Utrecht University
Faculty of Humanities
Cognitive Artificial Intelligence
Bachelor Thesis 7,5 ECTS

Opponent Modeling in Texas Hold'em

by
Jonathan Vrijhof

Abstract:

In this thesis several approaches to the problem of opponent modeling in poker are reviewed and some experiments concerning them are discussed.

Supervisor: Gerard Vreeswijk

Utrecht, 2011

Table of Contents

I. Introduction 3

II. Computer Poker Basics 4
- II.1 Hand Assessment

III. Action Frequencies 6
- III.1 Opponent Modeling
- III.2 Computing Initial Weights
- III.3 Re-weighting
- III.4 Results

IV. Probabilistic Knowledge and Simulation 8
- IV.1 Probability Triples
- IV.2 Simulation-Based Betting Strategy
- IV.3 Results

V. Neural Networks 10
- V.1 Architecture
- V.2 Accuracy
- V.3 Results

VI. Game Tree Search 13
- VI.1 Minimax and Expectimax Search
- VI.2 Miximax and Miximix Search
- VI.3 EV Calculation Example
- VI.4. Abstractions for Opponent Modeling
- VI.5 Results

VII. Related Work 19

VIII. Conclusion 20

IX. References 22

I. Introduction

In this thesis some techniques to deal with the difficult problem of opponent modeling in poker are explained, namely action frequencies, neural networks and game tree search.

Poker differs substantially from well-studied games such as chess and checkers, making it an interesting research subject. In games other than poker, players often have complete knowledge of the game state whereas in poker this is not the case. Poker involves *imperfect information* as players do not know the contents of their opponent's hand, making brute force search to determine a course of action a highly impractical option. Dealing with imperfect information is the main reason why research about bridge and poker has lagged behind other games. However, it is also the reason why they promise higher potential research benefits (Billings et al., 1998).

Furthermore, the game has a number of other attributes that make it an interesting domain for artificial intelligence research. These properties include multiple competing agents, risk management, opponent modeling, deception, and dealing with unreliable information. All of these are challenging dimensions to a difficult problem (Billings et al., 2002). Note that in this paper we will focus solely on Texas Hold'em poker, which is regarded as the most strategically complex variant that is widely played (Billings et al., 1999). We will only consider the Limit variant of Texas Hold'em, unless mentioned otherwise.
In this document, we will focus on opponent modeling and how it is implemented in computer programs, also known as 'poker bots'. Opponent modeling consists of two problems: to determine the cards that an opponent is likely to hold, and to predict the action that an opponent will take given a certain game state. Solving these two problems is essential to the performance of both human players and poker bots, though humans are naturally good at it. We will review different approaches to the problem of opponent modeling, their drawbacks and benefits, and the experiments performed to test their effectiveness. Finally, we will compare the approaches.

Most of the recent research on opponent modeling in poker has been done by the Computer Poker Research Group of the University of Alberta. Therefore this thesis will mostly concern their work. Before their work is described, we will go over some basic concepts in poker in Section **II. Computer Poker Basics**. Section **III. Action Frequencies** will review the approach detailed in the paper 'Opponent Modeling in Poker' by Billings et al., while section **IV. Probabilistic Knowledge and Simulation** describes the improvements made to this 'action frequencies' approach. Section **V. Neural Networks** will describe how Davidson et al. used artificial neural networks to model opponents and section **VI. Game Tree Search** will detail how the `Vexbot` computer program implements a game-tree search algorithm that uses real-time opponent modeling to improve its evaluation function. Lastly, some other research will be discussed in Section **VII. Related Work**.

II. Computer Poker Basics

Before we review some papers and the opponent modeling approaches they examine, we will first look at some basic concepts in poker concerning hand assessment. In this section we will discuss how hand strength and hand potential, two metrics for determining the overall strength of a hand, are calculated.

Hand Assessment

Determining the strength of a hand is an essential part of playing poker. Play before the flop has been studied extensively in poker literature (Sklansky and Malmuth, 1994). These authors have classified all the initial two-card combinations (pre-flop hands) into different categories. There are {52 choose 2} = 1326 possible two-card combinations which can be divided in 169 distinct hand types based on their strength against other hands. For example, 2-7 off suit is the worst possible hand, while A-A is the strongest.

After the flop, the overall strength of a hand can be determined by comparing it to all other possible hands, thus giving the probability of it being the best hand. Take the following situation, for example: Our hand is A♦Q♣ and the flop is 3♥4♣J♥. There are 47 unknown remaining cards in the deck and the opponent's hand. There are therefore {47 choose 2} = 1081 possible hands this opponent could have. We can now estimate our hand strength by comparing our hand to each of these possible hands. This yields a percentile ranking of our hand. Not taking into account opponent modeling, we regard each of those hands as equally likely. Each hand is either better than ours, equal to ours, or worse than ours. In this example, our hand is beat by any three of a kind, two pair, one pair or A-K high (444 cases). The remaining A-Q combinations are equal (9 cases) and the rest of the hands are worse (628 cases). Remember that hands higher than a three of a kind are not possible in this example. Counting ties as half, this yields a percentile ranking of 0.585. This corresponds to a 58.5% that our hand is better then a random hand. This calculation can easily be expanded to allow for multiple opponents by raising the percentile ranking to the power of the number of opponents. For instance, if there are 3 opponents in our example, our modified hand strength HS_3 would $0.585^3 = 0.200$.

Because a hand can have great potential for improvement, hand strength alone is often insufficient information. Consider the hand 5♥-2♥ with the same flop as described previously. Currently we only have a high card, but the odds of hitting a straight or flush are considerable. The opposite can also happen where a hand that is currently the strongest falls behind. The probability of our hand improving, or *positive potential* (PPot), can be calculated by simulating every combination of the community cards that have yet to be dealt. *Negative potential* (NPot), the probability of our hand falling behind, is calculated in the same way. This simulation of cards that have yet to be dealt is called a *look-ahead*. In our case, there are two cards that have yet to be dealt, so this is a two card look-ahead. For each of the 1081 possible hands, we consider the {45 choose 2} = 990 possible combinations of the next two community cards. The results of a two card look-ahead for our first example hand A♦Q♣ are shown in the table below:

5 Cards	**7 Cards**			
	Ahead	Tied	Behind	Sum
Ahead	449005	3211	169504	621720 = 628 x 990
Tied	0	8370	540	8910 = 9 x 990
Behind	91981	1036	346543	439560 = 444 x 990
Sum	540986	12617	516587	1070190 = 1081 x 990

The rows are labeled by the status on the flop; the columns by the final state after the community cards are dealt. For example, there are 91981 ways that we could be ahead on the river after being behind on the flop. There are 1036 possibilities to become tied after the river, and in the 346543 other cases we stay behind after the cards are dealt. So if we are behind pre-flop, we have about a 21% chance of winning.

Hand strength and hand potential can be combined into *effective hand strength* as follows:

$$EHS = HS_n + (1 - HS_n) * PPot$$

(HS_n is the modified hand strength with n opponents)

However, the assumption that all two-card opponent hands are equally likely is false. The next section will describe how Billings et al. used opponent modeling to take this into account.

III. Action Frequencies

In this section we will review the paper "Opponent Modeling in Poker" by D. Billings et al. In this paper the authors explained how they implemented both specific and generic opponent modeling in `Loki`. `Loki` is a poker bot, created by the University of Alberta's Computer Poker Research Group in 1997. The authors claim that the experiments discussed in the paper show that opponent modeling has a significant effect on the performance of the program. They used the so-called 'action frequencies' approach to implement opponent modeling in `Loki`. This computer program was the first successful demonstration of opponent modeling improving the performance of a poker bot.

Opponent Modeling

To account for opponent behavior we must calculate the probability for each hand that the opponent has played that hand in the observed manner. These probabilities are called *weights*. The accuracy of the calculations based on these weights can be improved by taking opponent actions into account. The opponent's betting behavior can be used to increase the weights for strong hands when the opponent raises, and reducing the weights of weak hands, for example. This is a form of *generic* opponent modeling as the model is identical for all opponents. It is also possible to keep a separate set of weights for each opponent, based on their betting history. This is a form of *specific* opponent modeling.

In Loki, each opponent is assigned an array of weights indexed by the two-card starting hands. This array is updated each time the opponent makes a betting action. As stated above, a raise could indicate that stronger hands are more likely, while weaker hands are considered less likely.

Computing Initial Weights

First, we must calculate reasonable estimates for the likelihood of each starting hand. The most important observed information for this calculation is the relative frequency of the actions folding, calling and raising before the flop. We calculate the *mean* (the median hand) and the *variance* (uncertainty) of the threshold needed for each player's observed action. These values are interpreted in terms of income rate values (return of investment), and mapped onto the set of initial weights of the opponent model.

Take, for example, an opponent who calls 30% of the hands and a median hand whose income rate is +200. If we assume a variance that translates to an income rate of +/-100, then hands with an income rate of +300 or higher would be assigned weight 1.0, weight 0.01 (unlikely but not impossible) to hands with an income rate lower than +100, and proportional values for hands with an income rate between +100 and +300. These initial weights provide a good early estimate for opponent behavior.

Re-weighting

To effectively model the opponent we must modify the weights after each betting action observed from that opponent. The opponent's actions can be classified into 36 different categories, based on the action (fold, call/check, raise), how much the action costs (0, 1 or >1 bets) and the betting round in which the action occurred (pre-flop, flop, turn or river). Some of these actions, namely folding when it is possible to check, do not normally occur.

After observing the frequencies of the opponent's actions we can update the weights in the `Loki` algorithm based on this new information. For example, if the observed action frequencies indicate that the opponent needs a median hand value of 0.6 to call a bet, with an uncertainty of 0.2, the lower bound will be 0.4 and the upper bound will be 0.8. In this case, all hands with an EHS greater than 0.8 are given re-weighting factors of 1.0, because the observed frequencies do not yield any new information about these hands. Any hand with an EHS less than 0.4 will be given a re-weighting factor of 0.01 because the opponent's behavior indicates that these hands are extremely unlikely. Linear interpolation is used for every hand with an EHS between 0.4 and 0.8. All the weights are then updated using these re-weighting factors. By the last round of betting, this algorithm will have the possible hands an opponent may have narrowed down to only a few with relatively high weights.

Results

In this section we will discuss the experiments performed by Billings et al., the results and what issues are left open.

Billings et al. tested the effectiveness of opponent modeling by letting 5 different versions of `Loki` play against each other. An older version of `Loki` was used to simulate 3 types of players: tight, loose and medium. This older version still has some generic opponent modeling, because having no information at all would be a large handicap (Billings et al. 1998). The other 2 versions used either generic opponent modeling (GOM) or specific opponent modeling (SOM), as described in the section above.
The authors came to the conclusion that both the SOM and GOM versions outperform the older versions of `Loki` by a large degree. The also state that "against players with very different styles of play, […] SOM's advantage over GOM will be magnified" (Billings et al. 1998). This is the case because SOM can keep track of all the different behavior patterns displayed by the opponents, while GOM cannot distinguish between different opponents. GOM has the advantage that opponent information can be acquired faster.

It is clear that poker bots with opponent modeling outperform those without, but the authors state that this implementation of opponent modeling has not yet provided conclusive results against human players. It is also worth noting that this opponent modeling technique (as implemented in `Loki`) implicitly assumes that the opponent also uses the hand strength and hand potential metrics to determine their betting strategy.

At this point, there is still room for improvements to the SOM implementation. As the authors state, "much of the relevant context was ignored for simplicity, such as combinations of actions within the same betting round." In the next section will we review two improvements to the action frequencies approach, probabilistic knowledge and simulation.

IV. Probabilistic Knowledge and Simulation

Because randomized strategies and misinformation are important aspects of strong poker play, it is of vital importance how a poker bot handles imperfect information. In this section we will review how Billings et al. used probabilistic knowledge to improve their poker bot, `Loki`, as detailed in their paper ‘Using Probabilistic Knowledge and Simulation to Play Poker’.

Probability Triples

The authors introduced a new way of representing knowledge: the *probability triple*. A probability triple is an ordered triple of values [f,c,r], where f + c + r = 1.0. These numbers represent the chance that the opponent will fold, call or raise, respectively. For example, the triple [0.0, 0.7, 0.3] indicates that the opponent will, in a particular situation, call 70% of the time, raise 30% of the time and never fold. Probability triples can also be used to determine which action the poker bot itself will take. The choice can be made by generating a random number, allowing the program to vary its play, even in identical situations. This is analogous to a *mixed strategy* in game theory, but here the probability triple implicitly contains contextual information resulting in a better informed decision which, on average, can outperform a game theoretic approach (Billings et al., 1999).

When `Loki` assesses a hand it compares that hand to every possible hand an opponent could hold. Probability triples are used to update the opponent’s Weight Table (discussed in the previous section) after each opponent action. For example, suppose the previous weight for Ace-Ace is 0.7 (which means that there is a 70% chance that the opponent would have played that hand in the manner observed, if it was indeed the hand dealt), and the opponent now calls. Given the probability triple [0.0, 0.2, 0.8] the updated weight would be 0.7 x 0.2 = 0.14. Thus, the likelihood that the opponent has Ace-Ace has been reduced to just 14%. This process is repeated for every possible hand. The probability triple can thus be used to represent uncertainty in our beliefs.

Simulation-Based Betting Strategy

Computing the decision tree for even one game of poker is prohibitively expensive. In effect, there is a branching factor of 3, as each player has 3 possible actions to choose from, and there may be several decisions in each betting round. `Loki` examines only a representative sample from all these possibilities, because we cannot consider all possible combinations of hands, future cards and actions (Billings et al., 1999).
`Loki`’s betting strategy now involves simulating likely scenarios to determine what decision is the most profitable. A simulation consists of the current hand being played out to the end twice, once with `Loki` calling, and once with `Loki` raising. (Folding requires no simulation, because the consequences are already known.) To select the candidate hands that our opponent might have Billings et al. introduced *selective sampling*. The selection of cards is biased according to observed information. For example, an opponent who raises a lot during a pot will likely have better cards than a player who just calls. In `Loki`, the opponent’s simulated hand is generated based on the probabilities stored in the Weight Table. Thus the cards selected for simulation are biased for the hands that have high probabilities. For each hand a probability triple is generated and a course of action is randomly chosen based on that triple. After a predetermined amount of simulations, the average *expected value* for each action is calculated. `Loki` then chooses the action with the highest expectation.

Results

Billings et al. performed experiments to study the effects of the 3 improvements made to `Loki`. These features are:

R: Changing the Re-weighting procedure to use probability triples.

B: Use probability triples to determine `Loki`'s action instead of a rule-based system.

S: Use simulations to determine the most profitable course of action.

The experiments showed that improvements **B** and **R** are nearly independent of each other. This is likely because the Re-weighting procedure is concerned with predicting opponent actions while **B** influences `Loki`'s betting strategy. Overall, all improvements greatly increased `Loki`'s performance, but enhancement **S** in particular. The selective sampling approach magnifies the quality of the evaluation function, achieving high performance with minimal expert knowledge (Billings et al., 1999). Another advantage of using simulations is that unlikely events are also taken into account (given enough trials) when calculating the expected values. We can conclude that the non-deterministic, imperfect information nature of poker is better represented by using probability distributions when making decisions. In summary, incorporating randomized strategies and simulation has a significant effect on the strength of a poker bot.

V. Neural Networks

This section will describe how neural networks can be used to model opponents in Texas Hold'em, as described in the paper 'Improved Opponent Modeling in Poker' (Davidson et al., 2000). Following that we will review the experiments described in that paper.

Every game of poker yields a huge amount of information about our opponent. The technique described in section **III. Action Frequencies** analyzes only a small part of this information (bets-to-call and game stage). It is possible that the opponent's decision-making process involves more context than just this small part, such as the size of the pot or the number of active players. The problem is determining what factors influence opponent behavior, and to what extent. This will of course vary from opponent to opponent. *Neural networks* are able to process all the context information present in a particular game of poker, and to cope with the amount of noise in such data (Davidson, 1999).
Neural networks can be used for both generic and specific opponent modeling, because you can use networks trained with data from certain players as well as networks trained with data from a balanced set of players. A broad range of players (weak and strong, loose and tight) is a requirement for a good generic opponent model. The next section will describe the architecture of the type of neural network used in the paper by Davidson et al.

Architecture

Neural networks consist of several layers of nodes: an *input layer*, 1 or more *hidden layers* and an *output layer*. In our case, the input layer contains one node for every game state parameter to be analyzed. The game stage (flop, turn or river), the number of players in the pot, or the pot ratio are examples of such parameters. This context information is translated to a real number between 0 and 1. The output layer contains 3 nodes corresponding to the possible opponent actions Fold, Call and Raise. If taken together and normalized, these values yield a *probability triple*. This is the output of the network.
Every node in a particular layer is connected to every node in both the preceding and the following layer, as shown in the image below. Every connection has a certain strength, or *weight*. These weights determine how strongly a node influences the following node. In a feed forward neural network the connections between nodes are one-way, going from the input layer towards the output layer.

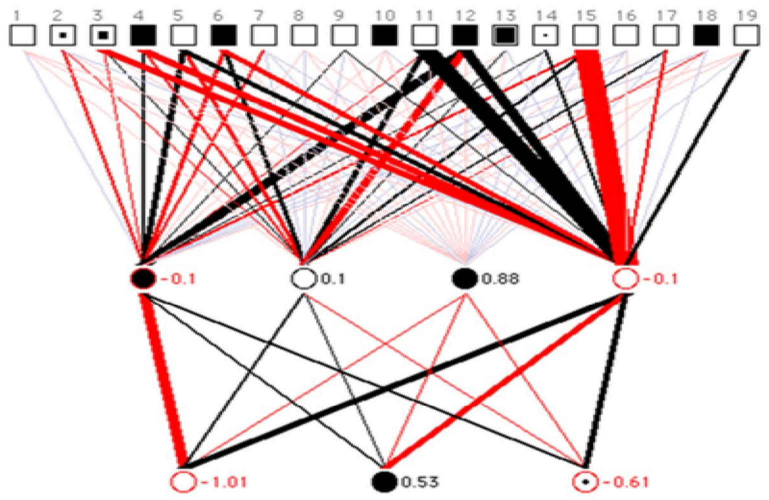

Fig 1. A neural network successfully predicting a call. (Davidson, 1999)

The network can be trained by putting *training sets* through the network. Each *training set* contains the input parameters of a certain game and the actual action the opponent made in that situation. Each weight is then updated so that the result of the network better matches the result specified in the training set. This process is called backward propagation of errors, or *backpropagation*, because it tries to reduce the discrepancy between the desired outcome and the actual outcome. For more information, see the paper 'Neural Networks' by Christos Stergiou and Dimitrios Siganos.

Accuracy

The accuracy of a neural network can be determined by running a test item through the network and comparing the network's output to the correct result. If the output node with the highest value is the correct action taken by the opponent, then the network has made a correct prediction. (Remember that each output node corresponds with one of the actions Fold, Call or Raise.) The error in the network is defined as the squared difference between the output and the correct answer.
A useful way to format the accuracy of a network is with a *confusion matrix*, as shown below.

	F	**C**	**R**	**Freq.**
F	13.0	0.3	0.3	13.6%
C	0.0	58.4	3.3	61.8%
R	0.0	10.5	14.1	24.7%
Freq.	13.0%	69.3%	17.7%	85.6%

Columns F, C and R represent the proportions in which the network predicted Fold, Call and Raise. Rows F, C, R show the actual action taken by the opponent. Therefore, the diagonal contains the times that the network predicted an opponent's action correctly. The Freq. column shows the distribution between the actions for the opponent, and the bottom row likewise shows the percentual distribution for the predictions of the network. The bottom right cell shows the percentage total of correct predictions, which is the overall accuracy of the network. An important advantage of using a confusion matrix is that it clearly shows what kind of mistakes the network makes. For example, in the figure it can be observed that when the opponent raises, the network quite often mistakenly predicts a Call (0.105), almost as often as a Raise (0.141). The information a confusion matrix yields about what kind of mistakes the neural network is likely to make can be used to update the probability triple to account for this type of mistake Thus we can update the probability triple to make a more conservative prediction (Davidson, 1999).

Results

Davidson et al. compared the results of the neural network approach with those of their old opponent modeling system, described in the previous section. The neural network was able to predict the opponent's action much more reliably, with an accuracy of about 81% whereas the old system had an accuracy of 57.3%. The results also showed that two important factors in the opponent's decision making process, the opponent's previous action and the previous amount to call, were not taken into account by the old opponent modeling system. These parameter were then included into `Loki`'s calculations. They were thus able to test 3 different systems: the old system, an enhanced version that took the 2 factors described above into account, and the neural network. The improved opponent model had an overall accuracy of

71.7%. As Davidson et al. note: “it is instructive to observe the significant improvements achieved with fairly simple enhancements.”

Sophisticated players will counteract our attempts to model them by changing often between styles of play. This is a problem for neural networks, because as a result they will be less able to identify patterns in opponent behavior. This can be dealt with by training several different networks at the same time, but in different ways. These networks will compete to be used, and are selected based on their accuracy in predicting the opponent’s most recent actions. A system such as this should give us the most accurate model possible at any given time (Davidson, 1999).

VI. Game Tree Search

The opponent modeling techniques described above have had only limited success, especially against strong human players. The following aspects of opponent modeling remain difficult:

1. The program must be able to learn quickly (within 100 games) from the information the opponent presents.
2. Good players often change their playing style; therefore our opponent model must be flexible and should not put too much emphasis on earlier opponent actions.
3. Because not every game leads to a showdown, we may never know what the basis was for the decisions an opponent made during a game in which he folded.

In the paper “Game Tree Search with Adaptation in Stochastic Imperfect Information Games” D. Billings et al. made the following contributions in an attempt to solve these problems:

1. *Miximax* and *Miximix*, two variations of the Expectimax search algorithm adapted for two-player stochastic imperfect information games.
2. Implement opponent modeling to refine the expected values in the game tree.
3. Introduce abstractions to translate a large amount of poker situations to a small amount of classes.
4. The poker bot `Vexbot`, which we will see is competitive with strong human players.

The following section will describe these contributions in greater detail and following that we will review the results of the experiments performed with `Vexbot`.

Note: for further information regarding tree search algorithms please consult “*Artificial Intelligence: A Modern Approach*”, by S. Russel and P. Norvig. Basic knowledge of these concepts is a requirement for the understanding of this section.

Minimax and Expectimax Search

A game tree is a tree that contains every action that each player can do, with the resulting leaves containing the payoffs for the players. Every odd layer in the tree contains choice nodes for player 1 while every even layer contains choice nodes for player 2 (the opponent). An example Minimax tree is shown below:

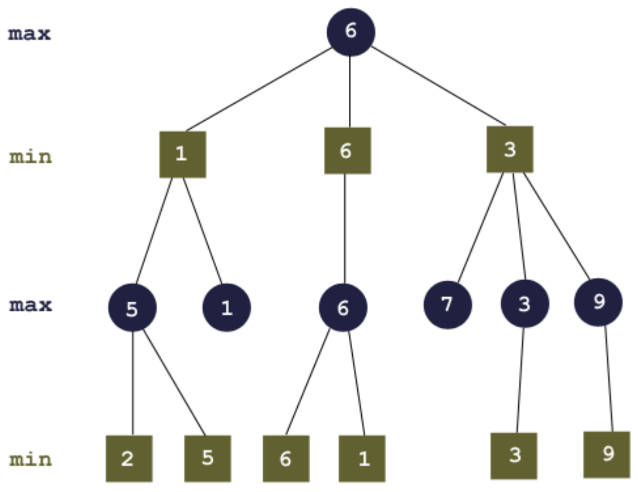

Fig 2. A minimax tree

Minimax search is a game tree search algorithm that assumes that the two players make the optimal decisions (for themselves) at each node in the tree. The Minimax theorem states:

> For every two-person, zero-sum game with finitely many strategies, there exists a value V and a mixed strategy for each player, such that (a) Given Player 2's strategy, the best payoff possible for Player 1 is V, and (b) Given Player 1's strategy, the best payoff possible for Player 2 is −V (von Neumann, 1928).

A zero-sum game is a game where the amount won by any player is equivalent to the amount lost by the other(s). Therefore, when player 1 tries to minimize his losses, this will in turn minimize the maximum payoff for player 2.

Expectimax is a variation of the minimax algorithm that combines the minimization and maximization nodes of Minimax search with chance nodes. The value of such a chance node is the sum of the values of each child of that node multiplied by the probability of that outcome. There are two reasons why Expectimax search is suboptimal for poker: firstly, for perfect information games such as backgammon, an opponent decision node is treated as a normal max (or min) node. However, this cannot be done for imperfect information games like poker, because the nodes of the tree are not independent (Billings et al., 2004). Secondly, taking the maximum or minimum value for a subtree is not always correct because the opponent may be using a randomized mixed strategy instead of a maximizing strategy.

Miximax and Miximix Search

D. Billings et al. implemented two variants of Expectimax suited for search on poker trees, namely *Miximax* and *Miximix*. These algorithms treat opponent actions as chance nodes with probabilities based on the information known about the opponent. Both algorithms perform a full-width depth-first search to the leaf nodes. In our case, since this is a two-player tree, a terminal node represents either a showdown or one of the players folding. The search tree comprises all possible player actions from the beginning of the game to the end. At the showdown leafs, the expected value (EV) is estimated with a heuristic evaluation function. As before (Section IV.), the EV calculation is used to decide the program’s action: fold, check/call or bet/raise. In Miximax we just select the action with the highest EV, so the tree would contain max nodes for the program’s choices and chance nodes for the opponent’s choices. However, always choosing the most profitable option can lead to a predictable play style, which might be exploited by the opponent. We can also use a mixed strategy ourselves, which is the case in the Miximix algorithm. Now, there are two unresolved issues:

1. How to determine the relative probabilities of the opponent’s possible actions at each decision node. This is based on the frequency counts of past actions at corresponding nodes (*ie*. given the same betting sequence so far).
2. How to determine the expected value of a leaf node. At fold nodes, computing the EV is easy – it is the net amount won or lost during the hand. At showdown nodes, a probability density function over the strength of the opponent’s hand is used to estimate our probability of winning. This histogram is an empirical model of the opponent, based on the hands shown in corresponding (identical or similar) situations in the past (D. Billings, 2004).

A Miximax search tree contains the following four types of nodes:

Chance nodes: The probabilities here correspond to the dealing of community cards. These probabilities are dependant on the cards that each player holds (since a card in an opponent's hand can no longer be dealt) but since our opponent's cards are unknown we make the assumption that the chances are uniformly distributed. The EV of a node *C* is calculated as follows:

$$EV(C) = \sum_{1 \le i \le n} \Pr(Ci) \times EV(Ci),$$

where *Pr(Ci)* is the probability of branch *i* occurring, and *n* is the number of branches.

Opponent decision nodes: Here the EV is calculated as follows:

$$EV(O) = \sum_{i \in \{f,c,r\}} \Pr(Oi) \times EV(Oi),$$

where *Pr(Oi)* is the estimated chance of branch *i* (fold, call or raise) occurring at opponent decision node *O*.

Program decision nodes: we can either use a mixed policy as in the node type above (Miximix), or we can always choose the action with the maximum EV (Miximax), in which case it is calculated as follows:

$$EV(U) = \max(EV(Uf), EV(Uc), EV(Ur)),$$

where *Uf, Uc* and *Uv* represent the action folding, calling and raising respectively.

Leaf nodes: At leaf nodes resulting from a fold the EV is the amount won during the hand if our opponent folded, and the amount lost if it was us that folded. In case of a showdown, the EV is calculated as follows:

$$EV(L) = (Pwin \times L\$pot) - L\$\cos t,$$

where *Pwin* is the probability of winning the pot, *L$pot* is the size of the pot and *L$cost* is the amount we invested in the pot.

EV Calculation Example

We will now look at an example of how the EV of each action is calculated in this algorithm. At each showdown leaf a *histogram* is stored containing a count of the hands our opponent has shown in previous games with the exact betting sequence (*ie*. series of choices) that lead to that particular leaf. The hands are sorted according to their *hand rank* (HR), between 0.0 and 1.0. In the following section betting sequences are represented as follows: a sequence of letters from {b, r, c, f} where our actions are in lower-case and our opponent's actions are in upper-case. So bRrF would mean that we bet, P2 raises, we re-raise and then P2 folds. For this example we will use 10-cell histograms for the sake of simplicity, so each cell has a range of 0.1. In the actual algorithm 20-cell histograms were used, however.

As before, we are Player 1 (P1) and our opponent is Player 2 (P2). The current pot contains 4 small blinds (sb) and we are in the final betting round. Suppose we bet (2 sb) and P2 re-raises in response. We want to predict what hands P2 is likely to have in this situation. Suppose that the histogram has (relative) weights of [1 2 0 0 0 0 3 3 1 0]. This histogram indicates that the previous times we observed this situation P2's raise was a bluff 30% of the time (HR in the range of 0.1-0.2) but that 70% of the time P2 had a strong hand (HR in the range of 0.6-0.9). In this case, the histogram for the leaf node after we re-raise and P2 calls will be related, being something like [0 0 0 0 0 0 3 3 1 0]. In other words, due to our re-raise, P2 would fold if he was bluffing but would have called our raise if he had a strong hand. Based on these histograms, we can infer that P2's decision after our re-raise can be represented by the probability triple {0.3, 0.7, 0.0}. Note that P2 will never re-raise in this scenario because that would only result in bigger losses if we have a hand with HR greater than 0.9 and has no beneficial effect otherwise. The betting tree for this scenario is shown below:

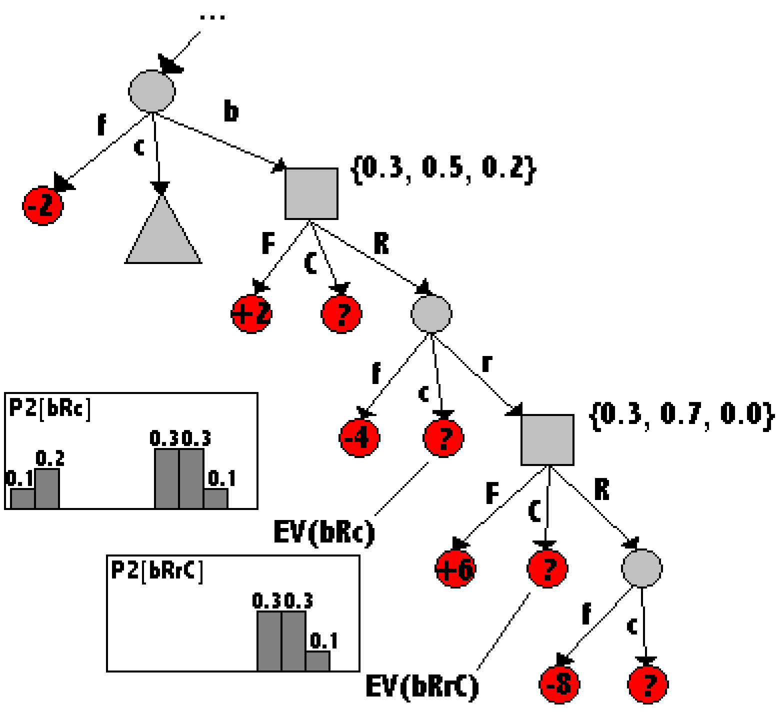

Fig 3. Betting Tree for EV Calculation Example

To decide what action to take we calculate the EV for each action of {fold, call, re-raise}. EV(fold) can simply be determined by looking at the pot size. In this case, folding would cost us -4 sb. EV(call) of course also depends on our hand rank. We can distinguish between 3 cases:

1) Our HR is in the range 0.2-0.6: we only beat P2 if he was bluffing, so Pr(win | bRc) = 0.30 (*ie*. the probability of winning when calling in response to P2's raise is 30%). The pot now contains 12 sb of which we paid 6, so EV(call) = (12 x 0.30) - 6 = -2.4 sb. In this case it is therefore more profitable to call than to fold.
2) Our HR is 0.1: we only beat P2 10% of the time, so in this case EV(call) = (12 x 0.1) – 6 = -4.8 sb. In this case we are better off folding.

3) Our HR is 0.7 or greater: if for example our HR is 0.8 we beat P2 90% of the time, because we beat P2's good hands with a HR lower than 0.8. In this case EV(call) = (12 x 0.9) – 6 = +4.8 sb.

To calculate the EV for a re-raise, we must determine the weighted average of all cases in that subtree, namely: bRrF, bRrC, bRrRf and bRrRc. Remember that the probability of a re-raise by P2 is considered zero, so the last two can be ignored. In the case of bRrF our hand strength does not matter (because our opponent folds), so the EV is 0.3 x 6 = +1.8 sb. The probability of winning in the case of bRrC is calculated using the histogram for that case, in the same way as described above. These values are then added together, yielding EV(re-raise).

Abstractions for Opponent Modeling

For the game of two-player Limit Hold'em, there are $9^4 = 6561$ showdown nodes for each player, or 13122 leaf-level histograms to be maintained and considered. This fine level of granularity is desirable for distinguishing different contexts and ensuring a high correlation within each class of observations (Billings et al., 2004). Updating these histograms is problematic however, since each case will only be observed very rarely. As a result, many thousands of games must be played before enough information is gathered. Also, as mentioned before, strong players will often change their play style, rendering older information useless. Therefore we must acquire knowledge very quickly, and incorporate a bias towards more recent games in our opponent model. Billings et al. implemented this bias in `Vexbot` as follows: each time an observation is made in a given context, the previous data is re-weighted by a history decay factor *h*, and then the new observation is added. If *h* = 0.95 for example, the new observation weights 5% of the total, while the last 1 / (1-*h*) = 20 observations account for (1- 1 / *e*) = 0.63 of the total, and so on (Billings et al., 2004).
To solve the problem of slow learning we must translate the large amount of possible poker situations to a relatively small amount of abstraction classes. This allows the program to apply its knowledge of similar situations to the current one. Billings et al. used a hierarchical system of abstractions. The abstraction level with the highest detail is a decision tree like the one in the example above, where every betting sequence has a separate class. Other abstraction levels group all betting sequences based on a common attribute; for example, all the hands where opponent made an equal amount of bets and raises throughout the hand, regardless of the order. In this hierarchical system each abstraction level is weighted based on the number of actual observations covered in that level, because these levels contain the most accurate information at that time.

Results

`Vexbot` was tested against both human and a variety of artificial opponents. Billings et al. concluded that `Vexbot` "easily exploited weaker players, and was competitive against expert level players". Against `Sparbot`, the previous strongest computer program for two-player Limit Hold'em, `Vexbot` eventually discovered an effective counter-strategy. Because `Sparbot` is based on game-theoretic optimal play it did not vary its strategy, making `Vexbot` more profitable by a small margin (+0.052 sb).
However, there is still a lot of optimization possible within `Vexbot`. Good default data is needed to ensure the program does not lose too much when it has yet to make observations of certain contexts. This is called the *zero frequency problem*. Billings et al. also noted that "there is a lot of room for improving the context tree abstractions, to obtain higher correlations among grouped sequences".

Summarizing, `Vexbot` “does the math” in order to make its decisions. As a result, many sophisticated poker strategies emerge without any explicit encoding of expert knowledge. The adaptive and exploitative nature of the program produces a much more dangerous opponent than is possible with a purely game-theoretic approach (Billings et al., 2004).

VII. Related Work

Though the approaches described in this thesis give a good overview of the field of opponent modeling in poker, other research has been done on this subject as well. Most of these papers describe how the authors used Bayesian networks (or belief networks) for opponent modeling, such as the paper "Bayesian Poker" by K. Korb, A. Nicholson and N. Jitnah. In this paper the authors explain how their poker bot, the Bayesian Poker Program, uses Bayesian networks to model the poker bots own hand, the opponent's hand and the opponent's playing behaviour conditioned upon the hand (K. Korb et al., 1999). They researched another poker variant however, named five-card stud poker.

Other approaches involve simplified variants of poker, such as Leduc Hold'em. The advantage of using such a simplified version is that the game theoretic results are known, whereas in normal Texas Hold'em the Nash (or minimax) strategy cannot be computed exactly (F. Southey et al., 2005). However, the results of these simpler variants can be used to approximate a solution for the more complex poker variants.

The problem of opponent modeling is not just limited to poker, though it usually does not play the central role it does in poker. The paper "Opponent Modeling and Commercial Games" by H. van den Herik et al. gives a good overview of the need for opponent modeling and ways to implement it in a range of games (two-person, multiplayer, commercial, etc.).

VIII. Conclusion

It is clear that opponent modeling is a very difficult problem that is essential to the performance of a poker player, both human and artificial. Opponent modeling is necessary to cope with the imperfect information aspect of poker, and because opponents will actively try to deceive you. We reviewed 3 different approaches to this problem: action-frequency predictors, neural networks and game tree search. Each of these approaches has advantages and drawbacks compared to the others.

The CPRG's first attempt at opponent modeling is based on the collection of simple statistical information, primarily on the betting frequencies in a variety of contexts. For example, a basic system distinguishes twelve contexts, based on the betting round (pre-flop, flop, turn, or river), and the betting level (zero, one, or two or more bets). For any particular situation, we use the historical frequencies to determine the opponent's normal requirements (i.e., the average effective hand strength) for the observed action. This threshold is used as input into a formula-based betting strategy that generates a mixed strategy of rational actions for the given game context (Billings et al., 2002). There is an important trade-off here between how quickly the system can learn from observations (more context means slower learning) and how quickly these observations can be applied. Despite using only a simple set of contexts, this opponent modeling system is reasonably effective.

Adding probabilistic knowledge and simulation improved `Loki` in a number of ways. Firstly, probability triples are a good way to represent both the uncertainty about the opponent's hand and his actions. They also allow `Loki` to use a mixed strategy. The probability triple framework allows the "messy" elements of the program to be amalgamated into one component, which can then be treated as a "black box" by the rest of the system (Billings et al., 1999). Secondly, simulation allows us to maximize the information extracted from a limited number of samples, which is useful considering the time constraints present in real-time poker. However, high variance in the results can be a major problem for the simulation approach. Simulation also reduces the amount of expert knowledge in the system, because the best action is determined by sampling instead of using a rule-based system. A rule-based approach is inherently flawed, resulting in a system that contains serious gaps and biases (Davidson et al., 2000).

The old opponent modeling system in `Loki` used only a very small part of the information that an opponent gives away during a game (bets-to-call and game stage). Using neural networks for opponent modeling has shown that they are a useful tool for analyzing which parts of this information are relevant for the opponent's decision-making process. These insights can then be used to improve existing opponent modeling systems. Different players will also react differently to the same contexts, making neural networks especially useful because they can learn and identify patterns in noisy data.
An important consideration with neural networks is how long it takes to get an accurate predictor for a given player. Here neural networks have the advantage over the action-frequencies approach (Section III.) because they can easily be trained on the hand histories of a given opponent. In addition, a default network can also be trained on all opponents to get a reasonable default model. The default model can be used until enough observations have been made to train a dedicated net for a player (Davidson, 1999). This does not work well for strong opponents who rapidly change their play style over the course of a game, as the hand histories would average each other out, reducing the quality of predictions.

Perhaps the greatest advantage neural networks have over the old approach is that the neural network can easily incorporate additional context features, whereas action-frequency predictors get increasingly more complex as game state information is added. This is because the table must be organized from the most general features to the most specific. Therefore, if more features are added, more work must be done to properly combine them to make an accurate predication. With neural networks the order of the context features does not matter.

A problem with both the action-frequency approach and neural networks is that the systems learn at a slower rate as information is added over time. If an opponent decides to change his style of play, the systems will struggle to adapt. A strong opponent who often changes his style of play magnifies this problem. This can be partially avoided by maintaining separate opponent models, one of which uses only more recent data. The opponent models can then compete to be used, and are scored by their prediction accuracy.
In the Miximax algorithm this problem is avoided by using an exponential history decay function so that more recent information is favoured over older data. This ensures that the program can adapt to an opponent changing play styles in a timely fashion. All the approaches described here share one drawback however: they need effective defaults while they are adapting to an (unknown) opponent.

Solving the zero frequency problem is essential for an effective Game Tree Search algorithm. A good way to mitigate the problem is by using good abstractions. This way, information of similar situations can be used to accelerate the learning process. With a good set of abstractions however, the Game Tree Search approach outperforms the others by a large degree.

IX. References

D. Billings, D. Papp, J. Schaeffer and D. Szafron, 1998. *Opponent Modeling in Poker*.

D. Billings, L. Peña, J. Schaeffer, D. Szafron, 1999. *Using Probabilistic Knowledge and Simulation to Play Poker*.

D. Billings, A. Davidson, J. Schaeffer, D. Szafron, 2002. *The Challenge of Poker*.

D. Billings, A. Davidson, T. Schauenberg, N. Burch, M. Bowling, R. Holte, J. Schaeffer, D. Szafron, 2004. *Game Tree Search with Adaptation in Stochastic Imperfect Information Games*.

A. Davidson, 1999. *Using Artificial Neural Networks to Model Opponents in Texas Hold'em.*

A. Davidson, D. Billings, J. Schaeffer, D. Szafron, 2000. *Improved Opponent Modeling in Poker*.

H. van den Herik, H. Donkers, P. Spronck, 2005. *Opponent Modelling and Commercial Games*.

K. Korb, A. Nicholson, N. Jitnah, 1999. *Bayesian Poker*.

J. Von Neumann, 1928. *Zur Theorie der Gesellschaftsspiele*. Math. Annalen. 100.

D. Sklansky and M. Malmuth, 1994. *Hold'em Poker for Advanced Players.*

F. Southey, M. Bowling, B. Larson, C. Piccione, N. Burch, D. Billings, C. Rayner, 2005. *Bayes' Bluff: Opponent Modeling in Poker*.