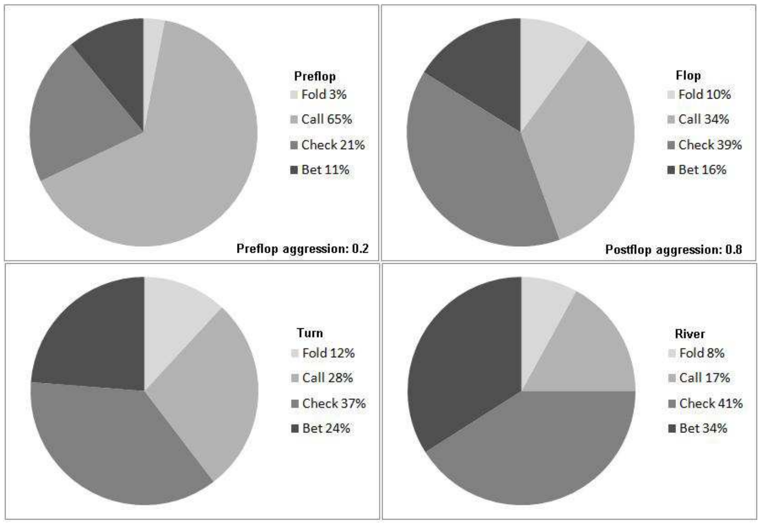
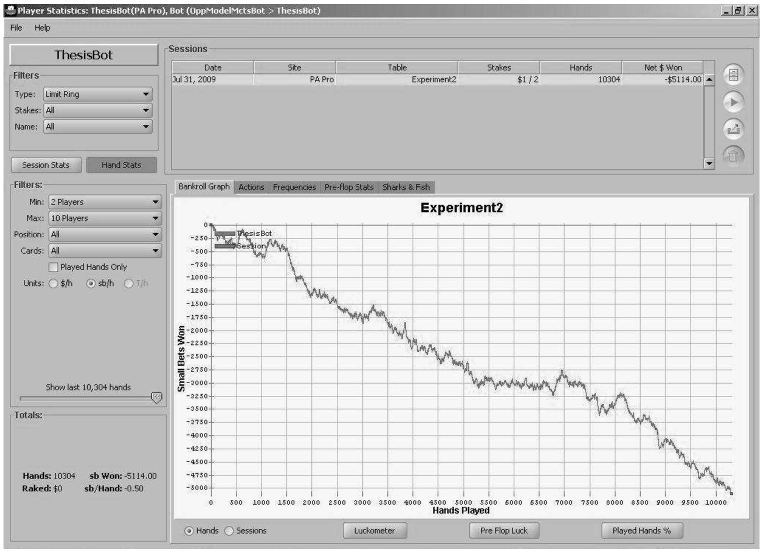


Figure B.2: Observed actions with opponent model.

In Figures B.3 and B.4, the amount of money (bankroll) of our poker program is shown over the course of 10,304 (Figure B.3) and 10,310 (Figure B.4) games. The x-axis represents the number of hands played and the y-axis shows the number of small bets won (the income rate). In Figure B.3, only MCTS is used in the program, while in Figure B.4, the opponent model is incorporated as well. With the numbers shown in the graph, the average income can be computed. In Figure B.3, the total amount of small bets won is -5,114 (a loss), so the average income is -0.50 bb./hand. And likewise, in Figure B.4, the total amount of small bets won is 612.50, which relates to an average income of 0.06 bb./hand.


Player Statistics: ThesisBot(PA Pro), Bot (OppModelMctsBot > ThesisBot)

ThesisBot

Filters

Type: Limit Ring

Stakes: All

Name: All

Sessions

Date	Site	Table	Stakes	Hands	Net $ Won
Jul 31, 2009	PA Pro	Experiment2	$1 / 2	10304	-$5114.00

Filters:

Min: 2 Players

Max: 10 Players

Position: All

Cards: All

□ Played Hands Only

Units: ○ $/h ● sb/h ○ T/h

Show last 10,304 hands

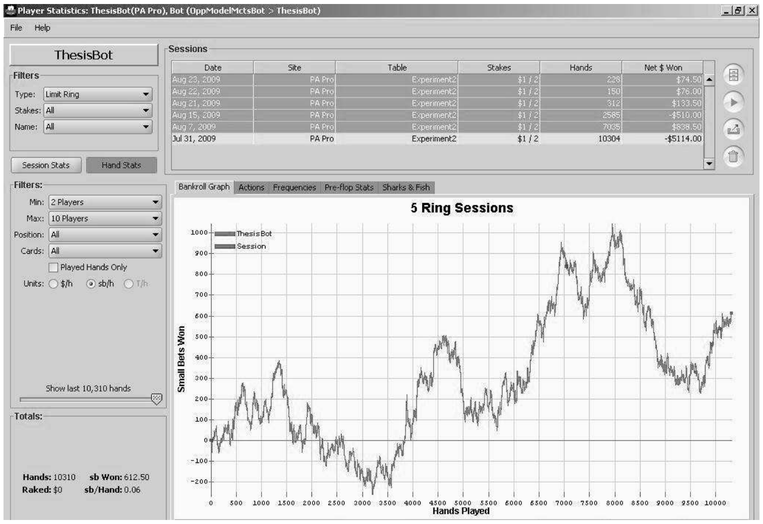
Totals:

Hands: 10304 **sb Won:** -5114.00

Raked: $0 **sb/Hand:** -0.50

● Hands ○ Sessions

Figure B.3: Bankroll without opponent model.


Player Statistics: ThesisBot(PA Pro), Bot (OppModelMctsBot > ThesisBot)

ThesisBot

Filters

Type: Limit Ring

Stakes: All

Name: All

Sessions

Date	Site	Table	Stakes	Hands	Net $ Won
Aug 23, 2009	PA Pro	Experiment2	$1 / 2	228	$74.50
Aug 22, 2009	PA Pro	Experiment2	$1 / 2	150	$76.00
Aug 21, 2009	PA Pro	Experiment2	$1 / 2	312	$133.50
Aug 15, 2009	PA Pro	Experiment2	$1 / 2	2585	-$510.00
Aug 7, 2009	PA Pro	Experiment2	$1 / 2	7035	$838.50
Jul 31, 2009	PA Pro	Experiment2	$1 / 2	10304	-$5114.00

Filters:

Min: 2 Players

Max: 10 Players

Position: All

Cards: All

□ Played Hands Only

Units: ○ $/h ● sb/h ○ T/h

Show last 10,310 hands

Totals:

Hands: 10310 **sb Won:** 612.50

Raked: $0 **sb/Hand:** 0.06

Figure B.4: Bankroll with opponent model.