Managing a Team of Poker Players

Ensemble Strategien im Poker
Bachelor-Thesis von Tobias Thiel
Mai 2013

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Managing a Team of Poker Players
Ensemble Strategien im Poker

Vorgelegte Bachelor-Thesis von Tobias Thiel

1. Gutachten: Prof Dr. Johannes Fürnkranz
2. Gutachten: Dr. Eneldo Loza Mencía

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 22. Mai 2013

(Tobias Thiel)

Abstract

Poker requires an agent to adapt its strategy to the opponents in order to achieve maximum outcome. Against unknown opponents effective counter-strategies have to be found among predefined strategies. This thesis will introduce new techniques for dealing with highly dynamic poker games in which an agent has to adapt his strategy to the current situation. The agent will have access to arbitrary existing poker agents and detect when each of them can be used beneficially. The techniques will be evaluated among themselves and against related work, to show their advantages and disadvantages.

Abstract

Im Poker ist die Adaption an die Spielweise der Gegner ein elementar Bestandteil, da so der eigene Gewinn maximiert werden kann. Im Spiel gegen unbekannte Herausforderer müssen die erhältlichen Strategien außerdem dynamisch evaluiert werden, um so die besten Strategien für die eigenen Situationen zu finden. Diese Arbeit wird sich diesem Problem annehmen und neue Möglichkeiten zur Adaption in dynamischen Pokerspielen einführen. Dabei können beliebige existierende Spieler zu einem Team zusammengeführt werden. Die vorgestellten Techniken werden dabei gegeneinander, aber auch gegen bereits existierende Ansätze evaluiert, um ihre Vor- und Nachteile zu demonstrieren.

Contents

List of Figures

List of Tables

1 Introduction

Games are a popular research domains in Artificial Intelligence (AI), which provide important knowledge necessary to solve challenging real world problems [Sch01]. They gained their popularity in academic research because they all provide beneficial features, which make the evaluation of agents more objective and therefore easier to measure:

- well-defined rules how the game is played
- clear objectives and overall goals
- direct feedback of success (won/draw/lost)
- usually experts exist as real world opponents

These features make the development and evaluation of an AI agent for a game easier and more transparent than for a complex real world problem. Nowadays software can therefore play some popular games at a level that is at least as good as the best known human players:

- The game *Backgammon* was successfully played by a computer program against "world-class human grandmasters" [Tes94].
- A computer named Deep Blue was able to beat *chess* Grandmasters in tournaments [CJhH02].
- Computers can play *Checkers* perfectly, so that the game results in a draw when the opponent also plays perfect [SBB+07].

Formally games can be classified by the following properties:

- *perfect or imperfect information*
  In games such as checkers with perfect information, all players have complete knowledge of the current state of the game. In contrast games like Battleship have imperfect information, because every player only has complete knowledge of his own situation and has to gain knowledge about the situation of the other player throughout the game.
- *deterministic or stochastic*
  Games which include stochastic events like rolling a dice or dealing of random cards make it impossible to precisely predict the future game states. When playing Backgammon, the AI agent can't determine which steps his opponent will take in the next turn, because the outcome of the next roll of dice is unknown. A chess player on the other side knows exactly which moves his opponent can take in his next turn.

This thesis will focus on Poker, specifically Texas hold'em Limit, which features imperfect information, because each player has private cards which are only known to him. Poker is also stochastic, since the cards are dealt from a randomly shuffled deck. It is a particularly interesting research game, because it additionally has a few unique properties compared to other popular games:

- *Exploitation is crucial*
  In poker it is not sufficient to win more than the own investment. If an opponent won more money by exploiting weak opponents, he will perform a lot better, although both players won throughout the game. A player therefore has to win by a large margin whenever possible.

- *Multiple competing players*
  Although poker can be played with only two players it is generally played with more players. Every player is competing against each other, sharing the money they invested. Usually the number of players also varies during a round. Overall this reduces the control an agent has on the game, since he does not necessarily play e.g. half the actions of the game, like during a chess game.

- *Partially observable*
  Private cards are not always revealed at the end of a round. Therefore a player can't always confirm whether his actions were the right choice.

- *Risk Management*
  Poker players have to decide if their current private cards are worth the risk of loosing their invested money, This is particularly interesting in phases of the game where a player doesn't have much money left and needs to decide on the hands worth playing to win money back. If the player is too defensive, he might never win his money back or take a long time to achieve it. When the player is too aggressive, he might quickly loose the limited amount of money he has left.

- *Deception*
  Especially human professional players don't always take actions which are appropriate given their private cards, therefore deluding other players in the game. This is an integral part of the way poker is played among human professional players.

Poker became a rather popular topic, much like chess was in AI research so that the "Association for the Advancement of Artificial Intelligence" (AAAI) holds annual competitions, called "AAAI Annual Computer Poker Competition" (ACPC), since 2006 where the state-of-the-art poker agents compete against each other to evaluate the work being done [acp12a].

The next chapters will introduce the structure of the thesis and describe its motivation and goals, as well as giving a short introduction into the terminology and rules of Texas hold'em Limit Poker.

1.1 Problem Description & Motivation

In Poker, there are many strategies to choose from, each one of them having advantages and disadvantages. One can use an ϵ-Nash equilibrium strategy, which is improbable to loose, because no player in the game can do better by changing their strategy. This allows agents to defend themselves against any opponent [Joh07], but also fails to exploit weaknesses the opponent might have, therefore not winning the maximum amount possible. An exploitive strategy on the other hand is only effective against the opponent strategy it was trained against and might even make itself vulnerable in the process, because it deviates from the safe equilibrium strategy. Especially when playing against unknown opponents this will become apparent since the optimal strategy is unknown beforehand and one has to hope the used strategy is good enough and that the opponents are unable to exploit it. Team approaches have already proven themselves as useful tools in Rock-Paper-Scissors competitions [Egn00]. Dan Egnor was able to develop a dominating player by employing a meta-strategy approach, which will be further introduced in Section 2.2 ("Related Work").

One approach to deal with this problem could be to combine as many different strategies into one overall strategy as possible, but this seems rather limited, because converse strategies, might not be compatible for combination. Since these kinds of strategies are prime candidates to reach an overall flexible strategy, other ways of combining different strategies might be more successful and straightforward.

Another approach to deal with the problem is to use a team of independent poker agents to play against the opponents in the game. The agent interacting with the other players in the game then is only a gateway dynamically deciding which strategy should be played in the current situation. If the team

contains a variety of strategies it has the ability to use the appropriate strategy exploiting an opponent in the current situation and can so maximize its winnings. Additionally strategies can be switched if the current strategy is being exploited by the opponents or the opponent changes its behavior. This can make the ensemble agent very adaptive to changes occuring during a poker game.

The thesis will deal with this last described team approach. This will be particularly interesting to tackle the described challenges in poker (Section 1) of exploitation and multiple opponents. The agent will be able to switch to other strategies which exploit the opponents more and also choose the strategies depending on the opponents left in the game. The agent can choose to play a safer strategy when facing all opponents, but in a heads-up situation with a particular opponent play more aggresively to exploit him. Deception is more present in games against human opponents and will therefore not be considered. Risk Management also won't be simulated in the automated games, since most evaluations and official competitions don't include it. Finally, partial observability is more of a general problem, which won't be solved by a team approach.

1.2 Goal

The goal of this thesis is to implement a reusable framework which allows custom implementations for team agents which make the decision which strategy should be used in the current game situation. This implementation will use the framework for developing intelligent poker agents previously implemented at *Technische Universität Darmstadt* by Zopf [Zop10].

Furthermore approaches of choosing strategies will be implemented and evaluated. They will be compared to each other and the benefits of using a team approach versus the traditional one strategy approach will be elaborated. Previous approaches to use teams of poker players will be summarized and be compared against the new methods introduced in this thesis.

1.3 Structure of Thesis

This first chapter introduces the team approach to poker in an abstract and short manner. The following Chapter 2 ("Preliminaries") will describe the rules to Texas Hold'em poker and introduce the related work in this field of research. It will conclude with a more detailed introduction of the particular team approach that will be used.

After that the developed implementation will be discussed and some of the used software shortly introduced in Chapter 3 ("Software & Implementation"). This is followed by Chapter 4 ("Ensemble Strategies") which introduces a classification of the common meta-strategies used in teams, also introducing the meta-strategies used in this thesis. After that Chapter 5 ("Game Statistics") describes in detail how informed decisions are made within the implemented team framework. This chapter also introduces the various methods which will be evaluated.

The penultimate Chapter 6 ("Evaluation") will describe all the evaluation which was done. The thesis will then conclude in Chapter 7 ("Conclusion & Future Work") summing up the work done and introducing some future work which could be done in this area.

2 Preliminaries

2.1 Poker Terminology & Rules

Poker involves two or more players, who play multiple rounds. Each player has private cards which they combine with public cards on the table to a five-card hand. The players then bet each round that their hand will be the highest ranked of all players at the end of the round. Whenever a player places a bet, the investment is added to the so-called pot, whose contents can be won at the end of each round.

Each round consists of multiple betting rounds during which each player can take the following actions:

- *Fold*
  The player puts no further chips into the pot and abandons his hand, therefore not contributing any more until this round is over and loosing the chance to win the pot.
- *Check/Call*
  The player puts the minimum amount of chips necessary to match the highest previous bet by another player into the pot. He can continue to contribute in this round. A check is a call with no chips, the player therefore already has matched all bets and doesn't want to increase the size in the pot.
- *Bet/Raise*
  The player puts more than the minimum amount of chips necessary to match previous bets into the pot. Could the player have checked this is considered a bet, otherwise a raise.

A betting round can only continue when all players individually have matched the highest bet or folded. A round is finished if the last betting round is over or all but one player folded. The structure and number of the betting rounds is different throughout the many variants of poker. In the following the betting rounds for the variant Texas hold'em which will be used throughout the thesis is described.

Texas hold'em is normally played with so called blinds which force two players in the round to make a bet before the round starts. One player always has the dealer button and is therefore the dealer. This button rotates clockwise after each played round. The small blind, forced to place half of a minimum bet is the player after the dealer and the big blind (placing a full minimum bet) is the player after that. If there are only two players the dealer has to post the small blind and the other player places the big blind.

A round in Texas hold'em is played in 5 stages: preflop, flop, turn, river and the showdown. The 5 stages of each round are executed as follows:

- *Preflop*
  The dealer gives each player two private cards and the blinds begin the round by placing their forced bets. Then the player to the left of the big blind, also known as under-the-gun, is the first to act.
- *Flop*
  The dealer deals three public cards into the middle which are called board cards or community cards. The player to the left of the dealer is the first to act.
- *Turn*
  The dealer adds another public card to the board cards. Another round of betting is started by the player to the left of the dealer.

- *River*
  The dealer adds another public card to the board cards. The last round of betting is started by the player to the left of the dealer.

- *Showdown*
  All players which are still in the round have to reveal their cards and the player with the strongest hand wins the pot, containing the betted chips this round. If multiple players have the strongest hand, the pot is equally divided between them.

This thesis focuses on 3-player Limit Texas hold'em. The limit restriction means that all bets have a fixed size, whereas the no-limit variant additionally allows the player to decide on the size of his bet, within restrictions, including the all-in bet, where all the chips of the player is put into the pot. This gives players more strategic choices, but also increases the complexity of the decision process.

The official ACPC rules [acp12b] in the 3-player limit games are 1000 hands in 3-player duplicate matches (permutations of positions with constant cards). The blinds are constant and consist of 5 and 10 chips. A bet during preflop or flop costs 10 chips, otherwise 20. The winners are determined through two different methods:

- *Bankroll instant run-off*
  After a set of matches is played where all players played against each other in all combinations, the players with the lowest total bankroll will be removed from the set of players. With this new set a new round of matches is started, which is repeated until only 3-players remain in the set. These three players will then be ranked according to their performance against each other. This measurement favors players who can compete a lot of different opponents.

- *Total Bankroll*
  The player with the highest bankroll combined after all games against all other players wins. This is a measurement for players which are especially good at exploiting certain opponents, since they will often score higher.

2.2 Related Work

Dan Egnor already successfully used team approaches to develop a very competitive Rock-Paper-Scissors player in "The First International RoShamBo Programming Competition" [roc99] in 1999. His winning entry "Iocain Powder" [Egn00] used multiple strategies for different types of opponents. A meta-strategy was then used to decide which of the available strategies would be the best against the current opponent. To make this decision he analyzed the (hypothetical) past performance of the various strategies and chose the one with the best chances of suceeding. In case none of the advanced strategies perform well, the meta-strategy falls back to a random strategy, which guarantees that the player doesn't loose much in Rock-Paper-Scissors.

In the domain of poker Johanson et al [Joh07, JZB07] already evaluated a team of agents in 2007, by training exploitive strategies as counter strategies for a few predefined strategies. These trained counter-strategies were then put into one team which challenged the strategies they were trained against, as well as some unknown strategies. Their team agent decided which strategy should be used and then queried it for its action for each decision to be made. The team agent therefore always queries only one strategy, which transforms the strategy choice to a multi-armed bandit problem.

The multi-armed bandit problem [ACBFS95, Rob52] describes a gambler facing multiple slot machines, each having an unknown distribution of rewards. Whenever the gambler plays a machine he receives a reward or looses money according to the distribution of the machine. The gambler obviously wants to maximize his rewards. This faces him with a trade-off between exploration and exploitation. When he exploits a machine, he might be missing out on an even more rewarding machine. If he plays too many

different machines, he might not get the best reward possible, because he plays the most rewarding machine less times.

As Johanson et al consult only one strategy for each decision, this formalized problem fits exactly to the choice between strategies. They use the UCB1 [ACBF02] algorithm (discussed in more detail in Section 4.2.1) to make the decision between the strategies, which is able to let the average regret approach zero [Joh07], since the regret of following one strategy grows logarithmically. UCB1 defines the regret as the difference between the reward received by always choosing the optimal strategy and the actual received reward. It therefore detects gradually which of the strategies is the optimal one. To evaluate the teams Johanson et al used an ϵ-Nash equilibrium agent, whose performance was compared against the two previously described sets of opponents (unknown opponents and opponents a team member was trained against). The team was able to match or outperform the ϵ-Nash agent against most opponents in some configurations [Joh07].

Rubin and Watson talked about the necessity of a team approach in 2011 [RW11a]. They argued that one single exploiting strategy can only function when facing the intended opponent, but as soon as other competitors, which operate different from the intended opponent are in the pool of adversaries, the strategy won't function well. This especially becomes noticeable when facing unknown opponents against whom no optimal counter-strategy exists. They then continued to evaluate a team approach in 2011 [RW11b] by transforming the strategy choice into a multi-armed bandit problem as done by Johanson et al [JZB07], also using UCB1 to rate the different strategies. They additionally evaluated a different approach from the multi-arm bandit abstraction, in which they used majority voting between all strategies to determine the action to be played. The teams in the evaluation were constructed using imitators from the AAAI computer poker competition in 2010 and consisted of 3 basic strategies:

- imitator trained on the winner of the bankroll division
- imitator trained on the winner of the equilibrium division
- entry by Rubin and Watson in this year

For each of these strategies a version choosing the action probabilistically and deterministically from the highest probability were used, resulting in a total of 6 strategies in the teams. The opponents were a Nash equilibrium agent, as well as an exploitive agent based on Monte-Carlo simulation. Although the majority vote approach was able to outperform the individual team members, the UCB1 method performed slighty better on average. Against the exploitive agent neither of the teams were able to outperform one of its members on his own.

Butolen and Zorman also successfully experimented with teams of poker agents in 2012 [BZ12]. In contrast to most other approaches they asked the opinion of all their team agents to current situations. To decide which action should be played, they mapped the actions to the discrete values 1 (fold), 2 (call), 3 (raise). The sum of these individual decisions $A(i)$ was then divided by the number of agents in the team n:

$$x = \frac{\sum_{i=0}^{n} A(i)}{n}$$

Whichever discrete value had the closest euclidean distance to x was then played by the team agent. Using this setup Butolen and Zorman were able to outperform all of agents used in the team on their own in a heads-up match with the whole team [BZ12]. They also experimented with other decision prodecures, such as a majority vote, but these were not able to defeat the described procedure.

2.3 Playing Poker with an Ensemble

There are multiple ways to play poker with a team of strategies. Investigations of these approaches mostly focused on only asking one strategy for every decision to be made [Joh07, RW11b]. These approaches were often able to match or outperform non-team agents and their results are described in

more detail in the Section “Related Work” (Section 2.2). Using such a technique leads to an exploration vs. exploitation problem. The agent has to decide whether he should explore using another strategy in the hope to find a more effective one, or if he should settle with the current one and not loose money by trying other ones. The goal of the agent should be to minimize the exploration of the available strategies so that the winnings can be maximized. The commonly used UCB1 algorithm solves this problem by achieving an average regret which approaches 0 [ACBF02].

Another approach is to ask every strategy available to the team for its action for every decision. This way the exploration can potentially be reduced, since the agent has complete knowledge what a strategy would have played in a particular situation. Therefore it could judge whether another strategy might have made a better decision. If the agent plays a hand until the showdown, only to loose, another strategy which would have folded earlier might have been the better choice.

The critical part of this approach is, to gain as much information as possible out of the decisions of other available strategies. If no usable knowledge is extracted from them, this comes down to the first approach, where the agent explores every strategy in isolation. This thesis will focus on evaluating this new approach, trying to extract meaningful information out of the additionally collected data.

3 Software & Implementation

3.1 TUD Poker Framework

The "TUD Poker Framework" is a Java framework which supports the development of poker agents. It was developed as part of a bachelor thesis in 2010 [Zop10]. It is compatible with the protocol defined by the "University of Alberta" [alb10], which is the most commonly used protocol for client-server communication during a poker game. It is also the protocol used in the "Annual Computer Poker Competition" (ACPC) [acp12a]. The framework provides a server implementation as well as a common interface for agents to implement. It also contains tools for easy access to the current gamestate and evaluation techniques.

In order to implement a framework for ensemble agents on top of this one, a few changes were necessary:

- Support for arbitrary ports for server and client
- Support for client resets once a new game starts

3.1.1 Arbitrary Port Support

Arbitrary port support was framework internally achieved by adding additional logic to the internal message senders and receivers in the package `pokertud.message.serverclient`:

- `MessageClient`
  Abstract base class for listening to and receiving messages from the server
- `ClientMessageClient`
  Implementation of `MessageClient` for a poker agent
- `ServerMessageClient`
  Implementation of `MessageClient` for a server
- `MessageServer`
  Creates `MessageServerSocket` to listen for new connections
- `MessageServerSocket`
  Accepts connections from clients

In order not to break the existing API, ommiting the port will use the standard port from previous versions. On the server side, these are all the changes needed to create a custom server on a different port. In order to enable arbitrary clients using the framework, to connect to servers on a different port, a few more changes are necessary. The class `TUDPokerClient` in package `pokertud.clients.framework` is used as the communication tool agents and initializes the `ClientMessageClient` mentioned above. `ClientRunner` in the same package is the entry point for every client using the framework which parses the arguments and bootstraps everything. `TUDPokerClient` was extended to initialize the `ClientMessageClient` with a custom port. `ClientRunners` argument parsing was largely rewritten to enable custom ports, but preserving backward compatibility, so that startup scripts didn't need to be modified.

3.1.2 Client Reset

When simulating multiple games or simulating the same game with different positions, the poker agents need to "wipe" their memory occasionally, such as opponent modelling, so that the achieved results in the new games are still correct. The framework didn't have direct support for such a command. Agents needed to detect the beginning of a new game by themselves, using meta information from the game state. To enforce such a thing and not alter the on-the-wire communication protocol, the detection of a new game was integrated on the client side in the mentioned `TUDPokerClient` class. When the class detects the beginning of a new game, it notifies the agent with a new reset command. The agent can then take the appropriate actions. Agents which support this, can now implement a new base class `PokerClientWithReset`, which forces it to override a new method, which is called once such a command is received.

3.2 Poker Bots

The evaluation of the various parts will rely on a few types of poker agents which will be used to demonstrate certain common situations which might occur during a poker game. They will be described next.

3.2.1 Requirements of Poker Bots

Theoretically all poker bots compatible with the PokerTUD framework can also be used in an ensemble with the implementation presented in this thesis. A restriction is that they must not assume that the action they sent to the ensemble server, will be played in the game. Because the ensemble server asks every team member for a decision in all situations, the action from all but one won't be used. If a bot models his own behavior or saves his actions for future considerations, he must wait for the next gamestate to see which action was really chosen. In practice this didn't turn out to be a restriction, because all tested bots didn't save their own actions.

3.2.2 All-In Equity Bot

The *all-in equity bot* is a mathematically fair bot. He uses a metric called *all-in equity* as a metric to decide which action to take. It describes the percentage at which the player can win or tie against its opponents given his hand cards. Unknown cards like the hand cards of opponents or not yet played community cards are chosen randomly. In order to sample a good amount of possibilities for the unknown cards, multiple iterations of random cards are used. For each iteration the outcome is checked and saved as win, tie or loss. The resulting all-in equity e is then calculated as follows:

$$e = \frac{w + \frac{t}{n_{tj}}}{i}$$

w denotes the number of wins in the i iterations and t the number of ties. n_{tj} represents the number of players which tied in iteration j.

The result e is then used to decide which action a to take:

$$a = \begin{cases} \text{raise} & e > \frac{1}{m} \\ \text{call} & e > \frac{p}{c} \\ \text{fold} & \text{else} \end{cases}$$

In this equation m denotes the number of players in the game, p represents the size of the current pot and c is the investment the player has to make to stay in the round. This means the player will invest more money if he has above-average chances of winning. If the expected outcome is greater than the investment necessary to stay in the round, he will call and otherwise fold.

3.2.3 Exploitable All-In Equity Bot

To demonstrate situations where opponents are exploitable, the *all-in equity bot* is modified to be exploitable in a certain situation. Whenever someone raises before the *exploitable all-in equity bot*, he will fold, leading to an immediate win for the exploiting player, when no other opponents are in the game.

3.2.4 Exploiting All-In Equity Bot

In order to demonstrate expoitable situations effectively, a bot which exploits this exact scenario is necessary. Existing accessible bots which used opponent modeling were expected to detect the flaw reliable and be able to exploit it. After some experiments the results were not as good as expected because of which a custom exploiting bot was implemented. The *exploiting all-in equity bot* keeps track of the reacting actions of all opponents and once he detects that an opponent starts folding reliably after a certain action, starts exploiting it. The exploitation is achieved by letting the exploitable opponent play and just calling his actions until the river is reached, to make the pot as big as possible. Then on the river the exploiting action is played to make the opponent fold. In situations where the bot has not detected a exploitable situation he plays just like the *all-in equity bot*.

3.2.5 Opponent-Aware Exploiting All-In Equity Bot

The *exploiting all-in equity bot* has the flaw that he starts exploiting even if there are still opponents who are not exploitable. This can be fixed by checking the exploitation scenario for every opponent and results in the *opponent-aware exploiting all-in equity bot*. He will be used to demonstrate that an ensemble can adapt to team members which are only effective against certain opponents.

3.2.6 Bad Exploiting All-In Equity Bot

Situations where a good team member is suddenly getting exploited are also imaginable. This will be demonstrated by using an *exploiting all-in equity bot* which at some point will start to choose bad actions. These decisions have the following characteristic:

- preflop: 40% fold, 60% call
- flop: 60% fold, 40% call
- turn: 80% fold, 20% call
- river: 100% fold

This characteristic accomplishes that the bot reaches all streets and looses his investments on all streets, not just e.g. the river.

3.2.7 Complexer bots

For a final evaluation some more complex bots which go beyond just calculating the all-in equity were used. The following chapters will shortly introduce them to get a sense of them without going into the details.

OwnBot

OwnBot is a player which was developed during a course at *Technische Universität Darmstadt* in 2011 and was submitted to the AAAI poker competition in 2011. Whenever he finds himself in a heads-up (2-player) situation because all other opponents folded, it uses a nash equilibirium strategy. This strategy is designed to be unlikely to loose, but also not win much. Otherwise the decision process involves modeling of the opponent behavior and monte carlo simulations. The opponent models contain information such as their aggressiveness/passiveness or behavior during blind steals.

When playing preflop the bot calculates the all-in-equity of his cards and retrieves a triple containing the probability of each action. This triple is then varied to not be predictable with support of the opponent models. This leaves the characteristics of the triple intact.

In postflop streets monte carlo simulation is used. This means the situation is simulated with different actions and unknown cards will be filled randomly. Opponent actions are simulated using their respective models. This will be done for all three possible actions in multiple iterations. The action with the best average outcome for the player will then be chosen. To reduce the possible simulations a trained neural network is used to predict the hands an opponent might have based on his behavior. This way the hand cards to be simulated can be significantly reduced.

Akuma

Akuma is an internal bot at *Technische Universität Darmstadt* which was developed in 2009. It uses monte carlo simulation just as *OwnBot* with opponent models containing counters for each action in each street depending on how many bets need to be called. The opponent model is then able to return an action triple according to what has been observed of the opponent in similar situations. In Preflop situations he uses a method like the all-in equity to decide what action to play. It also includes the expected benefit he thinks he is going to achieve, based on the likelihood to win. The bot was able to compete with *OwnBot* in internal evaluations. This means he can achieve similar performance and was therefore chosen as an adversary for *OwnBot*.

3.2.8 Other Bots

There were also experiments with an *exploiting all-in equity bot* which raises not only on river, but also on earlier streets, in cases where the *all-in equity bot* would raise. But this bot didn't cause all of the varying situations, where estimations are needed when calculating the outcome of a team member. This is the reason, this variation of an *exploiting all-in equity bot* was not used further. Some other trivial bots which are going to be used, will be shortly introduced when they are necessary.

3.3 Ensemble Server

The following chapter will describe the implementation of the ensemble server and how custom ensemble agents can be implemented using the framework. First a quick overview of the relevant packages in the implementation will be given, accompanied by a class diagramm (Figure 3.1). This will be followed by a more detailed explanation of the individual components, which are:

- *Ensemble server*
  The ensemble server provides the minimal functionality needed to connect to a real poker server and accepting connections from other poker agents using the PokerTUD framework.
- *Ensemble decision engine*
  The actual decisions which ensemble member should be used in a certain situation is delegated to an ensemble decision engine by the ensemble server. This allows changes to the decision process without changing the underlying server, whose tasks remain the same.
- *Startup*
  The framework provides a common startup behavior, with the ability to automatically launch ensemble members in separate processes. This allows the start of a whole team by just launching the ensemble server.
- *Gathering statistics*
  In order to make informed decisions on which ensemble member should be used, data about the performance of all members needs to be accessible. For this purpose, a common interface to retrieve information out of previous saved round is provided.

3.3.1 Package Overview

- `pokertud.clients.ensemble`
  Holds implementations of ensemble decision engines evaluated, which were implemented for this thesis.
- `pokertud.ensemble`
  Contains a default abstract implementation of an ensemble decision engine, taking care of common tasks, which will be described later (Section 3.3.3), but delegating the actual decisions to its subclasses.
- `pokertud.ensemble.init`
  Includes interfaces and default implementations to initialize an ensemble team in the first rounds, e.g. such that every team member gets to play a few situations.
- `pokertud.ensemble.outcome`
  Holds implementations to calculate estimations of the earnings of the team members whether they played or not.
- `pokertud.ensemble.server`
  Contains the implementation of the ensemble server and interfaces for the interactions with it.
- `pokertud.ensemble.startup`
  Embodies convenience methods to start an ensemble server and possibly team members.
- `pokertud.ensemble.statistics`
  Holds the statistics component which provides access to previously played rounds.

3.3.2 Ensemble Server

The main part of the ensemble server is realized in the class `EnsembleServer` in package `pokertud.ensemble.server`. It implements the `IServer` interface from the PokerTUD framework to act as a traditional poker server for members of the ensemble. It also extends the `PokerClientWithReset`

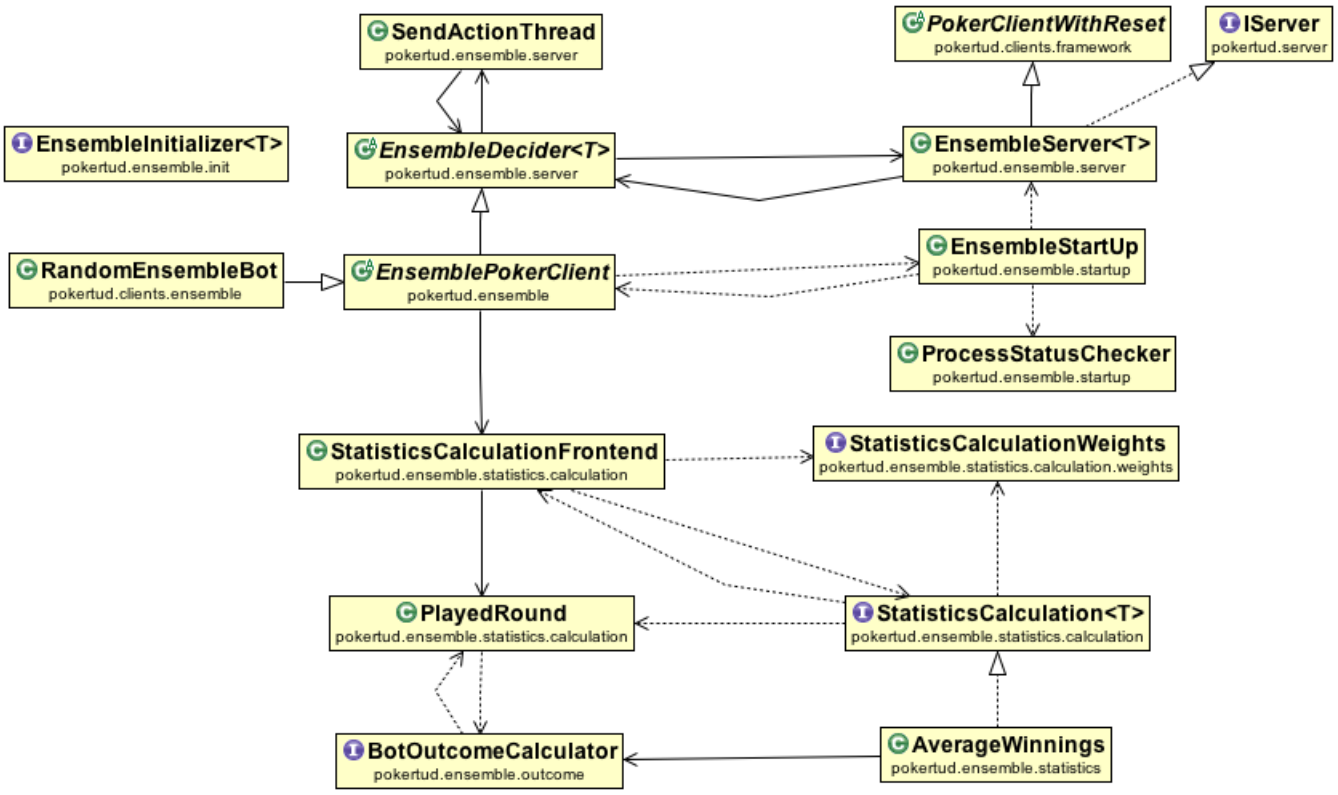

Figure 3.1: Class diagram of ensemble server framework

class, which was introduced in "TUD Poker Framework" (Section 3.1) as an extension to the base class `PokerClient` used for all poker agents. The ensemble server is therefore a server to the ensemble members, as well as an agent to the actual poker server. Using this approach has the advantage of making all poker bots already developed for this protocol automatically compatible with the ensemble approach. The disadvantage is that the ensemble has no further information about the members, but their choices and performance throughout one game, because the protocol only transports their one action to the server. This means the ensemble gets one action from each member for each decision and doesn't know how confident the member was in making this decision. It could be interesting for the ensemble to receive a triple from each member with the probabilities of each possible action, which could provide further information for the decision engine. This extended approach won't be used or evaluated in this thesis, since it would require to alter the server-client communication protocol.

`EnsembleServer` provides the minimal functionality to establish the communication between the poker server and the team members. On initialization it starts a server using the messaging classes from the PokerTUD framework and then needs to be connected to the real poker server. The server only reacts to `AccountLoginMessages`, `ClientDisconnectedMessages` and `ActionMessages` and takes the appropriate actions by updating its internal structures and notifying the deciding agent described below. When it receives a new game state from the poker server, the deciding agent also gets notified and the game state is relayed to the connected team members. When it is the turn of the ensemble server to make a decision, the members will recognize this, because they receive the unaltered game states and send their decisions to the ensemble server, since the ensemble is the server they are connected to.

The ensemble server relies on the abstract class `EnsembleDecider` in the same package to make the decision which actions are played. The class provides methods which are called before a game starts or for events during a round in the game. To not get confused with actions of team members received out of order, the `EnsembleDecider` always waits until all actions of the connected members are received. A dedicated thread `SendActionThread` is started, so that the threads of the ensemble server used for

receiving messages are not stalled. This thread holds onto decisions made by `EnsembleDecider` until all members sent their actions.

3.3.3 Deciding Agent

Whereas `EnsembleDecider` is a low level abstract class which provides a common interface the `EnsembleServer` uses, `EnsemblePokerClient` is the equivalent of the `PokerClient` class in the PokerTUD framework. It provides common functionality for implementations, to make the creation of deciding agents easy. It contains four abstract methods which must be implemented:

```
public void newGame();
```

This callback is called whenever a new game starts, so that the agent can reset his memory or do other initialization work.

```
public String chooseBot();
```

This method will be called whenever a new decision needs to be send to the poker server. The implementation needs to return the unique name of the team member it chose. The `EnsemblePokerClient` implementation then takes care of sending the corresponding action to the server.

```
public void botAction(String botName, Action action);
```

This callback is used whenever a team member sent its action decision to the ensemble server, so that the agent implementation can use it to gain information for future decisions.

```
public void finished();
```

This method is triggered when a round is over and the winnings for all players are known. With it the agent can gather performance information.

Additionally `EnsemblePokerClient` provides methods to access actions or decisions made during the round and saves previous rounds once they are finished for later access. This can be used to evaluate the performance of team members in the immediate past or over the whole game. The access to previous rounds is described in detail in "Gathering Statistics" (Section 3.3.5).

Generally it is necessary to let team members play a few rounds at the start of a game, in order to gain some initial information about them, to be able to judge which member would be beneficial. For this purpose the package `pokertud.ensemble.init` contains the interface `EnsembleInitializer`, which is common template for these kind of tasks. An useful implementation of this interface is provided in `EnsembleStreetInitializer`, which lets every team member play a certain number of times on each street, before switching to the real decision strategy.

3.3.4 Startup

To provide a common starting method for all implementations just like the PokerTUD framework does (thorugh the class `ClientRunner`), the class `EnsembleStartUp` exists. It is used to start the ensemble server and connect it to the poker server, but can additionally be used to start some team members. This allows a whole team to be started by just starting the ensemble server. The startup routine parses a host and port for the poker server from the arguments, as well as an port for the ensemble server and initializes everything. Team members can be started using two methods:

- *From the class path*
  When the classes required for the team member are all in the class path of the ensemble server and support the PokerTUD framework `ClientRunner` parameters, it can launch a separate java process with the same class path launching the member. This way only the fully-qualified class name of the new member is needed.

- *Using a start script*
  When the team member is not in the class path or is not even a java implementation, it can be started using a startup script which takes the arguments just like the extended PokerTUD framework `ClientRunner`. This way `EnsembleStartUp` needs a working directory for the new process, as well as the launch script. These values are read from a `.properties` file for all members to be launched. The necessary arguments will be automatically given to the subprocess in the format expected by the PokerTUD framework.

These methods don't need to be used, the user is free to start the team members on his own and connect them to the ensemble server. If one of the methods is used, a separate thread `ProcessStatusChecker` is launched, which checks the standard output and error output of the launched subprocesses. Errors are redirected to the output of the parent process. When the thread is instructed to terminate, it will also kill the subprocesses.

3.3.5 Gathering Statistics

`EnsemblePokerClient` saves finished rounds using `StatisticsCalculationFrontend` from package `pokertud.ensemble.statistics.calculation`. The class saves the round in a special data structure represented by the class `PlayedRound` which saves the showdown game state of a round. This class also holds all actions of all team members of the round and the decisions made by the deciding agent. `StatisticsCalculationFrontend` can be used to iterate over these saved rounds and extract the information needed, then eventually combining it over all interesting rounds. The rounds can be filtered by the following criterias:

- *time*
  only a given subset of n rounds starting with round s shall be considered

- *position*
  only rounds where the player was in a certain position (small blind/button/...) shall be used

- *team member chosen*
  only rounds where a given team member was chosen shall be used

- *opponents faced*
  only rounds in which the ensemble faces one or multiple given opponents will be used

- *action filter*
  rounds are filtered for situations where the team made identical decisions. For example: team member 1 called, team member 2 folded and all other members are irrelevant.

The interface `StatisticsCalculation` is used to process the filtered rounds. Implementations are initialized, to then be called iteratively for each selected round to process it. After that the combined result is retrieved from the instance. The processing implementation can also apply additional filtering, based on the following arguments:

- *team member*
  only process the round under the aspect of a given team member

- *street*
  only process the data for the given street (preflop/flop/...)
- *weighting*
  a weight in the range of $[0, 1]$ that should be applied to the processed value of the round. These are calculated using the common interface `StatisticsCalculationWeights` located in the package `pokertud.ensemble.statistics.calculation.weights` and will be further discussed in Section 5.1.

The extraction of some common information out of the game states is supported by `PlayedRound` through outsourced calculators. Outcome calculation for a given team member regardless of whether his actions were chosen is calculated by `BotOutcomeCalculator` implementations. Various provided implementations of this interface will be introduced in "Earnings Calculation" (Section 5.2) and "Evaluation of Earnings Calculation" (Section 6.1).

3.3.6 Logs

To make the evaluation of ensembles possible, critical parts produce log files which can be analyzed during or after a game. There are three types of logs for different purposes:

- *Timings log*
  The `EnsembleServer` by default creates a timing log for each game. Its purpose is to evaluate time-related issues. A common application of this would be to check the amount of time connected team members need to decide which actions they would want to take.
- *Ensemble log*
  The `EnsemblePokerClient` can create a log containing information on final decisions being made by the decision engine. It can be used to evaluate whether the decision engine works as expected by checking if the expected team members are being chosen.
- *Statistics log*
  Decision engines shipped with the framework using `StatisticsCalculationFrontend` for statistic calculations additionally create a log, which contains the raw values calculated and the final resulting rating which they calculate. It can also contain debug information such as the current estimate and the current values of filters used for the statistics calculation.

Timings log

The timings log is an event based log. Each line contains an event whose type is identified by the first capitalized string. Generally software working with these log files can only assume that events for a given round and street occur before the next round or street. The order of events during a street should not be considered deterministic. This does not include street containing multiple decisions. In such a case the events for one decision occur before the next decision. The following three events can occur:

- `DECISION`
  Describes the amount of time it took for the decision engine to make its decision which team member should be allowed to play. It also contains the round number and the street in which the event occured: `DECISION 0 PREFLOP 4msec`
- `BOT`
  Contains the information how long a team member took to sent its action to the ensemble server. The event contains information about the round number, the street and the name of the team member: `BOT 0 PREFLOP AllInEqPlayer 2572msec`

- WAIT
  Logs the amount of time the ensemble server waited between two decisions for opponents and the poker server. This event also contains the information about the round number and street at the start of the event: `WAIT 0 PREFLOP 58msec`

Ensemble log

The syntax for the ensemble log is inspired by the server logs produced by the PokerTUD framework. Each line contains all the information available for one round. These lines contain multiple parts separated by `:`. The content of the individual parts in their order is as follows:

- The first part contains the names of all the team members separated by `|`.
- This is followed by the round number starting at 0.
- After that come the actions the team members chose for each decision. The individual streets are separated by `/`. Each street contains the actions chosen in the order the members occur in the first part for each decision.
- The penultimate part saves the cards of the round. The format is exactly the same as in the server logs, so that the parser can be reused. This means the hand cards are duplicated for each team member, separated by `|`. These are then followed by the community cards whereas each street is separated by `/`
- The final part embodies the ratings which influenced the decisions. Each decision is separated by `/`. For each one the ratings of the members are separated by `|` and occur in the order given by the first part. The rating of the chosen team member is followed by the string `[C]`

A complete example of one line in the log looks like the following:

`Mem1|Mem2:0:cccc/cf/fc/:Kd9h|Kd9h/Ad8d6s/Ts/:1.0[C]|0.0/0.0|1.0[C]/1.0[C]|0.0/1.0[C]|0.0`

This means `Mem1` chose the actions cc during preflop, c and f during flop and turn in the first round. His first action during preflop, as well as the flop and river were actually used. `Mem2` was therefore only used for the second preflop actionand choose ccfc during the hand.

Statistics log

The statistics log have a very similar syntax as the ensemble logs, but whereas the ensemble logs focus on an overview of the rounds, these logs focus on statistics values. A major difference is that each line now only contains the information for one decision. Each line of a log consists of the following parts separated by `::`

- The first part contains the names of all the team members separated by `|`.
- This is followed by the round number starting at 0.
- The next part describes the street this line is for. The values are within $[0, 3]$ for preflop to river.
- The penultimate part saves the raw statistics values. For each member all the values are separated by `/`. The members are separated by `|` and occur in the order defined in the first part.
- The final part embodies the resulting ratings for each team member separated by `|` in the order defined by the first part.

A complete example of one line in the log looks like the following:

```
Mem1|Mem2:15:0:−4.761904761/−5.833333333|−5.238095238/−6.25:0.559244197|0.440755802
```

This means there were two rating methods and `Mem1` had the values −4.761904761 and −5.833333333 during preflop of round 16. These two values were combined to a final rating of 0.559244197. The final rating is a percentage, which is why the negative sign disappears.

4 Ensemble Strategies

Strategies to choose a team member for upcoming decisions are the crucial component of every poker ensemble. They directly influence the performance of the team. Using an underperforming strategy will probably lead to poor team performance, almost independent of its members.

The ensemble strategies can be divided into two big categories: *context-free* and *context-aware* strategies. In the following both of these categories will be defined and some representatives of them will be introduced and discussed.

4.1 Context-free Strategies

Context-free strategies settle with only using information of the current round to make a decision which team member should be allowed to play the current situation. Therefore they do not need a memory of previous decisions or performances of the team members or itself. The following strategies are representatives of this category:

4.1.1 Random Decision

The *random decision strategy* is arguably the simplest one. The deciding agent uses a random number generator to retrieve a connected team member to play the decision. Random number generators are expected to produce uniformly distributed random numbers, so that the connected team members should be uniformly chosen.

4.1.2 Majority Vote

Majority voting is another context-free strategy. The deciding agent waits until the decisions of all team members have arrived and then sorts the actions by the number of times they have been chosen. The action which was chosen the most is then send to the poker server.

This strategy can only be useful when more than 2 team members have connected, because otherwise it is just a somewhat random choice between the two members. Ties in the voting process cannot be complety eliminated, so they need special treatment. In order to reduce the complexity most of the time, the action which was sorted higher by the sorting algorithm is the one which gets chosen. This strategy was used among others by Rubin and Watson [RW11a] in their evaluation of poker team approaches, as discussed in “Related Work” (Section 2.2).

4.1.3 Fixed Transition

A *fixed transition strategy* is a useful tool for evaluation purposes. It uses a predefined set of team members and corresponding number of rounds, the member should play. The team members are then selected in the order they were specified. The following configuration

```
( (Callbot, 1000), (Raisebot, 1000) )
```

would use the team member “Callbot” for the first 1000 rounds, and then choose “Raisebot” for the following 1000. If the game lasts longer the strategy can be configured to start from the beginning once the end of the configuration has been reached. In the context of the example this would mean “Callbot” would be chosen again after the initial 2000 rounds, starting another cycle.

4.2 Context-aware Strategies

Context-aware strategies additionally use information of previously played rounds in the game to support their decision process. This information is then used to rate the team members performance, which influences the decision which team member is going to be selected.

4.2.1 UCB1

UCB1 is the decision strategy used by Johanson [Joh07] and Rubin [RW11a] in their poker ensemble evaluations, discussed in "Related Work" (Section 2.2). UCB1 tries to minimize the regret while acting in the multi-armed bandit problem [ACBF02]. The regret is defined as the difference between what could potentially have been won by always choosing the optimal strategy and what was actually earned. The strategy then acts as follows:

- In the beginning: select each team member once
- Afterwards: Select the team member i which maximizes the following:
  $argmax_i \left(\bar{x}_i + \sqrt{\frac{2ln(n)}{n_i}} \right)$
  $\bar{x}_i$ means the average earnings by team member i
  n is the total number of decisions played so far
  n_i is the number of decisions team member i was selected

With this strategy UCB1 is able to let the average regret approach 0, without any further knowledge of the underlying team members [ACBF02].

4.2.2 Game Statistics

These types of strategies extract information out of past rounds, by saving the whole state of the round and of the deciding agent at the end of each round. These saved rounds can then by iteratively accessed and every kind of information can be extracted. A high level overview of this was given in "Gathering Statistics" (Section 3.3.5). The framework provides a few default implementations for such information extractors:

- `AverageWinnings`
  The average earnings of a particular team member
- `AveragePotWinningsPercentage`
  The average percentage of the pot a particular ensemble member (would have) won. The potential pot is reduced by the amount the player already invested, because that is money he does not win. A loss is represented by 0%
- `AverageWinningsStandardDeviation`
  The standard deviation of the average earnings of a team member

All of these extractors can be fed with a filtered round list by the various criterias described in "Gathering Statistics" (Section 3.3.5). Once one or more of these extractors are finished, the resulting values are normalized into the interval $[0, 1]$ for every extractor over all calculated team members and are added together. This leads to a rating for each ensemble member in the interval $[0, n]$ where n is the number of extractors used. Optionally the individual extractors could also be weighted, if e.g. one of them would be considered more important than the others.

This thesis will focus on rating the team members using their estimated outcome using the `AverageWinnings` extraction. Using the estimated earnings is the obvious first choice, because it is

the metric one wishes to maximize, since a higher outcome results in a better choice to win against all opponents. `AveragePotWinningsPercentage` is similar in most situations to `AverageWinnings`, but is already normalized and has a defined upper limit of 1.0, which can be used to express potential (that a better strategy might exist). `AverageWinningsStandardDeviation` was found to distort the ratings rather than making them better, because its values are very unstable in the beginning and need a long time to stabilize. When they stabilized a lot of bad choices were already made. Leaving `AverageWinningsStandardDeviation` out of the calculated ratings often resulted in a better outcome for the ensemble. A few experiments using information extractors different from the outcome, such as the strength of the hand cards, were made. But in order to generally detect exploitable situations or situations were one gets exploited, they were not further investigated. The outcome was a sufficient metric to detect these situations and including a rating most team members in some way already use did not improve this detection.

5 Game Statistics

To go into more detail regarding the weighting methods and filtering by team members and streets, first the provided weighting methods are described. Afterwards the calculation of the earnings per team member is discussed. The last part of the chapter will then focus on the different decision strategies which can be used to choose a team member.

5.1 Weighting Methods

- *uniform weighting*
  Every round weights exactly the same and gets a weight of 1.0, as if no weighting is being used.
- *half-time*
  The weighting method is initialized with a round number h, called *half-time time*, at which the weight will have been declined to 0.5. The weight w_i of a round i is then calculated as follows:
  $w_i = 2^{\left(-\frac{m-i}{h}\right)}$
  In this equation, m represents the total number of rounds to be weighted.

5.2 Earnings Calculation

The calculation of the earnings of a given team member, whether chosen to play or not, is the crucial part of the decision strategy. It is the part where knowledge about the alternative strategies in a situation should be gained, to reduce the exploration costs of the alternative strategies. The following methods were developed and evaluated:

5.2.1 Simple

This calculation method is deliberately kept rather simple to get started, but therefore has issues gaining a lot of new knowledge. The following pseudo code represents the calculation rules for the earnings of a team member i:

```
memberReward = 0
perDecisionReward = ensembleOutcome / numberDecisions
for every decision in round:
        chosenMember = getChosenTeamMember(decision)
        if i == chosenMember:
                memberReward += perDecisionReward
        else:
                playedAction   = getActionOfTeamMember(decision, chosenMember)
                actionOfMember = getActionOfTeamMember(decision, i)
                if playedAction == actionOfMember:
                        memberReward += perDecisionReward
                else if playedAction != FOLD  && actionOfMember == FOLD:
                        memberReward += -perDecisionReward
                else if playedAction == CALL  && actionOfMember == RAISE:
                        memberReward += perDecisionReward
                else if playedAction == RAISE && actionOfMember == CALL:
```

Street	Member 1	Member 2	Chosen	Earnings Member 1	Earnings Member 2
preflop	call	fold	Member 1	10	-10
preflop	call	call	Member 2	20	0
flop	raise	call	Member 1	30	10
turn	call	raise	Member 2	40	20
river	raise	raise	Member 1	50	30
river	fold	raise	Member 2	40	40

Table 5.1: Example calculation of simple earnings calculation

```
                memberReward += perDecisionReward
return memberReward
```

In natural language the calculation can be described as follows: The total amount won in a round is divided by the number of decisions in this round, to get a measure how much each decision was worth. The team member i then gets rewarded or penalized using this number for each decision he made or would have made in the round. Depending on the outcome of the round this number might be negative. In the following the number will be called "per decision reward", but it might be a negative reward. If i was chosen for a decision in the round, he immediately gets the per decision reward. Would i have the same action as the chosen member, he also gets rewarded with the per decision reward. In case i would have folded instead of staying in the round, the inverted per decision reward is added to his score. This leads to a positive reward in case the betted money was lost. The only cases where no reward is given, are the ones in which the ensemble team folded, but the team member i would have stayed in the betting round. In this situation the per decision reward is always negative and it cannot always be analysed whether the fold was a good decision. This could only be done if all the other players in the game would have played until showdown revealing their cards. Then the cards could be compared and a defintive decision could be made.

To illustrate the calculations being done the following example calculation for one hand is provided in Table 5.1. It is assumed that the outcome for the ensemble after this round was 60 chips. The calculation is shown for the whole round, but could easily only be executed on a subset, such as the flop decisions.

This calculation is problematic since substantial differences in the earnings only occur if one or more team members would have folded. Otherwise, with only call and raise decisions present, this leads to very similar earnings and therefore not distinguing between the team members very well.

5.2.2 Known Fold

The *known fold* earnings calculation enhances the estimation of an early fold by a team member, when the actually used action was not a fold. In this case the loss can be accurately calculated by going over all previously played actions in the round and calculating the investment which was made up until the fold. This investment then overrides all previously accumulated estimations, because the accurate value is known at this point. This can be expressed in pseudo-code as follows:

```
memberReward = 0
perDecisionReward = ensembleOutcome / numberDecisions
for every decision in round:
        chosenMember = getChosenTeamMember(decision)
        if i == chosenMember:
                memberReward += perDecisionReward
        else:
                playedAction   = getActionOfTeamMember(decision, chosenMember)
```

Street	Member 1	Member 2	Chosen	Earnings Member 1	Earnings Member 2
preflop	call	fold	Member 1	10	0
preflop	call	call	Member 2	20	10
flop	raise	call	Member 1	30	20
turn	call	raise	Member 2	40	30
river	raise	raise	Member 1	50	40
river	fold	raise	Member 2	10	50

Table 5.2: Example calculation of known fold earnings calculation

```
        actionOfMember = getActionOfTeamMember(decision, i)
        investment     = getInvestmentUntil(decision)
        if playedAction == actionOfMember:
                memberReward += perDecisionReward
        else if playedAction != FOLD  && actionOfMember == FOLD:
                memberReward = -investment
        else if playedAction == CALL  && actionOfMember == RAISE:
                memberReward += perDecisionReward
        else if playedAction == RAISE && actionOfMember == CALL:
                memberReward += perDecisionReward
return memberReward
```

Just like for the simple method, an example calculation is provided in Table 5.2. The exact same scenario is used. The big blind is 10 chips and each bet is also 10 chips.

This calculation enhances the previous method, by using the actual known value when possible, therefore making the estimation more realistic.

5.2.3 Estimation

To further enhance the cases where the exact impact of a different decision is unknown, the statistics framework is used to estimate the outcome using similar situations. For this purpose a function `getEstimate(street, chosenMember, playedAction, otherMember, actionOfOtherMember)` is introduced. The estimate is calculated on previous rounds where the same ensemble situation was observed, but the other team member was chosen. On these rounds the average winnings are calculated for the whole round. Using the same team member twice in the function call is invalid.

```
memberReward = 0
perDecisionReward = ensembleOutcome / numberDecisions
for every decision in round:
        chosenMember   = getChosenTeamMember(decision)
        playedAction   = getActionOfTeamMember(decision, chosenMember)
        actionOfMember = getActionOfTeamMember(decision, i)
        investment     = getInvestmentUntil(decision)
        estimate       = getEstimate(street, chosenMember, playedAction, i, actionOfMember)
                         / numberDecisions
        if estimate does not exist:
                continue

        if i == chosenMember && exists(estimate):
                memberReward += perDecisionReward
```

Street	Mem1	Mem2	Chosen	Earnings Mem1	Earnings Mem2	Estimate value
preflop	call	fold	Member 1	10	-10	/
preflop	call	call	Member 2	20	0	/
flop	raise	call	Member 1	30	20	120
turn	call	raise	Member 2	30	20	n/a
river	raise	raise	Member 1	40	30	/
river	fold	raise	Member 2	30	40	/

Table 5.3: Example calculation of estimations earnings calculation

```
        else:
                if playedAction == actionOfMember:
                        memberReward += perDecisionReward
                else if playedAction != FOLD  && actionOfMember == FOLD:
                        memberReward += -perDecisionReward
                else if playedAction == CALL  && actionOfMember == RAISE:
                        memberReward += estimate
                else if playedAction == RAISE && actionOfMember == CALL:
                        memberReward += estimate
                else if playedAction == FOLD  && actionOfMember != FOLD:
                        memberReward += estimate
return memberReward
```

Since the estimation is for a whole round, the value is divided by the number of decisions in the round in which the estimation is used, so that the scale is the same as for other rewards in the round. In the case no estimation can be made, no member gets a reward for the decision, so that the difference between the members does not increase just because no good estimation can be made.

Just like for the previous methods, an example calculation is provided in Table 5.3. The exact same scenario is used. The big blind is 10 chips and each bet is also 10 chips.

This method improves some of the shortcomings of the *simple* approach, but also introduces a dependency to the decision strategy which team member gets chosen. If the strategy prefers a member, the estimations for other members might get inaccurate or not exist, because there are not enough rounds available for a good estimation.

5.2.4 Combined

The last earnings calculation method, combines the *known fold* and *estimation* methods into one earnings calculation. This is expressed in pseudo-code as follows:

```
memberReward = 0
perDecisionReward = ensembleOutcome / numberDecisions
for every decision in round:
        chosenMember   = getChosenTeamMember(decision)
        playedAction   = getActionOfTeamMember(decision, chosenMember)
        actionOfMember = getActionOfTeamMember(decision, i)
        estimate       = getEstimate(street, chosenMember, playedAction, i, actionOfMember)
                         / numberDecisions
        if estimate does not exist:
                continue
```

```
        if i == chosenMember && exists(estimate):
                memberReward += perDecisionReward
        else:
                if playedAction == actionOfMember:
                        memberReward += perDecisionReward
                else if playedAction != FOLD  && actionOfMember == FOLD:
                        memberReward += -investment
                else if playedAction == CALL  && actionOfMember == RAISE:
                        memberReward += estimate
                else if playedAction == RAISE && actionOfMember == CALL:
                        memberReward += estimate
                else if playedAction == FOLD  && actionOfMember != FOLD:
                        memberReward += estimate
return memberReward
```

5.2.5 Further Observations

There were additional attempts to weight the reward to set how much each decision is worth, instead of just distributing the earnings of the hand uniformly on all decisions. Unfortunately, this did not improve the estimations, but rather made them less accurate. To weight the individual streets, the investment made in it in relation to the total investment in the street was used. This caused folds to be undervalued in contrast to all other actions, because no investment is being made. Since folds were somewhat common actions, the resulting estimates turned out worse than without weighting. Experimations using the total investment of the hand up to the current street in relation to the total investment were also done, but did not change the behavior of the resulting estimations.

A comprehensive evaluation of these methods will be done in “Evaluation of Earnings Calculation” (Section 6.1).

5.3 Decision Strategies

Besides the ratings of the individual team members the other important aspect of the decision process is turning these ratings into the final decision in favor of a certain team member. Using UCB1 as introduced in earlier chapters doesn’t make much sense, since every team member gets chosen to contribute his action. Additionally every action influences the earning of a team member which doesn’t meet the assumptions UCB1 makes. This chapter will therefore introduce the alternatively developed strategies, which will then be evaluated in Section 6.2.

5.3.1 Max Value

The *max value decision strategy* just choses the highest rating among all team members. If multiple highest ratings exist the first one will be chosen. While this can be an effective strategy, it makes it difficult for other team members to take control, especially in combination with ratings based on estimations as introduced previously. The strategy to choose a team member c among the possible members M can then be expressed as:

$$c = \arg\max_{m \in M} f(m)$$

In this equation $f(x)$ calculates the rating of a team member x.

5.3.2 Max Value with Draw Handling

This strategy improves the previous *max value strategy* in situations where multiple team members obtain the highest rating. In these situations the strategy will choose uniformly among them instead of picking the first member. This enhances some of the disadvantages, but only in draw situations.

5.3.3 Probabilistic

The *probabilistic strategy* translates the individual ratings into probabilities that a certain team member gets chosen.

$$
\begin{aligned}
S &= \sum_{m \in M} f(m) \\
M_{min} &= \min_{m \in M} f(m) \\
M_{max} &= \max_{m \in M} f(m) \\
p(m) &= \begin{cases}
\frac{1}{|M|} & S = 0 \\
\frac{f(m)}{S} & M_{min} > 0 \wedge M_{max} > 0 \\
1 - \frac{f(m)}{S} & M_{min} < 0 \wedge M_{max} < 0 \\
\frac{f(m) + |M_{min}|}{|M_{min}| + |M_{max}|} & \text{else}
\end{cases}
\end{aligned}
$$

Each team member m then is associated with the probability $p(m)$. Since the ratings for the members can be negative, care has to be taken that they are correctly translated into $[0, 1]$. The individual cases of $p(m)$ identify the possible combinations of positive and negative ratings and take the appropriate action.

5.3.4 Probabilistic with Bias

In situations where the team members obtain somewhat similar ratings and therefore similar probalistic values, it can be beneficial to introduce a bias. *Probabilistic with bias* increases the values of probabilities greater than 0.5 and decreases smaller ones appropriately. To achieve this a transformation using the sigmoid function is introduced. In the following $p_{org}(m)$ describes the probability of the original *probabilistic strategy* for team member m.

$$
p(m) = \frac{1}{1 + e^{-S(p_{org}(m) - 0.5)}}
$$

The parameter S can be used to control the magnitude of the increases and decreases. A couple of values will be evaluated in “Evaluation of Decision Strategies” (Section 6.2).

5.3.5 Max Value with Action Groups

The previous methods work for an arbritrary number of team members, but might not lead to the desired results when more than two are involved. In such cases it is often more desirable to vote on the actions rather than members. If two members would call and the last one wants to fold it can be beneficial to listen to the two agreeing members rather than the other one, even if he has a slightly better rating. This strategy updates the *max value strategy* such that members are grouped by the actions they chose. Each action can then be assigned a rating equal to the sum of the ratings of the corresponding team members. At the end the action with the highest rating gets chosen.

$$
c = \arg\max_{a} \sum_{m \in A(a)} f(m)
$$

In this equation $A(x)$ for an action x denotes the set of team members which chose this action. The strategy can obviously also be distinguished by the handling of draws between the action ratings, just like before.

5.3.6 Probabilistic with Action Groups

The *probabilistic strategy* can be updated in the same way as the *max value strategy* such that voting occurs on the actions to be taken, rather than the team members.

$$p_{action}(a) = \sum_{m \in A(a)} p_{member}(m)$$

$p_{member}(m)$ can be the original probability to be manipulated with a bias as in the *probabilistic with bias* strategy. It is important to note that the bias is therefore applied before grouping the ratings.

6 Evaluation

The following chapters will describe the evaluation of the various parts introduced previously. All evaluation games will generally be done in all positional permutations, which are six different games in case of a 3-player game. Not all of the six games will be shown every time to keep the graphs readable. In almost all cases the positional permutations do not change the overall trend of the effect evaluated and therefore one representative will be used to demonstrate the results. In cases where significant variations can be observed, these cases will be shown additionally or discussed.

Overall all games are played with the following settings: The blinds are constant with a value of 5 and 10 chips. A bet during preflop or flop has a value of 10 chips and otherwise 20 chips. The stack size of the players is unlimited. During each street 4 bets are allowed.

6.1 Evaluation of Earnings Calculation

To evaluate the methods for calculating the earnings of a team member, a random choice of 2000 hand and board cards was made and used for all games. This ensured that all methods were compared in the same situations. The evaluation games used the following setup of poker agents:

- *Fixed transition ensemble*
  An ensemble transitioning from the mathematical all-in-equity strategy to the exploitable all-in-equity player (described in Section 3.2.3) after 1000 hands.
- *Fixed transition ensemble*
  Same as above
- *Statistics ensemble*
  An ensemble using the simple earnings calculation (Section 5.2.1) choosing the team member with the maximum average earnings with draw handling (Section 5.3.2). The estimates of the earnings calculation mtehod to be evaluated are calculated separately and logged for later comparison. The team consisted of the mathematical all-in-equity (AllInEqPlayer) and the exploiting all-in-equity (AllInEqPlayerCustomExploiter) player, both described in Section 3.2. The statistics were calculated by filtering previous rounds for each team member in the current street. All previous rounds were uniformly used. The ensemble is initialized by letting every team member play 10-times on each street.

These three players were used in games in all their positional permutations with each game using the same random 2000 hands. The logged estimations were compared to the actual earnings of the two members on their own against the two fixed transition ensembles. All in all to evaluate all three methods 18 ($3 \cdot 6$) matches were played, each containing 2000 hands. The all-in-equity calculations of the bots weren't deterministic, so that their results were cached and reused for all games and players.

The following chapters will describe the performance of the introduced methods invidually, to then conclude with an overview comparing all methods against each other.

6.1.1 Simple

Figure 6.1 plots the estimations of both team members alongside their real winnings. It shows that in the beginning when both members play identical, the estimation is identical as expected. Once the exploiting player starts to play different his estimation remains very accurate, because the real outcome is used for the estimation, since he gets chosen almost exclusively. Due to the lack of differentiating

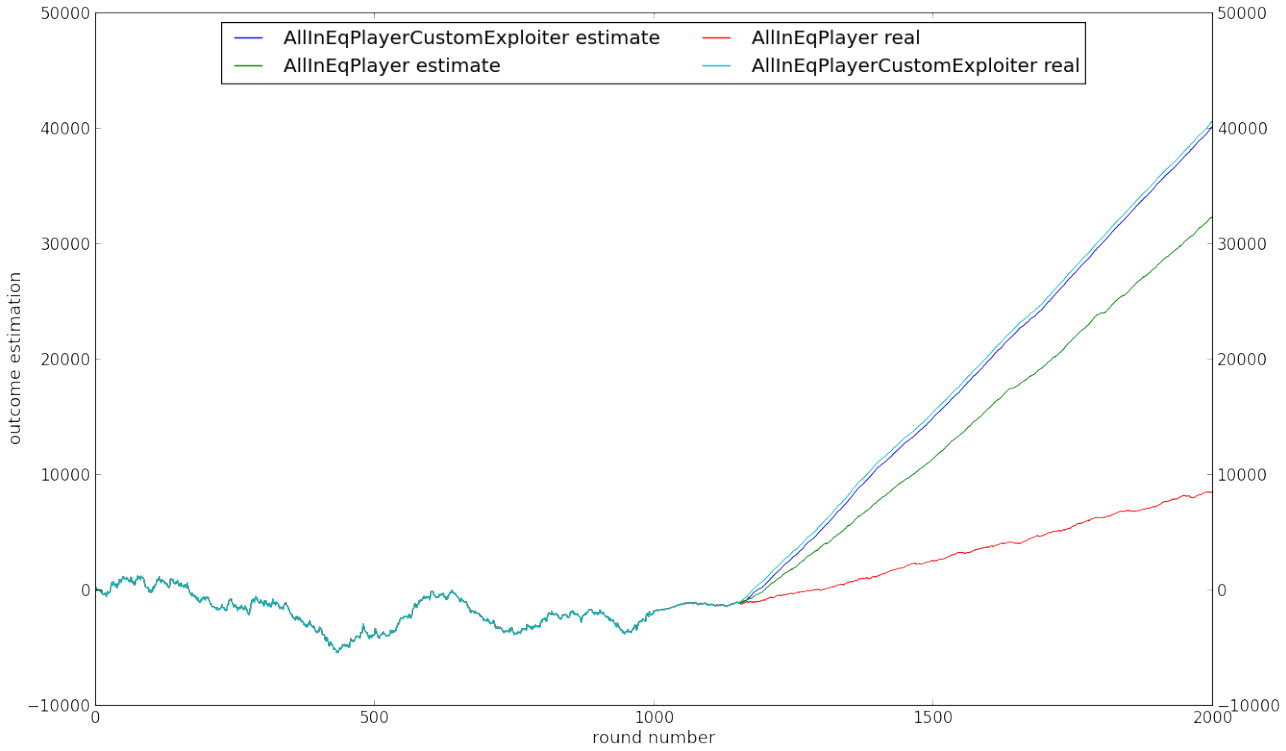

Figure 6.1: Earnings of team members in ensemble using the simple method

between members except when folding, the estimate for the not-playing team member is similar and far from the actual outcome.

6.1.2 Known Fold

When extending the simple method with the known loss when a not chosen team member folds, the estimate for the mathematical all-in-equity player goes down by approximately 9000 (900 big blinds), getting closer to the actual outcome. The absolute difference between the estimation and real outcome still adds up to 16000 (1600 big blinds), as can be seen in Figure 6.2.

6.1.3 Combined with Estimate

In order to further improve the earnings calculation, an estimation based on previous similar rounds can be used to approximate cases such as raising instead of calling. This was combined with the previous methods and resulted in the estimations shown in Figure 6.3. The estimation for the pure mathematical player improved significantly and is near the actual outcome. As a trade-off the estimation for the exploiting team member went down, because decisions are not rewarded, when no estimation for the other member can be made. Removing this restriction would make the calculation much more dependent on the actual decision strategy, since a team member can receive significant rewards when chosen, whereas other members receive no rewards, leaving their earnings unchanged. The overall earnings relation between these two team members is captured rather good by this method, so that no further improvements were made. Figure 6.4 shows that by plotting the benefit of using the exploiting player over the mathematical player using the three estimation methods as well as the real earnings (`CombinedFKE` identifies the combined with estimate method previously evaluated).

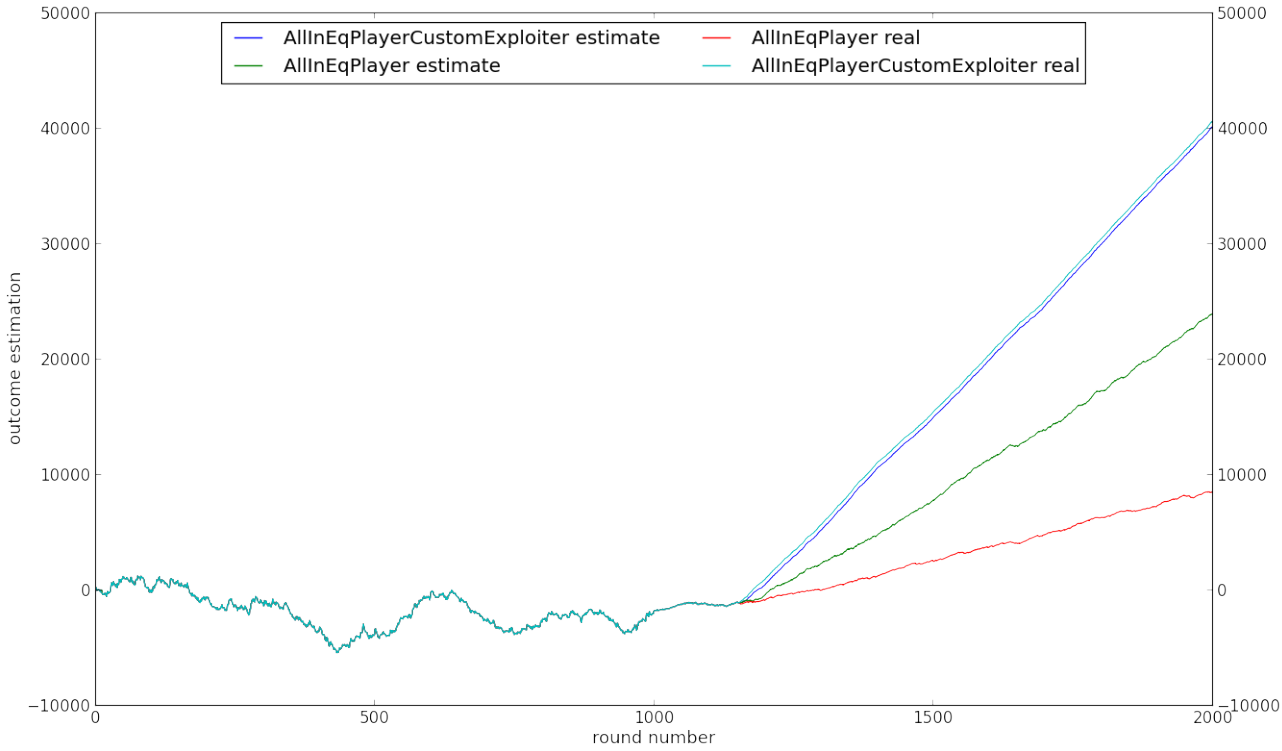

Figure 6.2: Earnings of team members in ensemble using the known fold method

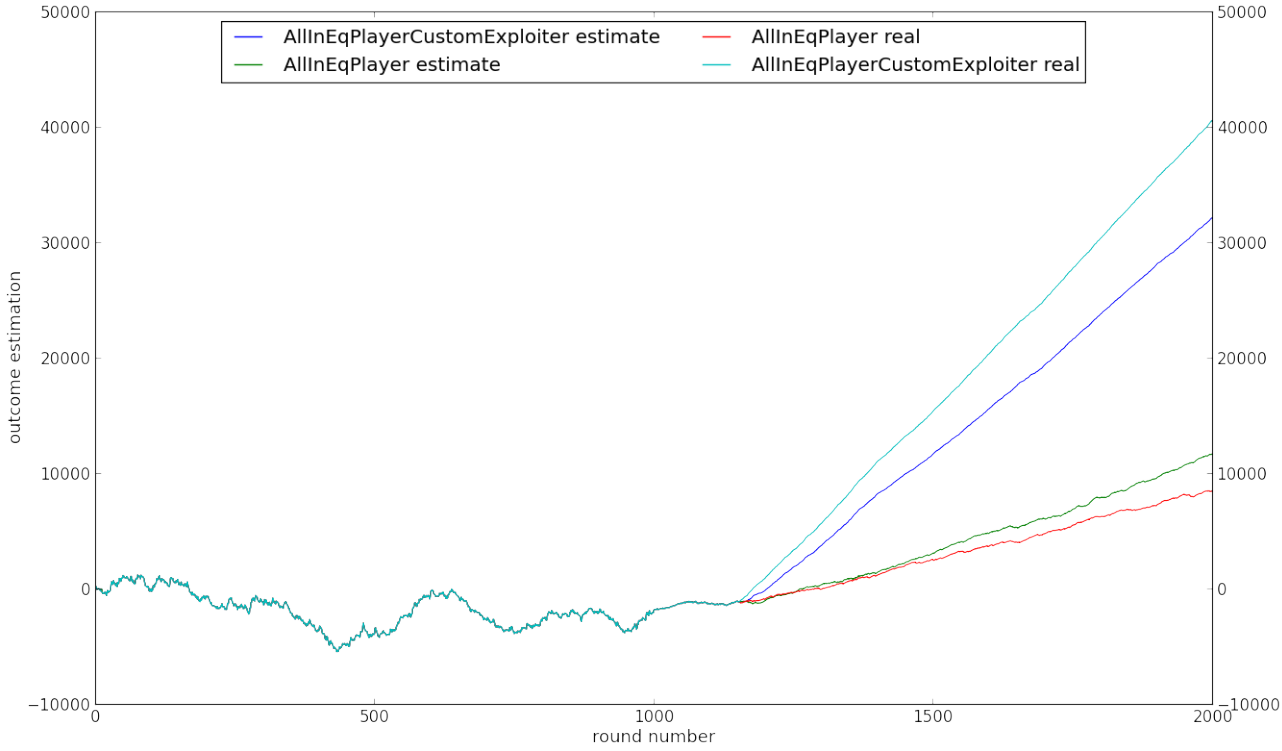

Figure 6.3: Earnings of team members in ensemble using the combined method

6.1.4 Summary

Overall the performance of the individual methods can be summarized and compared in Figure 6.5 (`CombinedFKE` identifies the combined with estimate method previously evaluated). It shows the rate of the calculated estimates divided by the actual earnings achieved. The optimal solution would therefore be a straight line at 1.0. Occasional values of zero are also okay, since they are caused by estimates or real earnings of 0.0. The simple and known-fold methods produce the same estimates for the exploiting player, because they just use the actual earnings he produces when being chosen in the second half of the game. The real difference between the approaches shows when looking at the estimates for the standard mathematical player, where a significant improvement can be seen. The third method which incorporates estimations makes the trade-off of a slightly less accurate estimation for the exploiting player, but also significantly increasing the accuracy for the other player.

6.2 Evaluation of Decision Strategies

To evaluate the decision strategies introduced in "Decision Strategies" (Section 5.3) a similar setup as for the earnings evaluation was used:

- random but fixed cards for 2000 rounds.
- two opponents switching to an exploitable strategy after 1000 hands.
- the ensemble contains a member exploiting the vulnerability

The differences are in respect to the ensemble player. The members are rated using the combined method with estimations. This calculation uses all previous rounds uniformly and estimations are filtered for the current street only for each team member. The initialization which lets every team member play on every street 10-times before switching to the real decision strategy remains the same.

6.2.1 Max Value

The max value decision strategy performs very poorly when combined with a rating calculation using estimations like in this case. The exploiting member gets only chosen in the beginning for initialization purposes, as can be seen in Figure 6.6. When the ratings should start to diverge, no estimations for the exploiting player can be calculated, since he only played a few rounds in the beginning. This causes the ratings to only be updated when no estimations are needed. Because the beneficial actions of the exploiting player need estimations, the members always remain on the same rating. This is the perfect example of the dependency of the ratings calculation on the decision strategy, in order to provide good estimations.

6.2.2 Max Value with Draw Handling

When introducing special handling of cases where team members draw, the issue is dampened. Looking at Figure 6.7 the issue seems to be gone. Shortly after the opponents switch their strategy, the ensemble recognizes the benefit of the exploiting team member and starts using him exclusively. This is the optimal reaction to this particular situation, but would any other non-optimal team member have gained the upper hand, the ensemble would probably never have recognized that a better solution exists. The cause is once again, that no other team member gets a chance to play, therefore no better estimations can be made for them in the rating process. If another team member could play beneficial in cases where the rating method does not use estimations, this would not be an issue because he could slowly gain the upper hand.

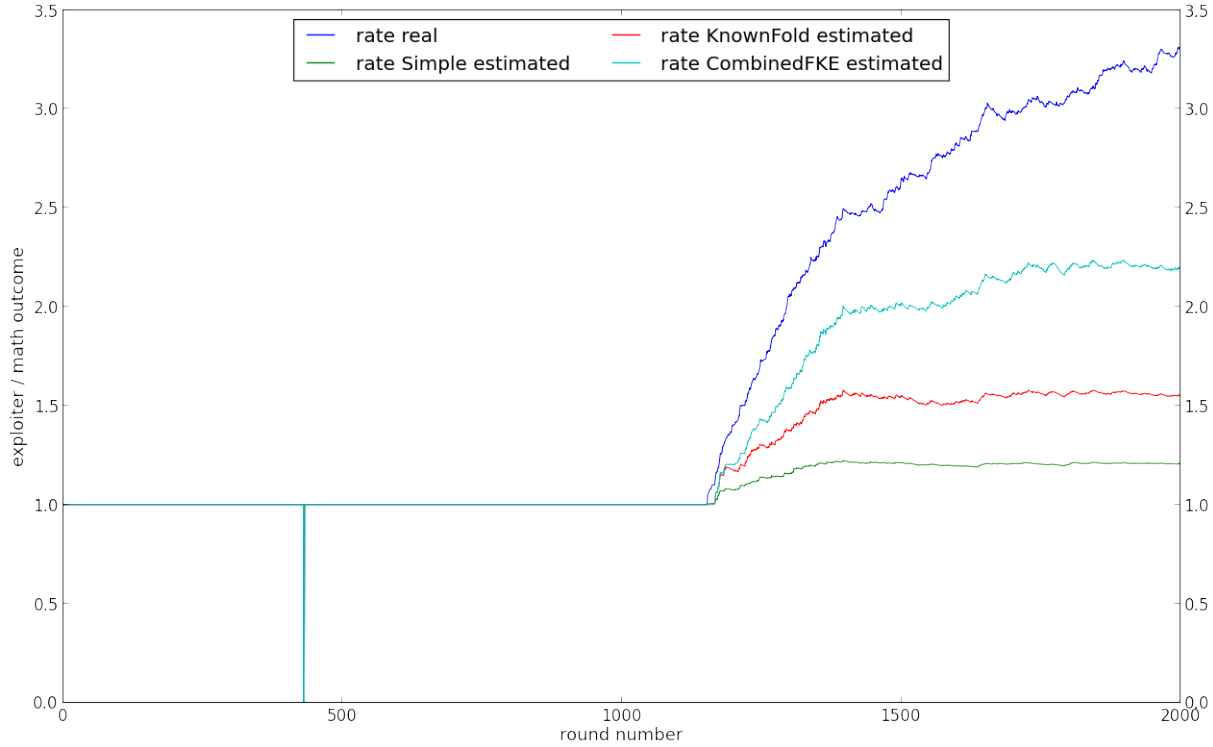

Figure 6.4: Benefit of using exploiter over mathematical bot using all methods

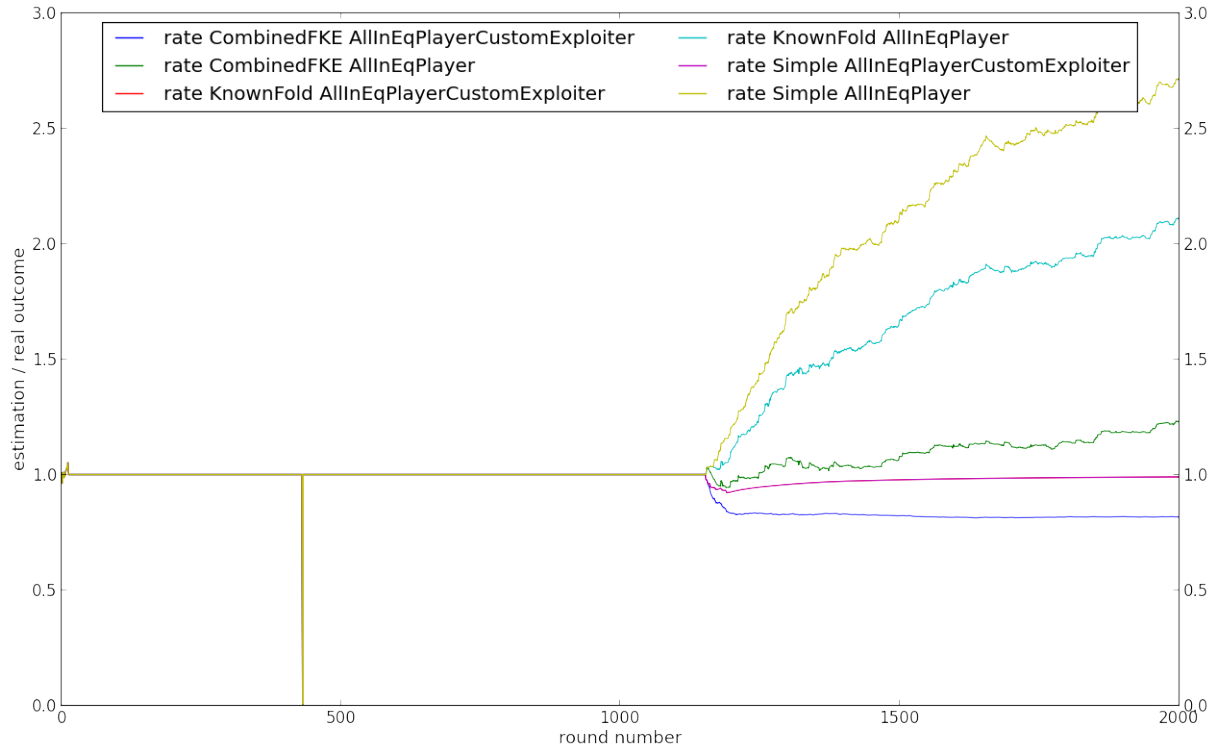

Figure 6.5: Rate of estimates vs. the real earnings using all methods

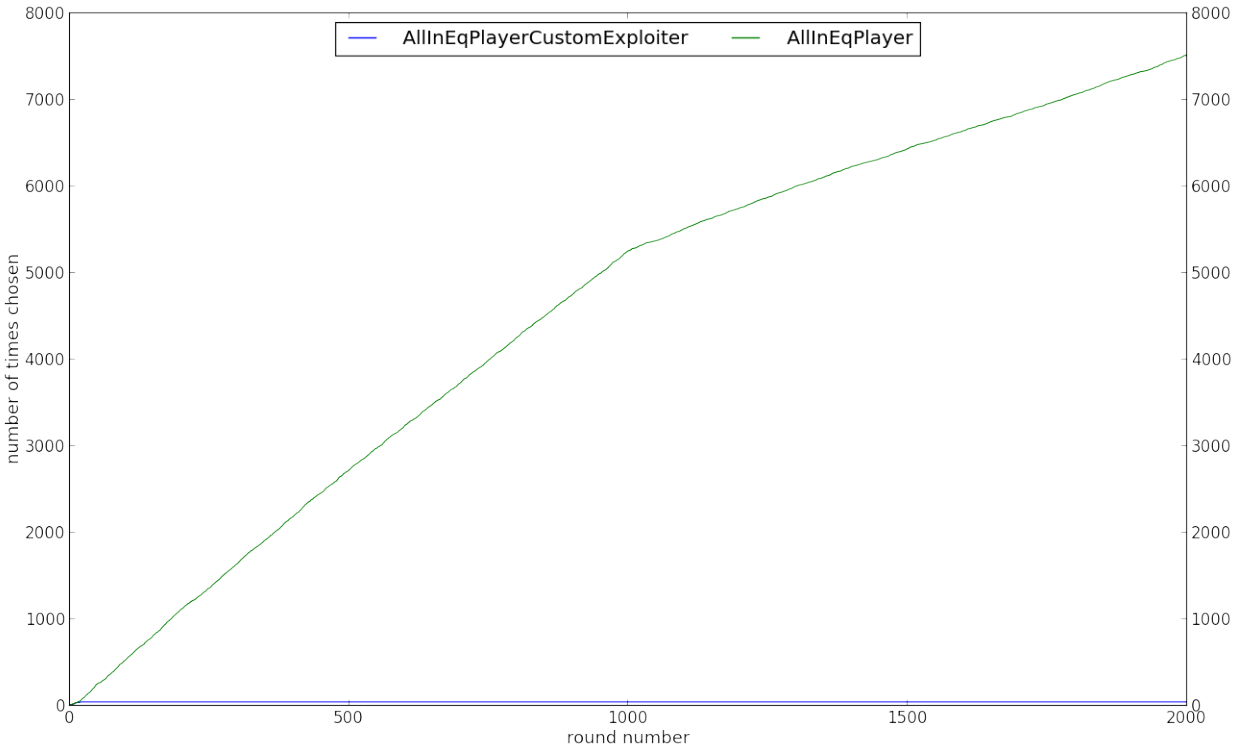

Figure 6.6: Decisions using max value

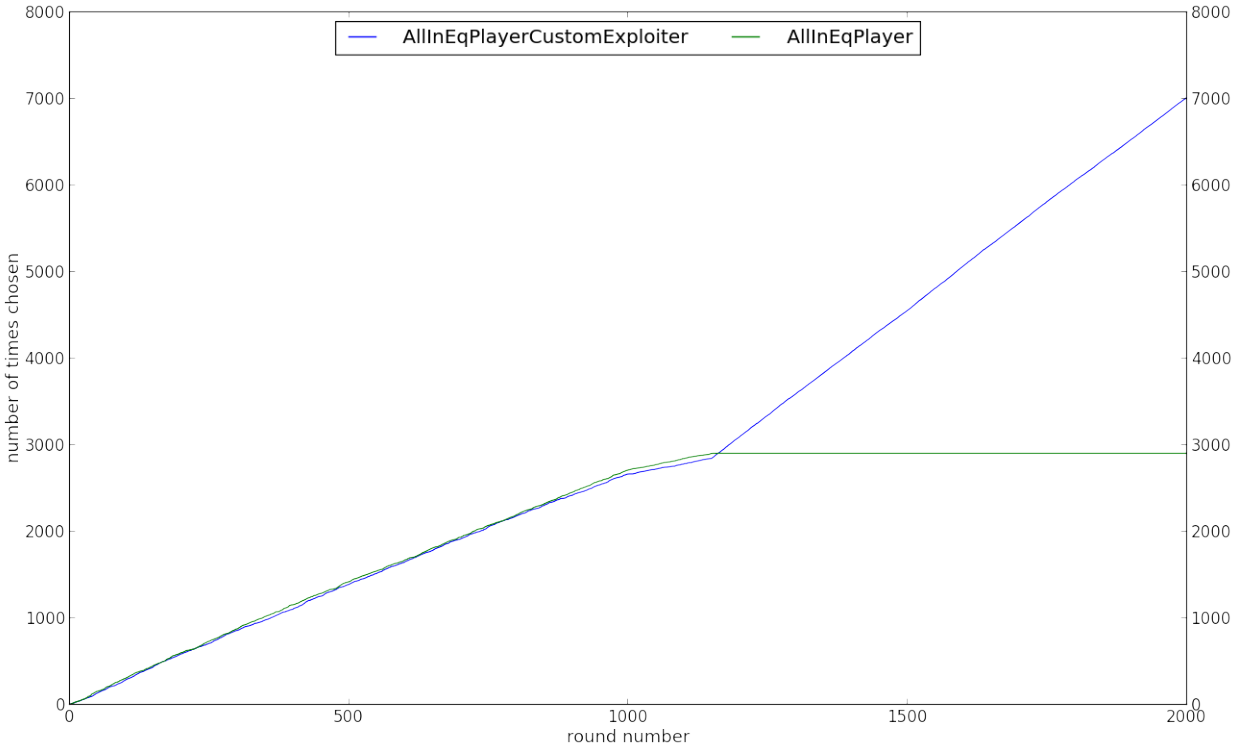

Figure 6.7: Decisions using max value with draw handling

6.2.3 Probabilistic

To prevent the discussed disadvantages of the max value strategies, the probabilistic approach can be used. It has none of the issues discussed, but the decisions could be improved, as show in Figure 6.8. Although the exploiting player gets chosen more often, he could be chosen significantly more. This would directly lead to more exploitation and therefore a higher final outcome for the ensemble. In order to improve the decision process a bias is introduced, which is evaluated next.

6.2.4 Probabilistic with Bias

This strategy improves the probabilistic choice by introducing a transformation which increases values greater than 0.5 and decreases smaller ones. This is achieved using a sigmoid function, which contains a parameter S used to control the intensity of the transformation. Figure 6.9 compares three values for the parameter. $S = 10$ shows a small improvement over using no transformation. When increasing the parameter to $S = 25$ the graph resembles the max value strategy very closely. $S = 20$ is a compromise between these two values. An always fitting value can't be provided, because it very much depends on the characteristics of the untransformed ratings. When using more team members the individual ratings might be closer together, where a higher value for S would make sense. When only using a few team members lower numbers can suffice, except when their ratings are very similar.

6.2.5 Majority Vote

The previous evaluations focused on ensembles with only two members. Whereas the methods can easily be used for far greater ensembles, some special decision strategies can be used where appropriate. To evaluate these three-player ensembles, an additional mathematical player is added to the ensemble.

The first three-player strategy is the majority vote. It does not use any game statistics, but might be beneficial in ensembles with a lot of players, where individual ratings aren't that different. Figure 6.10 shows the behavior of the strategy, which obviously can't use the exploiting player because the other two members often agree with each other, but not with the exploiting player, therefore overruling him.

6.2.6 Max Value with Action Groups

The max value strategy can be updated to rate chosen actions instead of the team members, as described previously. Just like in the two-player case, actions with the same rating will be chosen uniformly to reduce risks of missing estimations. The evaluation additionally to previous ones limits the earnings calculation to the 500 previous rounds. This leads to faster reactions and clearer overall ratings, because only a small amount of previous rounds, instead of the whole game is considered for the ratings. This will be evaluated in detail in Section 6.4.

Figure 6.11 visualizes the strategy in case of three team members. In contrast to the majority vote, the exploiting player eventually reaches a rating better than the sum of the two other players and gets chosen more often than them. In this case he is not chosen exclusively, because ratings are calculated for each street individually and his rating is not as good on all streets.

6.2.7 Probabilistic with Action Groups

The max value strategy has the discussed problem of not doing any exploration, which was handled by using a probabilistic decision process in the two-player case. This approach can also be updated so that ratings are grouped for possible actions. Just like in the max value evaluation, the ratings calculation

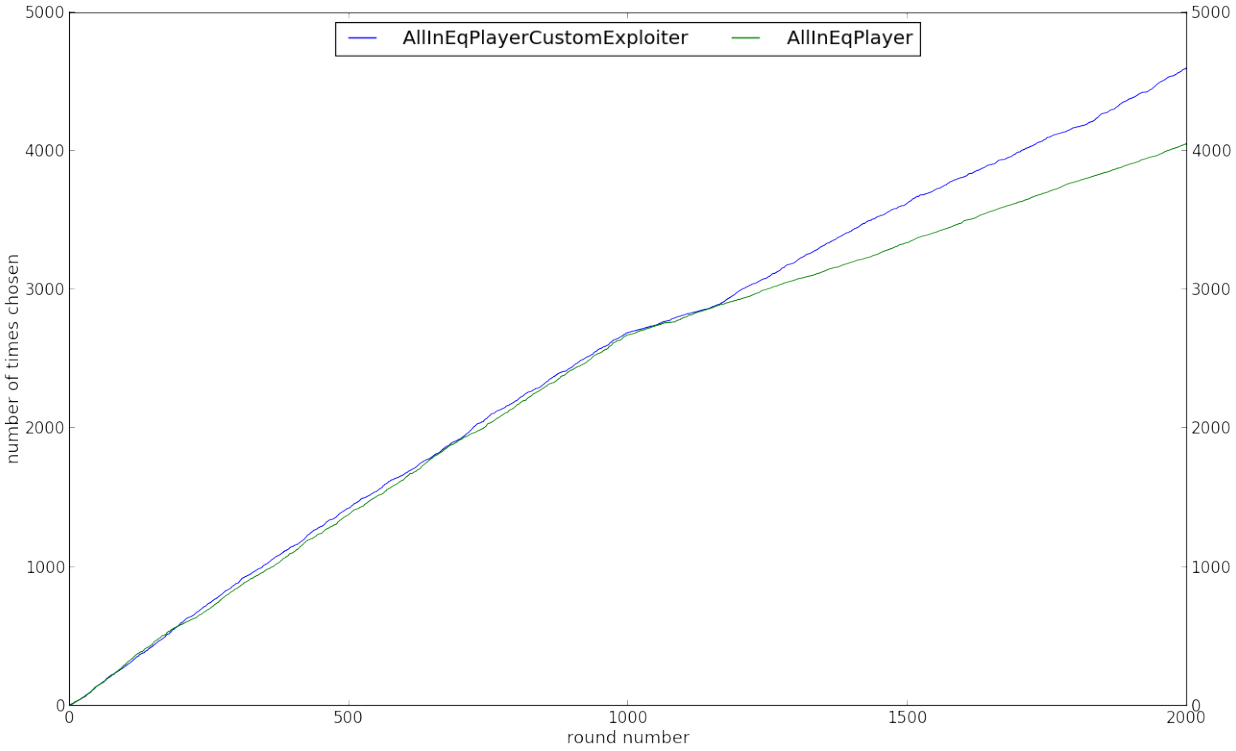

Figure 6.8: Decisions using probabilistic choice

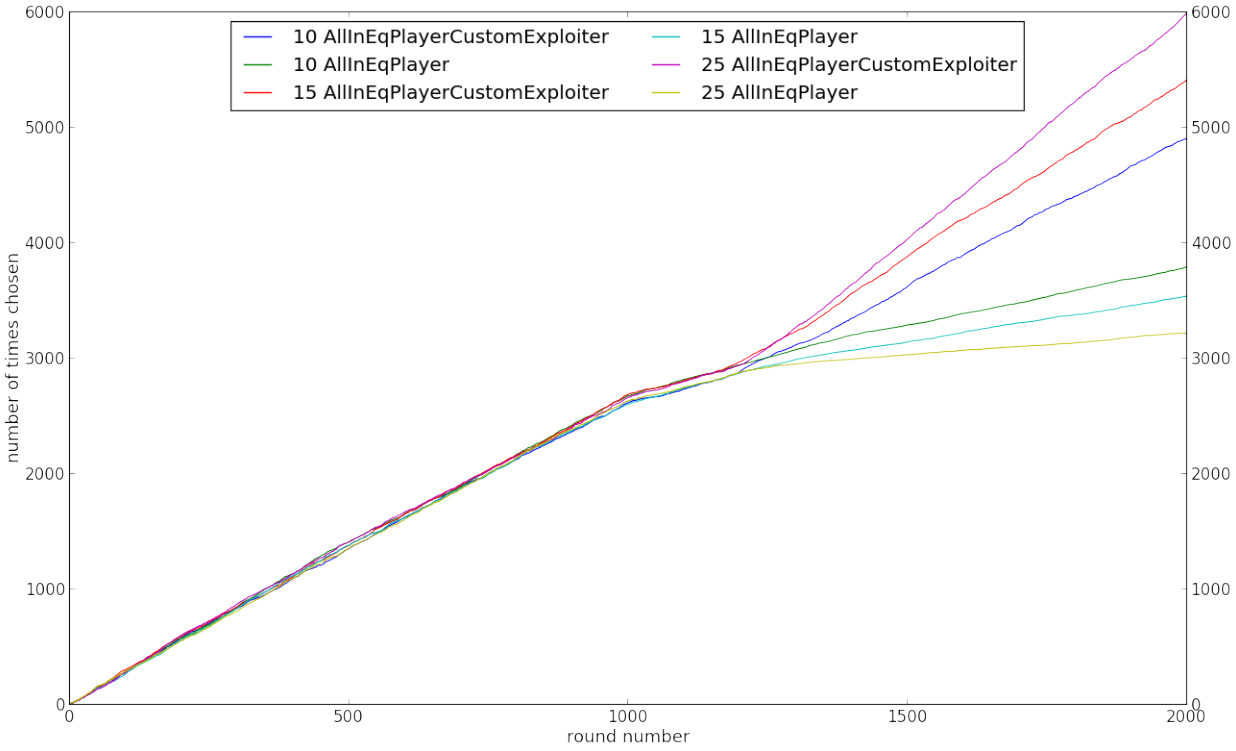

Figure 6.9: Decisions using probabilistic choice with a bias

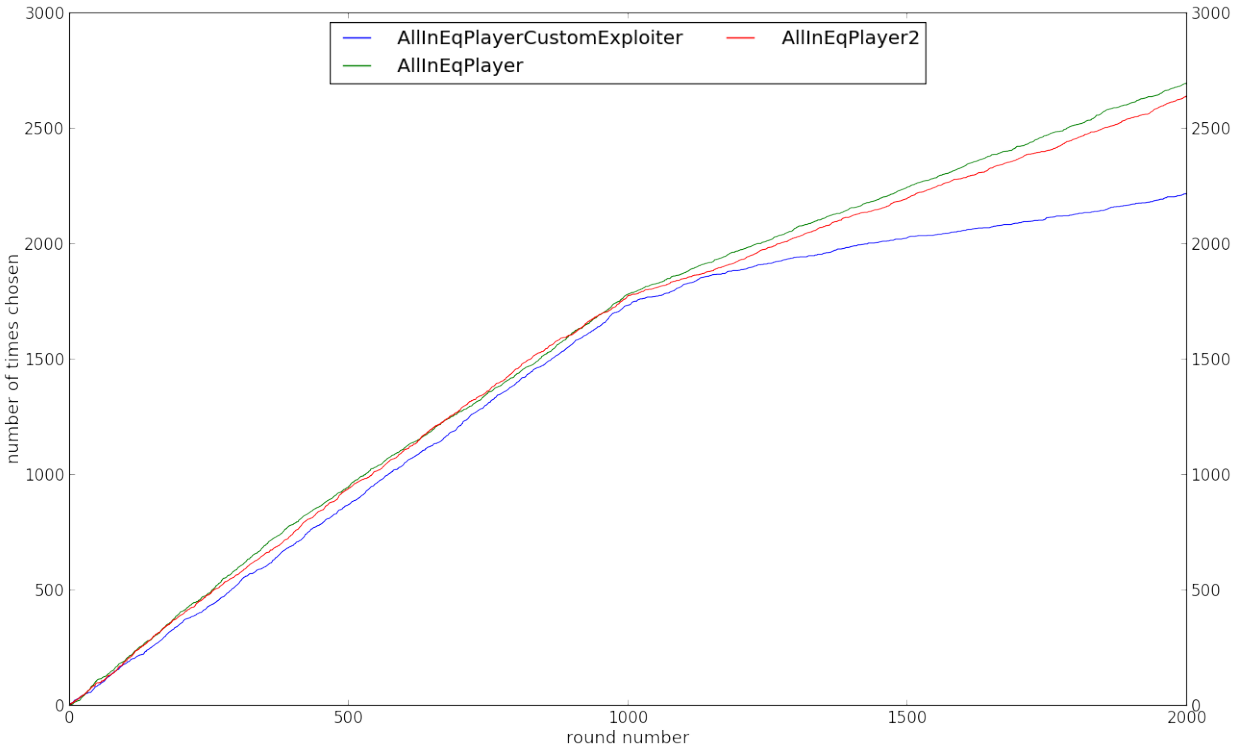

Figure 6.10: Decisions using majority vote

is limited to the previous 500 rounds. In Figure 6.12 the result of the strategy can be seen. The results are similar to the max value strategy and use the bias parameter $S = 10$. Changing the factor won't significantly alter the results, because the exploiting player overrules the other team members only on two of the four streets. This means increasing the factor will only make the decisions on the individual streets more clear, but the decision characteristic between the two types of players will remain the same.

The problematic aspect in this scenario is, that the exploiting player only really wins money when playing in the river, since he otherwise calls in the exploitable situations. To improve the outcome in this evaluation scenario, the outcome calculation would need to prefer the exploiting player even in previous streets, because his outcome on the river is significantly better. Another possibility would be to disable street filtering when choosing between the team players, because the overall outcome for the exploiting player is higher.

6.3 Evaluation of Exploitable Team Members

Another important aspect is the reaction of the ensemble to team members which get exploited or start to play bad. To evaluate the issue, an ensemble plays against two opponents which get exploitable after 1000 rounds. The ensemble consists of the following player:

- standard mathematical all-in-equity player
- exploiting player, but between rounds 1750 and 2500 he plays bad, as described in Section 3.2.6

The ensemble uses the introduced combined with estimates earnings calculation and the probabilistic decison strategy with a bias of $S = 20$. Additionally it was necessary to limit the amount of previous rounds on which to calculate the ratings. This decreased the necessary number of rounds to simulate the effect and provided clearer ratings. Figure 6.13 visualizes the resulting decisions when using a round limit of 500 and 250. The graph shows that the ensemble successfully switches to the exploiting player

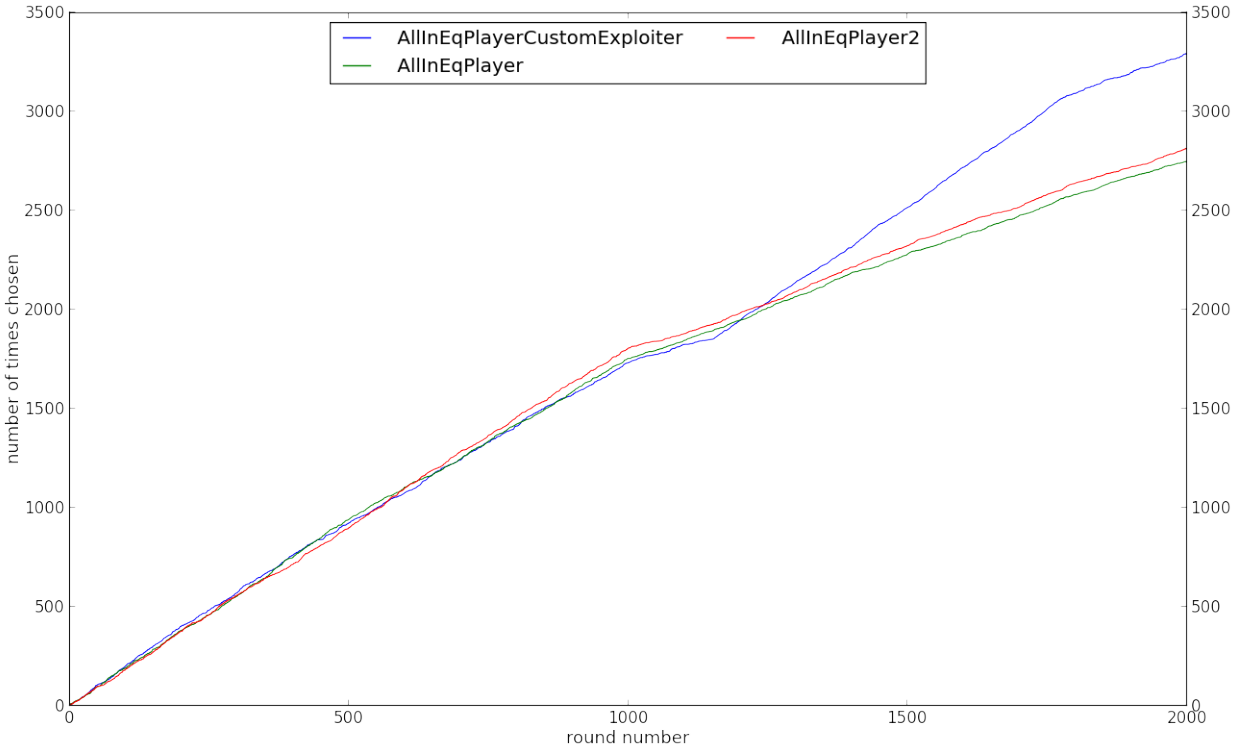

Figure 6.11: Decisions using maximum value action voting

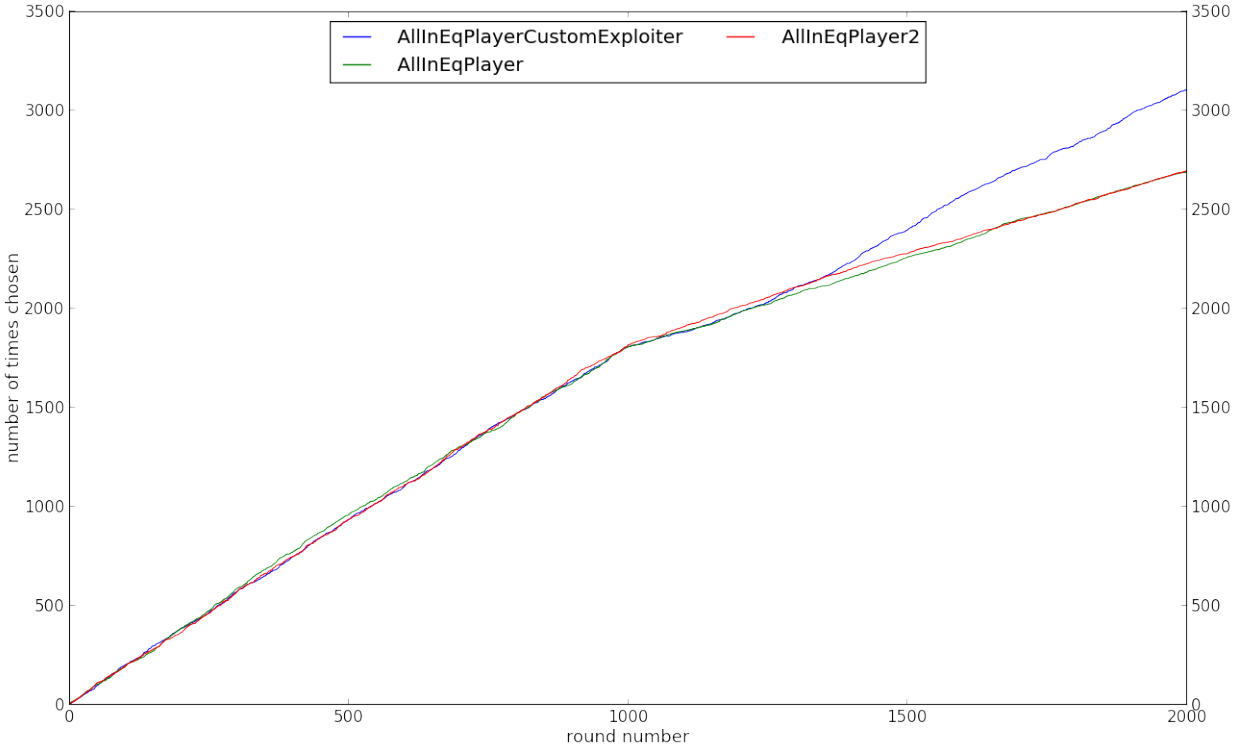

Figure 6.12: Decisions using probabilistic choice action voting

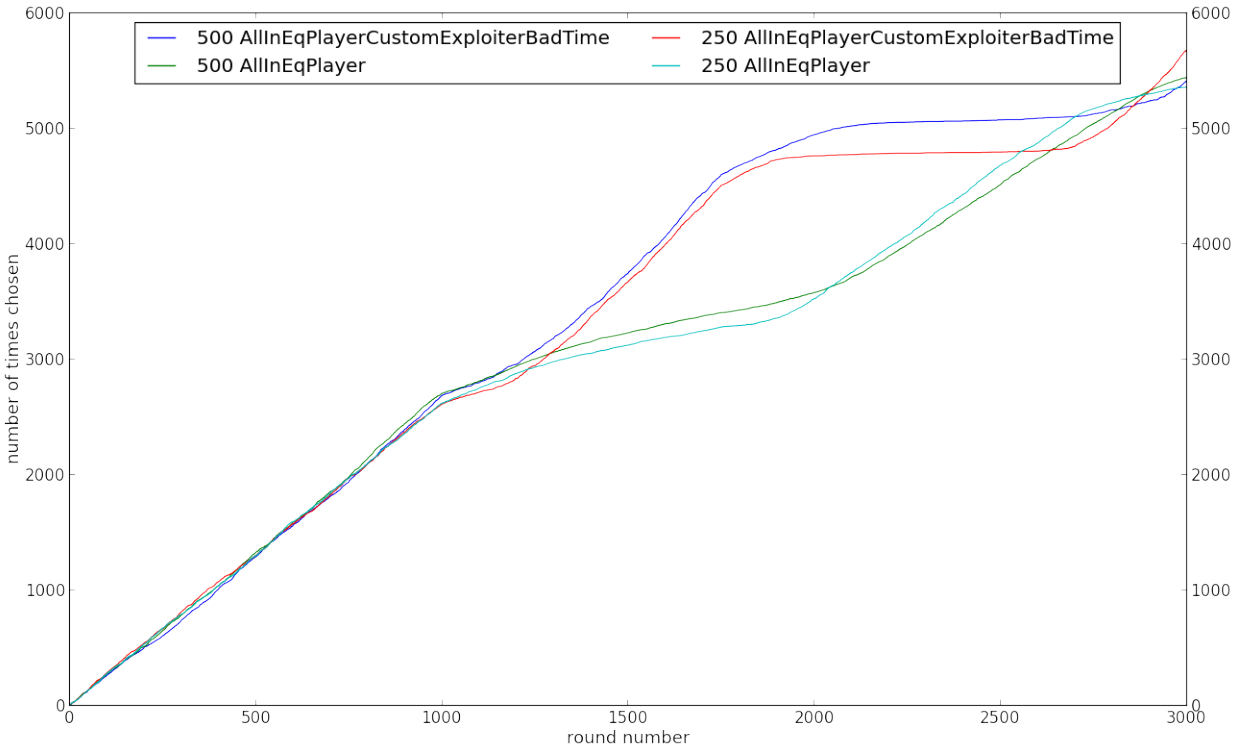

Figure 6.13: Decisions when a team member starts playing bad with different rating timeframes

after 1000 rounds in both cases and then almost exclusively to the all-in-equity player shortly after the exploiting strategy gets bad. Near the end of the game the ensemble starts switching back to the exploiting strategy. Using a smaller amount of previous rounds to rate the team members increases the dynamics of the ensemble, allowing the ensemble to react faster to changing situations. The effect can be seen very clearly when looking at the times the bad player gets chosen. Using a short rating timeframe, the ensemble switches a lot faster to the other player, mitigating further losses. This will be further explored in the next chapter. Lowering the rating timeframe even further could make the ratings for the players unstable, because not enough rounds are available to calculate a stable rating. This would result in a lot of unnecessary switching between the team members, so that even shorter timeframes were not evaluated.

6.4 Evaluation of Round Weighting

So far the focus was most of the time only on the detection of an exploitable situation and not how fast the detection was achieved. A faster detection can be easily achieved by using a weight function when calculating the ratings for the team members. In the following the traditional 2-player ensemble in a 3-player game will be used (2 exploitable opponents after half the time). The probabilistic decision strategy without a bias will be used alongside the combined earnings calcuation with estimates. The figures 6.14, 6.15 and 6.16 show the results when using a half-time weighting function (Section 5.1) with the half-times 500, 250 and 100 rounds respectively. The vertical red lines marks the round at which the exploiting player recognizes the opportunity and starts exploiting the opponents. The vertical turquoise line marks the round at which the ensemble starts choosing the exploiting player more than its combatants. The red line is always at round 1153, whereas the turquoise one is at different positions. With a half-time of 500 the spot is at 1340, which a shorter half-time can improve down to 1260. While this is a significant improvement, further lowering the half-time only slowly improves the speed. With a

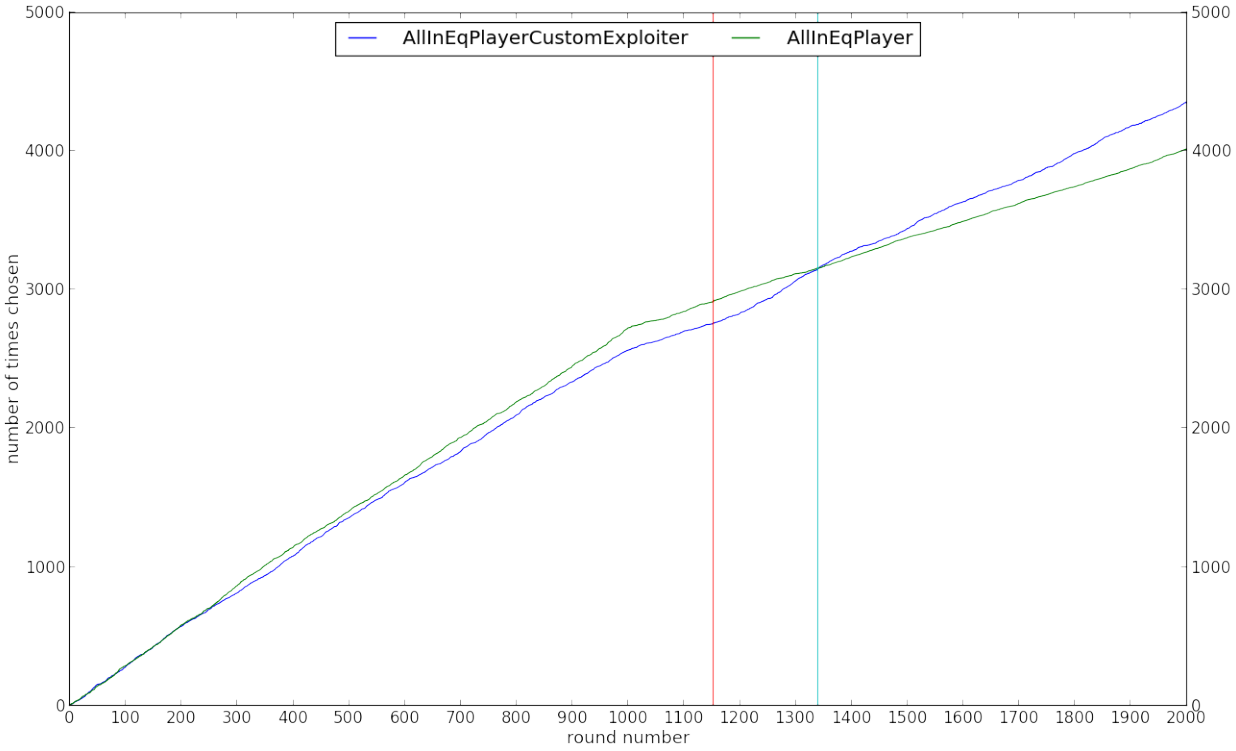

Figure 6.14: Decisions when weighting with a half-time of 500

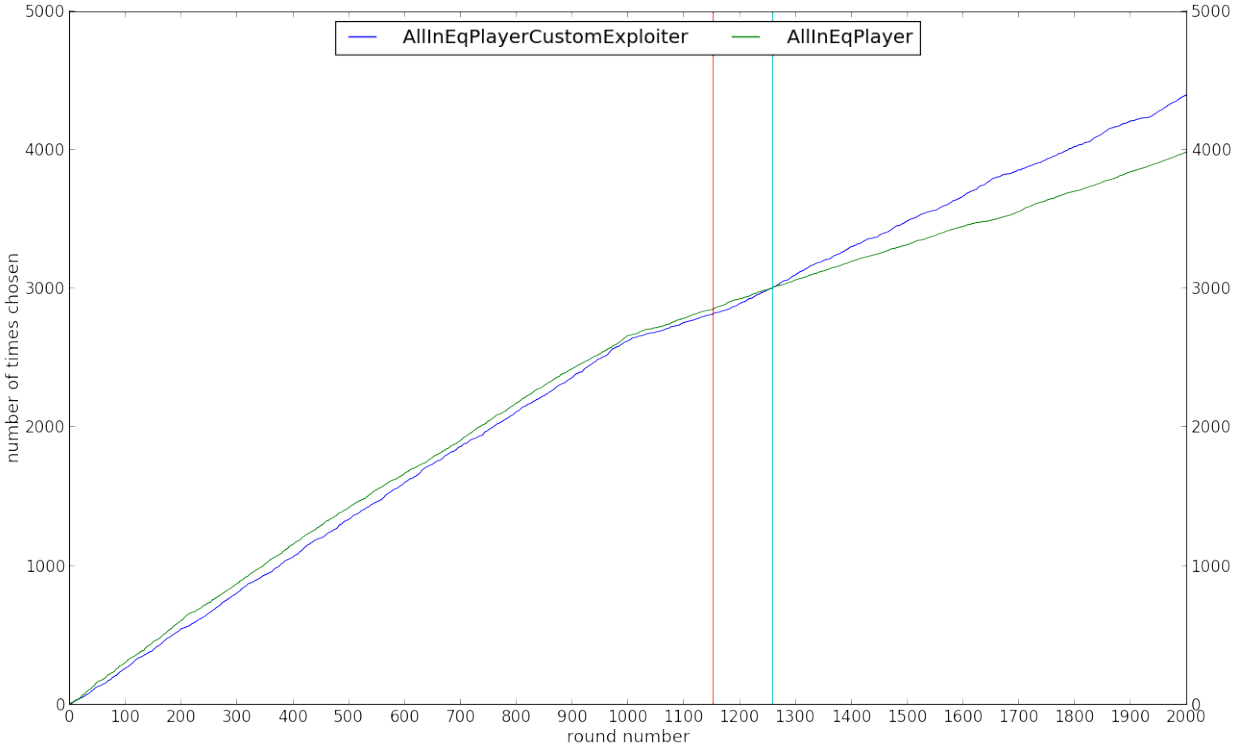

Figure 6.15: Decisions when weighting with a half-time of 250

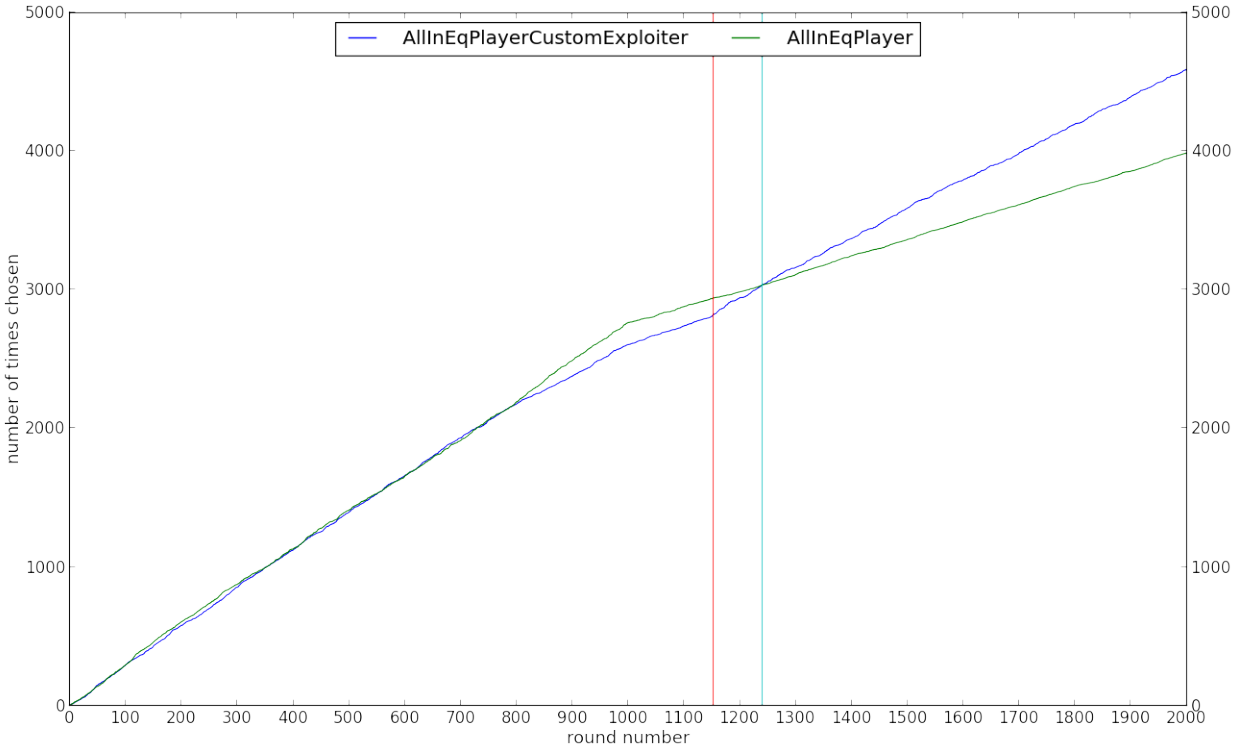

Figure 6.16: Decisions when weighting with a half-time of 100

very short half-time of 100 this can be brought down to 1240, resulting in less than 100 rounds to react on the change.

6.5 Evaluation of Opponent Filtering

In Section 3.3.5 opponent filtering was introduced, which can calculate statistics per opponent. To show the use of this feature an ensemble played against a mathematical all-in-equity player, as well as an exploitable bot. In order to have a baseline for comparison the ensemble containing a standard mathematical and the exploiting player challenged these opponents. The results of this ensemble can be better, because the exploiting player uses his exploiting strategy although he also faces an unexploitable player. The better ensemble therefore uses an exploiter which only deviates from his mathematical strategy when he faces the exploitable opponent alone. This ensemble improves the outcome significantly. To show the benefits of opponent filtering the same ensemble as in the baseline comparison is now used but has opponent filtering enabled. This should be able to replace the detection of opponents in the team member and Table 6.1 illustrates that. In comparison with the baseline this is a significant improvement, but the ensemble filtering is not always able to match the performance when detecting the opponents in the team member. This can be attributed to the fact that the ensemble needs some amount of time to detect the situation.

6.6 Evaluation against Traditional Approaches

In this chapter the performance of this new ensemble approach will be compared to the UCB1 method introduced in “Related Work” (Section 2.2). The evaluation was done with a random but fixed set of cards for 2000 hands. Each scenario was evaluated with the following ensembles:

- `Random`
  Always randomly choosing between the team members

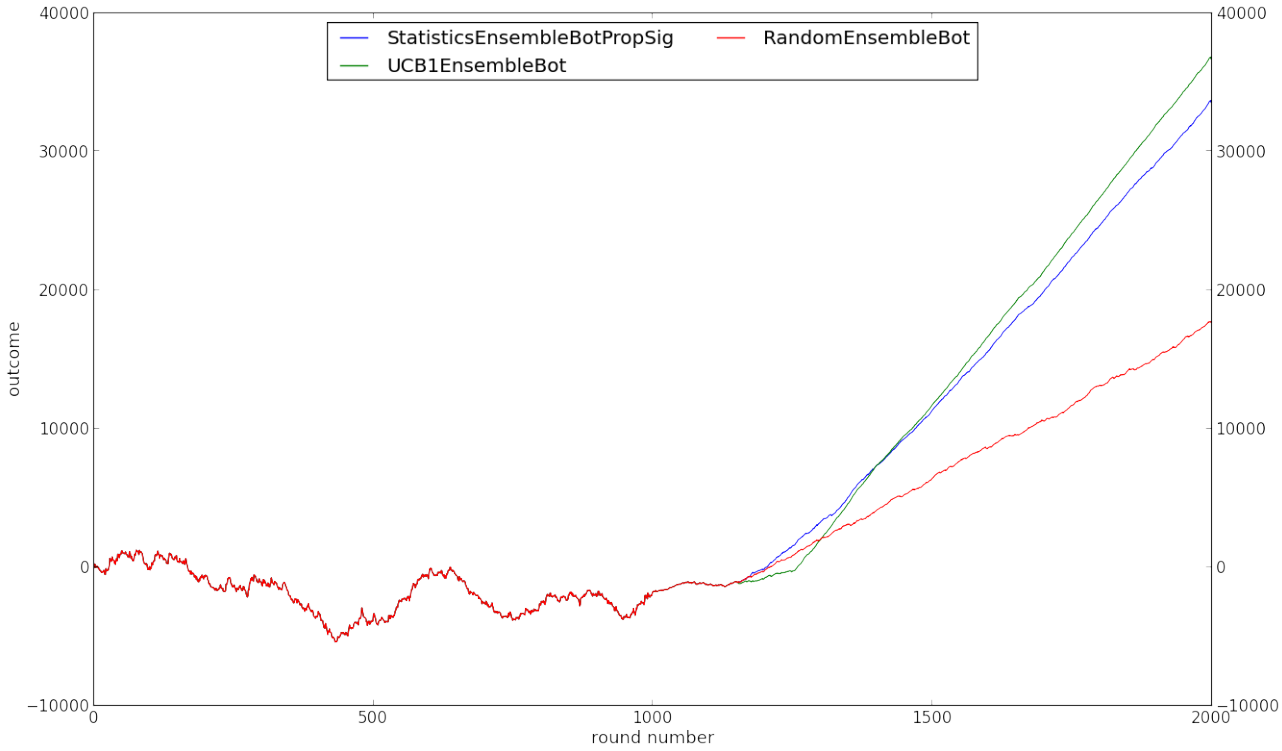

Figure 6.17: Outcome when reacting to overall strategy change

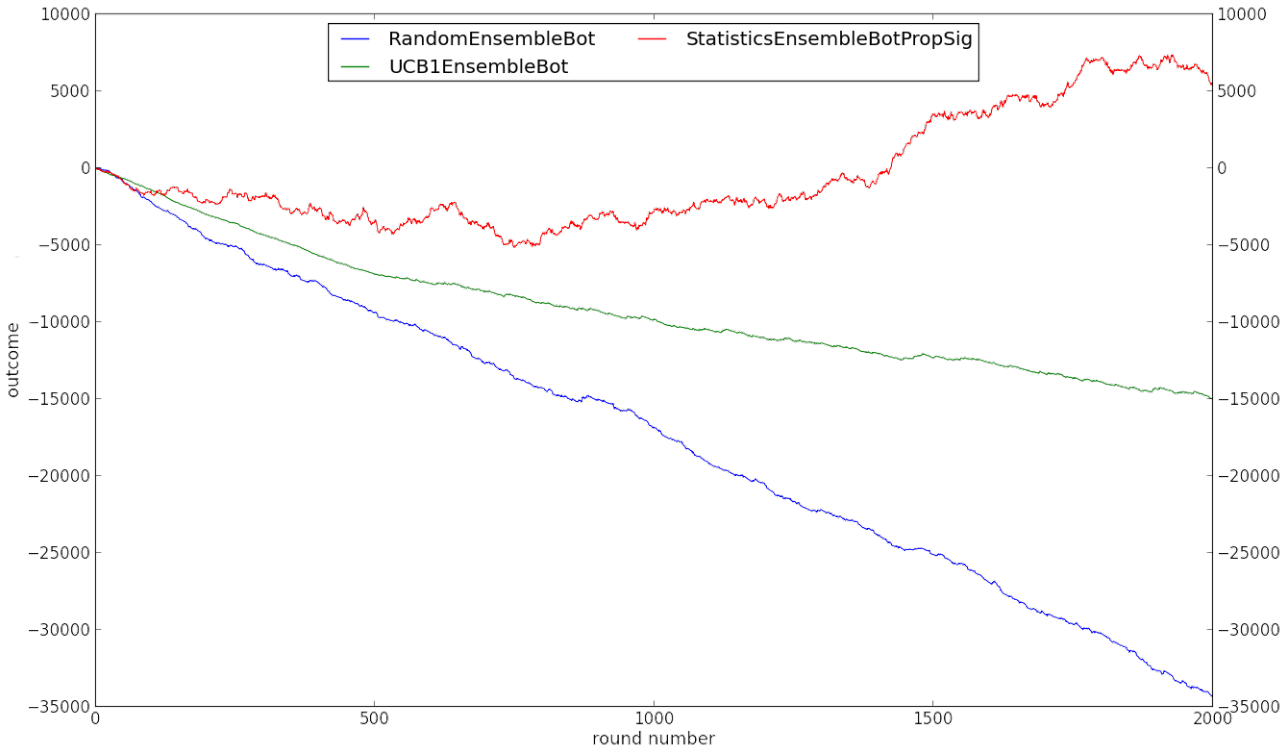

Figure 6.18: Outcome when adapting to game situations

	Baseline	Bot Opponent Detection	Ensemble Opponent Filtering
Outcome Game 1	-7315.00	10810.00	7865.00
Outcome Game 2	-10462.50	14695.00	2740.00
Outcome Game 3	-4627.50	16865.00	8697.50
Outcome Game 4	9145.00	21870.00	10425.00
Outcome Game 5	-6340.00	18715.00	5265.00
Outcome Game 6	-8882.50	14425.00	13325.00
Sum	-28527.50	97380.00	48317.50
Average	-4754.58	16230.00	8052.91

Table 6.1: Outcome when using no opponent detection, detection on the bot side or on the ensemble side

- `UCB1`
  Using the UCB1 approach to choose a team member
- `Statistics`
  Using the probabilistic strategy with a bias of $S = 20$ and the combined with estimations earnings calculation. The earnings calculation was done on all previous rounds uniformly and calculated for each street and team member.

At first a simple overall strategy change was tested. The ensembles faced two opponents which were vulnerable after 1000 rounds. They had the exploiting strategy, as well as a standard all-in-equity player in their repertoire. Figure 6.17 shows the results of the evaluation. It can be seen that the additional effort of the statistics calculation does not pay off, because it reaches a similar outcome like the UCB1 method. UCB1 detects the strategy change well and reacts appropriately and can therefore be considered an easier and equal choice. Although the statistics ensemble could be further tuned, to detect the change a couple of rounds earlier, as shown in Section 6.4. It already reacts earlier than the UCB1 method, but doesn't react as radically as it (exlusively choosing the exploiting player). Tuning these values for the scenario could allow the ensemble method to perform slightly better.

Another evaluation scenario which was tested, makes use of the opponent filtering feature introduced in Section 3.3.5. The ensembles face two opponents of which one is exploitable, whereas the other one is a normal all-in-equity player. The team members are also a normal all-in-equity player and a player which exploits once he only faces an exploitable player, otherwise he plays bad as described in Section 6.3. Figure 6.18 shows that the statistics ensemble now is able to significantly outperform the UCB1 approach by detecting which team member is good in which situations. Therefore the additional effort of the statistics calculations are beneficial whenever team members are especially good only in certain situations. Then the calculations can use their comprehensive filtering capabilities to detect these situations and react appropriately.

6.7 Evaluation with Trivial Bots

Another interesting aspect of an ensemble approach is whether trivial members can achieve a greater performance against complex opponents than the individual members on their own. To evaluate such a situation the following trivial players were used:

- `CallBot`: always calling
- `CheckNorris`: 75% call, 20% raise and 5% fold
- `RaiseBot`: always raising

	CallBot	CheckNorris	RaiseBot	PropSigAction	PropSig
Outcome	-36780.00	-42645.00	-76682.50	-43417.50	-35610.00

Table 6.2: Outcome for trivial bots versus all of them in an ensemble (with and without action groups)

These three were put into an ensemble using the probabilistic method with action grouping and played against Akuma (described in Section 3.2.7) and a mathematical fair player, using all-in equity. Table 6.2 shows the achieved outcome after 2000 hands of each trivial bot on his own against using all of them in an ensemble. The individual bots don't perform particularly good against the opponents as expected, but `CallBot` is clearly the best of them. The ensemble is also not able to achieve the outcome of `CallBot`, but rather of the second best player. The problematic aspect which limits the ensemble is that often the bots are rated almost uniformly or two of them dominate with similar ratings. This causes the action grouping mechanism to always choose the action of members who agree on an action. Table 6.2 also lists an evaluation without action groups using the normal probabilistic ensemble without bias. Now the ensemble reaches the outcome of `CallBot`, but is not able to significantly top it. This means the ensemble can identify the best available player and can play as good as him. Additionally it also recognizes that there are sometimes better options than this one player and uses other ones, so that it can achieve a slightly better result.

6.8 Evaluation with Complex Bots

After using the ensemble with trivial bots the immediate question which arises is how good the ensemble handles complex bots. For this purpose Akuma (Section 3.2.7) and OwnBot (Section 3.2.7) were put into an ensemble playing against themselves. A game consisted of 2000 hands and the ensemble used a probabilistic decision strategy with bias ($S = 15$). The ratings of the team members were limited to the 500 previous rounds. During test matches it became clear that the ensemble didn't always use the bots as expected and observed in previous games. The situation was improved by enhancing the outcome estimation a little bit further. Previously the difference between a call and a raise on the river was approximated by an estimate obtained through past experience. For this evaluation no estimation was used in this particular situation on the river, but rather the direct bet size was added or subtracted depending on the outcome in the round (positive or negative). A bit more detail will be given after this in Section 6.8.1. The described small change improved the performance of the ensemble and usage distribution of the team. The overall achieved outcome of the ensemble in contrast to its opponents can be seen in Table 6.3.

The ensemble at the end almost always played OwnBot the most, just using Akuma in preflop situations. Game 2 is an exception to this because Akuma gets to be used more than OwnBot. Although the ensemble settles on similar behavior, its performance is not always good. It is able to win half the games and looses one.

The overall evaluation with complex bots proved to be challenging, because of their non-deterministic nature. Attempts to make their behavior deterministic for the evaluation purposes didn't succeed. This would have made it possible to compare the performance of the ensemble against the team members on their own. This is the reason that no exact reason for the shortcomings of the ensemble on the last three games could be identified. In different games it was observed that two instances of OwnBot generally don't play good against each other, with one instance loosing a lot. On the other hand a similar evaluation result can be observed when using another bot instead of OwnBot. It may very well be the case that the used outcome estimation still isn't accurate enough to rate these complex players reliably. The method was already slightly tweaked as described above for this evaluation.

The achieved overall outcome could generally be improved by shortening the timeframes on which the team members were evaluated. Table 6.4 shows the same games but uses only the 250 previous rounds instead of 500 to rate the available players. It can be seen that this configuration ultimately achieves a

	OwnBot	Akuma	Ensemble	Ensemble Finishing Rank
Outcome Game 1	735.00	-2147.50	1412.50	1
Outcome Game 2	-205.00	82.50	122.50	1
Outcome Game 3	-962.50	-2610.00	3572.50	1
Outcome Game 4	2867.50	-1262.50	-1605.00	3
Outcome Game 5	3427.50	-2665.00	-762.50	2
Outcome Game 6	4712.50	-4877.50	165.00	2
Sum	10575.00	-13480.00	2905.00	
Average	1762.50	-2246.66	484.16	
Average Rank	1.66	2.66	1.66	

Table 6.3: Outcome of evaluation with complex bots

	OwnBot	Akuma	Ensemble	Ensemble Finishing Position
Outcome Game 1	3317.50	-1707.50	-1610.00	2
Outcome Game 2	330.00	25.00	-355.00	3
Outcome Game 3	-1182.50	-2360.00	3542.50	1
Outcome Game 4	2552.50	-2702.50	150.00	2
Outcome Game 5	2750.00	-2035.00	-715.00	2
Outcome Game 6	2790.00	-8040.00	5250.0	1
Sum	10557.50	-16820.00	6262.50	
Average	1759.58	-2803.33	1043.75	
Average Rank	1.33	2.83	1.83	

Table 6.4: Outcome of evaluation with complex bots with a shorter rating timeframe of 250

higher overall outcome and does so through different games, although the cards and positions remain the same. It might therefore be desirable to dynamically detect which rating timeframe is the most promising one in order to maximize the earnings of the ensemble.

It was observed that using different pairs of complex players in an ensemble playing against each other did not change the overall resulting picture of only winning two to three games. Another tested approach was to not let the ensemble play against themselves but different players. This also didn't change the characteristic of the results.

6.8.1 Improved Earnings Calculation

Much like in Section 5.2 where the other earnings calculation methods were introduced, this chapter will provide pseudo-code of the previously described improvement in the following listing. Additionally a small example calculation using this method is provided in Table 6.5.

```
memberReward = 0
perDecisionReward = ensembleOutcome / numberDecisions
for every decision in round:
        chosenMember   = getChosenTeamMember(decision)
        playedAction   = getActionOfTeamMember(decision, chosenMember)
        actionOfMember = getActionOfTeamMember(decision, i)
        estimate       = getEstimate(street, chosenMember, playedAction, i, actionOfMember)
                         / numberDecisions
        if estimate does not exist:
                continue
```

Street	Mem1	Mem2	Chosen	Earnings Mem1	Earnings Mem2	Estimate value
preflop	call	fold	Member 1	10	-10	/
preflop	call	call	Member 2	20	0	/
flop	raise	call	Member 1	30	20	120
turn	call	raise	Member 2	30	20	n/a
river	raise	call	Member 1	40	-10	/
river	call	raise	Member 2	30	0	/

Table 6.5: Example calculation of estimations earnings calculation

```
        if i == chosenMember && exists(estimate):
                memberReward += perDecisionReward
        else:
                if playedAction == actionOfMember:
                        memberReward += perDecisionReward
                else if playedAction != FOLD  && actionOfMember == FOLD:
                        memberReward += -investment
                else if playedAction == CALL  && actionOfMember == RAISE:
                        if street != RIVER:
                                memberReward += estimate
                        elif street == RIVER && ensembleOutcome >= 0:
                                memberReward -= bet_value
                        else:
                                memberReward += bet_value
                else if playedAction == RAISE && actionOfMember == CALL:
                        if street != RIVER:
                                memberReward += estimate
                        elif street == RIVER && ensembleOutcome >= 0:
                                memberReward += bet_value
                        else:
                                memberReward -= bet_value
                else if playedAction == FOLD  && actionOfMember != FOLD:
                        memberReward += estimate
return memberReward
```

7 Conclusion & Future Work

Traditionally poker agents were static and somewhat specialized. While there were some approaches to make poker agents more dynamic by using teams of agents, most of them didn't harness all the information available to them. Often only the opinion of one of the members was requested, whereas much more could be used. This thesis introduced techniques to gather all the available information from unknown team members. With the help of this information rating mechanisms were developed to be able to classify the team members and use them whenever they were the strongest. With this the ensemble can identify when it is being exploited and take counter-measures, notice that a team member can exploit opponents and maximize the earnings and realize when a player is only effective against certain opponents to use him only in these situations. Furthermore multiple decision strategies were tested and their advantages and disadvantages discussed, so that the right strategy can be used for each set of ensemble players.

Poker is a complex domain to evaluate because of the many factors involved in making a successful player. During this thesis it proved challenging to transfer the successes experienced in the constructed evaluation games in the same extent to real games with undeterministic complex players. While reproducibility is crucial during evaluation, it is difficult to achieve with more complex players with many different components involved in the decision process. As expected, it was shown that trivial bots can't be used to form an adaptable and competitive poker team. The ensemble just hasn't enough information about the impact of a particular action to the overall outcome of a round.

Ensembles can be very beneficial to adapt to changing situations during a game. The constructed evaluation scenarios showed the potential such a technique could introduce, but the performance proved to be challenging with real complex bots. It was shown that better outcomes can already be achieved by using an ensemble with them, but there are also cases where one could be better of playing without an ensemble. Further improvements to the ensemble techniques are necessary so that they are always beneficial. Enhancing the outcome estimations will certainly improve the performance of the ensembles, as already seen in "Evaluation with Complex Bots" (Section 6.8). The introduced estimation was good enough to detect all the evaluated scenarios, but could be further improved.

It was further shown that the configuration of an ensemble is very dependent on its members and must therefore be carefully chosen. In games with a lot of changes it is crucial to use round weighting to get fast reactions through clear ratings. Additional filters like opponent filtering can introduce negative side effects when they reduce the amount of rounds used for the ratings too much. In such situations the ensemble is often better off judging the team members with this feature disabled. Dan Egnor tackled this problem in his Rock-Paper-Scissors ensemble agent [Egn00] by using many meta-strategies with differing configurations and deciding among these different configurations. Egnor rated his strategies amongst others over multiple timeframes and let his algorithm decide which timeframe rating is the most accurate during the game. An approach like this could be further evaluated in the domain of poker to reduce impact of a team on the ensemble configuration.

Using many different strategies in one team increases the resources the team needs drastically. An ensemble could tackle this problem by disabling some team members which play very similar and therefore do not contribute much diversity to the ensemble. They might even hurt the ensemble when decision strategies such as majority voting or action grouping are used. Disabling them could then improve the performance of the team. A similar idea might be to let the ensemble detect that it could benefit from a certain type of player which e.g. raises more often, whenever the ensemble detects that it e.g. calls too often.

Bibliography

[ACBF02] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[ACBFS95] P. Auer, N. Cesa-Bianchi, Y. Freund, and R.E. Schapire. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 322–331. IEEE, 1995.

[acp12a] Annual computer poker competition. `http://www.computerpokercompetition.org/`, 2012. accessed january 2013.

[acp12b] Annual computer poker competition rules. `http://www.computerpokercompetition.org/index.php/competitions/rules/85-2013-rules`, 2012. accessed january 2013.

[alb10] ACPC protocol specification, version 2.0.0. `http://www.computerpokercompetition.org/downloads/documents/protocols/protocol.pdf`, 2010. accessed january 2013.

[BZ12] Bojan Butolen and Milan Zorman. Combining various strategies in a poker playing multiagent system. 2012.

[CJhH02] Murray Campbell, A.Joseph Hoane Jr., and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1-2):57 – 83, 2002.

[Egn00] Dan Egnor. Iocaine powder. *International Computer Games Association Journal*, 23(1):33–35, 2000.

[Joh07] M.B. Johanson. Robust strategies and counter-strategies: Building a champion level computer poker player, 2007. Masters's Thesis.

[JZB07] M. Johanson, M. Zinkevich, and M. Bowling. Computing robust counter-strategies. *Advances in neural information processing systems*, 20:721–728, 2007.

[Rob52] H. Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

[roc99] The first international roshambo programming competition. `http://webdocs.cs.ualberta.ca/~darse/rsb-results1.html`, 1999. accessed april 2013.

[RW11a] J. Rubin and I. Watson. Computer poker: A review. *Artificial Intelligence*, 175(5):958–987, 2011.

[RW11b] J. Rubin and I. Watson. On combining decisions from multiple expert imitators for performance. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume One*, pages 344–349. AAAI Press, 2011.

[SBB+07] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[Sch01] J. Schaeffer. A gamut of games. *AI Magazine*, 22(3):29, 2001.

[Tes94] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.*, 6(2):215–219, March 1994.

[Zop10] Markus Zopf. Ein Framework zur Entwicklung und Evaluation intelligenter Pokeragenten, 2010. Bachelor's Thesis.