Pokerbots Final Report

Solving Permutation Hold'em

libratus fan club

January 29, 2020

Contents

Abstract

Synthesizing techniques from inference and machine learning, we develop and optimize an approach to solving Permutation Hold'em, the poker variant introduced as part of MIT's annual Pokerbot competition.

1 Introduction

Pokerbots 2020 centers on the game of Permutation Hold'em, a variant of the popular Texas Hold'em in which the relative ordering $2 < 3 < 4 < \ldots < T < J < Q < K < A$ of suits is permuted. Such a permutation is generated at the beginning of a 1000-hand game according to the following algorithm:

Thus, in order to play Permutation Hold'em optimally, two distinct objectives must be considered: inferring the permutation P', and determining an optimal strategy for playing Texas Hold'em. In the sequel, we discuss our attempts to accomplish both objectives.

2 Permutation Inference

In order to determine the true permutation, information must be gleaned from "showdowns," situations in the game of Hold'em in which all cards on the board are revealed and both players

Algorithm 1 [`generate()`: Generate permutation of ranks]

$P \leftarrow [0, 1, 2, \ldots, 12]$
$P' \leftarrow []$
while P is not empty **do**
  $i \leftarrow$ Geometric(0.25) $\pmod{|P|}$
  append $P[i]$ to P'
  remove $P[i]$
end while
return P'

show their hands. The strengths of these hands are compared by the game engine with respect to permutation P', thus determining the winner of the showdown. However, showdowns often provide ambiguous information about P': multiple explanations may exist for why one player wins against the other, such as the presence of unexpected straights. Therefore, we had to capture the information provided by showdowns to conduct inference in a fashion distinct from direct construction of P'. This led us to explore methods that could deduce the showdown indirectly via the rejection or acceptance of randomly generated new permutations, as we explain in the following.

2.1 Method 1: Particle Filtration

Our first attempt at inference was a form of Particle Filtration, in which an initial list of permutations was generated from the prior distribution, then narrowed down by iterated deletion of permutations which did not agree with the results of all seen showdowns. The pseudocode is outlined below; we let P' be the true permutation.

Algorithm 2 [`particle1()`: Particle Filtration, Version 1]

$L \leftarrow []$
`threshold` $\leftarrow 100$
for $i \in [1, 10^6]$ **do**
  $Q \leftarrow$ `generate()`
  append Q to L
end for
for s in `showdowns` **do**
  for Q in L **do**
    if result of s assuming permutation Q is different than permutation P' **then**
      remove Q from L
    end if
  end for
  if $|L| < 100$ **then**
    break
  end if
end for
$Q' \leftarrow$ randomly chosen element of L
return Q'

Unfortunately, this initial attempt at a particle filter did not work especially well, due to the fact

that even with an initial permutation list of size 10^6, the proportion of permutations considered was still extremely small—equal to

$$\frac{10^6}{13!} \approx \frac{10^6}{6 \times 10^9} \approx 0.016\%$$

of the total permutations at hand. Therefore, our particle filter almost never returned the correct permutation. Furthermore, showdowns did not decrease the size of the list L of potential permutations significantly, causing our algorithm to run extremely slowly and sometimes even exceeding the computation time limit of our pokerbot. Therefore, we needed an algorithm that improved on two fronts:

- returning a permutation equal to (or at least highly similar to) the true permutation, without running out of candidates
- decreasing the amount of time required to converge on a permutation.

Thus, after the third lecture, we implemented a particle filter that sampled around 10^5 initial permutations. If the number of remaining permutations dipped below 200, we would take every possible permutation of swap distance 1 from our remaining and add them to the list if they pass all observed showdowns. This method worked maybe 40% of the time but it still gave a solid sample of permutations anyways. As for implementing it into the bot, we only replaced the original permutation ($\pi(i) = i$) if we had observed more than a threshold number of showdowns (on the order of 10^2).

Unfortunately, the modified particle filter often still failed to find the correct permutation. The issues were very similar to the ones we had before: we were often unable to reach a permutation sufficiently similar to the true permutation through our single swaps, thus causing our list of potential permutations to collapse after some number of permutations with a high probability (≈ 80, empirically); furthermore, our algorithm was still running far too slowly. Therefore, we had to find a new method by which to deduce the permutation—leading us to MCMC.

2.2 Method 2: Markov Chain Monte Carlo

After our failed attempts at using Particle Filtration to deduce the correct permutation, we turned to Markov Chain Monte Carlo methods. The idea of MCMC is simple: suppose we have a graph, where each node represents a single permutation. Each node is connected to a certain set of neighbors, which we determine manually. We take a random walk on this graph of permutations, selecting a neighbor at random and then either staying at the current node or jumping to the chosen neighbor with a probability α, which we are able to calculate. Specifically, we used the Metropolis-Hastings algorithm to guide the determination of our jump probability α.

Now, the question at hand was how to determine a good topology for our graph on permutations. Our goal was to make α small when considering a jump from a likely permutation given the information encoded by the prior distribution of permutations, along with the information gleaned from the showdowns, and vice versa when considering a jump from an unlikely permutation and a likely permutation. We did not want to waste time walking on unlikely nodes, but at the same time, we did not want to run into the same issue of not being able to reach the correct permutation due to potential disconnectivities of the permutation graph. Therefore, given the node v we are on and the candidate node v' we must decide whether or not to jump to, we set α equal the following:

$$\alpha(v \to v') = \min\left(1, \frac{P(v')}{P(v)} \cdot (\lambda)^{W(v')-W(v)}\right),$$

where $P(v)$ is the prior probability of generating the permutation which node v represents, $W(v)$ is the number of incorrect showdown results, and λ is a parameter we had to tune, which we initially set to 0.05.

Additionally, we had to determine what permutations were connected, and how many iterations of the random walk to run per showdown. Initially, we connected two permutations if they were a swap distance of 1 from each other, and ran the random walk for 1000 iterations. We then returned the permutation which was visited the greatest number of times in the last 750 iterations, where both stays and jumps were counted as visits to a permutation.

MCMC performed phenomenally well compared to our particle filter, allowing us to consistently determine the permutation. In the cases where we did not arrive at the exact permutation, we returned a permutation that was sufficiently similar to the true permutation that the discrepancies were negligible, such as a single swap of two indices that may not have been resolved by any showdowns within the 1000 hands that were played. As a result, our pokerbot improved significantly on the inference front.

In the final days, we made a few small optimizations to our method, finalizing our pokerbot with parameters:

- MH iterations: 5000
- λ: 0.07

and a graph with permutations connected iff they were a reinsertion distance of 1 apart. Thus, we succeeded on the inference front—what was left was to play poker.

3 Playing Texas Hold'em

Initially we considered how a Nash equilibrium strategy in the first few rounds would differ from a true Hold'em equilibrium strategy. However, upon learning that the permutation was generated based on a distribution that favors keeping low cards low and high cards high, we noted that playing Hold'em without straights would be a good idea. Only until the permutation is essentially determined will either player have information to play straights correctly. If the probability of a straight holding is too low, then it could essentially be ignored. We didn't know how many showdowns it would take to determine the permutation, but our initial guess was that it may not even be possible by the end of 1000 rounds.

3.1 Method 1: Aimbot

Aimbot was a strategy that evolved throughout the first week, performing at a top level until the first mini-tournament. It began as a bot that would shove good cards pre-flop and eventually evolved into a strategy that would utilize a lot of if statements. The if statements were determined based only on our holdings, the board, and the pot odds. We applied our knowledge from 6-max hold'em cash games to develop a bot that would be very aggressive early to exploit over-folding while tightening up when it came to putting stacks in due to under-bluffing from opponents.

We believe we would have been top 3 in the first mini-tournament had we not been plagued with build timeout issues.

We submitted a slightly optimized version of this bot at the end of week 2 as we began implementing CFR but had no bot for it yet. Again, we hit a build timeout issue so we weren't able to determine our true rank.

3.2 Method 2: Counterfactual Regret Minimization

We started to study CFR on the second week using a variety of sources. To learn exactly how it works, [1] was useful. A lot of useful papers came from [2]. and [3]. Also, [4] was useful for implementation. Finally, the first advanced topics lecture as well as office hours were of great help. We began by writing Monte Carlo CFR with external sampling for Kuhn Poker, and after seeing it generate an equilibrium, we decided to extend it to Heads up no limit hold'em. For the hold'em version, we also used linear CFR, which puts more weight into later iterations of CFR. We implemented it in C++, starting from the skeleton code which already had round states implemented. It took well over a week to slowly eliminate bugs one by one until it finally printed reasonable looking hold'em strategies, although to this day we are still uncertain whether it is actually correct. The time factor was furthered by the fact that we needed to generate our abstractions, which required creating a fast hand strength equity lookup table since we needed to use this many times in each iteration of CFR. For abstractions, we decided on the following:

- Preflop: 169 strategically different hands
- Flop: Hand strength vs random range histograms (every 2 percent), grouped by closest distance to centers generated by kmeans++ using earth mover's distance metric
- Turn: same as flop
- River: Opponent cluster hand strength vs 8 different ranges (from one of the ualberta papers), again 100 centers generated by kmeans++, except using Euclidean distance

100 buckets on flop, turn, and river were chosen rather arbitrarily and in hindsight should have been tuned, probably in a way where number of flop buckets is less than number of turn buckets which is less than number of river buckets. It did, however, combined with our action abstractions, lead to a good number of information sets (around 10^5) that allowed for a good number of CFR iterations to run.

Implementation wise, this required beginning with fast equity lookup tables. Without lookup tables, our other option was monte carlo simulations, but getting accurate results required too many iterations, and we would rather use lookup tables that only take a few hours to generate (they also ended up being only around 300 mb total). For 7-card lookups to compare showdowns, we initially used the OMPEval library (which is still present in our final version of Aimbot), but we realized that the 100 mb two-plus-two hand evaluator was a much better option. We decided to create similar tables for hand strength on flop, turn, and river by enumerating all boards and opponent hands that don't conflict with ours and counting the number of wins, ties, and losses. We stored these equities as floats in lookup tables.

With equity, a baseline CFR, and a round state structure out of the way, we could finally compute our card abstractions. Preflop doesn't require anything special: just check whether the cards are suited. For flop, we generated 500000 boards, created hand strength histograms for all of them (each bar spans a range of 2%), and then implemented kmeans++ with Earth Mover's Distance metric to cluster them into 100 centers. The same exact thing was done for the turn, and for the river we also did the same thing except instead of hand strength histograms we calculated equity against the 8 chosen ranges and used L2 norm metric.

Next, before we could run CFR we needed to create action abstraction and build a game tree. We initially decided on using bet sizes of $1/2$ (pot), $1, 2$, and all-in while using raise sizes of 1 and all-in in with a max of 4 actions per street before the next raise would be mapped to a call, fold or

all-in, but we eventually realized that extending raise sizes to include all bet sizes and no limit on number of actions per street didn't increase the game tree by much.

Finally we needed to bucket each node. This involved the card abstraction and the action history. At first we decided to map the action history to a tuple of (pot size, my chips put in pot this street, opponent chips put in pot this street), but for the purposes of later action translation we changed it to just (pot size, pot odds bucket), where pot odds were put into 6 different buckets $[0, 1/6), [1/6, 1/4), [1/4, 1/3), [1/3, 2/5), [2/5, 4/9), [4/9, 1/2)$. This allowed our final information set count to be around 12000.

We ran a multithreaded version of this CFR on a 96 core AWS EC2 instance (c5.24xlarge) for a couple of hours to hit around 400 million CFR iterations.

Finally, we needed to map opponent moves that would leave the game tree to states that CFR generated strategies for. We followed the advice of [3] and used a pseudo harmonic mapping of an action to the nearest bet size. We then updated an imagined pot size this way, and made sure that all of our actions were legal. We also had logic to care of edge cases (for example, if we map a raise to a call then we aren't allowed to raise anymore even if we have a strategy for this abstracted state). Of course, we don't know if these are the best ways to process our opponent's moves, and if we had more time we would fine tune this. Furthermore, we encountered an extremely large number of bugs in this step and fixed them one by one until our bot stopped running into problematic situations that we didn't account for versus random bots on the scrimmage server.

Another final note is that we don't actually sample an action purely based on probabilities produced by CFR. Due to inaccuracies with our abstraction as well as potential inability to converge due to changed made to speed up CFR that make it lose theoretical guarantees, we decided that it was best to throw away actions with low probabilities in the strategy. We are also experimenting with only using the action with the highest probabilities, as well as other similar methods. This process is still being tested and we have no idea what we will actually use in our final submission.

4 Implementation

Everything was implemented in C++, with the exception of using python to test some things.

Our code may be seen at the Github repository here (private until 1/31).

5 Final Comments

Parameters that still need to be fine tuned at the time of writing the report: Abstractions, MCMC thresholds, Action Sampling thresholds, number of showdowns until we use the permutation generated from MCMC.

References

[1] http://modelai.gettysburg.edu/2013/cfr/

[2] http://poker.cs.ualberta.ca/

[3] http://www.ganzfriedresearch.com/

[4] http://poker-ai.org/phpbb/