Foundations of Artificial Intelligence: Final Report

b11902038 鄭博允

June 12, 2025

Methods

All-In

This agent always goes all-in regardless of the round state, but I added a simple mechanism to prevent it from being overly aggressive.

Here’s the mitigation logic:

1. Evaluate the hand strength using the provided `HandEvaluator`, and identify the last action made by the opponent.
2. If the opponent has been raising frequently—indicating a possibly strong hand—and our hand strength is below a `FULLHOUSE`, we fold.

This agent relies heavily on luck. If the cards are strong, it wins big; if the opponent has better cards, it loses big.

Monte Carlo

The Monte Carlo agent is straightforward: it performs a Monte Carlo simulation for each decision to estimate the win rate of the current hand.

The process is as follows:

1. Run 1,500 iterations of simulation by randomly completing the community cards and generating possible opponent hole cards.

2. Compute both the **expected value (EV)** and **pot odds**.
3. If **EV** $>$ **pot odds**, then call or raise; otherwise, fold.

More simulations improve the accuracy of the prediction. However, since we are limited to 10 seconds per decision, we cap the simulation count at around 1,500.

While this agent is simple and effective (compared to training a full RL agent), it only considers the current hand and ignores other contextual factors like opponent behavior or overall pot size. In short, it calls when the cards "look likely to win", raises when they are "very likely to win", and folds otherwise.

Monte Carlo CFR (MCCFR)

The above agents make decisions in real-time, so their complexity is limited by computation time. To overcome this, we use training to "pre-compute" strategies based on game states. Counterfactual Regret Minimization (CFR) is a fundamental algorithm for computing approximate Nash equilibria in imperfect-information games such as poker. However, traversing the entire game tree is computationally infeasible, so we adopt Monte Carlo CFR (MCCFR), which samples only relevant parts of the game tree.

The key idea is to simulate game states using Monte Carlo methods and calculate the "regret" for each action—i.e., the difference in payoff if a better action had been taken. These regrets are accumulated into an **InfoSet**, which maps a game state (InfoKey) to a distribution over actions. The composition of the InfoKey is critical, as it defines how the agent interprets the game.

I experimented with several MCCFR variants. Below are three representative versions:

v1: Round State + Action Count

This version uses the round state and action counts as the InfoKey. Action count refers to the number of fold, call, and raise actions taken during the current street.

- $l1$: <street>, <hole-cards>, <community-cards>
- $l2$: <fold-count>, <call-count>, <raise-count>, <other-count>

The InfoKey is separated into two layers to provide generalization when certain combinations in $l1$ are unseen during training. For example, if a specific $l2$ is not seen under a given $l1$, the agent averages the strategies from other $l2$ values under the same $l1$.

v2: v1 + Card Strength + Opponent Action History

This version extends v1 by adding card strength and replacing action counts with the opponent's action history. We focus only on the opponent' s actions, assuming that our own prior actions have less influence on the next move.

- $l1$: <street>, <hole-cards>, <community-cards>, <card-strength>
- $l2$: <action1>, <action2>, <action3>, ...

For "call" and "raise" actions, the amount is also abstracted into `HIGH` (> 500) or `LOW` (500), as the bet size might signal the opponent's hand strength.

v3: v2 + Abstracted InfoKey + External Sampling

Training the above agents took around 4 hours per 1,000 iterations, largely due to the size of the InfoKey and time to build it. In addition, storing InfoSet as JSON files consumed a lot of memory (around 250MB).

To address this, I abstracted some of the InfoKey components to reduce state space. For instance, rather than storing actual cards, we store only their rank and strength.

- $l1$: <street>, <hole-highest-rank>, <community-highest-rank>, <hand-strength>, <hand-rank>
- $l2$: <action1>, <action2>, <action3>, ...

This reduces both memory usage and training time significantly—about 30 minutes for 10,000 iterations. InfoSet are now stored in `pickle` format. This version was trained for 500,000 iterations and used as the final model.

MCCFR + Monte Carlo

Since it is infeasible to cover the entire state space, some InfoKeys will be unseen during training. To mitigate this, I added a Monte Carlo agent to calibrate the MCCFR agent's decisions.

- If the MCCFR agent chooses to fold but the Monte Carlo agent estimates a high expected value, we override the action to call.
- If the MCCFR agent chooses to call or raise but the expected value is very low, we override it to fold.

This agent was used for the final submission.

Test Result

Each agent is tested with baseline agents in 10 games, each game containing 20 rounds.

	All In	**Monte**	**MCCFR**	**MCCFR+Monte**
Baseline0	3/10	5/10	5/10	7/10
Baseline1	6/10	5/10	7/10	6/10
Baseline2	6/10	6/10	4/10	6/10
Baseline3	2/10	5/10	6/10	5/10
Baseline4	5/10	3/10	6/10	4/10
Baseline5	3/10	4/10	2/10	5/10
Baseline6	4/10	3/10	4/10	4/10
Baseline7	4/10	6/10	2/10	3/10
Random	9/10	10/10	7/10	5/10
All In	(tie)	6/10	0/10	7/10

Table 1: Matching result of each agent vs. baseline