POKER AUTOMATION WITH COUNTERFACTUAL REGRET MINIMIZATION

An Honors Thesis Presented

By

ELI STANLEY BILDMAN

Approved as to style and content by:

**** Yair Zick 12/22/21 10:50 ****
Chair

**** Daniel Zhang 12/22/21 19:07 ****
Committee Member

**** Philip Sebastian Thomas 12/23/21 10:48 ****
Honors Program Director

ABSTRACT

Automated poker is a complicated game theory problem and an important benchmark for solving imperfect information games. As my portfolio, I created an intelligent Heads Up Fixed Limit Texas Hold'em agent. The agent uses Counterfactual Regret Minimization and some practical improvements to find an approximate solution. Training is done on an abstracted tree, created with two original processes: winrate grouping and history truncation. I test the agent to prove success and to demonstrate improvement with training. In all tests against basic opponents, the agent maintains positive average winnings. In tests between levels of training, the agent shows iterative improvement. These results show the validity of the algorithm, as well as the abstraction methods my agent uses.

1 Introduction

My portfolio is a Texas Hold'em Poker agent. The goal of this agent was to be able to automate intelligent poker decisions, and in testing, we find that is what is accomplished. The agent is currently limited to Fixed-Limit Heads-up (two-player) Poker but can learn bigger, more complicated versions of the game given more computing resources and some tweaking. This project explores game theory, abstraction, efficiency in computing, machine learning, and reasoning under uncertainty. It has been the most challenging and rewarding work I have done in my undergraduate program.

Much of my research is into artificial intelligence with limited computing and time since this project had to be built in a middle ground between accuracy and resources. The future of machine learning likely lies in finding more compact, practical solutions for widespread use, and this project shows some uses of the tactics that could lead there.

In this manuscript, I give an overview of the competencies I needed to create my agent. I start with important theories and variables (sec. 2). I explain my implementation of CFR+ and how it has been tailored specifically for this project, as well as some of the practices I use to increase efficiency (sec. 4). I introduce the abstraction methods I used to make the training process feasible, winrate grouping and history truncation, and cover the tradeoffs between gamesize and strategy accuracy I found (sec. 3).

1.1 Related Work

Much poker research focuses on Fixed-Limit Heads-up poker [3][1][5], as does my agent. Research on larger versions of the game focuses on finding approximations and practical solutions instead of fully solving the game [4]. These are some influential papers in automated poker research.

Burch [6] provides a strong starting place for research into automated poker play. Burch comprehensively covers the limiting factors of poker agents: training space and training time. Any

algorithm we create has to train in a reasonable amount of time within the memory the machine has access to. Burch presents Counterfactual Regret Minimization (CFR), an algorithm I will explore later in this manuscript to find solutions in a reasonable amount of time. He then explains practical modifications to CFR, like CFR+ and MCCFR, that further improve performance. Burch presents CFR-D, a variant of CFR that uses decomposition of the game tree to limit the amount of memory the solving machine needs access to in order to complete its computation.

A huge milestone for automated poker play is Brown and Sandholm [3] in which they present Libratus, the first Heads-up Poker agent to defeat top human players. Libratus uses a combination of pregame solving, game-time subgame solving, and updating based on opponents' play. In pregame solving Libratus uses MCCFR, as discussed by Burch, to solve an abstracted version of the game. This means instead of using the entire game tree, similar states and player decisions are grouped and solved as if they were the same. Abstracted solving means that pre-computation is less resource-intensive, but only provides an approximate solution to the game. To improve on this, game-time subgame solving is implemented: during the game, the agent is calculating a more accurate strategy for the remainder of the decisions it will need to make. Later into the game, there are fewer decisions left, meaning computation is faster. Lastly, the agent improves on its strategy based on the opponent's style of play. During the round, it takes note of how the opponent tends to bet and improves the accuracy of its strategy in these areas by reducing the level of abstraction. For instance, if the opponent tends to bet aggressively when a specific strong hand is possible, the agent might train itself further in real-time on how to counter aggressive play in this scenario.

Another important milestone by the same team is Pluribus [4]. This agent was the first to defeat top poker players in six-player Poker. Pluribus uses a similar strategy to Libratus, creating a precomputed "blueprint" strategy based on an abstracted form of the game, then finding finer-grain solutions as play progresses. A unique issue of many-player Poker is that there are multiple equilibria to be found. Players using different equilibria leads to unexpected results and exploitations. Pluribus copes by focusing on empirical strength instead of theoretic perfection. The algorithm implemented does not have mathematically guaranteed results, but performs well in practice.

Another relevant paper is Bowling et. al [1]. This paper introduces the agent that first found an equilibrium in Heads-Up poker in its full, un-abstracted form. This level of solving marks a conclusion for ground-breaking research into this form of poker, as no better solution can be found. This team used an augmented form of CFR and a supercomputer cluster to formulate the strategy.

2 Technical Introduction

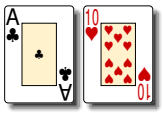
These are the necessary concepts and variables for studying automated poker. I cover extensive form games, Nash Equilibrium, and the variables used later to define necessary algorithms.

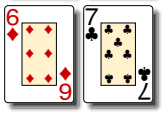
2.1 Extensive Form Games

Algorithms used to find solutions for Poker games use the extensive form model to represent games [12]. Extensive games are built around the concept of a game tree. The game tree has a node for every possible position and is rooted at the start of the game. Each non-terminal node has an associated player acting on it. In this model, *nature*, or random events are considered a player. Edges run from nodes to nodes representing positions that could be created in one move. Each terminal node has a value for each player equal to their take at that termination of the game.

Poker is an extensive form game of *imperfect information* [12]. These games have hidden information at some states and therefore a player is sometimes unable to tell what position the game state is within the game tree. This is in contrast to games like Chess or Go, where all the information is always available to every player.

In an extensive game *information sets*, or infosets, are defined as the set of game states indistinguishable to the acting player. Accordingly, strategies used by players must be the same over every state within an infoset. For example, in Poker, a player does not know what cards their opponent was dealt. In Fixed-Limit Heads-Up Hold'em, an infoset could include the position where player one is the small blind, and the players were dealt:

Player 1: 

Player 2: 

Player one is playing here but does not know what cards player two was dealt. Therefore the infoset player one uses includes every position where they were dealt this hole.

In extensive form games, a *strategy* is a function that maps information sets to probability distributions over available moves. In the example above, a strategy could include a distribution for this position such as [fold: 0.6, call: 0.2, raise: 0.2]. The player would fold 60% of the time, call 20% and raise 20%. If we have the strategies for both players, the expected value at any position can be calculated by multiplying the value at terminal nodes by the likelihood of the actions that lead there.

2.2 Nash Equilibrium

When solving games, we are looking to find a Nash Equilibrium. In a two-player game, such an equilibrium exists when both agents can expect no greater value if they changed their strategies [9]. In a Nash Equilibrium, neither player has an incentive to change, and the game is considered solved.

The algorithms used for training all search for this equilibrium. This is because if we have a strategy that is not a part of an equilibrium, by definition there is a better strategy to find since there is an incentive to change in some way.

2.3 Variables

To study these algorithms we define these standard variables and functions for imperfect information games. A is the set of all game actions. I is an information set, including several states that are identical in information to the acting player and possible decisions. $A(I)$ is the set of possible

game actions at information set I. T and t represent time, which increments once for each move by a player or random event. A strategy σ_i^t is the set of probabilities that player i at information set I_i will take any action $a \in A(I_i)$ at time t. All player strategies at time t create the strategy profile σ^t. $\sigma_{I \to a}$ is a strategy profile equivalent to σ, except action a is always chosen at I. The history h is a sequence of actions and random events starting from the beginning of the game. $\pi^\sigma(h)$ is the probability of history h ring given strategy profile σ. Similarly, $\pi^\sigma(I)$ is the probability of reaching information set I with strategy profile σ. The counterfactual reach probability of information set I, denoted $\pi^\sigma_{-i}(I)$ is the probability of reaching I with strategy profile σ except every decision player i makes is considered to have a probability of 1. Z denotes the set of terminal game histories. So, if $h \subset z$, $h \neq z$ and $z \in Z$ then h is a nonterminal game history. $\pi^\sigma(h, z)$ represents the probability of terminal history z given nonterminal history h. Lastly, $u_i(z)$ is defined as the utility, or value received for player i at terminal history z.

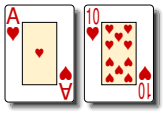
3 Abstraction

In my development, I found even the smallest form of Poker is too big for my agent to learn without unobtainable computing power and time. Luckily, it does not need to. Like in the related papers, if the agent can understand an abstracted version of the game, it can translate that understanding into an approximation of a full strategy [4][3].

This is intuitive as it is similar to the way humans make sense of big games. In chess, you would not take the time to learn an independent strategy for every position where your opponent has left their queen undefended. You would instead develop the understanding that every time there is an undefended queen, you can take it. In poker, we can use similar intuitions to group hands with similar strategies.

3.1 Hands Reduced to Winrates

The first abstraction my agent makes is to understand the cards on the table not as the actual cards, but as positions with estimated win-rates instead. The agent is trained on a gametree with only winrates estimated from the perspective of the player in question. For example, if the agent was dealt the following hand preflop:

Hole: 

it would immediately estimate the winrate for this hand, that is, the likelihood that this is the best hand at the table. In this case, the estimation is about 0.6619. From there, it rounds this winrate to the nearest winrates it is trained on and selects its strategy accordingly. In the current form of the agent, the winrates it is trained on are $[0.2, 0.4, 0.5, 0.6, 0.7, 0.8]$. This is based on empirical tests of the limit to the number possible with its current setup, and an intuitive understanding that winrates are normally distributed, centered around 0.5. In the example, the agent would consider this hand a 0.7 winrate.

This process is made possible with the use of Monte Carlo estimation. The Monte Carlo estimation runs a series of simulations of possible endstates for the round given the initial variables provided. For instance, if the agent only had access to its hole cards and three of the five community cards, it would fill in the other two cards randomly with uniform likelihood several times, taking note of at what rate it is winning. Over enough simulations, the sample winrate converges to the true winrate of the position [8]. The agent uses 1,000 simulations, which runs on my PC in about 0.2 seconds, making it feasible for decently quick play during testing. At this level of simulation, I found a standard deviation of about 0.01 in winrate, which is low enough to not make a notable difference after rounding. My agent also uses a running cache of positions is has seen before and the associated winrate. Almost all of the opening positions and many of the following positions are known and can be looked up instead of recalculated every time, increasing performance time.

Transition probabilities between winrates are found similarly. A graph with a node for each

winrate at each stage of the game and an edge between every node and those in the level below it is created. In my agent it looks like this:

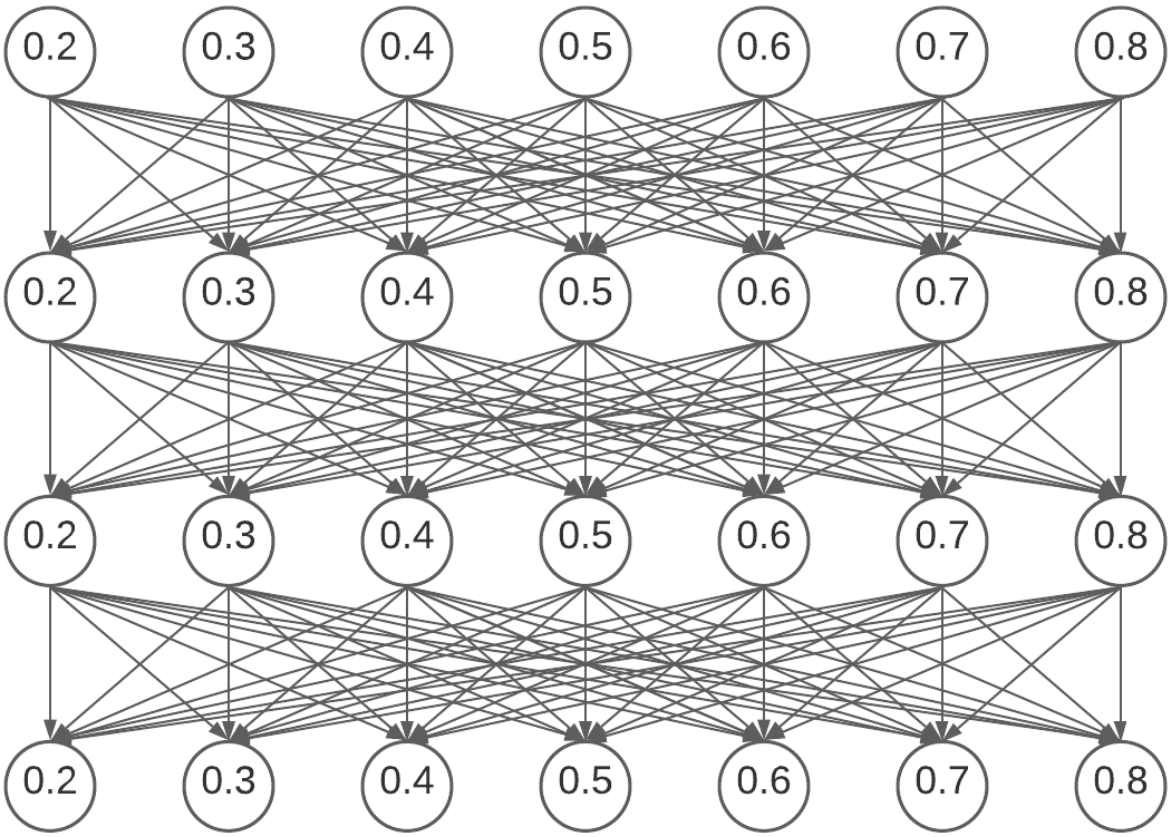


Figure 1: Winrate map

From here several simulations of games are run from the deal through the end, and the winrate at every position is recorded. The transition rate on each edge is taken as the rate at which states within the parent node become states in the child node. For instance, if during our simulation, 500 preflop hands had a winrate closest to 0.6, the breakdown of transitions and their corresponding rates could look like this:

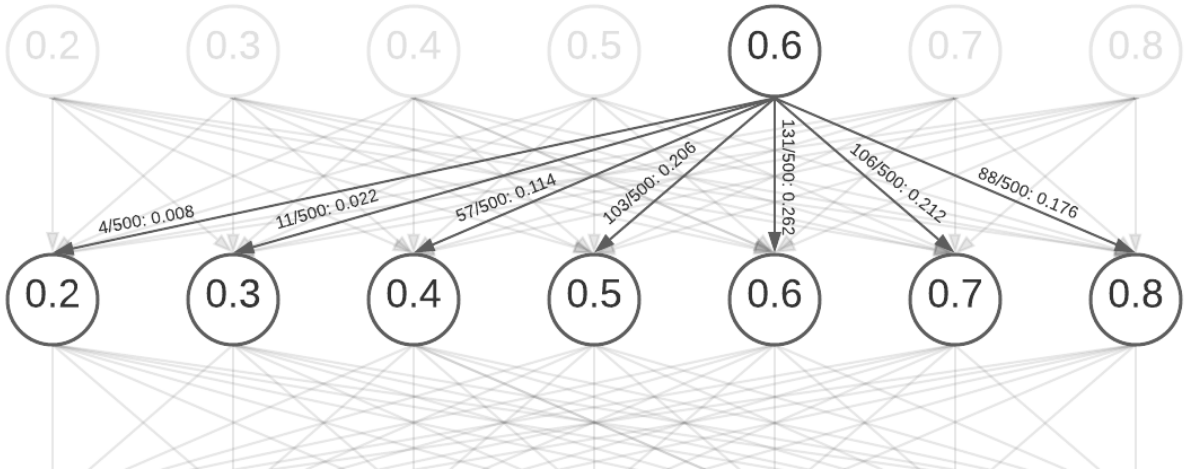


Figure 2: Example edge weights for winrate map

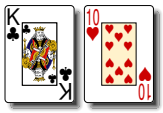
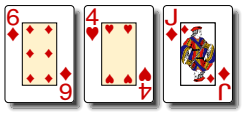
This is also Monte Carlo estimation, and therefore also converges to the true probabilities. When assessing the likelihood of a history that includes nature events for CFR, the agent can use these transition rates. To find the joint likelihood of two of these transitions, for example, player one being dealt a 0.4 winrate and player two being dealt a 0.6, a simple product is taken of the corresponding transition rates.

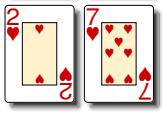
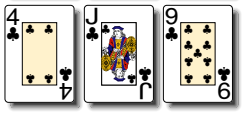
In the full gametree, nature events have by far the biggest branching factor. During the deal, there are $\binom{52}{4} = 270,725$ different ways the hole cards can come out. On the flop, there are $\binom{48}{3} = 17,296$ different options. Making this abstraction cuts the gametree down significantly and makes any learning possible.

Unfortunately, this abstraction reduces overall intelligence in several ways. To start, a finely tuned understanding of the winrate is lost. With the current implementation, a hand with a winrate of 0.54 would be treated completely identically to a hand with a winrate of 0.46. This has the potential to remove some nuance in the differences in how these situations should be handled and might decrease performance in the long run.

A more pressing issue is the loss of understanding of the context of moves. Two states may have the same winrate, but look completely different in terms of how their trees develop in the later stages of the game. For example, directly post-flop, these two positions have similar winrates,

around 0.4:

Hole:  **Community Cards:** 

Hole:  **Community Cards:** 

However, in the second position, there is around a 0.125 chance that your opponent is holding two clubs, creating a flush, one of the strongest hands in the game, and a sure loss for you. Strategy in this position would likely revolve around this possibility and your opponent's understanding that you might have the same. If someone was to raise, for example, directly after this flop is dealt, it should be treated as an indication they are holding the clubs, or they are bluffing. These strategy decisions are where much of the nuance of high-level poker comes from, and are lost on this agent.

3.2 Rooting the Gametree at Last Nature Event

The next major abstraction my agent makes is rooting the abstracted tree at the last nature event, and terminating before the next one. This means no decisions or nature events before the last card was turned are taken into account. This also means that multiple trees have to be created since there are differences in game logic between rounds of betting. The agent also creates different trees for different pot sizes going into a round of betting to preserve payoff values. For instance, four distinct trees are made of every winrate level for the flop round, since pre-flop betting can end with a pot of 2, 4, 6, or 8 big blinds.

Since extensive games need every terminal node of the tree to have a value, an estimation must be made at the leaves for trees representing the first three rounds of betting. If the agent knew the actual hands at play in a state, this estimation could just be another Monte Carlo winrate estimation over the remaining variables. It would find the approximate winrate, given both player's hole cards and the community cards, and assign value as the product of this rate and the current pot size. Since hands are represented as winrates instead, the agent just assigns the winner as the player with the

higher observed winrate and gives them the entire pot.

This cuts down the size of the gametree significantly. Since the trees for separate rounds are built separately, they do not branch into each other, meaning the number of states grows linearly with how many rounds are learned, instead of exponentially. However, the loss of historical events removes the agent's ability to consider the way its opponent has acted in the past.

3.3 Abstraction Results

With the abstractions mentioned, the agent creates 203 game trees. In the trees, there are 39,572 states in 2,016 different information sets. Stored in the python data structures I wrote them in and on my PC, the trees take about 1 second to generate and use about 5.3MB of space.

4 Counterfactual Regret Minimization

The most popular starting algorithm is called *counterfactual regret minimization* or CFR [10].

4.1 Algorithm Explanation

CFR is the process of trying to minimize *regret*, which is calculated as a loss of *counterfactual value*. Counterfactual value at a nonterminal history h with strategy profile σ is defined as:

$$v_i(\sigma, h) = \sum_{z \in Z, h \sqsubset z} \pi^{\sigma}_{-i}(h)\pi^{\sigma}(h, z)u_i(z) \tag{1}$$

This is the counterfactual probability of reaching an end state, meaning the actions of player i are considered given, multiplied by the value of that state, summed for every state reachable. This is similar to the expected value of a gamestate but done counterfactually. Counterfactual regret builds on this and is defined as:

$$r(h, a) = v_i(\sigma_{I \to a}, h) - v_i(\sigma, h) \tag{2}$$

Regret can be considered the difference in value between changing the strategy to always choose action a and not. If this regret is positive, it would be beneficial to change the strategy to make action a. Lastly, counterfactual regret at an infoset I is defined as:

$$r(I, a) = \sum_{h \in I} r(h, a) \tag{3}$$

This needs to be calculated because, as discussed earlier, players must consider states in infosets, and can never know the true history h they are in.

From here, CFR creates a vector of running regret sums for every infoset. Whenever a given infoset is reached, new regrets are calculated and added to these sums. Strategy σ_i is then updated to represent the positive regret sums normalized to sum to 1. For example, if we had a regret sum vector $[-5, 10, 8, -2]$, our resulting strategy for I would be $[0, \frac{10}{18}, \frac{8}{18}, 0]$. This is repeated several times as a training process, and the strategy profile eventually converges to the optimal values [6].

CFR removes the need to consider every possible state of the unknown variables of a game because all calculations are done retroactively when unknown variables are uncovered. However, CFR can sometimes take too long to converge on large, complex trees. To tackle this issue CFR+ is used [11]. CFR+ makes the key change of only considering positive regrets when calculating regret sums. This small difference means the training spends less time rewarding good decisions, and more time punishing bad ones which, based on studies in the same paper, makes the values converge faster in most cases.

The last addition to the algorithm is called "Regret Discounting" [2]. It is notable that for some payout distributions, CFR and CFR+ take an unreasonably long time to converge due to outlier regrets calculated in early iterations. An example given in Brown and Sandholm[2] is the case of three choices with payoffs $[0, 1, -1000000]$. In this case, the positive regrets would be $[333332.5, 333334.5, 0]$. Since the third option has zero regret, it will never be chosen, and the strategy will consist of picking between options 1 and 2 with about 50% odds. The difference in regret between these choices is very small, so finally converging to the point where option 2 is

chosen over 1 will take hundreds of thousands of iterations. The solution to this is to add a weight to the calculation of the normalized regrets. The normalization step in CFR is can be thought of as taking the average of the regrets of every iteration of the training so far. Since the regrets of early iterations are relatively less important than the regrets found more recently, these regrets can be less heavily weighted. The weight of a regret iteration is found with the equation $w = max\{T - d, 0\}$. Where d is some optional delay value to keep early iterations from being affected.

4.2 Implementation

In my agent to find the infosets, the game trees are traversed, and every state where a decision can be made is checked for the information available to the acting player. If two states have identical information, they are added to the same infoset. To formalize this and make hashing possible, unique strings are created out of this information. An example of one of these information strings is:

```
"P1R0B3.0|N:0.3|D:call|D:raise|"
```

Figure 3: Example information string

Breaking down this string, 'P1' means this is a state that player one plays in. 'R0' means this is the round with index 0, or preflop. 'R1' would be the flop, 'R2' the turn, etc. 'B3.0' means that there is currently 3 times the big blind in bets on the table. Everything within the '|' characters are the history of this position, taken to the last nature event. 'N' events are events of nature, followed by the winrate from the player's perspective afterward. 'D' events are player decisions, followed by the name of the move that was made. Since fixed-limit poker has only one raise amount, we do not need to specify the amount being raised or called to. The infosets are stored in a hashmap, with the information string as the key for quick lookup.

In every iteration of training, regret is calculated for each infoset for both players. The regret sums for each possible move and a new strategy are calculated accordingly. Once calculated, the strategy can be copied from the infosets into a dictionary relating the information string to a list of

tuples containing the name of possible moves and their associated probabilities within the strategy. When saved, these strategies are fairly lightweight, only taking about 150KB.

4.3 CFR Results

On my PC, with an Intel i5, each iteration of training takes about 3.2 seconds. All of the training I did for this project was on this computer.

5 Testing

Using the abstractions and algorithms described, I was able to achieve some positive results. To start, I created and saved separately strategy profiles for the first 30 iterations of training. To test for convergence, I calculated the overall change in strategies over all the infosets for each iteration of training.

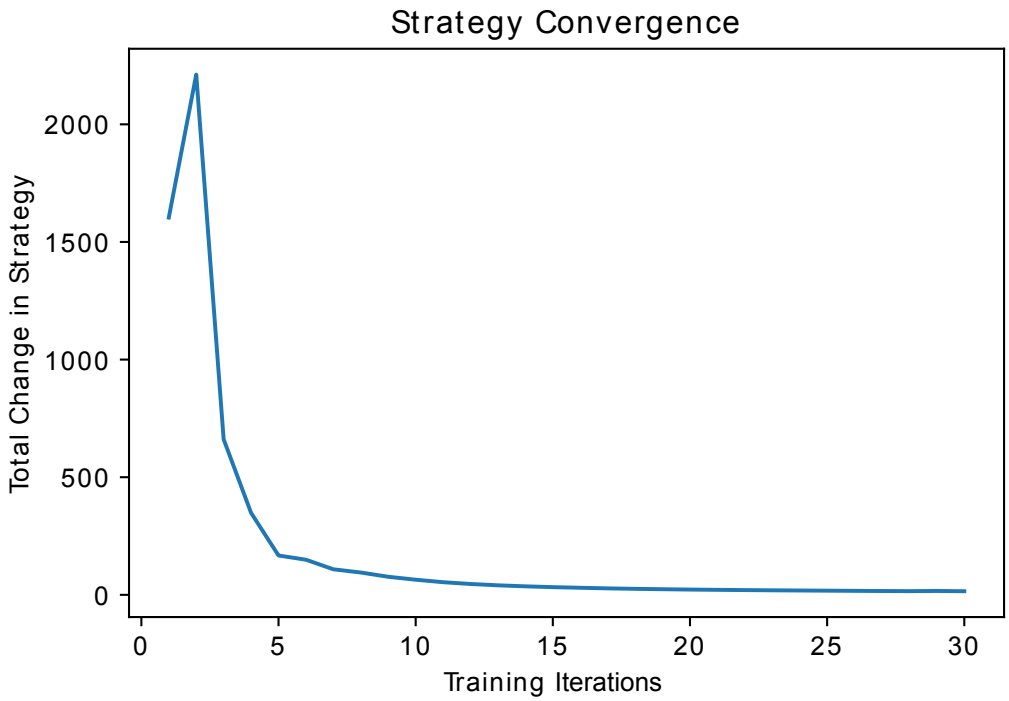


Figure 4: Strategy convergence graph

The strategy mostly converges in the first 10 iterations. By the time it is trained 15 iterations,

the change is almost none. Based on this, we would expect to see the most change in performance among the strategy profiles the first iterations produce. In testing, this is what we find.

To test, I used a script that plays the bot against a custom opponent. The testing I did was in sets of 10,000 rounds, resetting stack size at the beginning of each round and switching seat positions. For the following figures, I iteratively increased the training level of the agent to demonstrate improvement. I ran the bot against several simple agents and observed the average difference in stack size after the round.

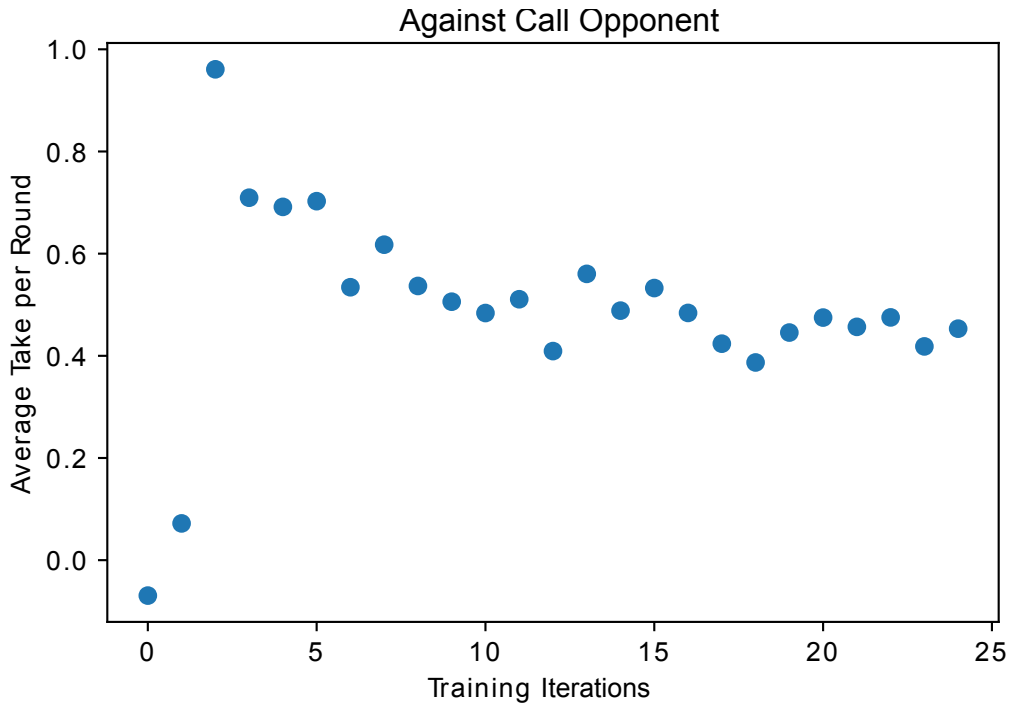


Figure 5: Average take against an agent that only checks/calls

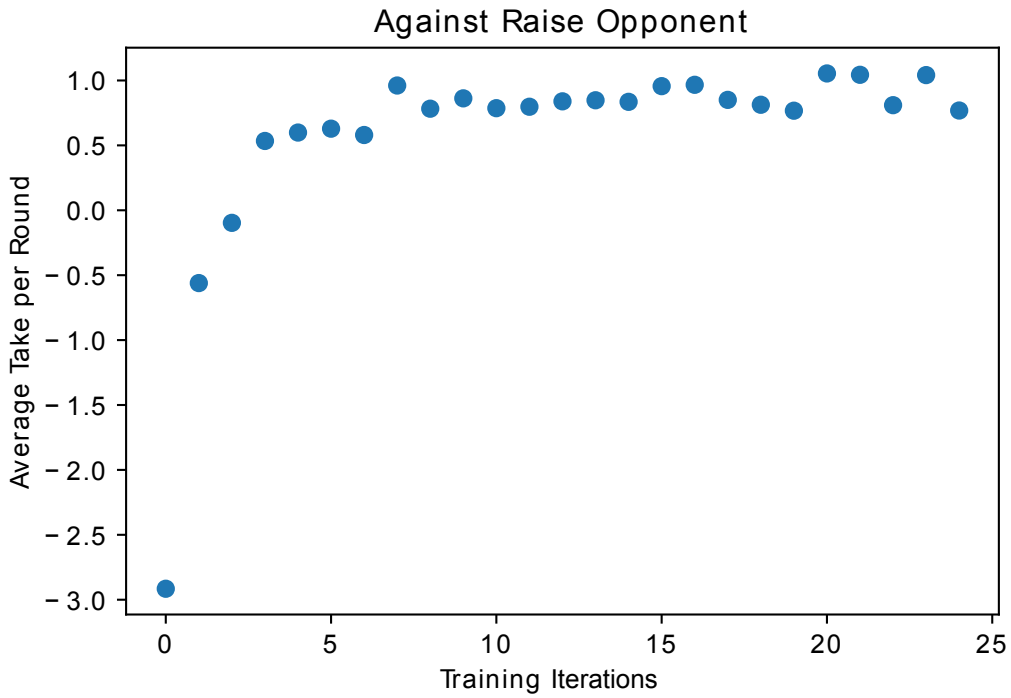


Figure 6: Average take against an agent that only raises

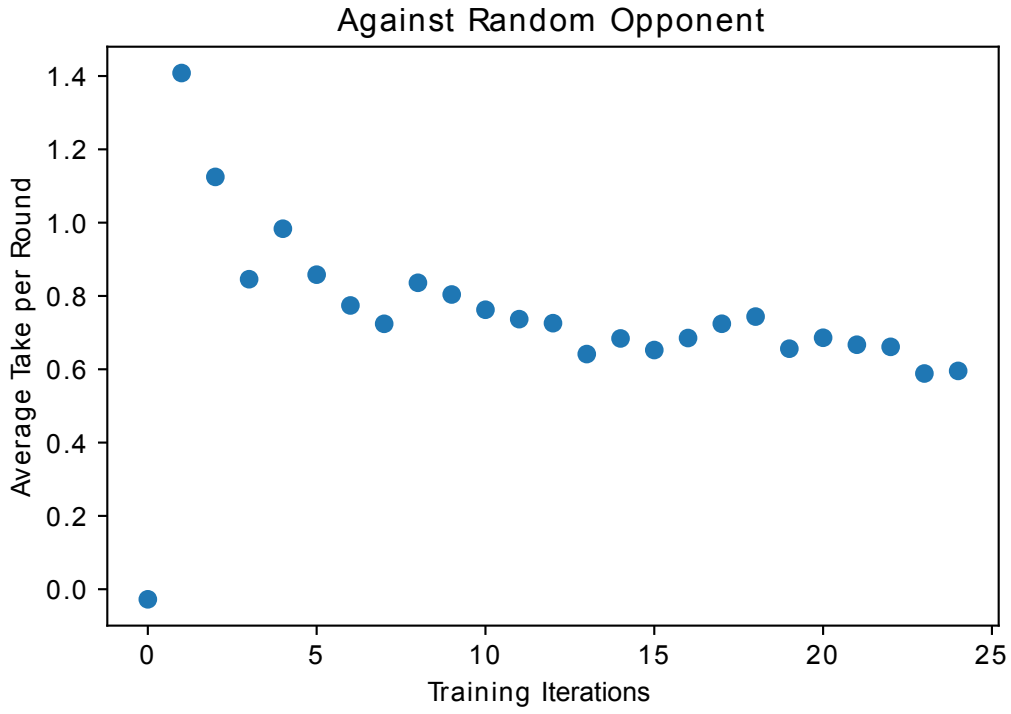


Figure 7: Average take against an agent that make uniformly random decisions

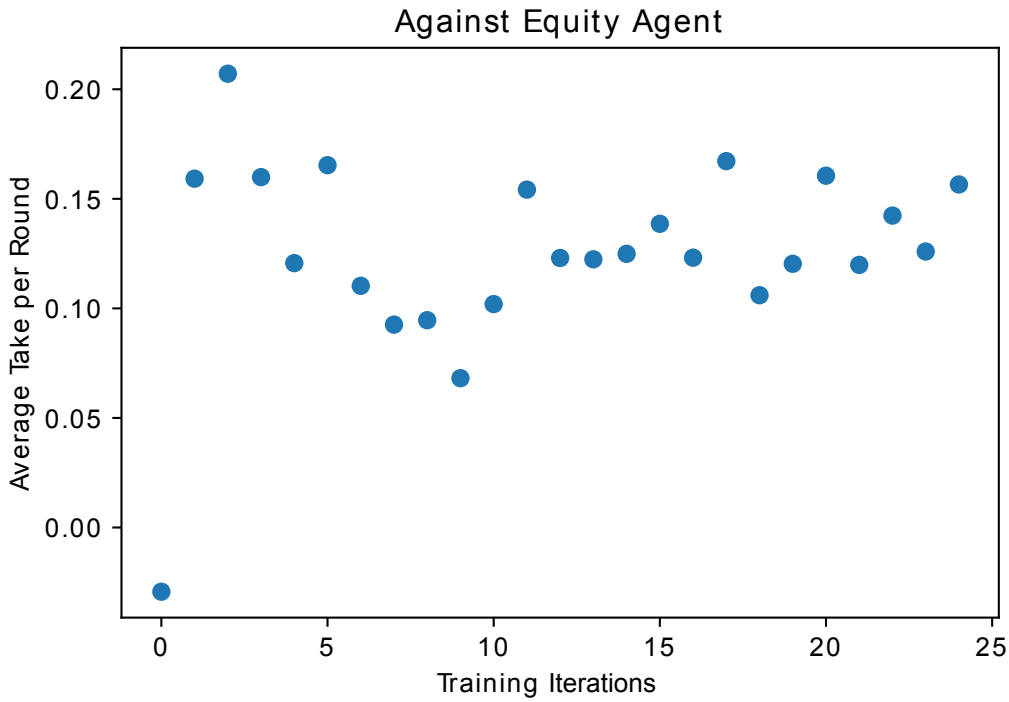


Figure 8: Average take against an opponent that call if it has over a 50% winrate and folds otherwise

I also ran a series of tests between the levels of training. These tests are against the agent trained one iteration, and 50 iterations. CFR initializes the strategy as uniformly random, so testing against a 0 iteration bot would be the same as against a random agent.

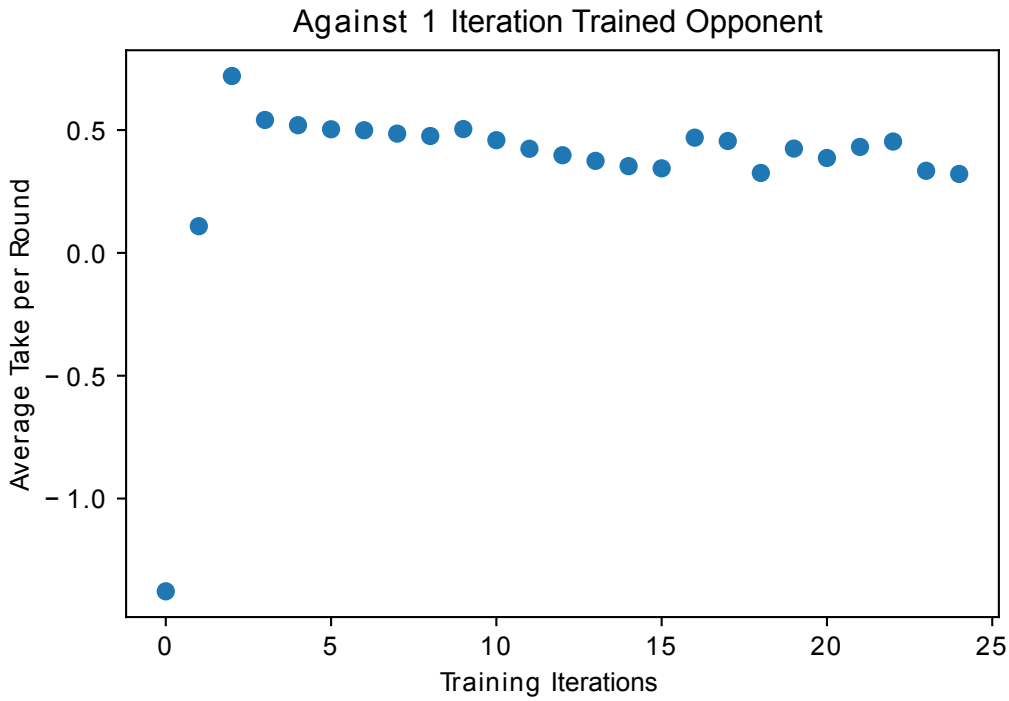


Figure 9: Average take against strategy with 1 round of training

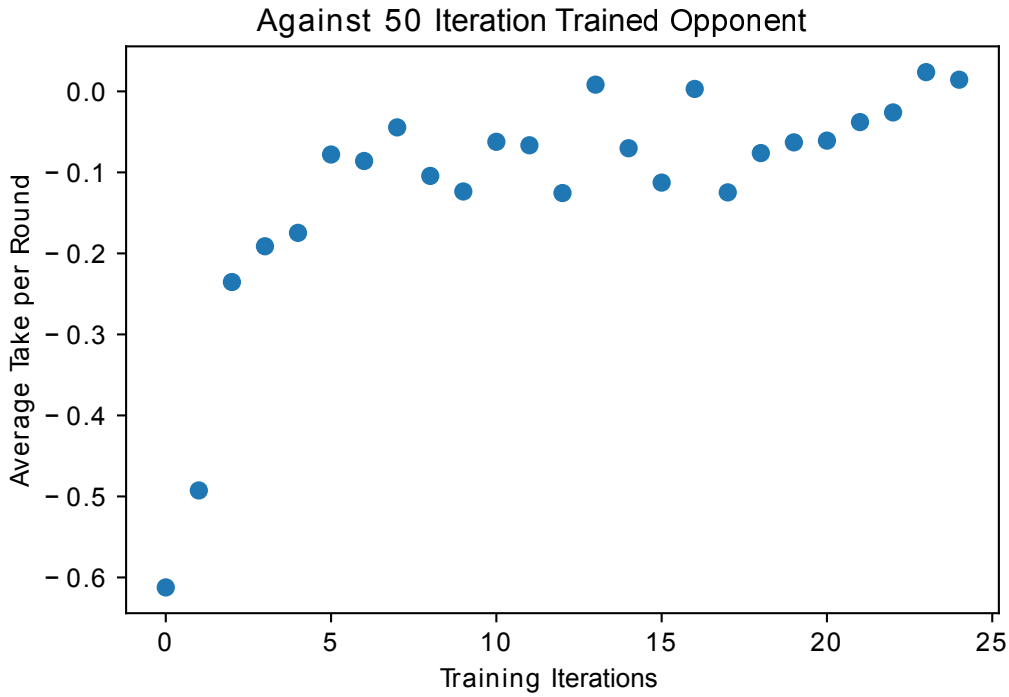


Figure 10: Average take against strategy with 50 rounds of training

The testing shows a variety of different results, but there are some common features. Firstly,

against all the opponents, the agent almost always won a positive average take, excluding against itself fully trained. This means that the agent was overall winning in these match-ups. Second, in all of the tests, the winrates level off around or before 10 iterations. This is consistent with the convergence data and shows the driving force behind the changes in winrate are the iterations of training.

There is an interesting effect in some of the tests, most pronounced in the random agent and call agent tests. In these tests, before leveling off the agent saw a brief period of decline in winrate. In these cases, it seems the 5 iteration strategy is outperforming the 25 iteration strategy. I have two hypotheses for the cause behind this effect. The first is that training on the abstracted tree too much leads to the bot developing nuances in the strategy that do not apply to the larger game. This would be something like an overfitting problem in AI.

I think the more likely reason behind this dip in performance is that as the bot improves, it also assumes it is playing against a better player. CFR trains against itself, which is necessary to find a Nash equilibrium [9], but means that it is developing to play another agent as intelligent as itself, which is not the case in these tests. There is an understanding in the Poker community that playing a game-theory-based game will not necessarily exploit weaker players as much as a more tailored, less optimal strategy could. I think that is what we are seeing here. The earlier strategy profiles are better at exploiting the random and call agents than the more refined strategies, although both are winning.

6 Conclusion

In this manuscript, I discussed abstraction and counterfactual regret minimization as they apply to automated poker play, and the state of game theory research in games of imperfect information. These were the competencies I had to build to create my Poker agent.

With more time, I will research finding abstraction methods, and try to find more accurate and nuanced ways to do the abstractions I have. The unfortunate catch-22 of finding abstraction

methods that preserve strategical integrity is it would be useful to have an understanding of the strategy the pre-abstracted and post-abstracted training will produce for comparison. These are hard to know as the initial reason for building abstractions is that training on the pre-abstracted tree is unreasonable. I will also branch my research into different sections of reinforcement learning in case there is an algorithm better tailored to what I am doing. However, it seems like the vast majority of Poker agents use CFR or some variation. Lastly, with more time, I can look into ways the agent can adjust its strategy to better exploit weak play in its opponents. Given how quick this model trains, it would be possible to model and train against opponents' play on the fly, using some of the tactics of Pluribus and Libratus.

Overall, building this agent was extremely challenging and rewarding. The process of design and redesign is not one we get to do often in traditional computer science classes, as projects are often completed and forgotten. Finding iterative improvements is challenging in a completely new way. I had to take things I already understood and grow those understandings further, and in other cases, I had to abandon what I had built and learn completely new ideas to make progress. I am grateful to have completed an honors portfolio and proud of what I created.

References

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

[2] Noam Brown and Tuomas Sandholm. Solving imperfect-information games via discounted regret minimization. *arXiv preprint arXiv:1809.04040*, 2009.

[3] Noam Brown and Tuomas Sandholm. Libratus: The superhuman ai for no-limit poker. *IJCAI*, 2017.

[4] Noam Brown and Tuomas Sandholm. Superhuman ai for multiplayer poker. *Science*, 2019.

[5] Noam Brown, Tuomas Sandholm, and Brandon Amos. Depth-limited solving for imperfect-information games. *arXiv preprint arXiv:1805.08195*, 2005.

[6] Neil Burch. *Time and Space: Why Imperfect Information Games are Hard*. PhD thesis, University of Alberta, 2017.

[7] Matthew Cluff. The ultimate guide to limit texas hold'em. *888Poker*, 2018.

[8] Paul Dagum, Richard Karp, Michael Luby, and Sheldon Ross. An optimal algorithm for monte carlo estimation. *SIAM Journal on Computing*, 29(5):1484–1496, 2000.

[9] John F. Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, 36(1):48–49, 1950.

[10] Todd W. Neller and Marc Lanctot. An introduction to counterfactual regret minimization. *Model AI Assignments*, 2013.

[11] Oskari Tammelin. Solving large imperfect information games using cfr+. *arXiv preprint arXiv:1407.5042*, 2007.

[12] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. *NeurIPS Proceedings*, 2007.

A Rules of Heads-up Fixed-Limit Texas Hold'em and Terminology

Fixed-Limit Heads-up Texas Hold'em poker is played by two players in a series of four betting cycles. Before the game starts, one player must offer up a *blind* bet, and the other player half of this value. These bets and their respective players are called the *big blind* and *small blind* accordingly. For the sake of notation, big blind is often shortened to *BB*. The round starts with both players being dealt two cards, face down. These cards are a player's *hole cards*. This event is the *deal*. After the deal, the first round of betting begins, called the *preflop* round. Betting in this round starts with the small blind. They have the option to *call* where they match the bet of the big blind, *raise*, where they increase their bet beyond the big blind's or *fold* where they give up their hand and sacrifice their bet to the opponent. Once the small blind player has made a decision, betting moves to the big blind. The big blind now must either call to the amount that the small blind has bet, raise the bet higher still, or fold. Betting continues back and forth until the players have bet an equal amount, or one has folded.

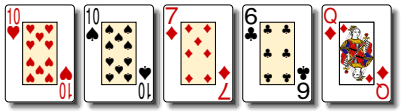
In Fixed-Limit poker, raises can only be done in a predetermined amount. In the first two betting rounds, this amount is equal to the big blind. In the final two rounds, this amount is twice the big blind. In addition, the bet of any individual player in any betting round cannot exceed four times the raise amount for that round.

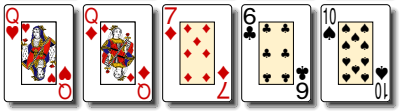
Once preflop betting is completed, three cards are dealt from the deck, face-up, in the center of the table. This is the *flop*. The flop is followed by a round of betting without blind bets and started by the big blind player. After betting, a fourth card is dealt, face-up, in the center of the table. This is known as the *turn*. The turn is followed by another round of betting. After this is one more card turn, known as the *river* and a final round of betting. If neither player has folded by the end of the last round of betting, players show their cards, and a winner is determined.

The winner is the player who can make the strongest *hand* of five cards using some subset of the cards in their hole and the cards in the center of the table, known as the "community cards."

Hand strength is determined by where the hand lies on the hand strength list in B.

If the players tie, the rank of the cards they used to make their hand is used as a tiebreaker. For example, if these hands faced off, the winner would be determined by the rank of the pairs, and player two would win.

Player One: 

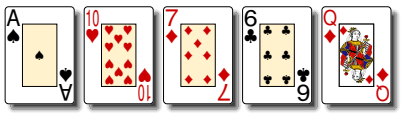
Player Two: 

If ranks are identical, a draw is called and bets are returned.

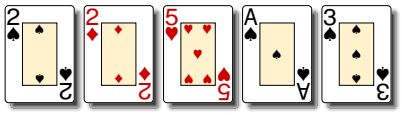
In the case where a player runs out of money and cannot call a bet, they have the option to go *all-in* or fold. In an all-in, they bet the remainder of the money they have. The flop, turn, and river are run through in succession without betting rounds, and a winner is determined based on hand strength. In the case where the all-in player wins, they regain their all-in bet and equal value from their opponent's bet. The rest of the opponent's bet is returned. In the case where the all-in player loses, they have lost the game. [7]

B Texas Hold'em Hand Rankings

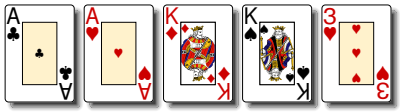
High Card: Strength determined by highest card. No other hand type.



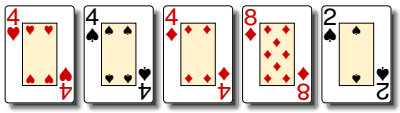
1 Pair: Two cards of same rank.



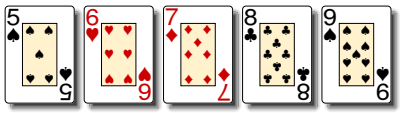
2 Pair: Two pairs of two cards of same rank.



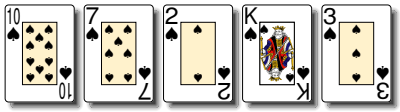
3 of a Kind: Three cards of same rank.



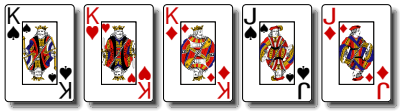
Straight: Five cards of sequential rank.



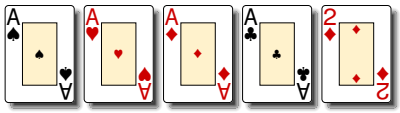
Flush: Five cards of same suit.



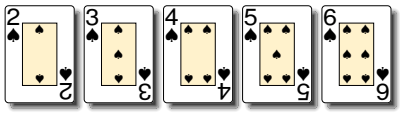
Flush: Two cards of one rank, three cards of another.



Four of a Kind: Four cards of same rank.



Straight Flush: Combination of a straight and a flush.



Royal Flush: Straight flush starting on ten.