COMPSCI 683 - Group 13 Project

Sri Rama Bandi sbandi@umass.edu
Mason duBoef mduboef@umass.edu
Amanda Ariceri aariceri@umass.edu
Satya Srujana Pilli spilli@umass.edu

GitHub Repository:

`https://github.com/mduboef/pokerBot`

1 Introduction

We experimented with a number of approaches in order to design a competitive poker agent. Having encountered the shortcomings of the hard-coded agent we ultimately decided to implement a learning-based approach using Monte Carlo Tree Search. The raw un-abstracted game tree would be far too large, so we abstracted our states to significantly reduce the branching factor. Once we designed our game tree, we trained our agent, experimenting with different coefficient values in the Upper Confidence Bound formula to balance exploration and exploitation and optimize our agent's decision-making.

2 Initial Approach

The first implementation of our agent was a hard-coded bot called PokerBotPlayer, which was based purely on heuristics, and well-known best game practices. The bot would first check if it can make a better hand type than could be made using the community cards alone. If so, it was set to raise immediately. If not, it took the sum of the rank of its hole cards. If the sum was 24 or greater it raised. If the total was between 19 and 23, it called. If the sum was below 19 it folded. These thresholds were set through trial and error, while testing performance against the provided random bot. Before folding, the bot always checked to see if it can call and continue playing for free. Despite our best efforts PokerBotPlayer failed to clear out RandomPlayer and was always getting cleared out by RaisePlayer (see Figure 3 in results). Observing this poor performance and hearing Professor Zick mention poker agents "learning" in lecture, we decided that we needed to change our approach. We spoke with Professor Zick at his office hours where he recommended we consider MCTS. Our new approach required a major overhaul of our existing code. However, our state abstraction system did make use of the "haveHand" helper functions we initially developed for PokerBotPlayer.

3 Abstraction

Due to the large branching factor of poker, we had to abstract the state space in order to train our AI within computational constraints. We believe that our state space abstraction system, while not perfect is novel and well-designed, capturing relevant distinctions while maintaining a relatively small branching factor.

3.1 First Abstraction Design

The first abstraction we implemented was for the pre-flop. Using standard hand rankings, we abstracted all possible pre-flop hands into eight distinct buckets, based on the reference from best game opening tactics as detailed in [pok]. They were numbered 1-8 with 1 representing the weakest sets of hole hands and 8 representing the strongest. We sourced these values from a pre-flop evaluation chart we found online, giving the relative value of all card combinations, both suited and non-suited.

All other streets required a more complicated system for state abstraction. We took some inspiration idea from SOTA poker agents LIBRATUS as discussed in [2]. Among the remaining streets, the river was the simplest, requiring the fewest abstract states. Since all cards are visible on the river, we are only concerned with what hands can be made at that moment. There is no need to speculatively evaluate a position based on what the value of hidden cards might be. We bucketed our state base on the best hand-type we could make. Furthermore, we differentiated between buckets based on the rank (high or low) of the cards in the hand. Cards of the rank 10, J, Q, K and A were considered high, while all others were considered low. For example, [6, 7, 8, 9, 10] was the lowest possible high straight.

At the turn, our abstract state space is larger with the addition of speculative “-1” states, that indicate when we are one card away from making a hand. For example, flush-1 represents our agent being one card away from a flush. If we are on the turn in the flush-1 state, there is a 25% chance that the hidden card completes our flush. At the flop, we introduce additional speculative "-2" buckets for cases where we are two cards away from making a hand.

This design had a few major limitations. The most notable being that we had no way of differentiating between hole cards and community cards. If our best hand was a flush, we would find ourselves in the flush bucket, even if our flush was entirely made of community cards. In such a case we have no advantage over our opponent despite being able to make a relatively good hand. The second limitation is that there are buckets that are unlikely to resolve to their prospective hand, such as straightHigh-2. These buckets increased our branching factor needlessly. Lastly, our speculative buckets were not street-specific, meaning that we could be in the straightHigh-1 bucket but not know if we were in the flop or the turn. Presumably, we would expect straightHigh-1 to be of greater value on the flop than on the turn since we have two opportunities to complete our straight, instead of just one.

3.2 Final Abstraction Design

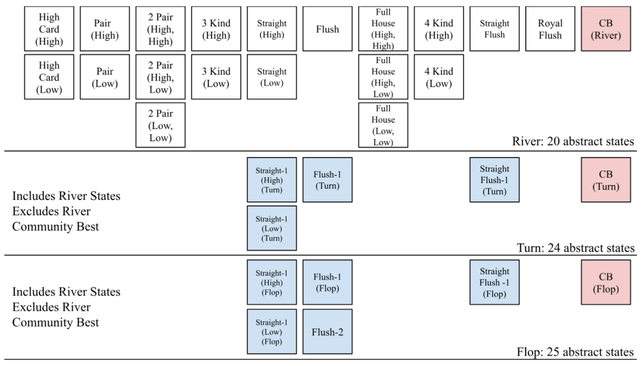

Figure 1: Diagram Of Abstract States In Our Final Design

Understanding these limitations allowed us to refine our abstraction. The abstraction shown in Figure 1 is our current implementation that addresses those limitations. The first improvement was adding a "community best" (CB) bucket for each street. States are placed into a CB bucket if the best hand we can make is entirely made of community cards. We expect CB buckets to be the worst of all abstract states. We differentate our CB states by street since a CB state on the flop or turn still has the opportunity to enter a better state abstraction when more cards are revealed. This doesn’t apply on the river so we presume that CB (River) states will have the worst outcomes. The second improvement was making our speculative buckets street-specific. For example, straightHigh-1 in the initial implementation became straightHigh-1F and straightHigh-1T in the current implementation. Lastly, we reduced the state space by removing speculative buckets that are too improbable, such as the straight-2 buckets.

Although this refined abstraction resolved limitations of the initial abstraction, there are still limitations. The first major limitation is that we do not speculatively check the community cards to see if our opponent is likely to make a better hand than we can. For example, say we are in Pair (High). Our agent will likely be happy, having made a better hand than can be made using the community cards alone. However, if there are four diamonds in the community cards, and there are no diamonds in our hole cards, it is quite likely that the opponent has a flush. This would be a high-risk position and we don't currently have any way of indicating that through our state abstraction. To solve this, we could triple our number of buckets, assigning each of our current buckets an additional flag representing if the hand is high risk, medium risk, or low risk. We did not implement this in our abstraction due to computational restraints. Another limitation is that we do not have an objective way of placing ourselves into one bucket when there are multiple valid options and it is ambiguous which is the most valuable. For example, if we are on the turn and our agent has a high pair but is also one away from a flush, we must decide which abstract state is superior. Our current method is a hard coded table of abstract state rankings (see Figure 5 in the appendix). However, this ranking is based on intuition rather than experimentation.

4 Game Tree

Our game tree (see Figure 4 in the appendix) starts with a nature move, representing whether we are big blind or small blind. Subsequent nature moves are used to represent the dealing/revealing of cards. These nature nodes have only one action, deal, but transition us to one of many abstract states for that street.

When it is our agent's turn, we choose to raise by a fixed amount, call, or fold. Our game tree is not explicitly built but rather generated as simulations expand new nodes. We tracks the limit on the number of raises for each player, ending the street by placing nature nodes when no more raising is allowed. Folding always results in a terminal node. There is no hard-coded functionality to call instead of fold when we can do so for free. It may have been a good idea to explicitly hard code this on top of our learned behaviour. However, we hope that this behaviour is learned since our agent is aware of its status as the big blind or small blind via its position in the game tree.

Our nodes' values are based on the money we won/lost in simulations that pass through that node. We choose this rather than basing the nodes' values on the number of wins. This was a strategic decision because we would like our agent to place more weight on winning big pots than small pots. For example folding five 5 pots and then winning a 100 pot is a good result despite having a low win rate.

5 Training & Testing

In order to determine the best exploration value for the Upper Confidence Bound (UCB) formula, we trained five agents with different values for the exploration coefficient, C. We used values of 1, 50, 150, 500, and 2,000. We chose these values so that MCTS would expand a wide range of nodes with our 2,000 agent exploring every node in the game tree. For each trial, our agent was trained by playing one million iterations with 100 simulations per iteration.

The our script used to train our agent (test_mcts_player.py) was not submitted to GradeScope seeing as it is not necessary for our trained bot to run. However, test_mcts_player.py can be viewed along with our hard-coded agent, pokerBotPlayer.py, in our GitHub repo.

6 Results

As we in increased the exploration coefficient used, number of explored nodes seemed to plateau at 2,389. We also noticed that the agent with a C value of 2,000 consistently beats the always-raise agent by the largest margin. Based on these results we chose to submit the agent with a C value of 2000. Figure 2 shows the performance of our different MCTS agents.

Figure 3 shows the result of testing our hard-coded agent against the two given bots. As you can see the hard-coded agent performs worse than MCTS against both the random and always-raise agents. This indicates that we succeeded in improving over our initial approaching by switching to

MCTS. However, as Figure 2 shows, our MCTS agent does seem to lose when matched up against our hard-coded agent. This is not what we expected and shows that our MCTS agent may not be strictly better than our hard-coded agent.

MCTS Agent Exploration Coefficient Value	# Nodes In Tree	Gain/Loss Against Random	Gain/Loss Against Always-Raise	Gain/Loss Against Hard Coded	Gain/Loss Against Hand Eval
C = 1	191	+$1,000,000	+$33,860	-$698,460	-$505,380
C = 50	2187	+$1,000,000	+$45,080	-$798,720	-$571,820
C = 150	2360	+$1,000,000	+$40,360	-$835.540	-$551,620
C = 500	2386	+$1,000,000	+$46,560	-$779,080	-$563,420
C = 2000	2389	+$1,000,000	+$86,540	-$826,340	-$533,960

Figure 2: Performance of MCTS Agents

	Gain/Loss Against Random	Gain/Loss Against Always-Raise
Hard Coded Agent	+$870,152	-$1,000,000

Figure 3: Performance of Hard Coded Agent (PokerBot)

7 Contributions

Mason duBoef (34960127): Mason fully designed and implemented our state space abstractions (state_abstraction.py). He documented and diagrammed his designs as they evolved. Mason and Sri brainstormed the design the game tree. Mason ultimately coded its implementation (MCTSTree.py) Mason and Sri also trained and tested agents with different exploration values. Mason developed the "haveHand" helper functions used in the hard-coded approach and in state abstraction. He created the team's Github repository and was active in creating and assigning issues. He independently went to Professor Zick's office hours several times to discuss the group's process and approach. Mason scheduled weekly meetings and attended each one in person. He wrote the majority of the project proposal and contributed to the final write-up.

Sri Rama Bandi (34965694): Sri worked with Mason on the design of the game tree. He fully developed search and backpropagation. He researched MCTS deeply and strategized how it could be adapted for use in poker. His deep understanding of the game of poker was useful at every step of the design process. He researched and implemented the system of pre-flop hand evaluation used in our hard-coded bot and state-abstraction system. He collaborated with Mason in training and testing agents with different exploration values. He arranged and attended group meetings.

Amanda Arcieri (33345420): Amanda created a randomized bluffing function in one of the preliminary versions of the bot. She also focused on testing the preliminary versions of the bot, and debugged the testing code. Amanda also assisted Mason with the "haveHands" helper functions, indicating when hands were being created using community cards alone. She also coded a function to call when it is free to do so. Amanda attended the weekly meetings. She also wrote the outline and the rough draft for this report.

Satya Srujana Pilli (34760606): Satya was tasked with calculating statistical probabilities based on visible cards. She wrote the simulation code that lets us estimate how often our hand wins, ties, or loses against other players, and how often each kind of poker hand appears. She also hard coded probabilities and did research on understanding the probabilities of hands. Helped with the implementation of MCTS. She also explored and researched Q-learning. Helped with testing agents with different players. Assisted on MCTS design and document write up. Attended group meetings. Generated visualization of our populated tree.

References

[pok] Preflop strategy: Open pushing charts, pokerstrategy.com.

[2] Brown, N. and Sandholm, T. (2017). Libratus: The Superhuman AI for No-Limit Poker (Demonstration). pages 5226–5227.

Appendix

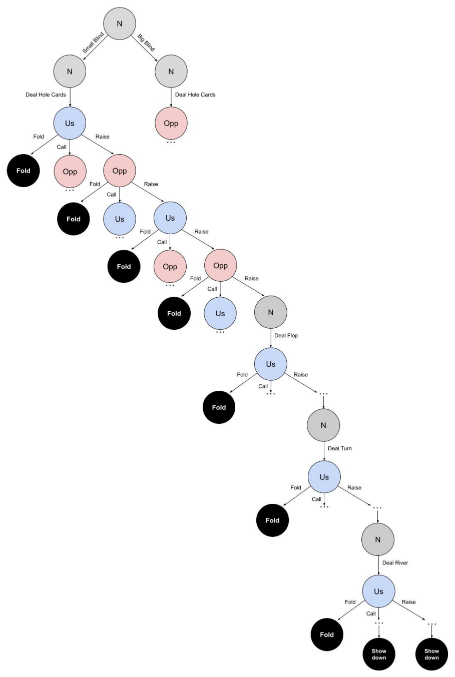

Figure 4: MCTS Tree Diagram

The diagram above shows only one arrow coming from the nature nodes tasked with dealing cards but in reality these nodes have a high branching factor (8 for preflop, 25 for flop, 24 for turn and 20 for river).

Flop	Turn	River
royalFlush	royalFlush	royalFlush
straightFlush	straightFlush	straightFlush
fourHigh	fourHigh	fourHigh
fourLow	fourLow	fourLow
fullHouseHighHigh	fullHouseHighHigh	fullHouseHighHigh
fullHouseHighLow	fullHouseHighLow	fullHouseHighLow
fullHouseLowLow	fullHouseLowLow	fullHouseLowLow
flush	flush	flush
straightHigh	straightHigh	straightHigh
straightLow	straightLow	straightLow
threeHigh	threeHigh	threeHigh
threeLow	threeLow	threeLow
straightFlush-1F	twoPairHighHigh	twoPairHighHigh
flush-1F	twoPairHighLow	twoPairHighLow
twoPairHighHigh	twoPairLowLow	twoPairLowLow
twoPairHighLow	straightFlush-1T	pairHigh
twoPairLowLow	flush-1T	pairLow
straightHigh-1F	pairHigh	highCardHigh
straightLow-1F	pairLow	highCardLow
pairHigh	straightHigh-1T	communityBest
pairLow	straightLow-1T	
flush-2	highCardHigh	
highCardHigh	highCardLow	
highCardLow	communityBestT	
communityBestF		

Figure 5: Abstract State Rankings