An Exploitative Monte-Carlo Poker Agent

Technical Report TUD–KE–2009-2

Immanuel Schweizer, Kamill Panitzek, Sang-Hyeun Park, Johannes Fürnkranz
Knowledge Engineering Group, Technische Universität Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Knowledge
Engineering

Abstract

The poker agent AKI-REALBOT described in this paper was designed to participate in the 6-player Limit competition which was part of the Computer Poker Challenge at the AAAI 2008 conference.It ended up in second place, its performance being mostly due to its ability to exploit weaker bots and still play fairly well against stronger players. This paper describes the architecture of the program and the Monte-Carlo decision tree-based decision engine that was used to make the bot's decision. It will focus the attention on the modifications which made the bot successful in exploiting weaker bots.

Contents

1 Introduction

Poker is a challenging game for AI research because of a variety of reasons (Billings et al., 2002). A poker agent has to be able to deal with imperfect (it does not see all cards) and uncertain information (the immediate success of its decisions depends on random card deals), and has to operate in a multi-agent environment (the number of players may vary). Moreover, it is not sufficient to be able to play an optimal strategy (in the game-theoretic sense), but a successful poker agent has to be able to exploit the weaknesses of the opponents. Even if a game-theoretical optimal solution to a game is known, a system that has the capability to model its opponent's behavior may obtain a higher reward. Consider, for example, the simple game of *rock-paper-scissors* aka *RoShamBo* (Billings, 2000), where the optimal strategy is to randomly select one of the three possible moves. If both players follow this strategy, neither player can gain by unilaterally deviating from it (i.e., the strategy is a *Nash equilibrium*). However, against a player that always plays *rock*, a player that is able to adapt its strategy to always playing *paper* can maximize his reward, while a player that sticks with the "optimal" random strategy will still only win one third of the games. Similarly, a good poker player has to be able to recognize weaknesses of the opponents and be able to exploit them by adapting its own play. This is also known as *opponent modeling*.

In this paper, we will describe the architecture of the AKI-REALBOT poker playing engine, which finished second in the AAAI-08 Computer poker challenge in the 6-player limit variant. Even though it lost against the third and fourth-ranked player, it made this up by winning more from the fifth and sixth ranked player than any other player in the competition.

2 Texas Hold'em Poker Basics and AAAI Poker Challenge Rules

Poker is played in many variants. The arguably most popular variant currently played is Texas Hold'em. In this variant, which can be played with up to ten players on one table, each player holds two cards, called the *hole cards*. In addition, five so-called *community cards* or *board cards* will be layed openly on the table. The winner is determined by forming the strongest possible hand consisting of five cards using the player's two hole cards and the five board cards. Thus, each player can use either both hole cards, one hole card or none (and accordingly three, four, or five community cards).

One single game of Texas Hold'em is called a *hand*. A dealer button is used to represent the position of the *dealer*. The position changes after each hand moving around the table clockwise. The player left to the dealer has to pay the *small blind* and the player left to the small blind has to pay the *big blind*. Usually the small blind is half the big blind, while the big blind is the minimum bet. The player sitting to the left of the big blind usually begins the first betting round.

The game consists of four different states. The first one being the *pre-flop*, followed by *flop*, *turn* and *river*. Each state ends with a betting round. At every players turn he can either *fold*, *check/call* or *bet/raise*. When the pre-flop betting is over the flop is dealt, three face-up community cards. Now, the player left to the dealer button starts the second betting round. Then, the turn, another face-up community card, is dealt, followed by a betting round. Finally, the last community card, the river, is dealt face-up. After the last betting round, the remaining active players have a *showdown*, where the winning hand respectively the winner is determined.

As each state has a betting round, there are different betting structures known in Texas Hold'em. The most popular variation is No-Limit Texas Hold'em which is also played at the famous World Series of Poker hold in Las Vegas. Almost as popular is Limit Texas Hold'em poker. Here the betting amount is restricted. In pre-flop and flop a bet or raise must be equal to the big blind, called a *small bet*. On turn and river it must be equal to twice the big blind, called a *big bet*.

The 2008 AAAI Computer Poker Competition[1] hosted a number of competitions. AKI-REALBOT participated in the 6-player Limit Competition. Here, the number of bets per state was limited to a maximum of four raises. As the money was practically infinite this was necessary to avoid a betting deadlock. The competition was played at stakes 10/20 which refers to the small bet of 10$ and the big bet of 20$.

There were 84 matches played among the six participants. Each match consisted of 6000 hands. The agents were assigned random positions at the beginning of each match to guarantee maximal fairness. This makes a total of 504,000 played hands which makes for a statistically relevant amount of poker games. The total amount of time that could be used by one agent for a match was seven seconds times the hands played. This makes a maximum of 980 hours of computing time per agent. How this time can be used efficiently will be described briefly in Section 5.

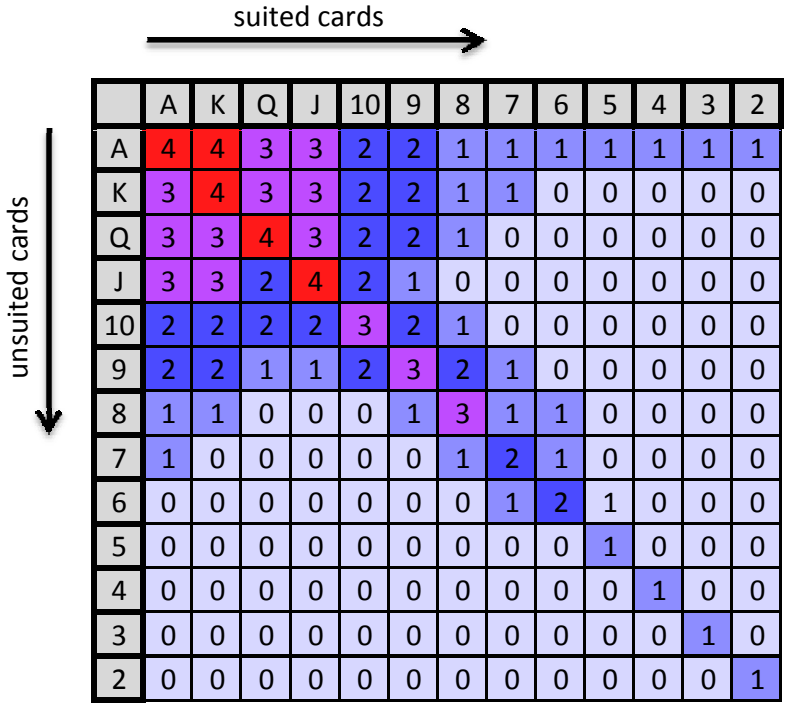
[1] http://www.cs.ualberta.ca/~pokert/2008/

3 Decision Engine

In this section, we describe the basic decision engine of AKI-REALBOT. We will describe bucketing, a basic mechanism for abstracting the state space in card games (Section 3.1), the basic Monte-Carlo search (Section 3.2), and the decision bounds which AKI-REALBOT uses for more aggressive play against weak opponents.

3.1 Bucketing

In order to reduce the space of possible card distributions, a bucketing system is used throughout all game states. The goal is to distribute all possible card combinations over b buckets (Sklansky and Malmuth, 1999). This is a popular method in designing a poker agent. In common literature 8 or 9 buckets are used. AKI-REALBOT uses just $b = 5$ buckets. Figure 3.1 shows into which of the five buckets 0 to 4 the two hole cards are mapped. The upper triangle shows the bucketing from cards in the same suit, whereas the lower triangle (including the diagonal) shows the same for cards of different suits. Higher bucket numbers represent stronger combination.



	A	K	Q	J	10	9	8	7	6	5	4	3	2
A	4	4	3	3	2	2	1	1	1	1	1	1	1
K	3	4	3	3	2	2	1	1	0	0	0	0	0
Q	3	3	4	3	2	2	1	0	0	0	0	0	0
J	3	3	2	4	2	1	0	0	0	0	0	0	0
10	2	2	2	2	3	2	1	0	0	0	0	0	0
9	2	2	1	1	2	3	2	1	0	0	0	0	0
8	1	1	0	0	0	1	3	1	1	0	0	0	0
7	1	0	0	0	0	0	1	2	1	0	0	0	0
6	0	0	0	0	0	0	0	1	2	1	0	0	0
5	0	0	0	0	0	0	0	0	0	1	0	0	0
4	0	0	0	0	0	0	0	0	0	0	1	0	0
3	0	0	0	0	0	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 3.1: Buckets for the hole cards

Obviously, the buckets are not evenly distributed, most hands will be in bucket 0. The probabilities of the 5 buckets that can occur in the pre-flop phase can be seen in Figure 3.2. So with a probability of 3% a player has a very strong hand which is assigned to bucket 4 (e.g. two kings).

0	1	2	3	4
65%	14%	11%	7%	3%

Figure 3.2: Bucket probabilities in pre-flop

3.2 Monte-Carlo Search

The Monte Carlo method is a commonly used approach in different scientific fields (Metropolis and Ulam, 1949; Allen and Tildesley, 1989; Frenkel and Smit, 2001). It was successfully used to build AI agents for the games of bridge (Ginsberg, 1999), backgammon (Tesauro, 1995) and Go (Bouzy, 2003; Coulom, 2006). In the context of game playing, its key idea is that instead of trying to completely search a given game tree, which is typically infeasible, one draws random samples at all possible choice nodes. This is fast and can be repeated sufficiently frequently so that the average over these random samples converges to a good evaluation of the starting game state.

Monte-Carlo search can be seen as being orthogonal to the use of evaluation functions. In that case, the intractability of an exhaustive search is dealt with by limiting the search depth and the use of an evaluation function at the leaf nodes, while Monte Carlo search deals with the problem by limiting the search breadth at each node and the use of random choice functions at the decision nodes. A key advantage of Monte Carlo search is that it can deal with many aspects of the game without the need for explicitly representing the knowledge. Many different factors influence the decisions of a world-class poker player. These include *hand strength*, *hand potential*, *betting strategy*, *bluffing*, *unpredictability* and *opponent modeling* (Billings et al., 1999). Most of these concepts are hard to model explicitly. By using Monte Carlo search, these concepts are modeled implicitly by the outcome of the simulation process. So there is a dynamic way of modeling the most important poker concepts without using explicit knowledge.

AKI-REALBOT will start the simulation process on his turn. In each game state, there are typically three possible actions, *fold*, *call* and *raise*.[1] AKI-REALBOT uses the expected amount of money it wins or loses to evaluate a decision. Folding will lose a fixed amount of money, the values of the other two options are determined via Monte-Carlo search (cf. Figure 3.3). Both subtrees, *raise* and *call*, have the same structure of the game tree and can be simulated independently. To take advantage of this fact multi-threading was used to speed up the simulation process. An increase in the number of simulated games also increases the quality of the expected values and therefore improves the quality of the decision. This is done by abstracting a simulation path to a general game being played, starting at different game states, one were the agent has called and one were it has raised. This is then done repeatedly until the simulation process is ended by the Time Management component described in Section 5.

Our Monte-Carlo search is not based on a uniformly distributed random space but the probability distribution is biased by the previous actions of a player. For this purpose, AKI-REALBOT collects statistics about each opponent's probabilities for folding (f), calling(c), and raising (r), thus building up a crude opponent model. This approach was first described in (Billings et al., 1999) as selective sampling.

For each played hand, every active opponent player is assigned hole cards. The selection of the hole cards is influenced by the opponent model because the actions a player takes reveal information about the strength of his cards, and should influence the sample of his hole cards. This selection is described in detail in Section 4.4. After selecting the hole cards, at each player's turn, a decision is selected for this player, according to a probability vector (f, c, r). The estimation of this vector is described in Section 4.2.

Each community card that still has to be unveiled is also randomly picked whenever the corresponding game state change happens. Essentially the game is played to the showdown. The end node is then evaluated with the amount won or lost by AKI-REALBOT, and this value is propagated back up through the tree. At every edge the average of all subtrees is calculated and represents the expected value of that subtree.

Thus, when the simulation process has terminated, the three decision edges coming from the root node hold the expected value of that decision. As we noted before, concepts like e.g. hand potential are implicitly modeled. In a random simulation, the better our hand is, the higher the expected value will be. This is still true even if we select appropriate samples for the opponents' hole cards and decisions as long as the community cards are drawn uniformly distributed.

So the expected value calculated by the simulation is a perfect basis for the following post-processing step, where expert knowledge is incorporated by imposing dynamical bounds that can change the overall behavior of AKI-REALBOT.

3.3 Decision Post-Processing

AKI-REALBOT post-processes the decision computed by the Monte-Carlo search in order to increase the adaptation to different agents in a multiplayer scenario even further, and to exploit every agent as much as possible. The exploitation of weak opponents is based on a simple consideration:

1. Weak players play too tight, i.e. they fold too often
2. Weak players play too loose (especially post-flop), which is the other extreme: they play too many marginal hands until the showdown

[1] According to the AAAI rules which allow only four bets per state, sometimes a *raise* may not be possible.

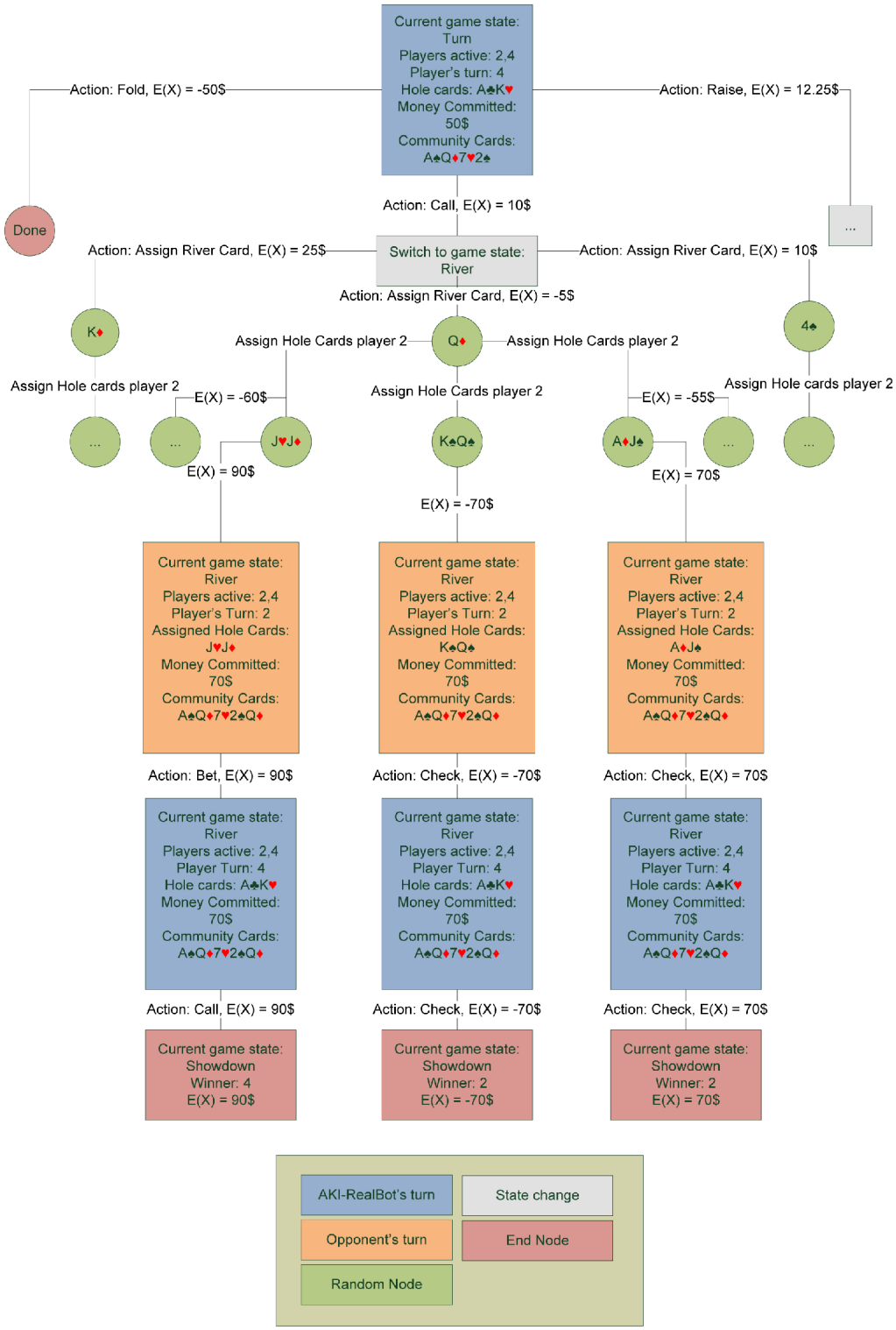


Figure 3.3: Monte Carlo Simulation: the figure depicts an example situation on the turn, where AKI-REALBOT is next to act (top). The edges represent in general the actions of players or that of the chance player. For the decisions *call* or *raise* (middle and right path), two parallel simulations are initiated. The path for the *call* decision for example (in the middle), simulates random games until the showdown (the river card is Qs (Queen of spades) and the opponent cards are estimated as KsQs (King and Queen of spades). Both players check on the river.) and the estimated loss of 70$ is backpropagated along the path.

These simply defined weak players can be easily exploited by an overall aggressive play strategy. It is beneficial for both type of players. First, if they fold too often, one can often bring the opponent to fold a better hand. Second, against loose players, the hand strength of marginal hands increase, such that one can win bigger pots with them than usual. Besides the aggressive play, the considerations implies a loose strategy. By expecting that AKI-REALBOT can *outplay* the opponent, it tries to play as often as possible against weaker opponents.

This kind of expert-knowledge, which is commonly known in poker, was explicitly integrated. For this purpose, so-called *decision bounds* were imposed on the expected value given by the simulation. This means that for every opponent, AKI-REALBOT calculates dynamic upper and lower bounds for the expected value, which were used to alter the strategy to a more aggressive one against weaker opponents. $E(f)$ will now denote the expected value for the *fold* path, while $E(c)$ and $E(r)$ will be the values for *call* and *raise* respectively. Without post-processing, AKI-REALBOT would pick the decision x where $E(x) = \max_{i=\{f,c,r\}} E(i)$. The decision bounds can change this behavior dynamically.

3.3.1 Aggressive Preflop Value

The lower bound is used for the pre-flop game state only. As long as the expected value for folding is smaller than the expected value for either calling or raising (i.e., $E(f) < \max(E(c), E(r))$), it makes sense to stay in the game. More aggressive players may even stay in the game if $E(f) - \delta < \max(E(c), E(r))$ for some value of $\delta > 0$. If AKI-REALBOT is facing a weak agent W it wants to exploit that weakness. This means that AKI-REALBOT wants to play more hands against W. This can be achieved by adding a $\delta > 0$ to our decision. We assume that an agent W is weak if he has lost money against AKI-REALBOT over a fixed period of rounds. For this purpose, AKI-REALBOT maintains a statistic over the amount of money, more precisely the number of small bets d, that has been lost or won against W in the last $N = 500$ rounds. For example, if W on average loses 0.5 SB/hand to AKI-REALBOT then $d = 0.5 * 500 = 250$ SB. Typically, d is in the range of $[-100, 100]$. Then, an *aggressive preflop value* δ for every opponent is calculated as

$$\delta(d) = \max(-0.6, -0.2 * (1.2)^d)$$

Note that $\delta(0) = -0.2$ (SB), and that the value of maximal aggressiveness is already reached with $d \approx 6$ (SB). That means, that AKI-REALBOT already sacrifices in the initial status $d = 0$ some EV (maximal -0.2 SB) in the pre-flop state, in the hope to outweight this drawback by *outplaying* the opponent post-flop. Furthermore, if AKI-REALBOT has won in the last 500 hands only more than 6 SB against the faced opponent, it reaches its maximal *optimism* by playing also hands which expected values were simulated as low as ≈ -0.6 SB. This makes AKI-REALBOT a very aggressive player pre-flop, especially if we consider that δ for more than one opponent is calculated as the average of the δ values for all active players.

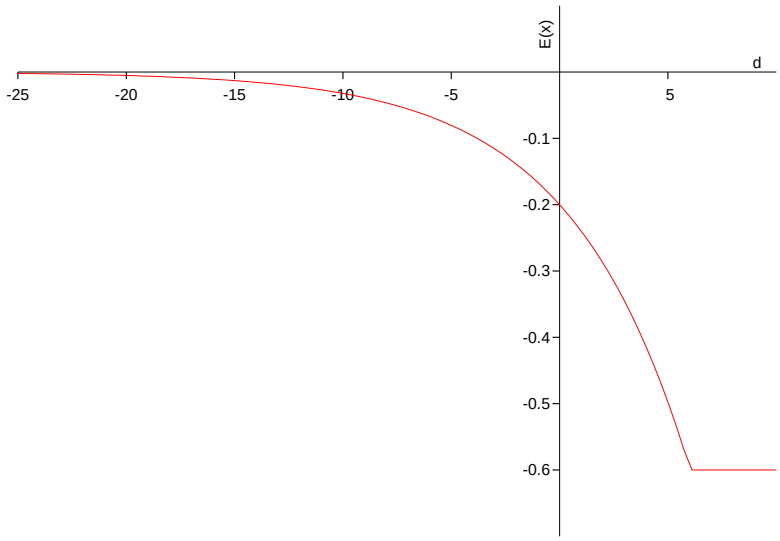


Figure 3.4: Aggressive Preflop Value $\delta(d)$

3.3.2 Aggressive Raise Value

The upper bound is used in all game states. It makes AKI-REALBOT aggressive on the other end of the scale. As soon as this upper bound is reached, it will force AKI-REALBOT to raise even if $E(c) > E(r)$. This will increase the amount of money that can be won if AKI-REALBOT is very confident about his hand strength. This upper bound is called the *aggressive raise value* ρ.

$$\rho(d) = \min(1.5, 1.5 * (0.95)^d)$$

Here, the upper bound returns $\rho(0) = 1.5$ for the initial status $d = 0$, which is 1.5 times the small bet and therefore a very confident expected value. In fact, it is so confident that this is also the maximum value for ρ. The aggressive raise value is not influenced if we lose money against a player. If, on the other hand, AKI-REALBOT wins money against an agent W, it will slowly converge against zero, resulting in a more and more aggressive play.

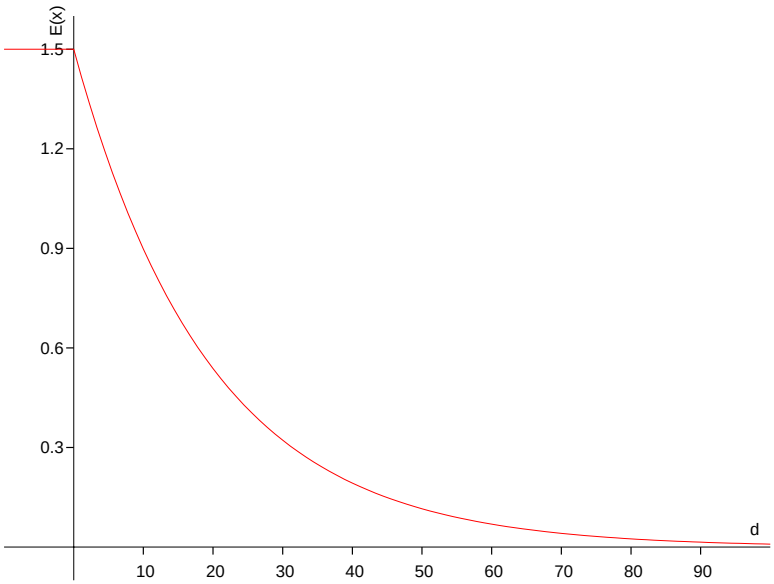


Figure 3.5: Aggressive Raise Value $\rho(d)$

As said before, the value of d is calculated based on a fixed amount of past rounds. It is therefore continuously changing with AKI-REALBOT's performance over the past rounds. The idea is to adapt dynamically to find an optimal strategy against any single player. On the other hand, it is easy to see that this makes AKI-REALBOT highly vulnerable against solid, strong agents. How this can be improved further will be described in Section 7.

Note that both bounds are only dependant of d and d is the win alternatively loss of one particular opponent against AKI-REALBOT. To compute d as the *overall* win/loss of one opponent against all opponents would yield more frequent updates, which could in turn lead to a faster adaption to the opponent. But it is especially important to know how well the opponents perform against the strategy of AKI-REALBOT and not their overall performance in the match. This value can change drastically for different setups, involving opponents with different strengths. Thus, the observed feedback of an adaption to an opponent would be disturbed.

4 Opponent Modeling

To adjust the implemented Monte-Carlo simulation to the individual behavior of the other players, information about their play is needed. This information can be used to rate the game situations in a more realistic way. This task cannot be accomplished by only considering the mathematical aspects of the actual game situation (Billings et al., 1998). Against a tight player raising might be more efficient than other actions because in that way the opponent could be urged to fold, as previously described. This is why opponent modeling is necessary.

In general, the opponent modeling of the AKI-REALBOT considers every opponent as a straight-forward player. That means, we assume that aggressive actions indicate a high hand strength and passive actions a low hand strength. Within the simulation, the opponent's hand strength is guessed based on the action he takes. So if a player often folds in the pre-flop phase but calls or even raises in one special game this means he has probably a strong hand. In addition the opponent modeling tries to map the cards to the actions every player takes. This gives a better understanding why a player makes certain decisions. Therefore all actions and cards need to be collected and stored accurately.

4.1 Data Structures

To gather all information about the opponents a data structure is needed. This structure is called HISTORY and is implemented as an observer of the game logic. That means every time the game state changes, this change is passed to the history. The history can then save all games, all game states and every action every player takes. To realize this, the history consists of two different inner data structures.

The first one is called ONEROUNDDATA. It maps one hand to the player. Every action a player makes, all his bets distributed over the different game states as well as his cards (if AKI-REALBOT is able to see them on showdown) are stored. And if the opponent wins a game, his winnings are saved as well. All these informations are gathered during one game.

The second data structure is called GLOBALROUNDDATA. This structure aggregates the single ONEROUNDDATA for every player. This enables the AKI-REALBOT to observe the behavior of one player in the long term. In addition every N games (in the actual implementation $N = 500$) a GLOBALROUNDDATA containing only the information about the last N games is stored separately. The function of this extra stored GLOBALROUNDDATA will be explained later.

But collecting only each opponent's actions per game or game state is not really meaningful to the simulation. We would like to refer to the taken actions in the context of the opponent's hole cards. The GLOBALROUNDDATA is able to save the hand strength of one player per game, per game state and per taken action. For this task, it uses the cards that the opponent reveals in a showdown. The opponent's cards are converted into a bucketing system and the probabilities for the different hand strengths (buckets) are adjusted. These probabilities are used to assign cards with a more realistic hand strength to the opponent during simulation.

Additionally the amount of money AKI-REALBOT wins from or loses to an opponent is stored in the history. This gives a good basis to minimize the losses to a strong opponent or to exploit a weak one. It is also important to save the game state an opponent folds in. This is needed to calculate the opponent's fold ratio.

But the history does not just store all this information. For AKI-REALBOT it is much more important to analyze the gathered information about the opponents and to use the data in the Monte-Carlo simulation. After collecting the data about all players the GLOBALROUNDDATA analyzes them and presents the results in different arrays.

4.2 Estimating Fold-, Call-, Raise-Ratios

A three dimensional array represents fold, call and raise ratios in the different game states and per player. Empirically we found that a straight-forward estimation of the fold ratio by the fraction of actions in which an opponent folded yielded too low values in comparison to the raise and call values, mostly because each player can only fold once in a game, whereas it can raise or call multiple times. Thus, we used the following heuristic approach for calculating the fold ratio f as

$$f(s) = \frac{N_f(s)}{N_p(s)}.$$

where $N_f(s)$ is the number of games in which an opponent has folded and $N_p(s)$ is the number of games in which the opponent was still at the table in a given game state s. These estimated values will be used in guessing the opponent's card bucket (Section 4.4).

The remaining probability mass $1 - f(s)$ is distributed according to the number of raises $N_r(s)$ and calls $N_c(s)$ to determine the raise ratio r and the call ratio c:

$$r(s) = \frac{1 - f(s)}{1 + \frac{N_c(s)}{N_r(s)}} \quad and \quad c(s) = 1 - f(s) - r(s) = \frac{1 - f(s)}{1 + \frac{N_r(s)}{N_c(s)}}$$

For many calculations another dimension is needed. That additional dimension represents the hand strength of the opponent. As said above, the hand strength is converted into a bucketing system. How the buckets are distributed over all possible cards and how the system works is described in the next section. If a satisfactory number of cards per game state is known, the average bucket is calculated and stored in this fourth dimension. This helps the simulator to better estimate the cards that particular opponent might have on his hand.

4.3 Adaption of Buckets to Opponents and Strategy Change Detection

The opponent modeling module estimates the hand strength of every opponent by recalculating the probabilities for every bucket. This is done according to the behavior of the particular opponent. Here, it is important how often the player has raised in the past in comparison to other actions he made. A raise in a certain game state means the player has a strong hand if he did not raise a lot in the past games and vice versa (Davidson et al., 2000).

Until now a constant strategy for every opponent is assumed, which is unrealistic. Especially other poker agents could adapt or change their behavior to confuse the AKI-REALBOT and win games. That is why the opponent modeler needs to check for changes in the behavior of every player. To recognize a change, the fold, call and raise ratios over all game states S from the current GLOBALROUNDDATA and the last N games are compared with each other.

$$v = \frac{1}{|S|} \sum_{s \in S} \Delta r(s) + \Delta c(s) + \Delta f(s)$$

If the variance v exceeds a certain threshold, a change in behavior of one player is recognized. In this case the GLOBALROUNDDATA of the opponent is replaced by the data of the last N games (cf. Section 4) and the old data is discarded. This check for changes in behavior is performed every N rounds (in the actual implementation $N = 500$).

4.4 Assigning Opponents' Hole Cards

The hole cards are what makes poker an interesting and challenging game because they are only known to the player and hidden from the other players. If a player could guess the opponents' cards, game-theoretically correct play would be relatively easy. Similarly, AKI-REALBOT's decision engine will return better results if it is able to estimate the opponent's hand strength. AKI-REALBOT has two different routines that enables it to guess hole cards according to the opponent model. Which routine it uses depends on the actual game state.

4.4.1 Pre-Flop Play

We assume that in pre-flop the actions a player takes are only based on his hole cards. He is either confident enough to raise or make a high call, whereas making a small call may indicate a lower confidence in his hand. A high call is indicated by committing more then two times the small bet. In either case his fold ratio f and his call ratio c are extracted from the history and used to calculate an upper and lower bound for the possible buckets.

In the first case, the upper bound U is set to the maximum possible bucket value ($U_h = 4$) because high confidence was shown. The lower bound is calculated by taking $l = c + f$ and relating this to the bucket. So if for example $f = 0.71$ and $c = 0.2$, which means that this player plays only 29% of the games ($1 - f = 1 - 0.71 = 0.29$) and raises in only 9% ($1 - (f + c) = 1 - 0.91 = 0.09$) of the games. Thus, only the 9% best cards are played. According to Figure 3.2, this maps to bucket 3. So the upper bound is set to $U_h = 4$ and the lower bound to $L_h = 3$.

If that same player had only called low, his new upper bound would be $U_l = L_h = 3$, calculated like the lower bound for raise. The lower bound would be $L_l = 1$, because $f = 0.71$ maps to bucket 1 according to Figure 3.2. The hole cards for that player are then assigned fitting in the interval $L_l \leq getBucket(hole) \leq U_l$.

4.4.2 Post-Flop Play

The second routine is used when the game has already entered a post-flop state. The main difference is that the actions a player takes are now based on hidden information, his hole cards, and visible information, the board cards. Therefore AKI-REALBOT has to estimate the opponent's strength also by taking the board cards into account. It estimates how much the opponent is influenced by the board cards. This is done by taking the number of folds for the game state flop. If a player is highly influenced by the board he will fold a lot on the flop and only playing if his hand strength has increased with the board cards or if his starting hand was irrespectively very strong.

This information is used by AKI-REALBOT to assign hole cards in the post-flop game state. Two different methods are used here:

- *assignTopPair* increases the strength of the hole cards by assigning the highest rank possible i.e. if there is an ace on the board the method will assign an ace and a random second card to the opponent.
- *assignNutCard* will increase the strength of the hole cards even more by assigning the card that gives the highest possible poker hand using all community cards i.e. if there is again an ace on the board but also two tens the method will assign a ten and a random second card.

It is important to note that for both methods the second card is always assigned randomly. This will sometimes strongly underestimate the cards e.g. when there are three spade cards on the board *assignNutCard* will not assign two spade cards.

These methods are used for altering one of the player player's hole card on the basis of his fold ratio f. We distinguish among three cases, where propability values p_{Top} and p_{Nut} are computed. These propability values state, how often one of the two methods (p_{Top} refers to *assignTopPair* and p_{Nut} to *assignNutCard*) are applied.

1. If the player folds less than 33% for a given state s:

$$f(s) < \frac{1}{3} \quad \Rightarrow \quad p_{Top} = \frac{1}{3} \cdot \left(\frac{f(s)}{\frac{1}{3}} \right)^2, p_{Nut} = 0$$

2. If the player folds more than 33% and less than 66%:

$$\frac{1}{3} \leq f(s) < \frac{2}{3} \quad \Rightarrow \quad p_{Top} = \frac{1}{3}, p_{Nut} = \frac{1}{3} \cdot \left(\frac{f(s) - \frac{1}{3}}{\frac{1}{3}} \right)^2$$

3. If the player folds at least 66% of the time in state s:

$$f(s) \geq \frac{2}{3} \quad \Rightarrow \quad p_{Top} = \frac{1}{3}, p_{Nut} = f(s) - \frac{1}{3}$$

To be clear, *assignTopPair* is applied with a probability of p_{Top}, *assignNutCard* is applied with a probability of p_{Nut} and with a probability of $1 - (p_{Top} + p_{Nut})$ the hole cards are not altered. As one can see in the formulas, the higher f is, the more likely it is that the opponent will be assigned a strong hand in relation to the board cards. If, for example, the fold ratio of player P is $f(s) = 0.32$, the simulation would use *assignNutCard* for around 30% of the games simulated to assign the hole cards for P. For the remainder of the games the pre-flop method would be used. The idea is to overestimate the hand strength of an opponent by using *assignTopPair* or *assignNutCard* and to underestimate it for the remainder of the games.

5 Time Management

To maximally exploit the time restriction, since more Monte-Carlo simulations yield naturally better results, a dynamic time management component was developed. The component tracks the time which AKI-REALBOT used over the past rounds and then assigns a fixed amount of simulation time to guarantee an average of 7 seconds per played hand. Amongst other things, it considers that many hands are played faster, e.g., when AKI-REALBOT folds in the first round, and exploits it for the following hands.

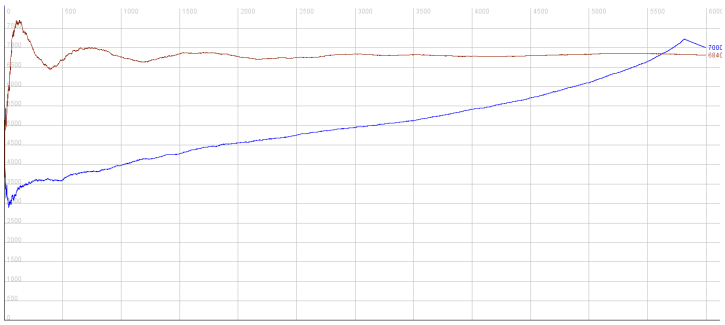


Figure 5.1: Average times per hand used by AKI-REALBOT (red line) vs. a naïve approach that divides the remaining time equally among the remaining games (blue line). The x-axis ranges from 0 to 6000 hands and the y-axis is denoted in milliseconds. Each point of both curves depict the average used time for the last 500 hands. Ideally, the curve should be a parallel at $y = 7000$ ms.

The first idea was to look at the different game states that are possible. There can be four rounds played in Texas Hold'em Poker. It starts with the pre-flop that is played in every game, then there is flop, turn and river which are only played if there are still two or more players in the game. From a simulation point of view it is obvious that the earlier the state of the game is the more possibilities there are to simulate. Therefore more simulation time is needed in the early states to make a good decision. This makes sense in more then one way. First, the agent needs more time to perform a simulation in the early game stages. Second, later game states might not always be reached because of a *fold* earlier in the game, so more time can be taken at the beginning of a hand. The third point is that early decisions are more important, because they may be hard or expensive to undo in the later game states. To reflect this, we split the round time over the four game states using 50% of the time on preflop, 30% on flop, 15% on turn and 5% on river. Note that for all states the simulation based decision engine was used, even for the flop. There are other approaches which distinguish between the pre-flop and post-flop phase and utilize different decision finding methods.

The next issue that needed to be addressed was that there are four possible betting rounds in every game state, so that in the worst case the bot would need to make four decisions within the time limit per game state. This was addressed by the same basic ideas that were true for the game states. Most of the time there is only one decision to be made in a game state. Moreover, the more players leave the game the smaller the simulation tree becomes so the time needed is always decreasing. So we chose to distribute the state times similar to the round time, but with 50% for the first, 25% for the second and 12.5% for the third and the fourth betting rounds. Also, there was a lower bound of 200ms for each round, introduced to guarantee a minimum amount of simulations made.

All of these ideas cover the worst case from a time management point of view. The case where the agent has to play over four game states making four decisions in every state. This is a very unlikely case since most of the time an agent folds in the first decision made. In this case only 25% of the possible round time would be needed and therefore 75% would be wasted if the round time would have been fixed.

The last idea was the concept of dynamic round times. The time management would start off with a round time higher than the average allowed by the rules. It would use that round time to assign simulation time to the different decisions made according to the scheme explained above. After every round finishes for the agent, the time management would track how much time was really used that round and would sum everything tracked in the past to calculate an average. If the average would be lower then the average allowed by the rules the round time would be increased by a small margin and vice versa. This system would allow the round time to converge to a reasonable value according to the game, while still ensuring that the average time would not exceed the average allowed by the rules of the competition. It would therefore give AKI-REALBOT a way of using the total time as efficiently as possible and keep the quality of the simulation at an almost constant high level. This is visualized by the red line in Figure 5.1. It starts to circulate around the 7 seconds but stabilizes quickly.ss

The first versions of the agent used a naïve approach which simply tracked the remaining time and divided it by the hands remaining to be played. This approach suffered from the fact that in the early games, the time for a hand was comparably small, and increased considerably over time, when more and more time was saved by early foldings in the games played, as can be seen by the blue line in Figure 5.1. Thus, the decision quality changed over time, which is undesirable.

6 Competition Results

The AKI-REALBOT was developed within a combined practical course and seminar on artificial intelligence in games (TUD Computer Poker Challenge) held at TU-Darmstadt.[1] In Total 7 bots were developed by 16 students. At the end, the bots were evaluated by playing 7 randomly seated matches, where every bot played exactly 6 matches. The setup was based on the one of the Annual Poker Competition of the AAAI, namely playing 6000 hands per match. So, every bot played in total 42.000 hands.

	AKI-REALBOT	HOKUSPOKUS	MCBOTPRO	BBHARDBOT	AETHON	ALPHACENTAURI	BRAINBOT
AKI-REALBOT	–	-9,924.17	-948.00	15,608.67	20,320.33	85,321.00	65,577.33
HOKUSPOKUS	9,924.17	–	1,145.67	4,631.50	4,573.50	31,792.33	15,949.00
MCBOTPRO	948.00	-1,145.67	–	7,465.33	4,684.50	21,015.33	13,112.00
BBHARDBOT	-15,608.67	-4,631.50	-7,465.33	–	1,843.00	4,062.17	11,717.50
AETHON	-20,320.33	-4,573.50	-4,684.50	-1,843.00	–	-3,692.83	19,236.83
ALPHACENTAURI	-85,321.00	-31,792.33	-21,015.33	-4,062.17	3,692.83	–	32,377.00
BRAINBOT	-65,577.33	-15,949.00	-13,112.00	-11,717.50	-19,236.83	-32,377.00	–
Total	173,362	70,608	46,083	-10,396	-15,524	-106,162	-157,971
avg. winnings/game	28,894	11,768	7,680	-1,733	-2,587	-17,694	-26,329
SB/Hand	4.128	1.681	1.097	-0.248	-0.370	-2.528	-3.761
Place	1.	2.	3.	4.	5.	6.	7.

Table 6.1: Results of the internal evaluation

Table 6.1 shows the results of the internal evaluation. Although AKI-REALBOT won by a large margin, it actually lost against the second and third-placed entries. Its overall winnings are only due to its superior ability to exploit weak opponents, which allowed to to win much more against the remaining four entries that any of the competitors. Based on these results, we decided to enter AKI-REALBOT and MCBOTPRO into the AAAI-08 competition. Both teams had a little time left to remove possible weaknesses, which was used for trying to improve AKI-REALBOT's play against stronger opponents. The second-placed bot HOKUSPOKUS was not submitted, because its authors could not continue to work on their code.

The two resulting bots, AKI-REALBOT and MCBOTULTRA, participated in the 6-player Limit competition part of the Computer Poker Challenge at the AAAI 2008 conference in Chicago. In addition to these two, there were four other entries:

- HYPERBOREAN08_RING from University of Alberta
- DCUBOT from Dublin City University
- CMURING from Carnegie Mellon University
- GUS6 from Georgia State University

Among these players, 84 matches were played with different seating permutations so that every bot could play in different positions. Since the number of participants were exactly 6, every bot was involved in all 84 matches. In turn, this yielded that every bot played 504000 hands. In that way, a significant result set was created.[2]

Table 6.2 shows the results over all 84 matches. All bots are compared with each other and the winnings from other bots as well as the losses to other bots are shown. Here it becomes clear that AKI-REALBOT exploits weaker bots because GUS6, the biggest looser, loses most of it's money to AKI-REALBOT. Note, that GUS6 lost in average about 1.5 small bets per hand, which is a worse outcome than by folding every hand, which results in a avg. loss of 0.25 SB/Hand. Although AKI-REALBOT loses money to DCUBOT and CMURING it manages to rank second, closely after HYPERBOREAN08_RING, because it is able to gain much higher winnings against the weaker players than any other player in this field, thus confirming the results of the internal evaluation. Nevertheless, if GUS6 did not participate in this competition, it is likely that AKI-REALBOT would have finished at one of the last positions.

Table 6.2 also shows how much money was won overall by every bot and how many small bets were won per hand. To establish a better overview the average winnings per match are shown in the table, too. The data is not precisely the average of the total shown in the table. It was calculated using our tool to evaluate poker matches. While all values are stored as integers some rounding errors have occurred on calculating the average over all 84 games. Using that tool a chart was created to show the average money development for every bot. This chart is presented in Figure 6.1.

One fact was discovered while evaluating the results of the matches. There is a little time discrepancy in the time manager. For some reason the clocks of the timer manager and the poker server are not always synchronized. The

[1] http://www.ke.tu-darmstadt.de/lehre/ss08/challenge

[2] The official results can be found at http://www.cs.ualberta.ca/~pokert/2008/results/

	HYPERBOREAN08_RING	DCUBOT	CMURING	AKI-REALBOT	MCBOTULTRA	GUS6
HYPERBOREAN08_RING		2,655.17	18,687.17	65,176.17	29,266.67	214,840.17
DCUBOT	-2,665.17		7,250.00	15,067.83	16,465.33	90,485.17
CMURING	-18,687.17	-7,250.00		2,768.50	7,548.50	92,453.33
AKI-REALBOT	-65,176.17	-15,067.83	-2,768.50		30,243.17	348,925.17
MCBOTULTRA	-29,266.67	-16,465.33	-7,548.50	-30,243.17		16,066.67
GUS6	-214,840.17	-90,485.17	-92,453.33	-348,925.17	-16,066.67	
Total	330,822	126,657	76,848	296,293	-67,529	-763,091
avg. winnings/game	3934	1512	939	3579	-800	-9042
SB/Hand	0.656	0.251	0.152	0.588	-0.134	-1.514
Place	1.	3.	4.	2.	5.	6.

Table 6.2: 2008 AAAI Poker Competition Results

server does not notify the bots about their remaining time and therefore the bots need to take care of this on their own. Although the stopwatch is started and stopped the moment a new round starts or ends the clocks show different times. There is even a buffer implemented to take care of the communication between bot and server. That little error also occurred while testing the bot. But it disappeared when the communication buffer was adjusted and never occurred again. This error can be seen in the last 200 rounds of the average game. There AKI-REALBOT loses money because a time out occurred and AKI-REALBOT was folded by the server.

Figure 6.2 shows the first of all games. It is quite typical for all other games. It illustrates that the bots do not win constantly over time from other bots. It is mostly an up and down. If a bot loses money for some games it does not necessarily mean it plays bad. Poker is still a game of luck and sometimes a bot does not have luck in that game. Here the little timer problem can be seen, too. It is the little flat bar in the last 50 rounds.

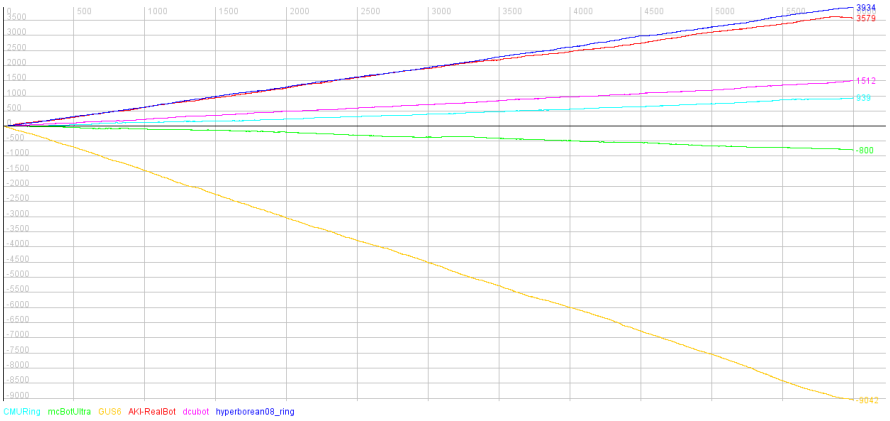


Figure 6.1: average game

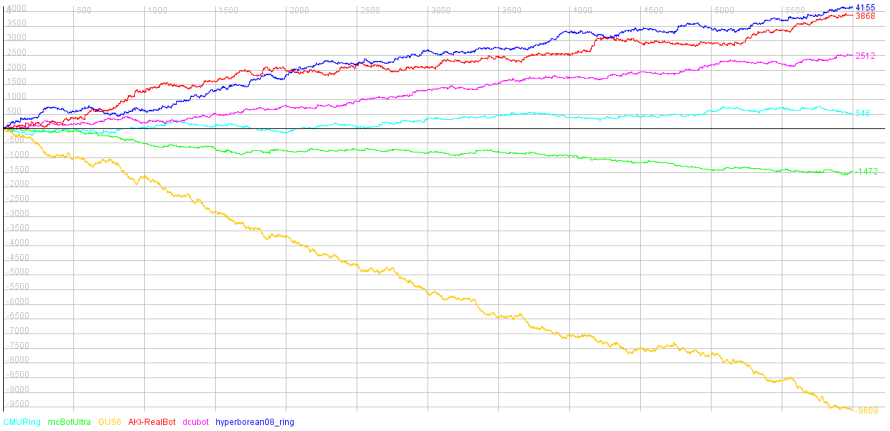


Figure 6.2: sample game

7 Enhancements

The results indicate that AKI-REALBOT's strength is to heavily exploit weaker bots, but they also show that the agent loses money against stronger bots. We have some ideas for enhancing the bot, which we believe would yield further improvements in performance.

7.1 Decision Engine

The main starting point for any further development should be the decision engine, since small performance changes might have a big influence on the simulated outcome. The more accurate the simulation works the less post-processing is needed and the more meaningful is any deduction made from the results. The easiest way to improve the results is to simulate more games. This would make the expected value more accurate. This can either be done by increasing the simulation time or by speeding up the engine. Since the time is set by the AAAI and the timing is optimized as much as possible there is no more time left. Speeding up the engine is the only option. Since the engine is all self-developed and the time window during the practical course until submission was so tight that there was no time to optimize for speed it should be possible to increase the speed drastically.

First tests showed that the speed can be easily increase from 10,000 simulations/second to 30,000 simulations/second almost effortless. So there should be a lot of speed left in the engine to squeeze out. The improvements made can be used to positively affect the engine in two directions. Either it is used to simulate more games or there is more time left for an extensive post-processing. Most promising should be a mixed approach with more simulations done and more time for the post-processing.

7.2 Decision Bounds

It does not make much sense to have more time for post-processing if there are no enhancements here. The decision bounds were the last enhancement built into AKI-REALBOT before the submission deadline and it proved to be the must promising for further exploration. But as much as it enables to exploit weak bots it enables other bots to exploit us. As noted in Section 3.3 most of the values are tweaked for maximal aggressiveness and must therefore be revised to make a more solid player.

One of the major weaknesses is that the bounds for more than one opponent are calculated as the average between both players. That means that for example if we play against two players one being weak W and one being very strong S with $\delta_S = -0.01$ and $\delta_W = -0.59$ we get an $\delta_{all} = -0.3$ which is a very aggressive value since overall, money is lost. What happens is that we might get drawn into the game by a weak player which then folds and leaves us with the strong player we might not have been played otherwise. So the method should value strong opponents higher than weak opponents. This can either happen drastically by using the $\delta_{all} = \max(\delta_i | i \in \text{Active Players})$ and $\rho_{all} = \max(\rho_i | i \in \text{Active Players})$. This on the other hand would mean that $\delta_{all} = -0.01$ for the example above which might scare AKI-REALBOT out of the game where we could otherwise have exploited W. A good approach would be to find a weighted function where the weight w is a function of d.

The second improvement must be the functions for ρ and δ themselves. For δ it must be important that $\delta(0) = 0$, so that AKI-REALBOT is not by default risking money. Also the value of maximal aggressiveness must not be reached for $d < 50$, and $\delta > 0$ must be possible for $d < 0$. All this would lead to a tighter game against strong bots while still exploiting weak bots. The exact values for δ must be derived by extensive testing and might even be dynamically adapted while playing.

7.3 Opponent Modeling

The implemented opponent model is still rather crude. It only logs statistics for the opponent, and the used numbers to select appropriate samples assume a straight-forward player. The relation between bucket and actions is not perfectly drawn. There is no component that guesses the bucket if a fold has occurred, therefore not that much data can be linked to the bucket. Most of the time the actions for an opponent are approximated. The samples are not selected appropriately because a folding opponent was not intended in the simulation and led to an decrease in performance. This should be

changed in future versions. Also, more sophisticated player profiles should be integrated, giving the post-processing engine a better understanding of strong or weak players and even help to assume the best strategy against any player.

The basis for a good opponent model is laid and the possibility to evaluate an opponent on a betting round basis is a major advantage. But this has to be improved even more to make the data more reliable and significant. The variance check to determine a strategy change has to be applied more often and should take more data into account. Last but not least, there is no way to model bluffing as of now. All of this could be used to further increase the implicit knowledge held by the simulation and improve the expected value.

The last step would be to model an opponent based on statistics and simulation so his future actions could be guessed based on our actions which would make the expected values for certain actions even more favorable.

7.4 Bucketing

A main disadvantage of the current opponent model is that bucketing was introduced last and is not yet fully integrated into the opponent modeling. In particular, there is no way to categorize hand strength into buckets in the post-flop phase. This is the main reason the methods described in section 4.4 for post-flop are needed. While the method in pre-flop is nice and easy, it was much harder to guess the cards in post-flop without a working bucketing system.

This only makes sense if the number of buckets can be increased so that the level of detail is also increased. All of this would lead to a better selection of the hole card sample. As a result, the quality of the expected value would increase and the post-processing could derive even more information. It would also lead to better statistics since they could also be derived based on the bucket for post-flop, giving a better selection of the future actions for any player.

7.5 Time Management

We have seen that AKI-REALBOT not only exploits weaker bots but also exploits the time that is given to simulate more games. However, the used time distributions are specified intuitively, and have not been verified on game data. One improvement is to see how many times the AKI-REALBOT plays a third and fourth betting round, and derive a new distribution from that data. Also, the time for all game states needs to be reviewed. The goal is to keep the quality of the decisions high by simulating a significant number of games. Therefore an evaluation of the time for the game states is needed. Also it needs to be evaluated how many simulations are needed for a certain number of players left in the game.

These two evaluations must lead to a better distribution of time over the game states. Another idea is to adjust the round times dynamically on how many players are left in the game and other factors. Here, a lot of testing and calculation is needed. The main point should be, however, to get the clock synchronized with the server. That is at the moment the main reason why AKI-REALBOT gets a timeout and loses money in the last 200 rounds.

8 Conclusion

In this paper, we have described the poker agent that performed second in the 2008 AAAI Poker Competition. Its overall performance was very close to the winning entry, even though it has lost against three of its opponents in a direct comparison. The reason for its strong performance was its ability to exploit weaker opponents. In particular against the weakest entry, it won a much higher amount than any other player participating in the tournament. The key factor for this success was its very aggressive opponent modeling approach, which allowed AKI-REALBOT to stay in the game against weaker opponents, even in cases where its simulation search recommended to fold.

Acknowledgments

We gratefully acknowledge the contributions of Frederik Janssen, Ulf Lorenz, Eneldo Loza Mencía, Jan-Nikolas Sulzmann, Lorenz Weizsäcker and all the participants in the TUD Computer Poker Challenge 2008: Timo Bozsolik, Bastian Christoph, Andreas Eismann, Thomas Görge, Björn Heidenreich, Benjamin Herbert, Michael Herrmann, Stefan Lück, Alexander Marinc, Lars Meyer, Arno Mittelbach, Hendrik Schaffer, Oliver Uwira, Michael Wächter, Claudio Weck, Marian Wieczorek.

We would also like to thank the organizers of the AAAI-08 Computer Poker Competition for their efforts in organizing this event.

Bibliography

M. P. Allen and D. J. Tildesley. *Computer simulation of liquids*. Clarendon Press, New York, NY, USA, 1989. 6

D. Billings. Thoughts on RoShamBo. *International Computer Games Association Journal*, 23:3–8, 2000. 3

D. Billings, D. Papp, J. Schaeffer, and D. Szafron. Opponent modeling in poker. In *AAAI/IAAI*, pages 493–499, 1998. 10

D. Billings, L. Peña, J. Schaeffer, and D. Szafron. Using selective-sampling simulations in poker. In *AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information*, pages 13–18. AAAI Spring Symposium, 1999. 6

D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of poker. *Artif. Intell.*, 134(1-2):201–240, 2002. 3

B. Bouzy. Associating domain-dependent knowledge and monte carlo approaches within a go program. In *Joint Conference on Information Sciences*, pages 505–508, 2003. 6

R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings Computers and Games 2006*. Springer-Verlag, 2006. 6

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

D. Frenkel and B. Smit. *Understanding Molecular Simulation (Computational Science Series, Vol 1)*. Academic Press, October 2001. 6

M. L. Ginsberg. Gib: Steps toward an expert-level bridge-playing program. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 584–589, 1999. 6

N. Metropolis and S. Ulam. The monte carlo method. *J. Amer. Stat. Assoc.*, 44:335–341, 1949. 6

D. Sklansky and M. Malmuth. *Hold 'em Poker for Advanced Players*. Two Plus Two Publications, 1999. 5

G. Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, 1995. 6