Poker Project

Jazib Ahmed
Colombe M'boungou

● Abstract

We have implemented a reflex agent and also a learning agent that learns statistics about the opponent's move. Both are utility-based : they choose the action that maximizes their expected utility. During the tournament, we played with our reflex agent and it ranked 2nd in the group.

● Introduction

This is a 5 card poker game in which each player gets a hand of five cards and it consists of two betting rounds and a showdown at the end of second betting round in each round of game.The player who has a better hand at showdown or if one player folds the other one,will win the pot and get all the money put into the pot by both players.

○ The poker game, rules, and settings

There are several rules for this 5-card poker game which are:

- Each game consists of two betting rounds.
- At the start of each game each player has to put a fixed amount of money (ante) in the pot to be able to play the game.
- At the start of each betting round the player gets to open (opportunity to bet first) or check(give the opportunity to the other player to open first).
- If every player checks, the betting round will end .

- If any of the players opens the other will have to either Call the amount equivalent to which the other player has put in order to stay in the game or Raise the amount by putting more money in the pot than the other player or Fold (in this case the player loses all the money he put into the pot) if he/she is left with not enough money to call and have a weak hand or All-in (in this case the player wish to continue with all the money left) if he/she wants to put all the money remaining in the pot.
- If one player raises the other will get a chance to Re-raise that.
- The betting continues until one of the players Folds or goes All in or any player calls.
- After the end of first betting a draw phase will get started in which each player gets a chance to throw some or all of his/her cards and get new ones instead in order to have a better hand.
- After the draw phase, the second betting round starts and it goes the same as the first round.
- After the second betting round showdown phase will get started and the player who has a better hand will win the pot.
- There are some special cases when any of the player goes all in the remaining the other two will keep the money equivalent to all-in in the main pot and they shift remaining and the next bets in the **Side Pot** continue their game and at the end the winner of the side pot will be among the remaining players and if that winner has a better hand than the player who went all in he wins the main pot as well otherwise the player who went all in will win the main pot.
- There can be several side pots in a single game.

Unlike chess, a 5-card poker game is a **partially observable** environment.
It's a multi-player stochastic sequential game with static environment and a discrete number of actions.

○ What types of agent and AI methods can be applied?

Search : In the laboratory 2, we did solve a poker problem using **search** algorithms only because we perfectly knew the strategy of the opponent and the cards. Here we do not know the next move of the opponent so the action sequence resulting from a search will be useless if the opponent's next move is not the one appearing in the sequence.
The randomness and partial observability of the game also disqualify the classic **MiniMax** algorithm. The min-nodes assume the opponent will act rationally knowing its hand but we cannot implement this since we ignore this hand. One solution could be to apply minimax with random hands from a relatively large set (from 100 to 1000 samples) and take the action that maximizes the sum of utilities for all opponent hands. However, the game environment adds a **real-time constraint that makes this strategy impractical**. Indeed, if the agent's response time exceeds 2 seconds, it is disconnected and loses the whole game. We would have to run minimax 1000 times over a tree with an unknown depth in less than 2 seconds.

Using Utility theory and uncertainty, one agent can make rational decisions under uncertainty as long as we model the environment correctly and provide a suitable utility function.

Bayesian Networks can help us learn the dependence between the opponent's actions (observable) and its current hand (unknown until the showdown), hence predict the opponent’s hand.
Reinforcement Learning : **Q-learning** could be a good fit because it finds the optimal policy even if the environment is stochastic and partially observable and it does not require any Transition or reward model. The state should at least contain the number of chips of the player, the number of chips in the pot, the player’s hand, the previous opponents’ actions.
The issue comes from the fact that the action-state space can be large and complex which may make it hard to converge and to implement under the time constraints.

○ Are any relevant work being applied to this problem?

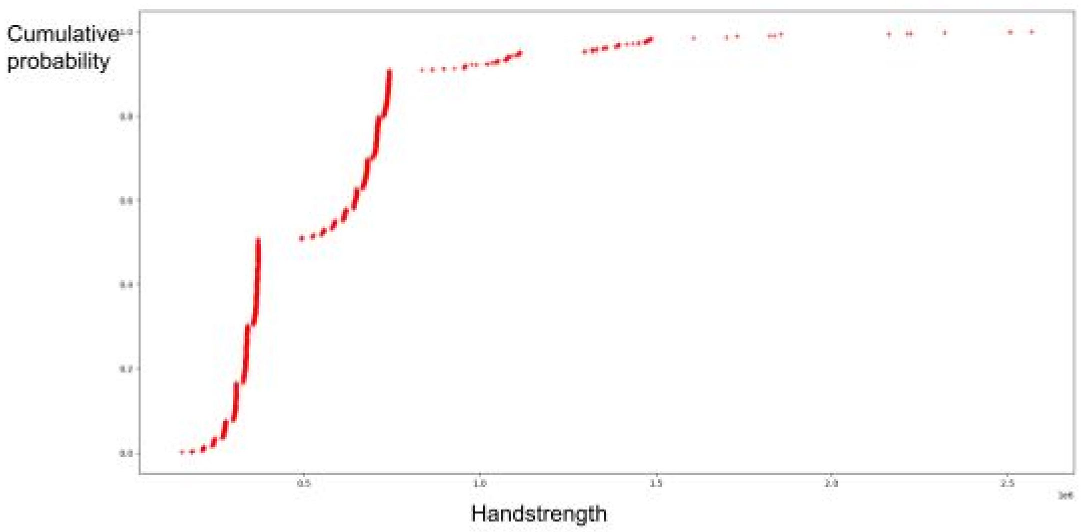
Texas Hold’em Poker have been studied by AI and Game theory experts since at least the 2000s. Even though the 5-card poker game has a different card dealing mechanism, some techniques can still be transferred from one to the other.
One common approach is to represent the remaining possible game states with a **game tree** and tree-search through it to find optimal decisions using Monte Carlo Tree Search, heuristics or Expectiminimax. One can also limit the depth of search to meet the real time constraints and use **machine-learned evaluation functions** to assess the value of the node instead of expanding it.
Opponent modelling enables to predict the opponent’s action and then exploit their weakness. **Artificial Neural Networks** and **Bayesian Networks** have proven to be successful at this.

● Method

○ Strategy and methods employed

The Utility-based agent is a good fit because it maximizes the utility function which, here, is the agent's number of chips.
The main idea of our strategy is to **estimate the probability of winning** at showdown and act **to maximize the expected utility**.
Since the dealing card phase, we know the agent's hand but the opponent's hand is unknown until the showdown, when it is too late. How can we estimate the probability of winning if we only know the agent's hand. To liberate ourselves from this lack of information, we have created an objective strength function so that, h1 and h2 being random hands, h1 > h2 only and only if Strength(h1) > Strength(h2). The hand strength function considers first the order of the combination (High Card, Pair...Straight Flush) and then the card ranks. The hand is ordered by ranks.
$Strength(h) = Value\ of\ Combination(h) + c1 \cdot b^4 + c2 \cdot b^3 + c3 \cdot b^2 + c4 \cdot b^1 + c5 \cdot b^0$
b is the biggest card strength, which is the Ace's.
The value of combination is chosen so that the weakest card with this combination has a bigger score than the strongest hand with a lower order combination.

Since, the opponent's hand strength can be modeled as a random discrete variable, we do not need to know the opponent's hand to approximate the probability of winning. This way, **evaluating the probability of winning can be reduced to computing the probability** that the agent's hand strength is stronger than a random hand strength. To do so, we have built a cumulative probability distribution of hand strength by taking 1048 samples from the game.



As we can see on the cumulative hand strength distribution, we can distinguish clusters that form the hand combinations. On the left, High Cards and Pairs are the most represented. Then this look-up table is stored in the code as a global variable in Client.py and the agent computes the strength of its hand and looks in the list for the probability of winning.

To compute **expected utility** of the action a, EU(a), we look one step ahead by **arbitrarily assuming** that the opponent will call after we call or raise.
EU(Fold) = -_playersCurrentBet

EU(Call) = p_win*(POT + _maximumBet - _playersCurrentBet) - (1-p_win) * (_playersCurrentBet + _maximumBet)

EU(Raise) = p_win * (POT + _minimumAmountToRaiseTo - _playersCurrentBet) - (1 - p_win) * (_playersCurrentBet +_minimumAmountToRaiseTo)

EU(All-in) = p_win * (POT + _maximumBet - _playersCurrentBet)
- (1 - p_win) * (_playersCurrentBet + _playersRemainingChips)

Then :
best action = ***argmax(EU(a), a)***
This expected utility encapsulates the greediness of the goal by expressing the fact the agent prefers to win chips instead of losing them.
On one hand, assuming that the opponent will always Call makes the agent overestimate the utility of Raise, Call and All-in then the agent might mistakenly choose them over Fold. But on the other hand, assuming that the opponent will always Fold makes the agent overestimate the utility of Fold, which might make it lose a lot of chips by folding.

Then we tried to **learn the other agents' moves** during the game in order to anticipate the opponent's next action. In a dictionary, we count the frequencies of all actions so that we can estimate (a priori) the probability that the opponent will call, raise, fold or all-in and apply them to compute the expected utility of the agent's action.
Now:

EU(*Call*) = p(opponent Folds) * EU(*Call*| opponent Folds) + p(opponent Raises) * EU(*Call*| opponent Raises) + p(opponent Calls) * EU(*Call*| opponent Calls)

Our throwing strategy discards the weakest cards that are not involved in a particular combination (Pair, 2 pairs …). This way, the quality of the hand can only increase.

○ PEAS description

A performance measure could be the number of chips at the end of the game.
The environment comprises the dealt cards, the opponent(s), the player's stacks. The environment is partially observable, episodic, static, discrete and deterministic even though the card dealing is stochastic.
The actuator is the decision made by the agent. The decisions that are available are *Open* and *Check* during the opening phase then *Fold, Call*, *Raise* and *All-in* depending on whether

the number of chips is sufficient. Lastly, the agent chooses which cards to throw during the drawing phase.

The input is basically everything the game server sends to the agent by the communication port : its own hand, the number of chips, the actions of the opponents, the ante and the showdown result.

The game state comprises every player's hand, number of chips and last actions and the common pot.

○ Expected behavior

The agent should only fold when the expected utility of *Call* is lower than *Fold*. And it should only *Raise* when the expected utility of *Call* is lower than *Raise*.

The action picked by the agent should in most of the cases give him the best utility it could get with its hand without taking too much risk.

● Experiment and tournament result

❍ What have you observed when playing against a random agent or any agent you have developed?

The expected utilities of call and raise get bigger when opponents put money in the pot.

When we played the reflex agent against itself, we saw that both were caught into a feedback loop of raising. Moreover, because of the current throwing strategy, the agent sometimes discards hands that have almost a Straight or a Flush.

❍ Observations

The probabilities of winning and utilities computed by the agent look sensible.

For instance, ['6h', '9h', 'As', '9d', '4d'] gets p(win) = 81,9% while ['Jh', '2s', '8d', '4h', '6d'] gets p(win) = 4,3%; which makes sense because the first has a pair and Ace the second only has low-rank cards.

The Reflex Agent raises or sometimes All-in when it has a good hand. It folds when it has a weak hand.

○ Observation from the pre-tournament

We have developed an Utility-based reflex agent and a Learning agent but we did the tournament with the Reflex Agent. In the tournament, the Reflex Agent sometimes got disconnected because of an I/O exception which is weird because there is no heavy computation in the code.

In the end, our reflex agent wins 6 times and gets second place at the pre-tournament.