AI Poker Agent Final Project

Ishita Chakravarthy
ichakravarth@umass.edu

Harsh Seth
hseth@umass.edu

Tirth Bhagat
tbhagat@umass.edu

Soniya Gaikwad
sgaikwad@umass.edu

1 Introduction

The process of creating a poker agent, "PokerMan," that plays optimally against its opponent in Limit Texas Hold'em required researching ideas of how existing poker agents play and implement abstraction, modeling the game tree's states and actions, and developing fine-tuned heuristics that aim to make the poker agent win and quickly bankrupt the opponent by predicting the comparison in potential hand strengths and risk levels given the amount of money each player has. The methods used to account for these goals include a game tree search using the Expectiminmax algorithm because poker is a partial information game and abstraction to create a sense of knowing some information, such as the "PokerMan's" expected hand strength and its opponent's potential hand strength. Some key results that were shown as there were main rounds of experimenting and testing included how PokerMan had a win rate of 84.18% against the honest player, and when playing against itself (self-play), it had a win rate of 49.53%. In the following sections, the design of the optimal poker agent will be further described by discussing the implemented Expectiminmax game tree model, how abstraction provides both players' hand strengths and variance, how the heuristic works with abstraction to choose appropriate actions to beat its opponent quickly, and how the experiments helped improve the parameters.

1.1 Our Contributions

During the development of PokerMan, the team worked together and contributed equally along the way. There were two regularly scheduled meetings weekly, in which everyone discussed the individual research regarding poker heuristics and abstractions as well as progress made. Apart from this, certain aspects of the project have been assigned to the group members to focus on. Abstraction of the state space and research on the same was done by Ishita. Developing the game tree with the states and implementation of the Expectiminmax Algorithm, which was used to pick an action for our agent, was managed by Tirth. Researching, picking essential metrics for the heuristic function, and updating the weights of the heuristic function were looked at by Harsh and Soniya.

1.2 Related Work

Limit Texas Hold 'em is a weakly solved game under incomplete information, i.e., the agent has reached a level of play where it is statistically unlikely that any other strategy would perform significantly better[2]. Modern poker agents use iterative counterfactual regret minimization CFR, which approximates the Nash equilibrium of incomplete games through self-simulations using a regret-minimizing algorithm (regret is the loss in utility when not taking the known best strategy) [2]. The possible number of states in a 2-player limit Texas Hold 'em Poker is $3.162 * 10^{17}$, which would give the possible number of actions as $8.221 * 10^{17}$ [4]. This would make computation to traverse the entire game tree expensive, which brings up the need for abstraction. Abstraction is typically done with Expected Hand Strength, which can be found from a variety of look-up tables. [9] Local Best Response (LBR) determines a lower bound on the exploitability for strategies without the need to introduce a layer of abstraction to the analysis[7]. The bot would then select the actions where it is least likely to be exploited, leading to good performance. This analysis acts as a starting point for developing a heuristic.

2 Model and Strategy: Game Tree, Abstraction, and Heuristic

2.1 Game Tree

To play optimally, the agent must choose a move that will lead to a goal state that will maximize the payoff. This is modeled using a game tree, where each agent can fold, call, or raise, depending on their money. The folded state always transitions to the goal state, and the call actions transition to the other player if played first in a round; otherwise, it leads to nature. The raise action will transition to the other player, but there is a limit of 4 raises in a round. Nature represents the cards revealed in the river and will have a branch for each possible card revealed.

If b represents the average branching factor of a non-goal state and d is the level of the final goal node, to fully represent all the states of the game, we need at least $b^{d+1} - 1$ states. Searching the entire game tree runs in exponential costs, which makes it an infeasible approach even with abstraction. To stay under the 100ms time limit, the agent will traverse the abstracted game tree starting from the current state to level k (where k is determined by computational constraints) and cut the tree off. For all states that do not have a utility assigned at level k, PokerMan will estimate the utility/payoff for both players using a heuristic utility function. In the case of PokerMan, the tree is cut off at nature nodes or gameplay depth of k=3 nodes.

The best move the agent could make are moves that follow the Bayes-Nash equilibrium for limit hold'em poker, which is a strategy profile that maximizes the players' expected utility/payoff for each player given their beliefs and the strategies played by the other players [1]. However, due to the computational constraint difficulties of solving the Bayes-Nash equilibrium using normal-form linear programming or CFR [1], the best move was decided by running the Expectiminmax algorithm (a variant of minimax algorithm). Similar to the minimax, PokerMan will try to maximize its payoffs, and the opposing agent will try to minimize the payoff. However, when nature plays, we want to calculate the expected utility using the probability of action and the associated utility of the state (we assume that nature has no preference for choosing an action; all are equally likely). Also, since the opposing player may have a preference for certain actions (e.g. more likely choosing call over the fold), when the opposing player is minimizing the utility, PokerBot weighs the utility by the opposing player's action distribution which will be learned by watching the opposing player moves (this gives a better idea what the opposing player would pick). Another motivator for the Expectiminmax algorithm is that it allowed us to solve for the good move without storing the entire game tree in memory[8]. Normally, an agent would need to construct a tree and then perform the necessary operation to pick the next move. However, with Expectiminmax, it allows us to get the best move recursively using a DFS algorithm as each parent node returns a utility solely dependent on the children's utility (b^{k+1} states in memory without the overhead cost of full tree construction).

2.2 Abstraction

To reduce the number of states in the game tree, abstraction will be employed. The goal of abstraction is to get a score for each hand, and based on this score, cards with the same score would be grouped, which helps reduce the branching factor of the game tree. This score would have to be recomputed at each round as cards are revealed since the cards in the river would play a crucial part in the calculated score.

Initially, the approach of average rank strength [9] was used, which was later swapped for Cactus Kev's [5] hand strength evaluator, which ranks each hand into 7462 values, where rank 1 represents a Royal Flush, and 7462 represents a high card of 7 (Note that scores of [7, 5, 4, 3, 2] with each card belonging to any suit is the lowest possible hand in a game of poker). Details on Average Rank strength and the decision-making are included in Appendix A.

A hash-table-based Python library that ranks 5, 6, and 7 card hands, with values derived from Cactus Kev's table, was used for the implementation [6]. In the case of 6 and 7-card hands, the output rank is the rank of the best possible 5-card hand. The ranks of the hands are normalized into a range of [0,1], and with the help of the frequency distribution over the ranks, the expected hand strength and variance are calculated. The frequency distributions of each rank can be found in Appendix A.0.3, where details of the distribution are discussed.

With this implementation of abstraction, the nature nodes are modeled with their respective scores. The weights of these branches are assigned as the frequency of the ranks. In the pre-flop round, along with the 2 cards in hand, the abstraction iterates through all possible 3 river card outcomes to generate the distribution. In the flop and turn round, 2 and 1 additional cards are modeled, respectively. This ensures that the agent considers cards that are revealed in the future while making a decision about his current hand.

While the above implementation considers PokerMan's expected hand strength, it does not model the opponent's hand strength. Since the cards in the river are a part of both players' hands, PokerMan's hand strength is not independent of the opponent's hand strength. Similar to the modeling done for PokerMan's hand, the opponent hand is also modeled at every nature node, i.e., the average opponent hand strength and variance are calculated.

Since there is a 100ms cutoff before which PokerMan folds, it becomes important for the abstraction to reduce the tree size to ensure that the requirements are met. For this, 1000 trials are run for each of the rounds, and the average time elapsed is calculated. For the pre-flop round, the average time elapsed for 1000 trials is 79ms. For the flop round, the average time elapsed is 77ms, the turn round is 9.6 ms, and the river is 5.1ms. For further details on speedup and suggested improvements for abstraction, refer to Appendix A.0.4.

2.3 Heuristic

For the poker agent to play optimally against its opponent, it was important to consider two factors - (i) determining if our agent's dealt hand was stronger than the opponent's and (ii) determining if our opponent was attempting to portray their hand as stronger than it actually was (i.e. bluffing via aggression). Based on these two factors, we had to select the action we deemed to yield the best outcomes for the agent - i.e., gave the highest return (pot value). However, given that the agent would never have the knowledge of the opponent's hand, such a decision would have to factor in statistical measures of the dealt hand distribution and account for its variance. Additionally, the "bluffing via aggression" was not a quantitative factor, so we could not employ purely score-based techniques. This informed our decision to develop a heuristic function for our game tree, which would instead provide a decision about our agent's likelihood of victory.

The basic structure of the designed heuristic function accepts information about the player's hands, the community cards, the stack value for both player 1 and player 2 at a given state of the game, and a history of actions taken to arrive at a given state. With this information, the function would return the expected utility, here the winnings or losses for the agent if it resulted in a showdown then and there.

To arrive at a statistically sound decision if we did indeed have the stronger hand, analysis and methods described in the Abstraction section above were employed to arrive at expected hand strengths and the possible variance in the calculated values for both the agent and the opponent. From there, a set of risk levels were identified (zero risk - guaranteed victory, low risk, medium risk, and high risk), and different weights were assigned to each piece of information identified above for each risk level. These weights would also help identify confidence intervals to codify the different risk levels. However, given that the distribution of hand strengths does not follow any standard, well-studied distributions (plots in Appendix A.0.3), parameters were tuned to find the ideal values. Details about the tuning process are specified in the section below.

In cases where the variance between predicted hand strengths would be large or the expected hand strengths themselves would be relatively close by (as determined by the tuned confidence intervals), the player aspects of the risk (i.e. bluffing via aggression) would have to be considered as well. This leads us to encode the game's action history into the decision as well. We distilled it into two components - a relative wealth parameter and a threshold for "risky plays". The relative wealth parameter gives a score to our agent's ability to take a higher risk in a bet (i.e. allow a looser confidence interval) because it can afford to take a risk from an economical perspective. The threshold for "risky plays" indicates if the play at any given state has reached a specific number of overall raises, indicating higher stakes and, in turn, higher potential losses. Both these factors have also been weighted into the decision via tuned weights.

The final heuristic function has the following structure (described in pseudocode)

```
# psuedocode for the heuristic function
def determine_victory():
    return
        we should have a stronger hand than opp (given a confidence level)
        AND (
            we should have enough relative wealth to risk this play
            OR this should not be a high-stakes play
        )

# psuedocode for the utility function
def get_state_utility():
    if determine_victory():
        return expected winnings
    return expected losses
```

—

2.3.1 Fine-Tuning

When determining the player utility and fine-tuning the parameters, the approach was to understand the risk levels that came from both players' hand strengths and their variance. With this information, there is a critical value that was fine-tuned according to the performance against a given opponent.

Given the nonstandard nature of the hand strength distribution and the qualitative aspects of the risk that we were considering in our heuristic, we had to employ tuning of weighted parameters to be able to arrive at the desired behavior of the agent. This was done largely manually, with the exception of runtime tuning of the action probability distribution described in the Game Tree section.

Taking the standard HonestPlayer as a baseline opponent for our tuning process, we put the agent through a sequence of 1,500 rounds with a starting stack of $10,000, running this for each configuration we desired to test. These configurations were sets of weights assigned to each parameter in the heuristic function. Throughout these runs, we recorded and aggregated several key metrics. Of them, the following were found to be most significant - (i) Average Winnings, (ii) Average Losses, (iii) Average Hand Strength in Rounds Won, and (iv) Average Hand Strength in Rounds Lost. We selected configurations that maximized (i) and (iii) and minimized (ii) and (iv). The plots for this tuning process can be found in Appendix B. We found that the configuration labeled 12 performed the best. This configuration performed significantly better in minimizing (ii) and (iv) than the others tested and thus was the basis of selection.

3 Experimental Results, Observations, and Analysis

3.1 Experimental Results

Opp.	# Rounds	Win Rate	Avg. Hand Str (Won)	Avg. Hand Str (Lost)	Avg. Winnings	Avg. Losses
Random	255	0.8980	0.4105	0.0692	216.8778	-611.7692
Fold	3300	1.0	0.4437	N/A	15.0	0.0
Fish	5029	0.4927	0.5542	0.3509	55.6416	-53.6417
Honest	2572	0.8418	0.4610	0.3878	21.7737	-54.5455
Clone	3089	0.4953	0.5468	0.3531	89.5294	-88.4734

Table 1: Key Performance Metrics against Standard Opponents

3.2 Observations and Analysis

While tuning the hyper-parameters, the goal was to build a stable poker agent with low average losses in money and high average winnings. The ideal behavior of PokerMan would be as follows:

- Hands with relatively high hand strength- either calls or raises as the game progresses.
- Hands with relatively low hand strength- fold early

Based on the tuning done, data was collected against 5 players over 10 trials each: Random, Fold, Fish, Honest, and Clone. With these 5 players, the game was played till either player bankrupted the other. Against the random player, PokerMan typically played raise (seen by the high winnings and losing) and wins nearly consistently, winning nearly 216 dollars per game or losing significantly. In fold, the bot wins always since the fold player gives up immediately. Against the fish and clone player, we had roughly a 50-50 chance of winning with similar pot wins and losses. With Honest players, we can beat them be them consistently.

An interesting analysis that should be pointed out here is that of the clone. This shows us the behavior of how PokerMan performs when played against another PokerMan. In this case, the win rate was very close to 50%, with a relatively high average winning per round and losses per round. This would indicate an aggressive player, which was the approach detailed in the heuristics.

4 Novel, Significant Contributions

For abstraction, a significant contribution was in the approach taken to model the opponent's hands in the same way that the river cards were modeled. With the help of this approach, if the initial complexity of calculating the expected hand strength was O(n), the new complexity was O(2n), i.e., there was no exponential increase due to opponent modeling. The importance and complexity of opponent hand modeling show up due to the fact that the river cards belong to both players' hands. For example: In the pre-flop round where PokerMan's cards= [S2, D3] river= [C9, D9, D10], the score returned by the evaluator would indicate a score equivalent to a pair of 9s. However, since the pair of 9s is in the river, it is also a part of the opponent's hand. Ensuring that both hand strengths are included in the heuristic led to an improvement in the agent.

References

[1] Michael Bowling, Neil Burch, Michael Johanson, and Oskari Tammelin. Heads-up limit hold'em poker is solved. *Science*, 347(6218):145–149, 2015.

[2] Michael Bowling, Neil Burch, Michael Johanson, and Oskari Tammelin. Heads-up limit hold'em poker is solved. *Science*, 347(6218):145–149, 2015.

[3] Sam Ganzfried and Tuomas Sandholm. Potential-aware imperfect-recall abstraction with earth mover's distance in imperfect-information games. *Proceedings of the National Conference on Artificial Intelligence*, 1:682–690, 06 2014. doi: 10.1609/aaai.v28i1.8816.

[4] Michael Johanson. Measuring the size of large no-limit poker games. *CoRR*, abs/1302.7008, 2013. URL `http://arxiv.org/abs/1302.7008`.

[5] Cactus Kev. Cactus kev's poker hand evaluator. URL `http://www.suffe.cool/poker/evaluator.html`.

[6] Henry Lee. phevaluator library. 2024. URL `https://pypi.org/project/phevaluator/`.

[7] Viliam Lisy and Michael Bowling. Equilibrium approximation quality of current no-limit poker bots, 2017.

[8] Jonathan Rubin and Ian Watson. Computer poker: A review. *Artificial Intelligence*, 175(5):958–987, 2011. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2010.12.005. URL `https://www.sciencedirect.com/science/article/pii/S0004370211000191`. Special Review Issue.

[9] Luís Filipe Teófilo, Luis Paulo Reis, and Henrique Lopes Cardoso. Speeding-up poker game abstraction computation: Average rank strength. 2013. URL `https://api.semanticscholar.org/CorpusID:15195749`.

Appendix

A Abstraction

A.0.1 Average Rank Strength

Average Rank Strength aims to provide quick computation, along with accurate scores and aims to be a better qualitative metric than existing expected hand strength calculations[9]. The main idea of average rank strength was to use existing TwoPlusTwo evaluator tables and look up the value that is returned in a pre-computed "Average rank strength" Table. Unfortunately, the paper does not have a working link to the pre-computed table as mentioned above, and neither does it provide sufficient detail on how this table was computed. However, this source led to the exploration of the TwoPlusTwo evaluator, which has been detailed below

A.0.2 TwoPlusTwo Evaluator

TwoPlusTwo evaluator tables were chosen initially. This lookup table, of size 130MB, provides a value for each 5, 6, and 7 card hand. With the TwoPlusTwo algorithm, each card is initially assigned a value from 0-52 based on the below equation.

$$TPTIndex(Rank, Suit) = Rank * 4 + Suit * 1 \tag{1}$$

where Rank is 0 for Two, 1 for Three, ... 12 for Ace, Suit is 0, for Clubs, 1 for Diamonds, 2 for Hearts, and 3 for Spades. With the newly assigned card values, the player's hand and the river are iteratively looked up in the tables with the formula

$$T_7[T_6[T_5[T_4[T_3[T_2[T_1[53 + c_1] + c_2] + c_3] + c_4] + c_5] + c_6] + c_7] \tag{2}$$

where T_n is the n^{th} lookup table and c_n is the n^{th} card of the hand. Once the lookup is complete, the score returned is a value between [0,7931]. The major advantage of TPT tables is that the tables provide iterative lookup, that is, once a value is assigned to a 5 card hand, the cards themselves can be discarded, and this value can be used in future rounds.

On experimenting with TwoPlusTwo evaluator tables, it was found that the speedup was achieved. For the pre-flop round, the average time elapsed for 1000 trials is 4.4 ms. For the flop round, the average time elapsed is 4.2 ms, the turn round is 2.8 ms, and the river round is 0.12 ms However, to use the TPT table, it needs to be loaded from a file, which takes 6 seconds. This boot-up operation would have to be done every time the PokerMan agent was initialized. Since the allocated time for a player is 100ms, this approach was discarded. In case the above constraint was not present, TPT tables offer a significant boost in average time performance once the load operation is done and should be preferred.

A.0.3 Distribution over chosen abstraction metrics

For the phevaluator library, which was chosen for abstraction, there are a total of 7462 equivalence classes for 5 card hands, 6075 equivalence classes for 6 card hands, and 4824 for 7 card hands. It is important to look at the distribution of scores across all possible card values to learn details about the significance of scores. Similar to plots generated for Hand Rank in [9], plots were generated for 5 and 6 combinations. Running the computation for 7 card combinations is extremely expensive, but can be generated similarly. Note that for the plots generated in the figure, 1 indicates the strongest hand, whereas in the plots below, 1 indicates the weakest hand. There are a total of 7462 equivalence classes for 5 card hands, 6075 equivalence classes for 6 card hands, and 4824 for 7 card hands.

A.0.4 Suggested Improvements

One suggested improvement would be to use the average rank strength metric as opposed to expected hand strength. The average rank strength does two consecutive look-ups in TwoPlusTwo evaluator tables and a pre-computed "Average rank strength" Table. With this, the values for the ranks used of [1,7462] can be replaced with the newly found values of [0,7931]

Another improvement that can be made is in the comparison of distributions across PokerMan and the opponent. In the naive approach taken above, the weighted average and variance of the distribution

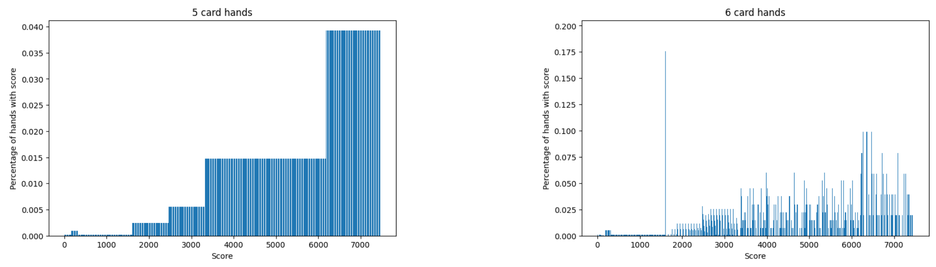

Figure 1: Hand rank relative frequencies for 5 and 6 card hands. All possible sub-ranks are represented in the horizontal axis, ordered by their score.

were compared. Earth-mover's distance [3] can be used to compare the two distributions instead. While clustering the ranks, k-means, along with the earth mover's distance metric, could potentially significantly improve the performance of the abstraction.

B Heuristics

There were 4 hyper parameters which were tuned manually as detailed in the fine-tuning section. These values were selected after considering two plots

- Average winnings and average losses.
- Average PokerMan Hand strength vs Average opponent Hand strength.

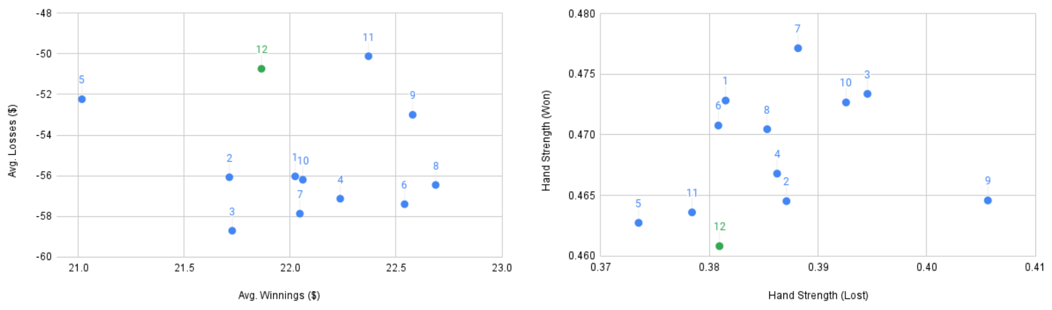

Figure 2: Metrics evaluated to select a most performant configuration (configurations numbered, with selected in Green)

The plot below shows details of how the pot value changes at every round while playing different opponents. It is clear to see that PokerMan does very well against the random player, while it is significantly worse performance against the Fish Player

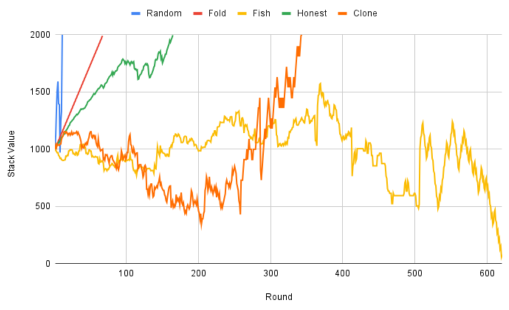

Figure 3: Average Player Stack against Standard Opponents