NTNU

Det skapende universitet

Department of Computer and Information Science

Report: Poker AI Implementation in Java

Henning Funke & Thomas Kinnen

IT3105 - Kunstig intelligens programmering

Table of Content

1 Implementations 1

2 Opponent Modeling 4

3 Simulation Results 5

4 Discussion of results 7

1 Implementations

We decided to use an object oriented design with Java to implement the poker simulation and players.

1.1 Phase I

Code structure

To control the flow of the poker simulation we use one main class **PokerGame**. It owns the players and a deck of cards and manages their interactions like giving cards, querying players for actions and evaluating the chosen actions. Additionaly it possesses information about the status of the game like the common cards, the current highest bet and credits in the pot.

The Players are stored in a fixed list, additionally only the active players of a round are stored in another list that is rotated in each round. We thereby implemented that a different player starts each betting round and the big- and small blind are bet alternatingly. The core procedure in this class is *bettingRound(GameState, Blinds)*. The parameters specify the phase of the game and whether betting of blinds is required. With this information the active players are queried to settle their bets and the betting is looped until there are no changes for a whole round. When all betting rounds are finished, the power ratings for each player are calculated and the credits are awarded to the winner.

In the first phase we implemented a naive class **Player** that is base-class for all players we implemented later. It owns the players´ two hand cards and the amount of credits. Further it knows the game object to query certain information about the game's status for betting decisions. The class provides a function *getBet(GameState, minBet, force)* that returns the action chosen by the player in the game phase specified by *GameState*. The parameters *minBet* and *force* indicate basic conditions for the bet the player has to settle on.

To wrap the different actions a player can take in one turn, we introduced an abstract base class **PokerAction** that stores the properties of a player action. It contains a procedure that executes the action by calling procedures of **PokerGame** like *call(), foldPlayer()* or *raise()*.

The concrete implementations for this class are **CallAction, RaiseAction** and **FoldAction**.

The class **Deck** implements the 52 playing cards available in a poker game. The cards are stored in an *ArrayList* and also provides the functionality to shuffle the deck. There is a variety of constructors to support the simulations for the AI Players and a single or more cards can be drawn.

The card itself has no functionality and only stores the suite and the value.

For evaluating sets of cards according to poker rules we had to implement the power ratings newly since we dont use python. Therefore we introduced 2 classes: **CardSet** and **PowerRating**.

A **PowerRating** stores the rating for a set of cards as described in the lecture and provides functionality to compare these ratings in order to find out which one is better.

CardSet contains a number of cards and is used to find the best rating for these cards. This is done by checking if the set contains a constellation of cards (e.g. StraightFlush). If yes the related power rating is returned and if not the set is checked if it contains the next lower evaluated constellation.

Betting decisions

To create simple AI players that profit from some of our preceding methods, we implemented **PlayerIRaiser** and **Player**. In pre flop situations **PlayerIRaiser** chooses randomly whether to raise or to call since the only tool of assessment is power ratings and for that five cards are needed. In every situation after pre flop, the rating for the player cards combined with the available common cards is compared to a "constant" rating. If the cards are rated less or equal to a pair **PlayerIRaiser** folds. Otherwise he calls.

Player uses similar betting decisions but with different constants. He folds when the rating is worse than two pairs and in pre flop situations he raises more often.

1.2 Phase II

Code structure

The first new feature in phase 2 is pre flop rollout simulations to find a measure for the quality of the two sole hand cards. To calculate the pre flop propabilities and to store them in a file we added two more classes: **RolloutSimulation** and **PreFlopPropability**.

As recommended we calculate the pre flop propabilities only for the 169 equivalence classes. **PreFlopPropability** stores the information for one prototype of each of the equivalence classes. Additionaly it is able to encode the information to a string and to parse it from a string. Thus it is easy to read and write the table containing the propabilities.

The class **RolloutSimulation** is used to extract the values from the table and has the ability to calculate them newly. When it is instantiated it checks if a file with the propability information exists. If yes, it is read line by line and instances of **PreFlopPropability** are created and stored in a map. If not, a new table is generated with the help of many rollout simulations. For each rollout a shuffled **Deck** without the players cards is created and the unknown cards are fed from the deck.

To access the table the procedure *double GetPropabilityFromList(Hand, numberOfPlayers)* is used. The parameter *Hand* provides a list of cards that specifies the hand for which the pre flop propability is required. Inside the procedure it is mapped onto the prototype of the fitting equivalence class. The other parameter indicates the number of players and by that the required column of the table.

The second new feature of phase two is handstrength calculations. For that we added the class **HandStrength**. It provides a static procedure *calcHandstrength(playerCards, commonCards, numberOfActivePlayers)*. It being static makes it independent from the context. Thus it can be used easily by the AI players that profit from the handstrength calculations, as well as the opponent modeling classes in phase three. The three parameters suffice for the calculations and can be easily extracted from a **PokerGame**. After encountering performance issues we added buffering for the handstrength results using a *HashMap*.

To include the new methods in the poker simulation two new players were equipped with an instance of **RolloutSimulation**, which they use to consider the propability in their betting decisions. Only one of them uses the *calcHandstrength()* methode to be able to assess the profit.

Betting decisions

We introduced two new Players that profit from the new features: **PlayerAI** and **PlayerAIHandStrength**. While **PlayerAI** only uses rollout simulations to calculate the betting behaviour, **PlayerAIHandStrength** also employs handstrength calculations. Therefore the betting decisions of **PlayerAIHandStrength** depend strongly on the newly revealed common cards during the later rounds.

PlayerAI uses a constant threshold for betting. If the pre flop rating is below 0.3 the player folds. If it is higher or equal, he bets an amount proportional to the pre flop rating or, if that is below the current bet he calls. **PlayerAIHandStrength** considers the ratio between the amount of credits in the pot and the amount needed to stay in the game, the pot odds, strongly. Only if the hand strength rating is larger than the pot odds the player stays in the game. If the rating is larger than the pot odds plus a constant amount, the player raises. If it is only larger he calls. The pre flop betting behaviour is carried over from **PlayerAI**.

1.3 Phase III

Code structure

To implement opponent modeling, we introduced one class **Context** that stores the game situation at a point of time. The equals method is used in a way that all contexts that differ in a certain range are equal. Thus the way of implementing the equals method effects on how narrow or wide a context is. The class **OpponentModelTable** is instanciated for each player. It records the players actions and if the player reaches showdown the hand strengths are mapped onto the context bins. For that purpose an additional class **ContextAvarage** is used that helps with calculating the average. As stated before each player is equipped with an **OpponentModelTable** a context can be created out of a game situation that can be used to query the map, to fetch the average. The AI players that use the opponent modeling technique can access each players **OpponentModelTable** by extracting the active players from the game and using their **OpponentModelTable** object.

Betting decisions

We introduced two new AI Players in phase three using opponent modeling in their betting decisions. The first one is **PlayerAIModeling**. For each player it calculates the average estimated hand strength for all actions in the current round. If the players own rating is higher than each of these avarages of estimated hand strengths, in others words the player thinks he should win, the player raises a big amount. If the player does not win against all opponents, but the hand strength is still bigger than the pot odds plus a constant summand, he raises a small amount. If the hand stength is only a little higher than the pot odds the player calls and otherwise if it is lower he folds.

PlayerAIModeling calculates the average estimated hand strengths in the same way as the first player, but uses the results in a different way. This player folds if he looses against all opponents if the number of opponents for which results from the opponent model are provided is at least two. Otherwise he behaves in the similar way as **PlayerAIModeling**. The constant that is added to pot odds to differentiate between raising and calling is bigger and the raised amount is linear to the current bet in the game.

2 Opponent Modeling

In this section we explain our opponent model and the contexts on which it is based.

2.1 Model Details

We have one model per player, which every other player can access. The model contains a mapping for *context* → *handstrength* pairs played by the player. The pairs are only recorded if the player is part of the showdown at the end of a single game. If a context appears multiple times, the average of all recorded handstrengths is used. All context-handstrength pairs are saved in a *HashMap* for very fast access to the data.

2.2 Context Details

For our opponent modeling we use quite specific contexts. A context is implemented by the **Context** class. Our contexts carry the following information to identify them:

- The game-state (PreTurn, PreRiver, etc.)

- The action the player playen (raise, call, fold)
- The player this context is valid for
- The common cards' values on the table, the suite is ignored
- The current pot-odds, devided into four different bins
- The number of raises that have been played in this betting round

Having the game-state and the current action in the context seemed natural, as they cleary differentiate the different situations inside a game and should thus be treated by different contexts. As we expect to have different behaviour per player, we included the player into the context.

The common cards on the table are one of the main differences between every round, thus it is essential to have this information present in each context. Because there are many possible combinations of suites for the same values of cards, we ignore the suites for the common cards. This is a basic form of equivalence class seperation.

As the pot-odds is a double value, using the exact value would lead to almost infinite numbers of contexts, which is not very useful for the AI to base decisions upon. Therefore the pot-odds is discretized into four different bins: pot-odds < 0.1, pot-odds < 0.2, pot-odds < 0.3 and pot-odds ≥ 0.3. Because the pot-odds provide a lot of information concerning the risk the player has when choosing an action, we think that it can be benificial to use it to differentiate between different contexts.

We also included the number of raises that have been made in a round into the context as we believe that the number of raises is a good indication of the general game situation, many raises implies that many players believe to have many cards. This can influence the players decision on which action to take and should thus be included in the context.

3 Simulation Results

In this chapter we present the results of our simulations. For each phase we ran five simulations of 2000 hands each.

3.1 Phase I

The results in phase I are very random. This was to be expected as the betting behavior of the phase I players is mainly random.

Table 1: Phase I Results - 5 simulations of 2000 hands each

Simulation	1	2	3	4	5
Player					
Phase I - raiser - 1	456	-5212	-17941	22541	-1153
Phase I - raiser - 2	7700	-1936	28227	6591	1720
Phase I - raiser - 3	-20680	-537	9323	-18126	-18186
Phase I - simple - 1	-6776	11900	2034	-39223	9172
Phase I - simple - 2	8223	-633	-6517	14217	4860
Phase I - simple - 3	17031	2360	-9185	19965	9534

3.2 Phase II

In phase II we can clearly see that the players using rollout simulation results are much better than the random players. Adding handstrength calculations results in an even greater gain and makes the player always win against his simpler opponents.

Table 2: Phase II Results - 5 simulations of 2000 hands each

Simulation	1	2	3	4	5
Player					
Phase I - raiser	-19305	-8125	-15256	-17774	-8243
Phase I - simple	-6707	-13404	-11603	-4350	-15067
Phase II - only rollout	8627	6346	2651	3535	10016
Phase II - rollout + HS	21335	19145	28160	22528	17247

3.3 Phase III

In Phase III players using an opponent model were added. This did not lead to an improved performance compared to the Phase II players.

Table 3: Phase III Results - 5 simulations of 2000 hands each

Simulation	1	2	3	4	5
Player					
Phase I - simple	-41870	-62525	-46869	-77551	-44061
Phase II - only rollout	-4884	16297	2030	4480	-3097
Phase II - rollout + HS	42133	28763	40565	42673	36456
Phase III - Modelling	3067	23382	24950	31286	32294
Phase III - Modelling V2	6468	-977	-15717	4050	-16663

4 Discussion of results

In this section we discuss the results of our simulations in the previous chapter. We give reasons for the performance of different strategies used by different players.

4.1 Phase I

In phase I we used two basic players. A very basic simple player who used some random numbers and the powerrating of his cards to decide when to raise or to call. The second player is very similar but uses a more aggressive strategy, which means he'll try to raise more often and doesn't fold as easily.

Interestingly it is not possible to see a clear distinction between the two players looking at the results. Since especially for the first round the random numbers are the only changing number, this could be the reason for optaining very random results over the course of many simulations.

4.2 Phase II

In phase II we added two new players. One uses the rollout simulation results as his only resource for reasoning about the bets he is about to make, the other also uses the handstrength-calculations which we implemented. We notice two things:

1. The rollout simulation clearly adds a lot of benefit to the players reasoning, even if using it for the late game phases.

2. The handstrength-calculations seem to provide a very good indication about how good the players hand is, as the handstrength-player always wins.

Because the phase II players use real game knowledge it seems reasonable to believe that it provides a real advantage over basic random numbers. The information is especially useful because the player can now fold more often and with a good reason, thereby he doesn't loose a lot of money because of staying in the game to long.

4.3 Phase III

In phase III we added two players making use of an opponent model. The results indicate that the current model, or how it is used, does not help in winning games of poker against less "evolved" computer players. The phase II player using the rollout-simulation results and the handstrength calculations is still much better. We see a number of possible reasons for this:

1. We use too broad or too small contexts

2. The players draw the wrong conclusions from the information gathered in the model

3. The information gathered by handstrength-calculations is more valuable than our model's data

Interestingly the first model player is much better than model player V2, although they use very similar reasoning paths. Small changes in the limits of when a player folds/raises/calls seem to have a large effect on the player's performance compared to the other players. A lot of fine-tuning is necessary to result in a very good poker player.