Integrating Opponent Models with Monte-Carlo Tree Search in Poker

Marc Ponsen[1] and **Geert Gerritsen**[1] and **Guillaume Chaslot**[1]
1: Department of Knowledge Engineering, Maastricht University, Netherlands

Abstract

In this paper we apply a Monte-Carlo Tree Search implementation that is boosted with domain knowledge to the game of poker. More specifically, we integrate an opponent model in the Monte-Carlo Tree Search algorithm to produce a strong poker playing program. Opponent models allow the search algorithm to focus on relevant parts of the game-tree. We use an opponent modelling approach that starts from a (learned) prior, i.e., general expectations about opponent behavior, and then learns a relational regression tree-function that adapts these priors to specific opponents. Our modelling approach can generate detailed game features or relations on-the-fly. Additionally, using a prior we can already make reasonable predictions even when limited experience is available for a particular player. We show that Monte-Carlo Tree Search with integrated opponent models performs well against state-of- the-art poker programs.

Introduction

Board and card games are interesting research domains for Artificial Intelligence (AI) researchers. Strategic decision-making of computer players can easily and rapidly be evaluated in such constrained domains. For many games research has lead to expert computer players that at least match humans in playing skill. An exception is still the game of poker, offering new research challenges. The complexity of the game is threefold, namely poker is (1) an imperfect information game, with (2) stochastic outcomes in (3) an adversarial multi-agent environment.

Recently, for the smallest of poker variants, namely a two-player game with fixed betting amounts, strong computer players have been implemented that are competitive with human experts. Current state-of-the-art poker algortihms use a game-tree search to decide which actions to take (e.g., (Billings et al. 2003), (Billings et al. 2006) and (Zinkevich et al. 2008)). Various game-tree search algorithms are shown to compute or approximate Nash-equilibria, and as such produce *rational* players. Such a player would play perfectly logical and would be oblivious to opponent mistakes.

In this paper we will design and evaluate a poker playing program (i.e., bot) that can both play an approximated rational strategy and a best-reply strategy given an estimation of opponent mistakes. We will empirically evaluate both approaches against two different opponent bots. The contributions of this paper are twofold: first, we employ the Monte-Carlo tree search (MCTS) algorithm in poker. MCTS uses a different selection and update procedure as compared to current state-of-the-art game-tree search algorihms used in poker. MCTS has shown to converge to a Nash equilibrium and has achieved succes in the perfect-information game of Go (Sturtevant 2008) (Scoulom 2006). We will evaluate its effectiveness in the game of poker against challenging opposition.

Our second and main contribution is integrating opponent models with the MCTS algorithm. Given the enormous complexity of poker we believe that players, and in particular human players, are uncapable of playing a perfect rational strategy. This then violates the assumption made by rational computer players. Playing a best-reply may yield more profit. We need an opponent model to play a best-reply, which in poker requires two estimations of opponent information, namely a prediction of the opponent *cards* and a prediction of opponent *actions*. In this paper we consider a relational Bayesian modelling approach that uses a general prior (for predicting cards and actions) and learns a relational regression tree to adapt that prior to individual players. This approach was first introduced in (Ponsen et al. 2008). We evaluate the impact of the learned models by running experiments with MCTS boosted with domain knowledge of the opponent.

We will first explain the rules of poker and discuss related work. Then we will describe the MCTS algorithm and how we integrate opponent models in it. After having explained how the opponent models are learned, we finish with experiments and a conclusion.

Poker

Poker is a card game played between at least two players. In a nutshell, the objective in poker is to win games (and consequently win money) by either having the best card combination at the end of the game, or by being the only active player. The game includes several betting rounds wherein players are allowed to invest money. Players can remain active by at least matching the largest investment made by any of the players. This is known as *calling* or *checking*. Players may also decide to *bet* or *raise* a bet, which increases the stakes. Finally, they can choose to fold (i.e., stop investing money and forfeit the game). In this paper we focus on the most popular poker variant, namely Texas Hold'em, and more specifically on the limit variant (i.e., a fixed amount for betting). This game includes 4 betting rounds (or phases), respectively called the pre-flop, flop, turn and river phase.

Related Work

MCTS led to excellent results in creating computer game-playing programs, for instance in the games of Go (Coulom 2006) and General Game Playing (Finnsson and Björnsson 2008). However, it has been shown that MCTS can be enhanced considerably by using domain-dependent knowledge learned by extraneous algorithms. This knowledge can be learned by using various methods,

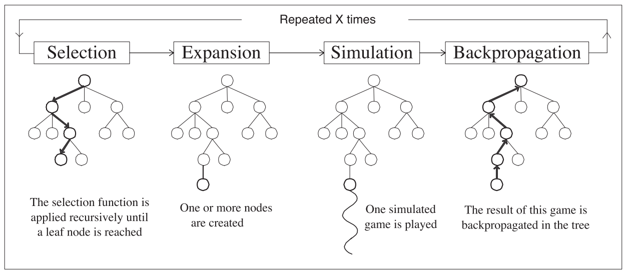
Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 1: Outline of Monte-Carlo Tree Search.

and integrated in different parts of MCTS. For instance, in (Bouzy and Chaslot 2006) it was proposed to learn a simulation strategy by reinforcement, which increased significantly the level of play of the program. Later, (Coulom 2007) proposed to learn Go knowledge by using algorithms developed for calculating ELO ratings. The knowledge was used both in the simulation, and in the selection parts of MCTS. Other improvements resulting from the use of extraneous knowledge in the selection were achieved in (Gelly and Silver 2007) (Chaslot et al. 2008). We will also boost MCTS with knowledge in the form of opponent models in the game of poker.

Poker is a perfect domain for opponent modeling, since the ability to anticipate an opponent's move highly influences the outcome of the game. The Adaptive Imperfect Information game-tree search algorithm (Billings et al. 2006) has an opponent model integrated with it. They keep track of statistics for both the outcome of the game and actions at opponent decision nodes for every possible betting sequence. The problem with this approach is that it uses little generalization and hence the frequency counts are limited to a small number of situations. Another drawback is that it assumes that both cards and strategies of the opponents are independent.

A more general system for opponent modelling was obtained by training a neural network. (Davidson et al. 2000) use nineteen different parameters as input nodes and three output nodes representing the possible actions. The input parameters include information about the players, information about the betting history and information on the community cards. Using a relational approach allows us to use the same features while easily incorporating relational properties of the game.

(Southey et al. 2005) uses prior distributions over the opponent's strategy space and computes a posterior using Bayes' rule and observations of the opponent's decisions. It also investigates several ways to play an appropriate reply to that distribution.

Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a search algorithm based on Monte-Carlo simulations that was developed in 2006 by (Chaslot et al. 2006) (Coulom 2006) (Kocsis and Szepesvári 2006). MCTS was successfully applied in several games, and it is now recognized as the current paradigm for computer Go (Gelly et al. 2006) and General Game Playing (Finnsson and Björnsson 2008).

MCTS performs a large number of simulated games in self play, from the initial position to the end of the game. The underlying idea of MCTS is to improve progressively the quality of simulations by taking into account the games previously played. More specifically, MCTS directs its simulations to that part of the game-tree that is most relevant assuming players take rational decisions. When few simulated games have been played, then moves are chosen randomly. The result of each game is back-propagated on the visited path. Progressively, the program concentrates its search on the best moves, leading to a deeper look-ahead ability. We will now present this mechanism in more detail.

In MCTS, every node in the game-tree represents a state of the game. Statistics of nodes which are important for MCTS are:

- the *value* of the game state. This is the average of the reward of all simulated games that visited this node.
- the *visit count* of the game state, i.e., the number of simulations in which this state was reached.

The starting state is represented by the root node, which is initially the only node in the tree. MCTS consists of repeating the following four steps (illustrated in Figure 1), as long as there is time left. We will now detail these steps and also discuss some implementation details in poker:

1. *Selection*: Actions[1] encoded as nodes in the tree are chosen according to the statistics stored in a way that balances between exploitation and exploration. On the one hand, the task is to select the game action that leads to the highest expected value (exploitation). On the other hand, less promising actions still have to be explored due to the uncertainty of the evaluation (exploration). We use Upper Confidence Bound applied to Trees (UCT). In UCT, $i \in I$ is the set of nodes (possible actions) reachable from the current node, p. Using the next equation, UCT selects the child node k of parent node p which has the highest expected value. The equation for UCT is:

$$k \in argmax_{i \in I}(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}}) \tag{1}$$

   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 coefficient that balances exploration and exploitation. A higher value encourages longer exploration.

2. *Expansion*: when a leaf node is selected, one or several nodes are added to the tree. The tree is then expanded by one node for each simulated game.

3. *Simulation*: from the newly expanded node, nodes are selected according to some simulation policy until the end of the game. More realistic simulations will have a significant effect on the computed expected values. Heuristic knowledge can be used to give larger weights to nodes that look more promising. We employ roll-out simulations starting from the expanded leaf node. More specifically, all remaining active players call or check until the end of the game. Then community and private cards are dealt to determine the winner(s). The rewards that are the result of the simulation consist of the amount of money a player loses or wins. The process of simulation is executed twice every MCTS iteration: once from the perspective of the MCTS player itself, and once for its opponents. For respectively the first simulation, we use known cards of the MCTS player, whereas for the second simulation we sample them (i.e., we can't assume that opponents know the cards of the MCTS player).

4. *Backpropagation*: after reaching the end of the simulated game, we update each tree node that was traversed during that game. The visit counts are increased and the values are modified according to the outcome.

[1]We define three types of actions in our game-tree for poker, namely 'fold', 'call' or 'bet'. One can imagine that in a no-limit variant of the game (i.e., variable betting amounts) this would involve a various number of raise amounts to be implemented. However, since we focus on the limit-variant this is no issue.

When available computational time is over, the 'best' move is selected. There are multiple selection strategies to do this, including a node having the highest expected value or the highest visit count. We respectively choose the first strategy.

Learning an Opponent Model

MCTS as discussed in the previous Section produces an approximated Nash equilibrium strategy, i.e., the best possible strategy against itself. It treats opponents as rational players (in the game-theoretic sense) and is oblivious to opponent mistakes. Given the immense complexity of Texas Hold'em poker, it may be assumed that players, and in particular human players, do not play a perfect rational strategy. When facing a predictable and inferior opponent, a tailored counter-strategy (best-reply) will earn more profit than a rational strategy. In order to play a best-reply, we need an accurate estimation or model of the opponent strategy.

Learning an opponent model can be approached as a pattern recognition task (Bishop 2007), wherein a model is learned based on experience (in this case, previous poker games of players). The model is then used to estimate the behavior of opponents in unseen situations. In the game of poker this implies having an estimation of opponent cards and actions. These two sources of information can easily be used in a game-tree search, as we will explain in the case of MCTS. An established assumption in the field of pattern recognition is that more training data leads to more accurate models. However, for an opponent model in the game of poker to be practical, we must be able to learn from limited amount of training data. Typically, in large poker tournaments or online poker games one faces a wide range of opponents, and it may not be assumed that many training games have been collected for each individual player. To overcome this difficulty one could learn opponent models for player types. Poker-experts often categorize players based on statistical features (Ponsen et al. 2009). However, in this paper we focus on learning models for specific players. We choose to start from a prior opponent model, so we can already make reasonable predictions even when little data is available.

Before we explain our approach in more detail, we describe our notations first: consider a player p performing the i-th action a_i in a game. The player can *fold*, *call* or *bet*. For simplicity, we consider *check* and *call* to be in the same class, as well as *bet* and *raise* [2]. When deciding on an action, the player will take into account his private cards c_p, and the game state S_{i-1} until time point i. S_{i-1} is the betting history of all players in the game and each card on the table (i.e., the *community cards* that apply to all players).

We will use the term *example* to describe the training data that can be used to learn a model. An example references a tuple $(i, p, a_i, c_p, S_{i-1})$ of the action a_i performed at step i by a player p, together with the private cards c_p of player p in the game, and the game state information S_{i-1} up until time point i.

We can collect training examples and then train a classifier that uses as labels the observed private cards c_p or the action a_i, and then search for patterns in relation with the game state S_{i-1}. More formally, the learning task now is to predict the cards for player p at time i given the observed state information S_{i-1}:

$$P(c_p|S_{i-1}) \tag{2}$$

and player p's action (given a guess about the private cards):

$$P(a_i|S_{i-1}, c_p) \tag{3}$$

[2]Note that this is an identical action representation as used in the game-tree for MCTS

Approach

We propose to start the opponent model with a prior distribution over possible action choices and cards. When no training examples are available to us, we assume that the opponent plays according to some prior distribution. Using a prior will allow us to make accurate predictions from the start as long as the choice of prior is reasonable. We then adapt that prior to the specific opponent using a so-called *differentiating function* as soon as more training data becomes available. The current approach was first proposed in (Ponsen et al. 2008) and we will explain it here further.

We propose a two-step learning approach. First, we must settle on a prior distribution. This can be achieved in many ways. We can learn functions that predict cards and actions for poker players in general, for certain player types or for rational players. Second, we learn a differentiating function that learns to correct the prior according to the actual observed behavior of a particular player.

Consider the mixture D_{p+*} of two distributions: the distribution D_* of arbitrarily drawn examples from our prior distribution and the distribution D_p of arbitrarily drawn examples from a particular player p. Then, consider the learning problem of, given a randomly drawn example x from D_{p+*}, predicting whether x originated from D_* or from D_p. For a given learning setting, namely either predicting cards or actions, it is easy to generate examples from D_* and D_p, labeling them with $*$ or p, and learning the function $P(D_p|x)$, giving for each example x the probability the example is labeled with p. From this learned 'differentiating' model, we can compute the probability $P(x|D_p)$, for every example x by using Bayes' rule:

$$P(x|D_p) = P(D_p|x) \cdot P(x)/P(D_p) \tag{4}$$

Since we have chosen to generate as many examples for D_* as for D_p in the mixture,

$$P(D_p) = P(D_*) = 1/2 \tag{5}$$
$$P(x) = P(D_*)P(x|D_*) + P(D_p)P(x|D_p) \tag{6}$$

and substituting (5) and (6) into (4) gives:

$$\begin{aligned} P(x|D_p) &= \left(P(D_p|x) \cdot \left(\frac{1}{2}P(x|D_p) + \frac{1}{2}P(x|D_*)\right)\right)/\frac{1}{2} \\ &= P(D_p|x)P(x|D_p) + P(D_p|x)P(x|D_*). \end{aligned}$$

From this, we get:

$$P(x|D_p) = \frac{P(x|D_*) \cdot P(D_p|x)}{1 - P(D_p|x)} \tag{7}$$

Here $P(x|D_*)$ is the prior probability of x and $P(D_p|x)$ is the learned differentiating function. $P(x|D_p)$ is the posterior probability that describes the probability that example x belongs to player p, and can be used to query action and card probabilities. Suppose we are interested in the probability distribution over actions for our opponent given a certain game state. We have a number of examples equal to the number of actions, for which only a_i differs. We plug the examples in our learned differentiating function and known prior to respectively compute $P(D_p|x)$ and $P(x|D_*)$. We can then compute the posterior probability, e.g., the probability that an opponent performs an action given the current game state.

The key motivations for such an approach are that learning the difference between two distributions is an elegant way to learn a multi-class classifier (e.g. predicting distributions over (52*51/2) possible card combinations) 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 (assuming the prior is reasonable).

Relational Decision Trees

A lot of study has already been done to derive important features for the game of poker, mostly by domain experts (e.g.,(Slansky 1987) (Harrington 2004)). Some examples of important features are a numerical ranking of a player's private cards, a player's position on the table or some simple to compute features about the game such as pot odds. In existing opponent modelling techniques, as for example in (Davidson et al. 2000), a manual selection of such *propositional* features are used for learning the models. We instead use a relational decision tree learner that can automatically discover important relations in a game, and as such can generate relational features on-the-fly. More specifically, we will employ the relational probability tree algorithm TILDE (Blockeel and De Raedt 1998) to learn the differentiating function $P(D_p|x)$.

Given a set of examples, the algorithm incrementally builds up a tree (using top down induction). The set of examples contains all observed examples for player p, labeled with $player$. This effictively reflects the player distribution D_p. We then draw an equal amount of examples from the prior distribution D_*, and label them as $prior$. For example, when learning an opponent model that predicts actions, we duplicate all examples labeled with $player$. We then replace the current action a_i entry in the example with one that we randomly draw from the prior distribution.

TILDE then starts with an empty tree with one leaf, which contains all stored examples. Then, when the algorithm reaches a leaf node, candidate tests (or a conjunction of several) defined in the so-called language bias are evaluated. In poker, each relational test describes a small part of the game state, e.g., $game\_number\_bets(Example, Bets)$. We can also use relational tests to generalize over private cards, e.g., $sum\_hole\_cards(Example, Sum)$. The tree may be expanded by candidate tests that reduce variance among the two distributions sufficiently. The best among these candidate tests is then selected to expand the tree, and the examples are then sorted out among all terminal nodes. Internal nodes contain a test which is a conjunction of first order literals. These tests should be read as the existentially quantified conjunction of all literals in the path from the root of the tree to that node. In the left subtree of a node, the test of the node is added to the conjunction, for the right subtree, the negation of the test should be added. Such a relational decision tree can be easily translated into a Prolog decision list. Figure 2 gives an example of tree learned by TILDE. Note that this tree represents the differentiating function $P(D_p|x)$.

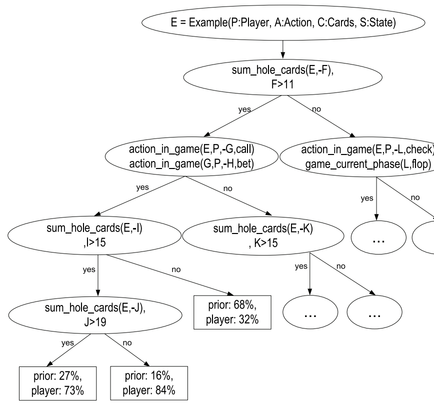
E = Example(P:Player, A:Action, C:Cards, S:State)
sum_hole_cards(E,-F), F>11
yes
no
action_in_game(E,P,-G,call) action_in_game(G,P,-H,bet)
action_in_game(E,P,-L,check) game_current_phase(L,flop)
sum_hole_cards(E,-I) ,I>15
sum_hole_cards(E,-K) , K>15
...
sum_hole_cards(E,-J), J>19
prior: 68%, player: 32%
prior: 27%, player: 73%
prior: 16%, player: 84%

Figure 2: An example of a relational decision tree encoding the differentiating function for predicting cards. Internal nodes include relational tests that partition the state space. Terminal nodes contain zero or more examples of the two distributions, effectively denoting a probability distribution over both distributions.

Integrating an Opponent Model with Monte-Carlo Tree Search

In this Section, we will discuss the way in which MCTS is influenced by the opponent model. As mentioned in the previous Section, the opponent model predicts two sources of information: opponent's private cards and future actions. The prediction of cards for players is used to determine the reward for each player at the end of each MCTS iteration. The prediction of actions is used for the selection process during the iterations.

First, we look at how card predictions are integrated with MCTS. 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, the opponent model may state that the likelihood of higher ranked cards is much greater after having observed a bet for this opponent. Attaching a higher probability to these cards leads to more MCTS iterations executed with them. This improves the expected value as computed by MCTS.

At the start of MCTS in some game state wherein the MCTS player must act, we first compute a probability distribution of the opponent's private cards given that game state (according to Equation 2). This is done only once for a set of MCTS iterations starting in a specific game state. Then for each single MCTS iteration we draw the private cards for the opponent according to the computed distribution. The sampled opponent hand will be used throughout the current MCTS iteration. In a next iteration, new opponent cards will be sampled.

Second, action probabilities are queried (according to Equation 3) from the opponent model whenever the opponent has to take an action in the game-tree. These probabilities are used for selecting actions in opponent nodes. For non-opponent nodes (i.e., nodes belonging to the MCTS player itself), standard UCT selection is done (see Equation 1). Note that action probabilities are dependent on the game state as well as the sampled cards in the current iteration. There are no changes in expansion, simulation nor backpropagation. The pseudo-code of MCTS with integrated opponent model is given in Algorithm 1.

Experiments and Results

We compare the results for our bot using standard MCTS and MCTS integrated with opponent models in a one-on-one game against two different bots. Our MCTS bot is implemented in Java and evaluated in the Poker Academy Pro (PAP) software tool. The first opponent bot is a simple rule-based poker bot (ACE1). ACE1 uses a very straightforward policy, wherein actions are based on either a numerical ranking of the cards (preflop), or the the sum of the two private cards (all other remaining phases). During later phases the sum must be higher in order to execute aggressive actions (i.e., bet or raise). For example, during the flop ACE1 bets when the sum of the two private cards is higher than 15, whereas for the river the sum must be higher than 21 to bet.

Our second opponent is Poki, or more concretely the bot named 'Anders' (an instance of Poki) which is provided with PAP. The results from experiments conducted with Poki showed that it is a fairly tough opponent: it yielded a result between $+0.10sb/h$ and $+0.20sb/h$ against novice human poker players. The term sb/h is the income rate, and describes the small bets won per hand which is used (also in our experiments) to reflect the program's playing skill. In games with stronger human opposition Poki's play re-

```
Data: game_state $S_{i-1}$
Data: root_node
Data: card_probabilities $P(c_p|S_{i-1})\forall c_p$
Result: best_move
while (has_time) do
    $current\_node \leftarrow root\_node$
    $Sample\ c_p\ from\ card\_probabilities$
    while ($current\_node \in \mathcal{T}$) do
        $last\_node \leftarrow current\_node$
        if $current\_node \neq opponent\_node$ then
            $P \leftarrow UCT\_Prob\,\forall a_i$
        else
            $P \leftarrow P(a_i|S_{i-1}, c_p)\,\forall a_i$
        $current\_node \leftarrow Select(current\_node, P)$
    end
    $last\_node = Expand(last\_node)$
    $R \leftarrow Play\_simulated\_game(last\_node)$
    while ($current\_node \in \mathcal{T}$) do
        $current\_node \leftarrow current\_node.parent$
        $Backpropagation(current\_node, R)$
    end
end
return $best\_move = argmax_{N \in \mathcal{N}_c(root\_node)}$
```

Algorithm 1: Monte-Carlo Tree Search overview with integrated Opponent Model

sulted between a $+0.07sb/h$ and $+0.10sb/h$. For a more detailed explanation of Poki and its experiments, see (Billings 2006). We will first describe the process of learning the opponent models for the opponent bots, and then discuss the parameters used for MCTS. Then we will present our results, and finish with a discussion.

Learning the Opponent Model

We collected 5000 games for each opponent bot that were used as training data for learning the models. We learned one decision tree (i.e., opponent model) per phase. We chose a uniform distribution as prior. Clearly, this is not an accurate model to start with, but we ensured that we already have a large amount of games to adapt the prior. The language bias used by TILDE (i.e., all possible tests for learning the decision tree) includes tests to describe the game state S_{i-1}. More specifically, we have tests that check for the phase, for the number of bets in a game or phase, for previously executed actions by players, pot odds (ratio between amount for calling and pot size) and pot ratio (ratio between amount invested and pot size). We also have tests that generalize over cards, namely checking for the numerical ranking of a hand, the sum of the rank of the two private cards, and checking whether the cards are of the same suit or form a pair. Please note that this is only a small subset of all possible tests. More elaborate tests can easily be added.

A small part of the model learned for the river phase and bot ACE1 is illustrated in Figure 2. Let us examine this tree. Given an example E that includes instantiated values for the modelled player, its private cards and the game state, the most-left terminal node satifies the following Prolog decision list:

```
SUM_HOLE_CARDS(E,F),F>11,
ACTION_IN_GAME(E,P,G,CALL),
ACTION_IN_GAME(G,P,H,BET),
SUM_HOLE_CARDS(E,I),I>15,
SUM_HOLE_CARDS(E,J),J>19, !.
```

This now states the case that the player holds cards with a sum higher than 19 and at some point in the game called (at time denoted with the variable G that references an earlier game state), and prior to that betted (at a game state H that occurred before game state G). If this is the case, it is more likely that we are dealing with the modelled player ACE1 (73%), than with (in this case) a uniform prior (27%). This nicely confirms the static policy used by ACE1, namely after having observed a bet it is more likely that this player is holding high ranked cards. As we can see, complicated and detailed relational tests are learned on-the-fly using only the few simple tests stored in the language bias.

Experimental Setup

In our experiments, a number of parameters are used for MCTS. First, the number of **iterations**. MCTS is a technique of which the accuracy improves with more iterations. Unfortunately, this parameter is the most limiting factor in terms of time. In particular querying the opponent model (which is written in Prolog) is very time demanding because it demands a Java-Prolog interface. Therefore we ran MCTS for a max of 1000 iterations at each decision. This is relatively low compared to the research with MCTS applied in Go (Chaslot et al. 2008) where 20000 MCTS iterations were allowed per move.

Second, the **UCT coefficient *C*** which influences the balance between exploration and exploitation during the process of selection. We found in preliminary experiments that a value of 2 for C results in a good amount of exploration before exploitation takes over.

Results

Results obtained for MCTS with and without integrated model are displayed in Table 1. Before we discuss the results, we will first explain the entries of our result table. The column SF stands for 'seen flops', this shows the percentage of games a bot participated in a game (i.e., did not fold preflop). The columns P-AGR and AGR respectively represent the aggression preflop and postflop (all phases after the preflop). Aggression is computed as $(\%bets/\%calls)$, and tells us something on how defensive or offensive someone is playing. As mentioned before, the sb/h entry is the income rate which we use to measure the playing skill of the bot.

	sf	p-agr	agr	sb/h	games
ACE1	0.64	0.5	4.7	1.09	10010
ACE1-M	0.47	7311	8.3	1.67	10010
POKI	0.945	0.7	1.9	-0.50	10304
POKI-M	0.916	0.2	0.8	0.06	10310

Table 1: Results of experiments against two bots using MCTS and MCTS integrated with an opponent model (rows with added '-M')

We can see that both runs against ACE1 are won with large amounts, which comes as no surprise since this bot is very weak. MCTS with opponent model wins on average $0.58sb/h$ more than without model. Striking is the much higher aggression using an opponent model compared to standard MCTS. Since ACE1 is likely to fold at some point in the game (namely, it will only remain active until the end of the game when the sum of the private cards is larger than 19, which only occurs very rarely), MCTS with integrated model decides to bet constantly, in particular in early phases.

The results against the more challenging opponent, Poki, indicate that standard MCTS loses quite severely, namely $-0.5sb/h$. This result suggests that the chosen parameter values for MCTS (i.e., number of iterations and UCT-constant) are not optimal, and thus the resulting strategy is clearly inferior compared to strong poker-playing programs. An almost identical performance improvement is observed as before when using models against POKI, an improvement of $0.56sb/h$. Despite the low number of iterations per decision, MCTS with model actually wins by a small margin.

Conclusions

We presented the Monte-Carlo tree search (MCTS) algorithm in poker. We integrated opponent models with the algorithm. Our Bayes-relational opponent modelling system predicts both actions and cards in Texas Hold'em poker. Both these sources of opponent information are crucial for simulation and game-tree search algorithms, such as MCTS. The Bayes-relational opponent modeling approach starts from prior expectations about opponent behavior and learns a relational regression tree-function that adapts these priors to specific opponents. Experiments show that our opponent modeling system can learn detailed, relational features on-the-fly. Furthermore, by using a prior we can make reasonable predictions when only few training data are available.

We ran experiments with MCTS, both with and without opponent models against two opponent poker bots. Among them is the challenging limit player, Poki (Billings 2006). We 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. However, against Poki standard MCTS does not perform well. The reasons for this can be the low number of iterations, or wrong choice of parameters. Secondly, we conclude that the Bayesian opponent model proposed by (Ponsen et al. 2008) improves the playing style which results from MCTS by adapting to a specific opponent. Although the number of games are too few to make any definite statements on the relative skill of the MCTS player compared to the opposition, we do see a large improvement when using an opponent model. Using the opponent model, MCTS actually wins against Poki by a small margin, which is remarkable given the low number of MCTS iterations at each decision. The learned opponent models effectively guide MCTS to relevant parts of the game-tree, and as such lowers the computational demand of the technique. Increasing the number of iterations, and providing more elaborate tests for learning the opponent model will almost certainly improve performance further. Also computing confidence intervals or calibration corrections on the estimated probabilities is interesting future work.

Acknowledgements

We thank Karl Tuyls, Jan Ramon, Tom Croonenborghs and Kurt Driessens for collaborating with us on the topic of learning the opponent model.

References

Billings, D.; Burch, N.; Davidson, A.; Holte, R. C.; Schaeffer, J.; Schauenberg, T.; and Szafron, D. 2003. Approximating game-theoretic optimal strategies for full-scale poker. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, 661–668. Morgan Kaufmann.

Billings, D.; Davidson, A.; Schauenberg, T.; Burch, N.; Bowling, M.; Holte, R. C.; Schaeffer, J.; and Szafron, D. 2006. Game-tree search with adaptation in stochastic imperfect-information games. In *The 4th Computers and Games International Conference*. Ramat-Gan, Israel: Springer.

Billings, D. 2006. *Algorithms and Assessment in Computer Poker*. Ph.D. dissertation. University of Alberta.

Bishop, C. M. 2007. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition.

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

Bouzy, B., and Chaslot, G. 2006. Monte-Carlo Go Reinforcement Learning Experiments. In *IEEE 2006 Symposium on Computational Intelligence in Games, Reno, USA*, 187–194.

Chaslot, G. M. J.-B.; Saito, J.-T.; Bouzy, B.; Uiterwijk, J.; and van den Herik, H. 2006. Monte-Carlo Strategies for Computer Go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*.

Chaslot, G.-B.; Winands, M.; Uiterwijk, J.; van den Herik, H.; and Bouzy, B. 2008. Progressive strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation* 4(3):343–357.

Coulom, R. 2006. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th International Conference on Computer and Games*. Springer-Verlag, Heidelberg, Germany.

Coulom, R. 2007. Computing "elo ratings" of move patterns in the game of Go. *ICGA Journal* 30(4):199–208.

Davidson, A.; Billings, D.; Schaeffer, J.; and Szafron, D. 2000. Improved opponent modeling in poker. In *Proceedings of The 2000 International Conference on Artificial Intelligence (ICAI'2000)*, 1467–1473.

Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008*. AAAI Press.

Gelly, S., and Silver, D. 2007. Combining online and offline knowledge in uct. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, 273–280. New York, NY, USA: ACM Press.

Gelly, S.; Wang, Y.; Munos, R.; and Teytaud, O. 2006. Modifications of uct with patterns in monte-carlo go. Technical Report 6062, INRIA.

Harrington, D. 2004. *Harrington on Hold'em Expert Strategy for No Limit Tournaments*. Two Plus Two Publisher.

Kocsis, L., and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Artificial Intelligence*, 282–293.

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. In *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI-08)*, 1485–1487. Menlo Park, CA, United States: AAAI Press.

Ponsen, M.; Tuyls, K.; ; Kaisers, M.; and Ramon, J. 2009. An evolutionary game-theoretic analysis of poker strategies. In *Entertainment Computing*. Elsevier.

Slansky, D. 1987. *The Theory of Poker*. Two Plus Two Publisher.

Southey, F.; Bowling, M.; Larson, B.; Piccione, C.; Burch, N.; Billings, D.; and Rayner, D. C. 2005. Bayes' bluff: Opponent modelling in poker. In *Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence (UAI '05)*, 550–558.

Sturtevant, N. 2008. An Analysis of UCT in Multi-player Games. *ICGA Journal* 31(4):195–208.

Zinkevich, M.; Johanson, M.; Bowling, M.; and Piccione, C. 2008. Regret minimization in games with incomplete information. In *Advances in Neural Information Processing Systems 20 (NIPS)*.