Cribbage Counterfactual Regret Minimization

Mykyta Solonko

2023-04-21

Abstract

The goal of this project was to research techniques that are utilized in programming agents for imperfect information two-player zero-sum games and apply them to Cribbage. Particularly, Counterfactual Regret Minimization (CFR) was applied to both the throwing and pegging stages of the game. The best CFR agent could beat the provided greedy opponent by 0.39 match points, with some simpler & faster ones beating greedy by 0.32-0.36 match points. Other methods, such as using Monte Carlo Simulations and Schell's Discard Tables were explored with mixed results.

Motivation

One of the inspirations for this project has been reading about the successes of Libratus, a program designed to play heads-up no-limit Texas hold 'em (Brown, Sandholm 2017). In 2017, Libratus was able to defeat several professional poker players over the course of thousands of hands, proving that it is possible to make a very good agent for two-player zero-sum games with imperfect information. Online, I was unable to find any papers regarding writing a Cribbage-playing program, with the exception of a few forum posts where users conclude that it will be too difficult given the large number of states. Existing online AIs typically use simple strategies or implementations of Cribbage "rules of thumb." With a thoughtful approach to this imperfect information game, it should be feasible to make an agent that is not too computationally expensive and that beats humans and these simplistic AIs alike.

What is Cribbage

Cribbage is a game best played with two people. Each player is dealt six cards face down from a standard 52-card deck. Both players then select two cards to send to the "crib," which belongs to the dealer of the current round. The goal of the game is to reach 121 points before the opponent does the same. Points can be earned in three ways. First, various combinations of the four cards left in the player's hand and the "starter" card can earn that player points, such as a pair, a straight, or a group of cards that sum to 15. The four cards in the crib (two from each player) and the starter card also can make various combinations; the points of these combinations go to that round's dealer. Finally, during the play, players alternate placing down their cards into the middle face-up, also attempting to make combinations, reach a sum of 15, and be the last one to put down a card. For more information on cribbage scoring and rules, this tutorial can be consulted.

Cribbage is an imperfect information game because it is not known what cards the opponent has as well as which card will be the starter card. Cribbage can be viewed as a zero-sum game because one player always wins and the other always loses. You can also "skunk" your opponent (by beating them by a certain margin) to get more than 1 match point (which makes Cribbage not zero-sum in terms of match points). For this project, I will be evaluating the success of the CFR approach using match points as a metric, but I will also mention win percentage

If viewed as a zero-sum game, and due to having imperfect information, methods that have worked well for solving poker, specifically CFR, could be a good fit for Cribbage.

Overview of Counterfactual Regret Minimization

Regret matching is an algorithm that allows players to reach an equilibrium by keeping track of the regrets of each action. In each iteration of training, an action is chosen proportional to the cumulative regret of that action. After a sufficient number of training iterations, the average strategy converges to a mixed Nash Equilibrium for a particular game.

Counterfactual Regret Minimization (CFR) applies the regret matching algorithm to sequential imperfect information games. In these games, a strategy consists of a mapping of each information set (information available to a player at that point in the game) to a set of probabilities of taking each legal action at that point in the game. Whenever an information set is visited during the training, the regret matching algorithm is used to compute the mixed strategy. For games with many information states or chance events, each iteration of CFR can be very slow. Monte Carlo Counterfactual Regret Minimization (MCCFR) uses sampling to speed up each iteration. Also, abstraction (like information set abstract or action abstraction can be used).

Approach

First, I will need to define an appropriate level of abstraction to keep the number of states manageable. Afterward, CFR will be used to train an agent on the throwing and pegging versions of the game *separately*. The reason I am separating this training is two-fold. First, given the high number of actions in a round of Cribbage, the size of the state space as well as the training time required per iteration will be very high. The regrets for the latter actions may also not be making much of an impact due to the realization weight multiplier that the CFR algorithm uses. The second reason is curiosity. By training on the two stages separately, we can see how much impact improving the throwing algorithm has versus improving the pegging algorithm over the greedy. As a small spoiler, decisions on which cards to put in the crib seem slightly less important than which cards to play during pegging, although the effects are similar in magnitude.

For the most part, all CFR agents' performance will be compared to greedy policies as a baseline. The provided GreedyThrower generates every possible 4|2 card split. Then the score of 2 cards that are discarded is added (or subtracted if the crib is on the opponent's side) to the score of the 4 cards we are keeping in our hand. This has decent performance over random, but is still not considering the playability of the kept cards in during the pegging phase of the game and does not account for the interaction of the 2 cards we are throwing with the cards that the opponent may put in the crib.

GreedyPegger makes the move that will immediately lead to the most points, breaking ties randomly. Another pegging agent that will be occasionally used is the RulePegger agent, which is stronger than GreedyPegger. RulePegger acts similarly to greedy but considers additional advantages of certain plays, such as playing a low card to prevent the opponent from getting 15, being able to get a triple if the opponent pairs the board, and being able to pair the card the opponent plays if they make 15.

The training will be done in 4 steps. First, we will train a throwing agent (throw_v1).

throw_v1 training

After being dealt 6 cards, the player has 15 possible actions of which two cards to throw to the crib. To evaluate the outcome and properly calculate regrets, the CFR algorithm will play out that round of cribbage. The utility at the end will be measured as the difference between the scores of both players. This incentivizes players not just to have a higher score than the opponent, but also maximize the difference in the score in order to try to earn extra match points through skunking. After the CFR algorithm chooses which cards to discard, the play will be handled by the given GreedyPegger, which is not the best pegging algorithm, but it is a start, and it is much better than random.

peg_v1 training

With the trained throw_v1, we will now have a very good way of determining which cards should be tossed. This method is much better than GreedyThrower (more on this in the results section). Now, we can train the CFR algorithm on pegging, knowing that both players are making fairly intelligent throwing decisions. Here, the utility will be calculated the same as above: the difference between the scores of the two players. At each stage of the round, CFR will be run recursively on every legal action that player can take, and use that to compute regrets along with the probabilities of taking those actions based on the current CFR strategy.

At this point, we will have an agent with CFR-trained throwing and pegging, ready to be compared with the greedy agent. However, I also wanted to know whether an additional round of training will improve performance. My hypothesis is that it should improve performance since a more accurate throwing policy will be used to train the new pegging policy and a more accurate pegging policy will be used to train the new throwing policy. However, it is possible that the v2 agent will perform well against v1, but still perform worse against greedy than v1, since it learned well how to play against another CFR agent rather than against a greedy one. We will test this by training v2 as well.

throw_v2 training

Now, we will take train another throwing agent, but this time, the play will be handled by the peg_v1 agent instead of GreedyPegger. Since throw_v1 decisions were evaluated using the suboptimal GreedyPegger, it did not have the most accurate perception on which cards are good to discard. This time, throw_v2 will be able to evaluate its choices using a CFR-trained pegging agent, leading to more accurate throwing performance.

peg_v2 training

peg_v1 was trained when both players were making throwing decisions with throw_v1, which is inferior to throw_v2. This affected the probabilities with which some cards can end up in the hand of the opponent, impacting the game theory optimal play. This time, we train the pegging agent again, but it will now be given cards to play using throw_v2.

Technically, we could keep doing this for v3, v4, etc, but I did not see any performance improvements beyond these 4 steps.

Action Selection

There are two ways to select actions once the CFR algorithm has been run sufficiently. Since the Nash Equilibrium is a mixed strategy, in order to play at the Nash Equilibrium, at each step, the action should be sampled from the corresponding probability distribution. This keeps the strategy less deterministic and less exploitable. Another strategy is to play the action that should be played with the highest probability. Here is a real example from throw_v2:

We are dealt 3489JK (ignoring suit) and we are the dealer (so the crib is on our side). We have 15 choices of what to do, but there are only 3 choices that CFR believes should be done with more than 0.01 probability:

1. Discard JK (84%)
2. Discard 9J (10%)
3. Discard 89 (3%)

If we were to sample, we would roughly be playing the above strategy (with the remaining 3% split among the other 12 choices). If we were to play what seems to be the "best" move, we would just discard JK every time. In this case, discard JK seems like the better move since lower cards typically have better pegging playability, which throw_v2 knows since its choices were evaluated using the fairly strong peg_v2 agent.

Here is another pegging example. Suppose we make the first play and we play a 4, ensuring the opponent cannot hit 15. Then, our opponent makes a somewhat suboptimal play and pegs a 3. We have the following cards left in our hand: 28J. Here is what peg_v2 thinks we should do:

1. Play the Jack (~0%). This makes sense - the only move that scores no points.
2. Play the 2 (35%)
3. Play the 8 (65%)

While both 2 and 8 seems reasonable, the agent wants us to play the 8 twice as frequently. And it makes sense why. Even though playing the 2 would earn us 3 points (by completing a 432 straight), it leaves us vulnerable to the opponent completing a longer straight by playing an A or a 5. On top of that, the opponent can play a 6 and get 2 points for themselves by hitting a 15, netting us only 1 point. On the other hand, playing an 8 gets us 2 points (by reaching 15), and does not leave us vulnerable to any of opponent's plays (except hitting a pair of 8s, which we can't really avoid). The argmax agent would play the 8 here 100% of the time, likely reaching better results. Many of these agents have been trained for a long time, but it is still unclear whether these probabilities would become more polarized with additional training.

What I found through testing is that the gap between the performance of sampling vs. argmax narrows over time (when the agent faces itself with the first using sampling and the second using argmax), but does not narrow to 0. The gap is fairly wide for throwing agents. Through testing, I found that throwing decisions are much more polarized, and from the 15 decisions one can make, the "best" decision usually has at least a 70-80% probability, meaning that playing even the second best decision occasionally is suboptimal. On the other hand, the argmax pegging performance only slightly beats pegging argmax. Pegging probabilities usually are not as polarized. The above two strategies are actually good examples of this. The best throwing decision should be played 84% of the time while the best pegging decision only 65% of the time. This pattern is common in other nodes. In some CFR papers that I have read, a thresholding technique is applied, where actions with probabilities below some threshold are not considered and the rest of the actions are adjusted to sum to 1. Usually, the described threshold is very low. For my agents specifically, I have found that just using argmax is still the most effective.

So, for the purposes of this project, I will *not* be sampling the actions. Instead, to evaluate the agents, I will be using the argmax. Under the assumption that the opponent is not actively adjusting to my strategy (which is reasonable since GreedyPegger, the Monte Carlo agent, or other CFR agents that I test against do not observe the opponent's strategy), it makes sense to make the best action every time without worrying about exploitability. The resulting strategy will not be a Nash Equilibrium, but it will have better performance against other agents. I have tested this. When comparing a certain CFR Thrower and Pegger against itself, with one sampling actions and the other doing the argmax, the argmax action selection outperforms by more than 0.1 matchpoints (95% confidence: 0.095-0.105). This same argmax agent beats greedy by 0.327 matchpoints, while the sampling version only beats greedy by 0.218 matchpoints, further demonstrating that taking the most probable action every time leads to better results.

Abstraction

It is crucial to choose a proper level of abstraction to use CFR. The reason is because of the large number of states that Cribbage has. Just to start, each player is dealt 6 cards. We can calculate the number of ways of doing this using the combination formula: $\binom{52}{6} \times \binom{44}{6} \approx 1.9 \times 10^{14}$, which is already not feasible to keep track of. There are two abstraction decisions to make: one for the throwing and one for pegging.

Throwing Abstraction

We received 6 cards and must choose 2 cards to throw to the crib. That is $\binom{52}{6} \approx 20 \times 10^6$ possibilities, which is fairly high. One way to simplify this is to ignore suits (and thus, flush possibility). We take the 6 cards, and sort them by rank. Using this technique, there are about 18k possible states (determined by looking at the number of nodes CFR creates when running it with this level of abstraction). In this case, each state is specified with a sorted rank string such as "478JJK," representing the six card ranks in sorted order. We also need to know in this case who the dealer is (and thus whether we are throwing cards into our own crib or the opponent's), which doubles the amount of states fo 36k, which is fairly small.

We can also treat flushes more intelligently. We do not care about the suits themselves. We only want to know whether there are at least 4 cards of the same suit, and if so, which ones they are. In the case that fewer than 4 cards share a suit, the state is represented the same way as the unsuited case. However, if 4+ cards share a suit, we separate them from the others. For instance, upon being dealt 4s, 4d, 5s, Ts, Qh, Ks (in any order), the state will be represented as "45TK|4Q" where the divider separates the suited cards from the others and both parts are sorted by rank. This leads to 100k unique states and 200k taking the dealer into account (this is determined empirically).

Pegging Abstraction

In order to choose which card to play, the player must know the cards in their own hand, which will be represented as a sorted rank string (similar to throwing abstraction). You cannot peg a flush in cribbage, so suits are irrelevant here. We also want to know which cards are in front of us in the current round (in order, since it matters for straights, pairs, etc). Lastly, it will be useful to keep track of cards that have already been played in prior rounds (by both players), since that would change the probabilities of the potential cards that the opponent may have remaining in their hand (this is the imperfect information part).

After writing and running an enumeration script in Python that keeps track of every unique info state, this level of abstraction leads to roughly 26 million states, assuming every possible card combination can be used for pegging. This is not a correct assumption, since with an intelligent throwing agent, certain card combinations will never occur and thus the number of states will be lower. We can also simplify this and have CFR only choose the first two cards that will be played, letting the greedy agent decide how to play the remaining two cards in the hand. This would reduce the number of states to about 1.6 million (again, assuming every combination is possible). I will start with the simpler case and potentially try the other one if computational resources permit.

Speed and Space Considerations

When first beginning this endeavor, I had a rather inefficient way of storing CFR nodes. Here, I want to elaborate on some space and time-saving decisions that I took to reduce space and speed everything up.

While the throw and peg nodes are slightly different, most of the functionality is identical, so they inherit from the CFRNode class in Java. Every such node must keep track of the regretSum, strategy, and strategySum arrays, whose length is the same as the number of actions possible (which is always 15 for throwing and up to 4 for pegging).

One initial inefficiency I had was that I set the number of actions to be 4 for pegging in all cases, even though sometimes there are fewer than 4 legal actions. Setting the number of actions to be the number of legal actions instead of hard-coding to 4 allowed me to save some storage space. On top of that, I previously had the three arrays be of double[] type, which I ended up changing to float[] since this saves a lot of space without meaningfully reducing the precision that we need. This is due to the fact that a double takes 8 bytes of space while a float only needs 4 bytes.

These nodes are stored in a hashmap indexes by the infoset, which is a String. At first, I made these node classes implement the Serializeable interface and have the infoset be one of the class attributes as well. Then, I was able to save all this to a .ser file and rebuild the hashmap using the saved infoset upon loading the file back into memory. For PegNodes, loading a fully trained agent into memory took over 1 minute, which significantly slowed down testing. I thought that an improvement could be made since this was only an 80 mb file.

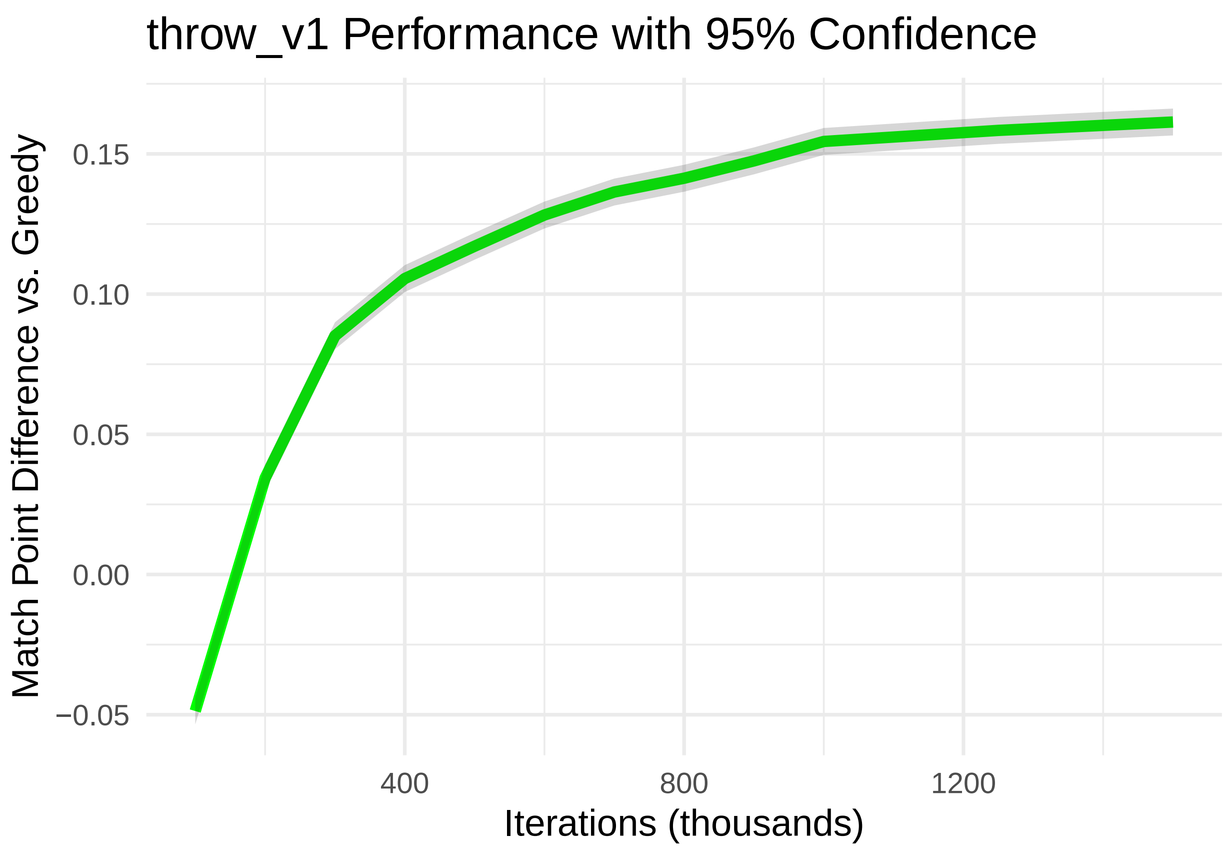
The solution was simple. I used the Gson library to convert the entire hashmap into a String (now, the infoset no longer needed to be a CFRNode attribute). Then, I used a lossless String compression algorithm found online that relies on the DeflaterOutputStream class to compress the String and save is as a txt file. This reduced the memory for a PegNode file required from 86mb to just 33mb and reduced the time to load it into memory from 60 seconds to under 4 seconds. This made testing much easier. All of this loading and compressing code is located in the NodeLoader class.

Training

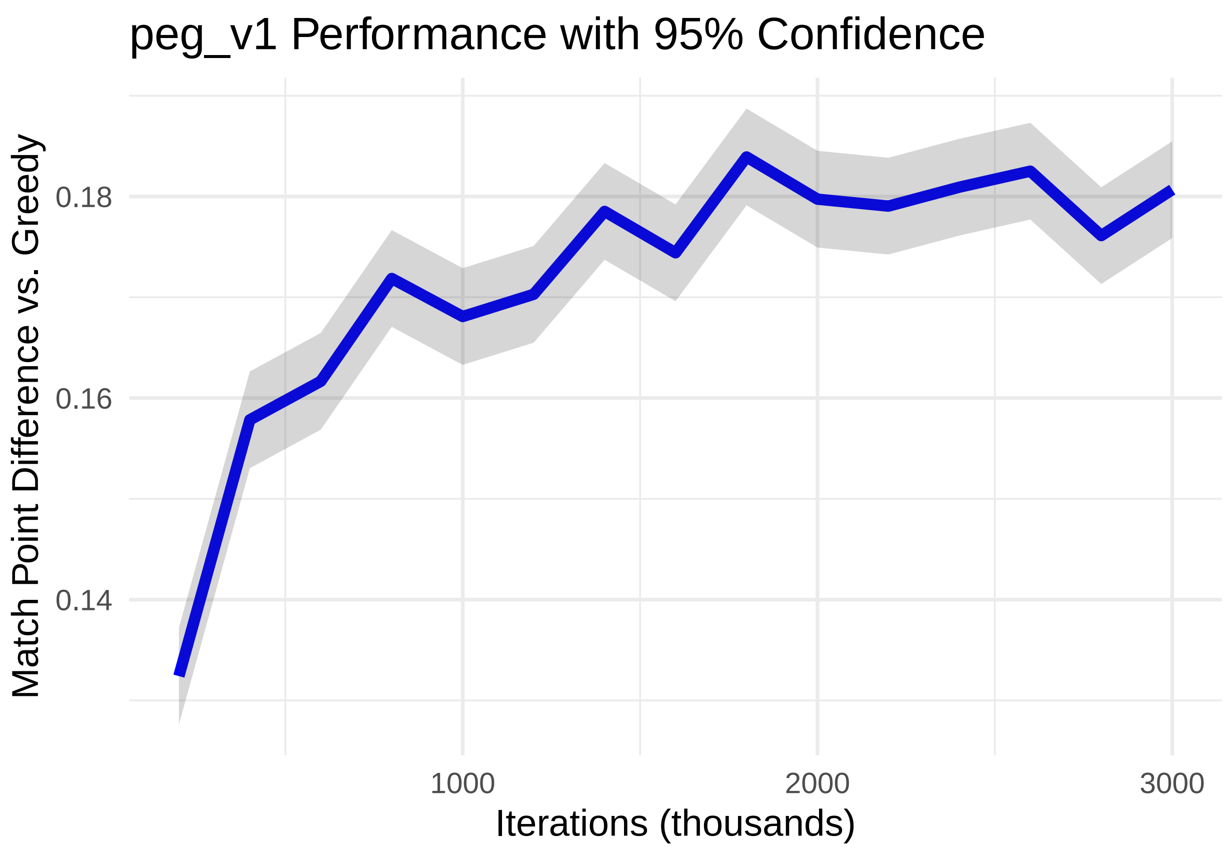
v1 Results

I followed the four step process outlined above for the suit-blind throwing abstraction and 2-card-play pegging abstraction. The following graphics demonstrate the performance of throw_v1 and peg_v1 separately (paired with a greedy counterpart) against a fully greedy agent (greedy throwing and greedy pegging) as a function of the number of iterations.

```r
throw_training <- data.frame(iterations = c(100, 200, 300, 400,
    500, 600, 700, 800, 900, 1000, 1250, 1500), matchpoints = c(-0.048688,
    0.034376, 0.0851280000000001, 0.105536, 0.117076, 0.128208,
    0.136404, 0.141324, 0.147512, 0.15444, 0.158384, 0.161376))
throw_training$lower <- throw_training$matchpoints - 0.0048
throw_training$upper <- throw_training$matchpoints + 0.0048


peg_training <- data.frame(iterations = c(200, 400, 600, 800,
    1000, 1200, 1400, 1600, 1800, 2000, 2200, 2400, 2600, 2800,
    3000), matchpoints = c(0.132412, 0.15784, 0.161664, 0.171864,
    0.168088, 0.170292, 0.17852, 0.174412, 0.183924, 0.179732,
    0.179044, 0.180924, 0.182512, 0.176116, 0.180676))
peg_training$lower <- peg_training$matchpoints - 0.0048
peg_training$upper <- peg_training$matchpoints + 0.0048

ggplot(throw_training, aes(iterations, matchpoints)) + geom_line(color = "green",
    linewidth = 2) + geom_ribbon(aes(ymin = lower, ymax = upper),
    alpha = 0.2) + labs(title = "throw_v1 Performance with 95% Confidence",
    x = "Iterations (thousands)", y = "Match Point Difference vs. Greedy") +
    theme_minimal(base_size = 14)
```

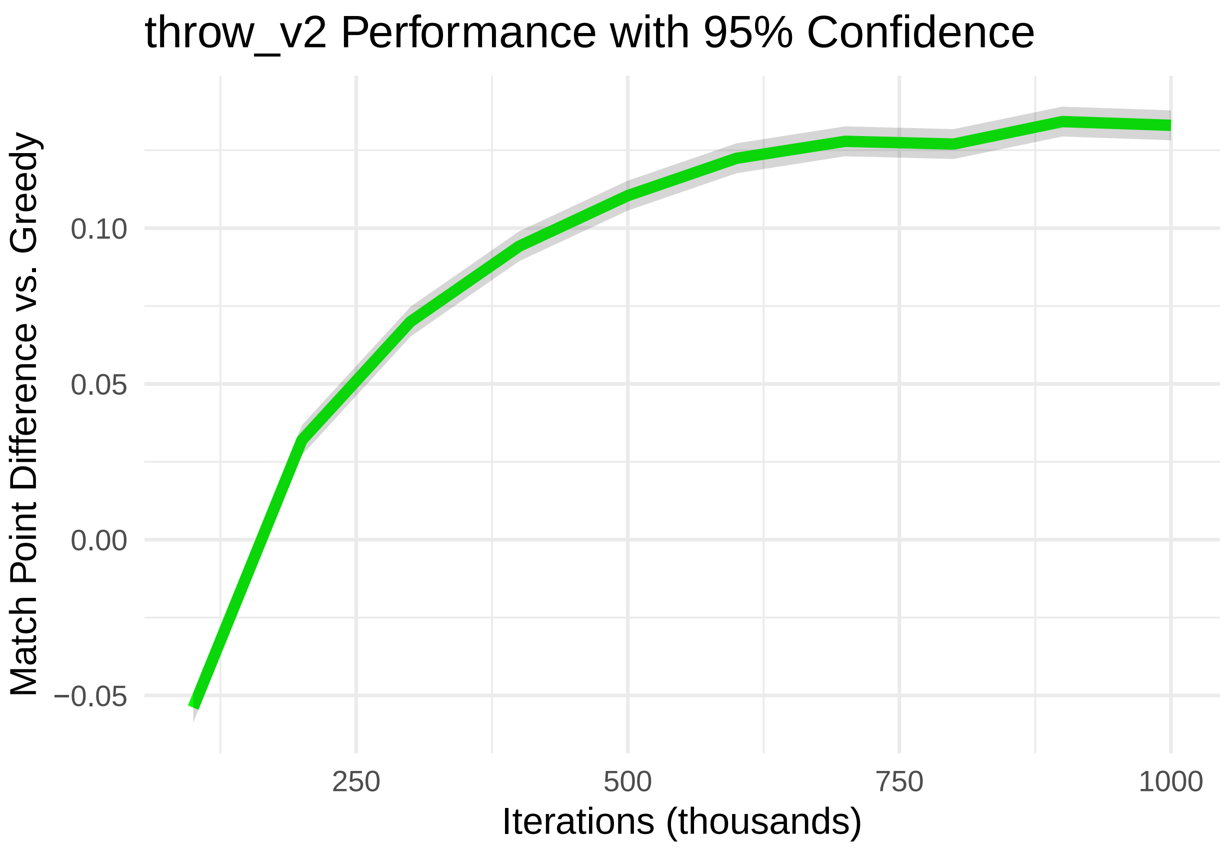


```r
ggplot(peg_training, aes(iterations, matchpoints)) + geom_line(color = "blue",
    linewidth = 2) + geom_ribbon(aes(ymin = lower, ymax = upper),
    alpha = 0.2) + labs(title = "peg_v1 Performance with 95% Confidence",
    x = "Iterations (thousands)", y = "Match Point Difference vs. Greedy") +
    theme_minimal(base_size = 14)
```

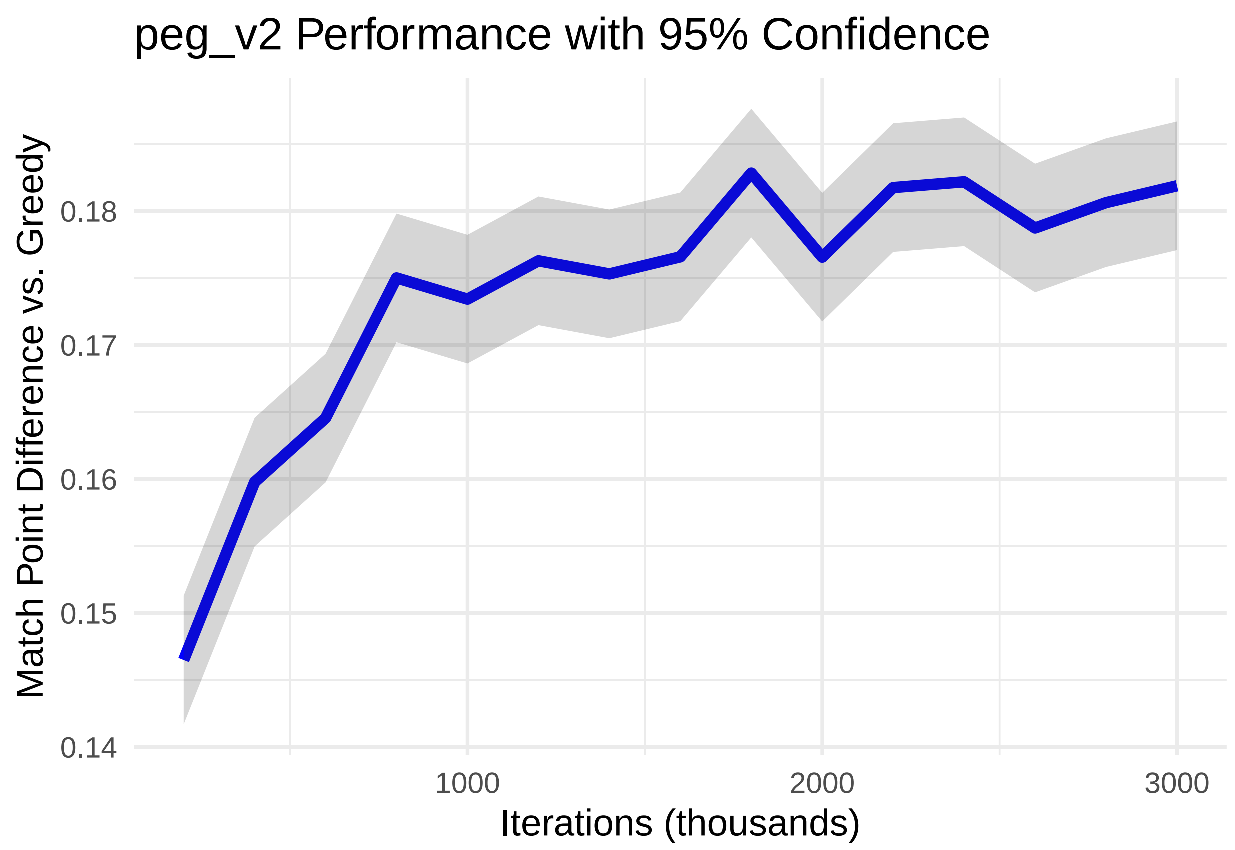


As seen from the graphs, throw_v1, after 1 million training iterations, can beat the greedy agent by about 0.16 match points on average. To train for 100k iteration, I must run the algorithm for about 10 minutes on my laptop. For peg_v1, the number is about 0.18 match points. When working together, they can beat a fully greedy agent by 0.3404 (95% interval 0.211-0.221). While throw_v1 may not have completely converged yet, I will show performance of various versions when they are fully trained further in the report ### v2 Results Below, you can now find the v2 performance against greedy.

```
throw_training <- data.frame(iterations = c(100, 200, 300, 400,
    500, 600, 700, 800, 900, 1000), matchpoints = c(-0.0538959999999999,
    0.0319360000000001, 0.0699920000000001, 0.094184, 0.110432,
    0.122404, 0.127856, 0.12698, 0.134196, 0.133))
throw_training$lower <- throw_training$matchpoints - 0.0048
throw_training$upper <- throw_training$matchpoints + 0.0048


peg_training <- data.frame(iterations = c(200, 400, 600, 800,
    1000, 1200, 1400, 1600, 1800, 2000, 2200, 2400, 2600, 2800,
    3000), matchpoints = c(0.146496, 0.159772, 0.164548, 0.175008,
    0.173424, 0.176284, 0.175308, 0.17658, 0.182832, 0.176556,
    0.181748, 0.18218, 0.178732, 0.180624, 0.181884))

peg_training$lower <- peg_training$matchpoints - 0.0048
peg_training$upper <- peg_training$matchpoints + 0.0048

ggplot(throw_training, aes(iterations, matchpoints)) + geom_line(color = "green",
    linewidth = 2) + geom_ribbon(aes(ymin = lower, ymax = upper),
    alpha = 0.2) + labs(title = "throw_v2 Performance with 95% Confidence",
    x = "Iterations (thousands)", y = "Match Point Difference vs. Greedy") +
    theme_minimal(base_size = 14)
```



```
ggplot(peg_training, aes(iterations, matchpoints)) + geom_line(color = "blue",
    linewidth = 2) + geom_ribbon(aes(ymin = lower, ymax = upper),
    alpha = 0.2) + labs(title = "peg_v2 Performance with 95% Confidence",
    x = "Iterations (thousands)", y = "Match Point Difference vs. Greedy") +
    theme_minimal(base_size = 14)
```



Here, it seems that there is no performance increase in v2 over v1 when playing against the greedy agent. In fact, while the pegging performance against greedy stays the same, it seems that throwing got worse (only 0.13 advantage instead of about 0.16 from v1). Could it really have gotten worse?

I had throw v1 play against throw v2 for 250,000 iterations (using greedy for pegging). The result was throw v1 winning by 0.016. However, when we do the same matchup but use v1 pegging instead of greedy, the results change. Now, the throw v2 + peg v1 combination beats throw v1 + peg v1 combination by 0.021 match points. I claim that throw v2 plays closer to the Nash equilibrium, but for it to be effective, it also needs a decent pegging policy.

Peg v2 did not see any improvement over peg v1. When facing each other, using either a greedy throwing policy or throw v2, their performance difference is not statistically different from 0.

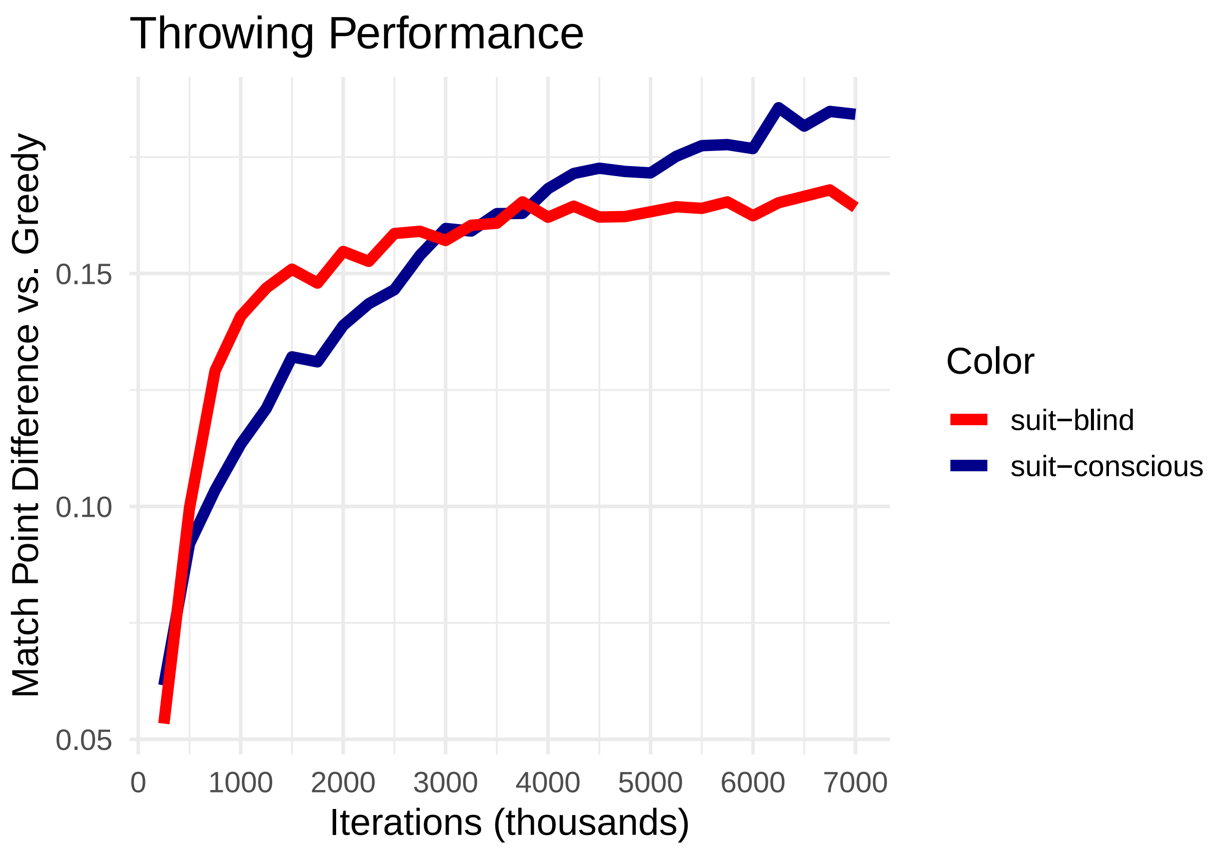
Playing against each other throw_v2 / peg_v2 has an advantage of 0.0167 matchpoints compared to throw_v1 / peg_v1. On top of that the v2 agent beats the greedy agent by 0.3265 match points.

Do flushes matter?

I wanted to investigate the extent to which paying attention to suits matters when throwing. As a reminder, the infoset will be the same as before if fewer than 4 cards share the same suit. Otherwise, the infoset will have it designated which 4+ cards share the suit with each other.

This training was significantly longer. Part of the reason is that there are not about 200k different infosets (instead of 36k). I trained for flush-conscious agent for 7M iterations (about 14 hours). I will compare its performance (using greedy pegging) against a flush-blind throwing agent as evaluated against a fully greedy agent.

```r
throwing_training <- data.frame(iterations = c(250, 500, 750,
    1000, 1250, 1500, 1750, 2000, 2250, 2500, 2750, 3000, 3250,
    3500, 3750, 4000, 4250, 4500, 4750, 5000, 5250, 5500, 5750,
    6000, 6250, 6500, 6750, 7000), flush = c(0.061572, 0.091964,
    0.103512, 0.113348, 0.121024, 0.132112, 0.131036, 0.138836,
    0.143516, 0.146528, 0.153924, 0.159664, 0.159112, 0.162852,
    0.162912, 0.168216, 0.171468, 0.172604, 0.171912, 0.171592,
    0.175148, 0.177448, 0.177672, 0.176816, 0.185596, 0.181636,
    0.184796, 0.184176), noflush = c(0.053376, 0.0996680000000001,
    0.129136, 0.140844, 0.146888, 0.150936, 0.147936, 0.154732,
    0.152608, 0.15858, 0.15906, 0.157092, 0.160356, 0.1608, 0.165384,
    0.162076, 0.164452, 0.16212, 0.162224, 0.163256, 0.164328,
    0.163988, 0.165376, 0.1624, 0.165196, 0.166572, 0.167944,
    0.164188))

ggplot(throwing_training) + geom_line(aes(x = iterations, y = flush,
    color = "suit-conscious"), linewidth = 2) + geom_line(aes(x = iterations,
    y = noflush, color = "suit-blind"), linewidth = 2) + labs(title = "Throwing Performance",
    x = "Iterations (thousands)", y = "Match Point Difference vs. Greedy") +
    scale_color_manual(name = "Color", values = c(`suit-conscious` = "darkblue",
        `suit-blind` = "red")) + scale_y_continuous(breaks = seq(-0.05,
    0.2, length.out = 6)) + scale_x_continuous(breaks = seq(0,
    8000, length.out = 9)) + theme_minimal(base_size = 14)
```



At 7M iterations, the suit-conscious version beat the unsuited version (both using v2 pegging) by 0.0189 match points. When using the most complex pegger we have (16M infosets - trained and explained below), the edge is 0.0244. Against a greedy agent, the suit-conscious throwing has a bigger edge than the suit-blind (0.375 vs. 0.364). While difficult to estimate the exact importance of flushes, it seems than 0.01-0.03 match points is close to that value. I will refer to this suit-conscious agent as the v3 throwing policy.

Improving Pegging

It is clear that we are achieving very high performance for throwing (>0.18 match points). However, it is possible that improvement could be made for pegging. v1 & v2 of pegging only allowed CFR to choose the first 2 cards to throw, letting greedy make the next choice. We can try increasing the complexity of the agent by letting it make all 3 decisions (since the 4th decision is forced). This will lead to at least 10X the infosets.

Another idea is to make the agent simpler to see if faster training helps performance. For this, I will only include the current run sequence as well as cards in our hand in the infoset, meaning that previously played cards by either us or the opponent will be ignored.

3 Cards - Current Run Sequence Only

I tried allowing CFR to make all the throwing decisions. This would require more infosets. This 3-card version would only include the current hand and the current run in the infoset (not the cards played previously). This would still need 6M states, 6X more than previously. Using throw v2 and this agent, the performance against greedy is 0.339 (0.0125 better than throw v2 and peg v2).

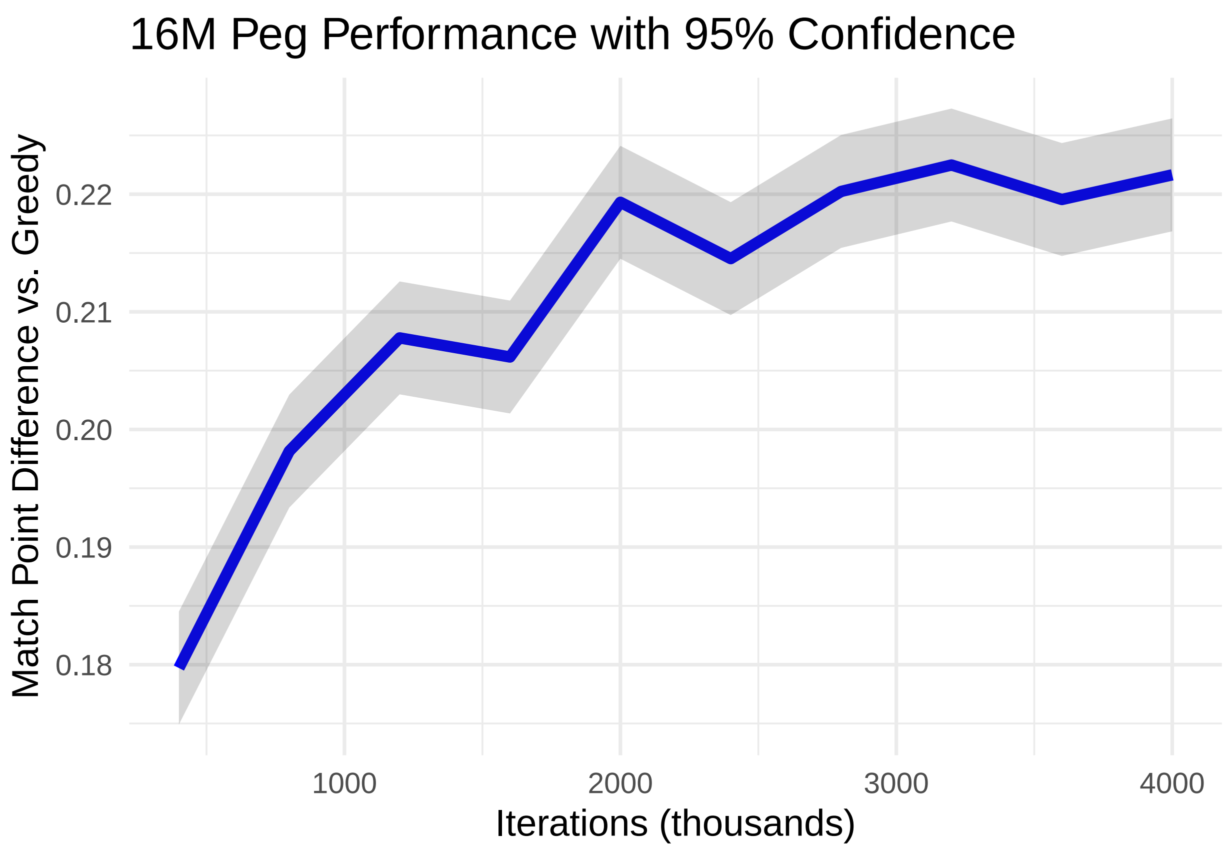
Against greedy using greedy throwing, this agent beats greedy by 0.2237. This agent against peg_v2 (both using throw v2 throwing) has a 0.0373 advantage. Indeed, it seems that having CFR make all the pegging decisions is better than just some of them, which is not surprising. I will refer to this agent as the v3 pegging agent.

All 3 Cards - Including Previous Cards

To have the CFR pegging agent make the first 3 pegging decisions (since the 4th is forced), more infosets are needed. Specifically, there will be about 16M. Due to the increased size, I am unable to train this version on my laptop because Java quickly runs out of heap space.

The solution was to build a .jar file and run it on the Zoo instead, which has 4X the RAM of my laptop, and this worked.

```r
peg_training <- data.frame(iterations = c(400, 800, 1200, 1600,
    2000, 2400, 2800, 3200, 3600, 4000), matchpoints = c(0.179716,
    0.198156, 0.207784, 0.20616, 0.219316, 0.214524, 0.220236,
    0.222484, 0.219552, 0.221648))

peg_training$lower <- peg_training$matchpoints - 0.0048
peg_training$upper <- peg_training$matchpoints + 0.0048

ggplot(peg_training, aes(iterations, matchpoints)) + geom_line(color = "blue",
    linewidth = 2) + geom_ribbon(aes(ymin = lower, ymax = upper),
    alpha = 0.2) + labs(title = "16M Peg Performance with 95% Confidence",
    x = "Iterations (thousands)", y = "Match Point Difference vs. Greedy") +
    theme_minimal(base_size = 14)
```



When using v2 throwing, this agent has a 0.038 advantage over peg v2. However, it actually has no advantage over the previous 6 million infoset agent. This would lead us to conclude that knowing which cards were played in previous runs is not very important for optimal decisions now. All that is needed is the hand we have as well as the current run.

With this being our most complex agent, when combined with the suit-conscious throwing policy, it can beat greedy by 0.3895 match points (0.3845-0.3945). Against throw v2 + peg v2, this combination of the most complex throw and peg agent has an advantage of 0.0957 match points. At the same time, the downside is much more time and space required. I will refer to this agent as the v4 pegging agent.

Other Methods

I also made attempts to use other algorithms to play Cribbage and compare that performance to the existing agents that we have (CFR, Greedy, and Random).

Using Schell's Tables for Throwing

At this link, there are tables that show an expected value on how many points the crib will contain if a player discards two specific cards into it (Schell, 2002). There is one table for if a player discards those cards to their own crib versus their opponent's crib. The provided greedy throwing algorithm tried to maximize the sum of our hand score and the points we get from the 2 cards we put in our crib or the difference between our hand score and the points from the 2 cards that are thrown into the opponent's crib, depending on who is the dealer. Of course, this is simplistic, since the 2 cards thrown into the crib interact with other cards there as well, not just with each other.

Instead, we will now consider the score of our hand as well as the expected points we will receive (or give to the opponent) by discarding the other two cards for every partition. This still does nothing to evaluate how good the cards we keep are for the pegging phase of the game, but it is a definite improvement.

Testing this throwing agent against the greedy thrower (both using greedy pegging), yields a 0.122 (0.117-0.127) match point advantage. This is a large improvement. At the same time, our v2 thrower outperforms 0.02 match points (0.015-0.025), with our stronger, flush-conscious thrower outperforming by even more.

Monte Carlo Pegging

One idea was to use simulations to decide which card to use for pegging. When the MC agent has more than one move available, it tries making both moves, and then simulates the rest of the game some number of times (20 in my case). Whichever move leads to better results on average, it ends up doing.

For the simulation, the difficulty is figuring out what the opponent's moves may be. Even at the beginning of the game, we know 7 cards that the opponent does not have - the 4 in our hand, the 2 we discarded, and the turn card. Thus, we know that our opponent has some combination of cards that do not include those. So, as a start, for each simulation, we can generate a random hand for the opponent from those 45 possible cards. At later stages of pegging, when the opponent has already played some cards, we can take that information into account. For example, if the opponent has already played a 7, we will keep generating hands for the opponent until we get such a hand that after running some intelligent thrower on it, the kept cards still include a 7. Thus, these generated hands are not completely random; instead, they ensure that they are consistent with previously played cards by the opponent.

Once the opponent hand is generated, pegging decisions will be decided by some pegging policy. For testing, I used the RulePegger for pegging and the SchellThrower (introduced in the previous section) for the throwing policy to help simulate the opponent's hands.

One major difficulty was the "Go!" part of the game.

The performance was 0.17 better than RandomPegger (with GreedyThrower). However, GreedyPegger outperforms it by 0.11 match points, so it is not very effective at choosing the next move.

Human vs. CFR Results

While I was unable to play my agent tens of thousands of times to get statistically-significant results, after playing 10 games against the v2 thrower and pegger, I lost 8 of those games, twice via skunking. At the same time, I am by no means an excellent Cribbage player, so as a future extension, it would be interesting to have a professional play one of the CFR agents for a few hundred games.

CardGames Bill

One of the most popular websites to play Cribbage on against other players or against an AI is cardgames.io. I reached out to the creator via Twitter and was able to get the logic for the strategy behind Bill, the Cribbage AI that the site uses. On top of that, the creator added a way for me to try my agents on the website directly with Selenium; the video of one of these attempts is on the HTML webpage that I submit.

First, I will share the results of playing Bill through the UI with Selenium. Essentially, I had a Python script interact with the Javascript on the CardGames website and send the infosets to my Java code through a subprocess. I ended up having my throw v2 / peg v2 agents play 10 games against Bill in this fashion. These agents were slightly weakened because keeping an accurate history of plays was difficult. So, I had my peg v2 agent make the first 2 pegging decisions as usual, but the next decision was not made my GreedyPegger (since it requires a complete PeggingHistory object). Instead, I just played the highest value legal move. After 10 games, the score was 5:5, with the CFR agents winning 3.7 points more, on average. Of course, this is nowhere close to statistical significance, so below, I am sharing the results of playing the Bill policy in Java.

Bill's throwing and pegging implementations are modest improvements over the greedy policies we have here. Both of our greedy policies break ties randomly. Bill takes all the tied moves and tries to evaluate their potential for making future straights, future 15 sums, etc (more can be viewed in submitted source code). When compared to a greedy agent, Bill beats it by 0.08 matchpoints, with a 53% win rate. Of course, our CFR agent is very strong, so over a large number of trials, it beats Bill significantly. The v2 policies win by 0.21 matchpoints with a 58% win rate, while the suited v3 thrower and v4 pegger win by 0.28 matchpoints with a 61% win rate. Bill also loses to the Schell/RulePegger policy by about 0.1 matchpoints. Overall, Bill is not very strong.

I am very grateful to the creator of CardGames for providing me with Bill's logic and updating the website code to allow my Selenium script to interact with it.

Results Summary

I have mentioned many different match points advantages in this report. In this section, I want to summarize them. I also added some comparisons of the best non-CFR combination (Schell Thrower + RulePegger) against some CFR agents. All of these are run for 250k games and have a 95% confidence interval +- 0.0048 match points.

Thrower 1	Pegger 1	Thrower 2	Pegger 2	Agent 1 Match Point Advantage
v2	v1	v1	v1	0.021
v2	greedy	v1	greedy	-0.016
greedy	v2	greedy	v1	0.0
v2	v2	v2	v1	0.0027
greedy	v3	greedy	greedy	0.2247
v2	v3	v2	v2	0.0373
v2	v4	v2	v3	0.0
v2	v4	v2	v2	0.038
v2	v3	greedy	greedy	0.339
v3	v2	v2	v2	0.0486
v3	v4	greedy	greedy	0.3895
v2	v2	greedy	greedy	0.3265
v3	v4	v2	v2	0.0957
schell	rule	greedy	greedy	0.19
v2	greedy	schell	greedy	0.01957
v2	v2	random	random	1.828
v3	v4	schell	rule	0.247
v2	v2	schell	rule	0.174
v2	v2	cardgames	cardgames	0.21
v3	v4	cardgames	cardgames	0.28

Agent Memory Usage

It is clear that throw v3 and peg v3 & v4 outperform all other agents by a decent margin. Here, I wanted to quickly summarize the storage required for the different CFR agent types so that their performance can be considered in the context of its complexity. The storage is rounded to the nearest megabyte.

Policy	Storage
peg v1	21 mb
peg v2	20 mb
peg v3	75 mb
peg v4	257 mb
throw v1	5 mb
throw v2	5 mb
throw v3	28 mb

Is Cribbage a Game of Luck or Skill?

In poker, many beginner players incorrectly claim that the game is mostly luck, since no amount of skill can make up for the fact that you are being dealt worse cards than your opponent orbit after orbit after orbit. In reality, in the long-term, poker is a game of skill since neither player is favored by the cards dealt.

Cribbage is similar. Even the purely random player beats our v2 CFR agent more than 1% of the time. As in poker, in cribbage, serious opponents never play randomly, so looking at just one game, there is always a possibility that the stronger player will lose. However, over the long term, it will become more and more clear which player is stronger.

Sometimes, serious Cribbage players play Cribbage for money. I want to simulate a simple scenario involving two very skilled, but not equally skilled players.

The first player will be a CFR agent that uses v2 throwing and pegging. It is very strong, beating the greedy agent by 0.32 match points.

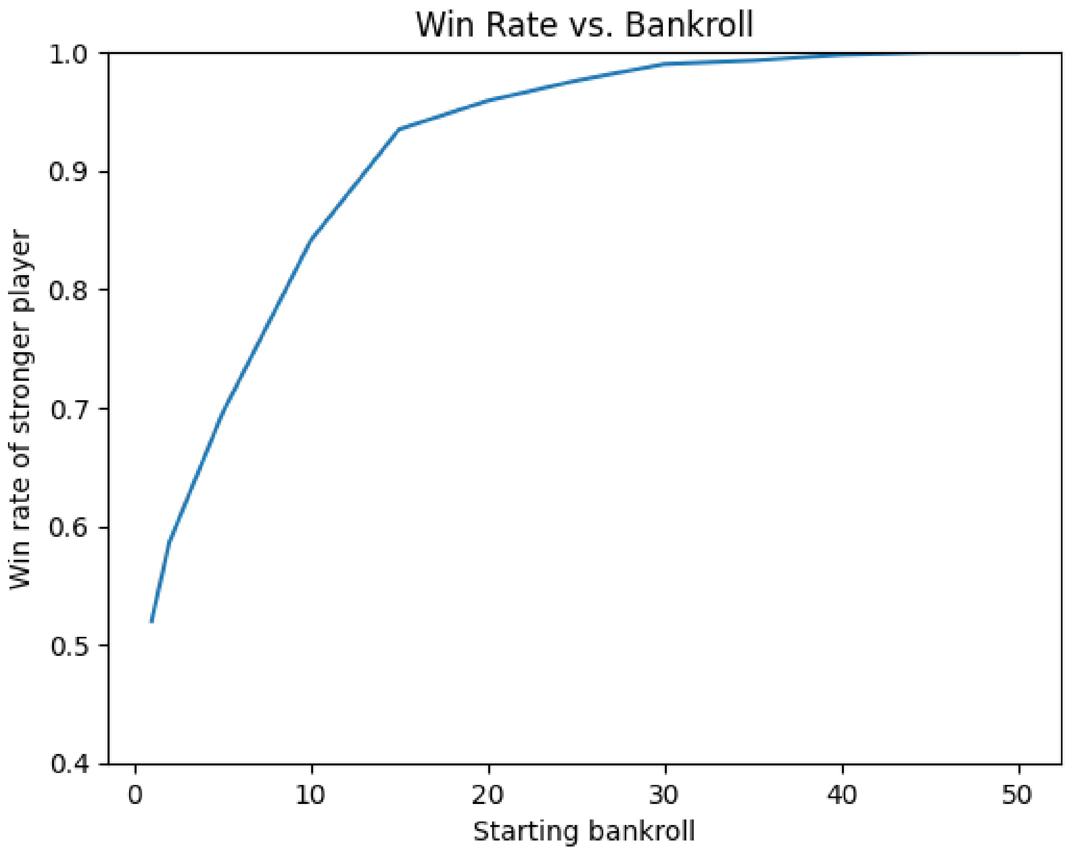
The second player will be our best CFR agent: one that considers suits when throwing and makes all pegging decisions with 16M infosets. This one beats greedy by 0.39 match points.

According to a matchup of the two against each other, the second agent is about 0.1 matchpoints stronger. Here are the results for 10,000 games: 0.10350000000000004 0.6149-0.5114 {-1=4082, 1=4669, -2=510, 2=725, -3=4, 3=10} (9.1058 hands/game)

If these players were to bet money on this, what would happen? Which player would run out of money first? How does this vary with the starting bankroll?

I will simulate this. We will make the following assumptions. There is no rake, like in poker. Both players put in $1 and the winner takes all (without regard to skunking). Both players start with the same bankroll and everything ends once one player has no money left.

For this simulation, we will say that the second player has a 54% win rate, which we calculated from the results above. Let's see what happens. I will do this simulation in a Python Jupyter notebook for starting bankrolls between 1 and 50. I will run 1000 simulations for each case to see how many times the stronger player will win. Here are the results:



When the starting bankroll is just 1, the stronger player, as expected, wins just over half the time. With a bankroll of just 5, the stronger player wins 70% of the time. For bankrolls of 30 or more, the stronger player has over a 99% win rate. The number of games played drastically reduces the variance in the results, making it very clear who the stronger player is. Despite both of these agents being very strong and one only having a small (although significant edge), they would win a lot of money from the first player with this small edge as long as they got to play a good number of games.

Conclusion and Future Research

In conclusion, Counterfactual Regret Minimization is very effective in playing Cribbage, beating the greedy agents with a large margin, and even the smart rule-based pegging and Schell's throwing. For someone attempting to write a very good program for Cribbage, the costs associated with using CFR are low - at most a few dozen hours for training the most complex agents and a few hundred megabytes of storage.

In the future, it would be interesting to have a professional play against these CFR agents. It would also be interesting to see whether the professional would learn to exploit the CFR agents after seeing that they are not playing the Nash Equilibrium and are playing the most probable action instead. On top of that, it would be interesting to see if, over the long term, a professional could beat the Nash-Equilibrium version of our agent, or whether the chosen level of abstraction is still insufficient to beat the best in the world.

References

Brown, N., & Sandholm, T. (2017). Libratus: The superhuman AI for no-limit poker - ijcai. International Joint Conferences on Artificial Intelligence Organization. Retrieved September 19, 2022, from https://www.

ijcai.org/Proceedings/2017/0772.pdf

How can I easily compress and decompress strings to/from Byte Arrays? Stack Overflow. (n.d.). Retrieved April 11, 2023, from https://stackoverflow.com/questions/10572398/how-can-i-easily-compress-and-decompress-strings-to-from-byte-arrays

How to play cribbage - how to play. Bicycle Playing Cards. (n.d.). Retrieved September 19, 2022, from https://bicyclecards.com/how-to-play/cribbage/

Schell, M. (2002, February). Schell's discard tables. Retrieved April 11, 2023, from http://www.cribbageforum.com/SchellDiscard.htm