Playing Cribbage with Reinforcement Learning and Minimax

Ethan Partida

April 26, 2022

Abstract

This project created a program to play the card game cribbage. Previous work on cribbage has focused on the discard phase of the game, using only a basic algorithm for the play phase. However, determining a hand's potential performance in the play phase affects the optimal choice in the discard phase of the game. By ignoring this effect, current cribbage algorithms make un-optimal choices. We seek to alleviate this by creating an efficient and intelligent algorithm for the discard phase of cribbage. We explored two approaches to this algorithm, a reinforcement learning algorithm and a minimax algorithm. Due to the randomness of the play phase and our lack of computing power, we were not able to successfully train the reinforcement learning algorithm. By specifically tuning the minimax algorithm to suit the game of cribbage, we created an efficient algorithm which consistently out-preformed the naive greedy algorithm. We also introduce a variety of new heuristics to further tune the minimax algorithm for the play phase of cribbage.

1 Problem Description

Cribbage is a two-player, competitive, round-based card game split into two distinct phases, a discard and play phase. Each player is first dealt a six card hand which they reduce to a four card hand in the discard phase. They then play against each other and score points using their four card hands in the play phase. Finally, each player separately scores points based on whether their hand contains special combinations such as pairs, straights, or groups summing to fifteen. See [Con18] for a more detailed explanation of the rules.

We focused on creating an efficient and accurate algorithm, called Knoddy[1], for the play phase of cribbage. The play phase of cribbage involves the players taking turns laying down one of the cards from their hands. The order of these laid down cards is kept track of and used to create a shared card history. The players can score points when they lay down

[1]Named after the sixteenth century card game Noddy which was a precursor to cribbage. The addition of the 'K' is both for stylistic reasons and to avoid any association with the moderately controversial children's book character created by author Enid Blyton.

their cards by creating special card combinations with the shared card history. Examples of these combinations include pairs, straights, and sums to 31 and 15. Since players take turns contributing to this shared history, the players must balance scoring points from the shared history while preventing their opponents from doing the same. Since each player does not know the cards in their opponent's hand, it is difficult to determine what card choices will prevent their opponent from scoring. This is why previous cribbage algorithms, such as [Lan18], opt for a greedy algorithm. Each turn they lay down the card which maximizes their own score without considering how it will benefit or hinder their opponent. We hoped to improve this by making Knoddy account for how its actions might benefit or hinder its opponent. Knoddy not only improves the way in which a program is able navigate the play phase of cribbage, it will improve the program's performance during the discard phase as well. Since a player's choice of discard will affect their performance in the play phase, the ability to properly evaluate a hand's potential in the play phase will lead to more intelligent discard choices.

2 Algorithm Description

Mathematically, we sought to create a function, eval : GAME STATES $\to \mathbb{R}$, which assigns each game state in the play phase of cribbage a score. Using this score, we can make Knoddy always choose whatever action leads to a game state that maximizes the score. This simple procedure is given in Algorithm 1.

Algorithm 1 A basic play phase algorithm

```
function playPhase(gameState, playerHand)
    maxScore = −∞
    maxCard = null
    for each card in playerHand do
        cardScore = eval(gameState.playCard(card))
        if cardScore > maxScore then
            maxScore = cardScore
            maxCard = card
        end if
    end for
    return maxCard
end function
```

To actually implement this algorithm, we needed to answer three main questions:

1. "How do we represent a game state in the play phase?"
2. "What should the score given to a game state represent and how can we calculate it?"

3. "Cribbage is not a perfect information game. At every game state, there will be some vital game information which the player or the opponent cannot know. How can we adapt our algorithm to account for this?"

We will now address each of these questions in order.

2.1 Game State Representation

There are three main components in every game state: the player's hand, the opponent's hand and the shared card history. Both the player's and the opponent's hands consist of a collection of cards with a maximum size of four. The shared card history is also a collection of cards, this time with a maximum size of eight. Since all of these collections are small, we chose to represent them using a list. Furthermore, during the play phase of cribbage only the values of the cards matter, not the suits. This lets us represent cards using integers where Jacks are 11's, Queens are 12's, Kings are 13's and $1 - 10$'s are represented by their own numbers. An example of a play phase game state and its internal representation is given by Figure 1.

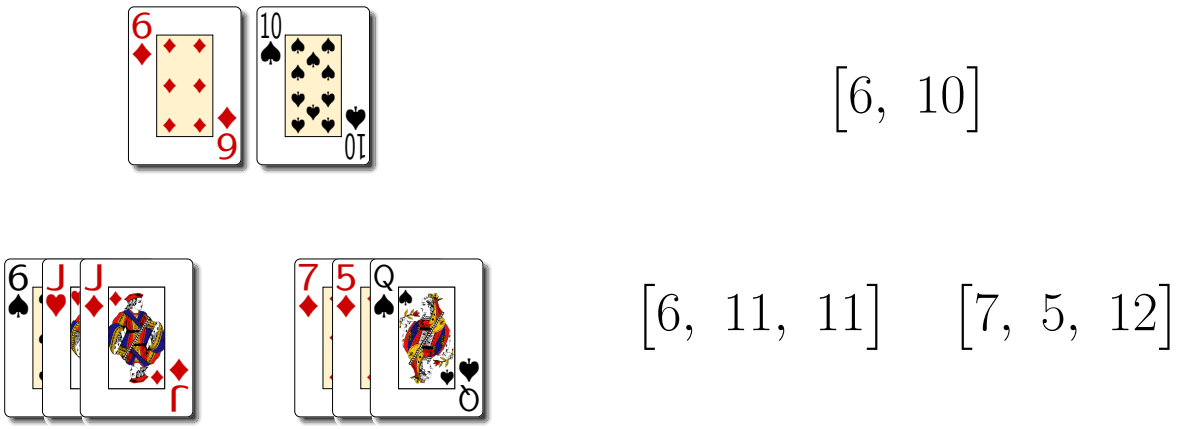

Figure 1: A game state with the card representation on the left and our internal representation on the right. In each representation the top section is the shared card history, the bottom left section is the player's hand and the bottom right section is the opponent's hand.

2.2 Score Representation and Calculation

The first score representation we considered was an evaluation function which behaved identically to the greedy algorithm. That is, the score assigned to a game state is the amount of points the most recently played card would have received. Figure 2 gives an example of this evaluation applied to a game state and its possible children. Using a game state scoring function, we can compute this evaluation with Algorithm 2.

This was the score representation used by [Lan18] and the one we hoped to improve on. Its simple design and implementation gave us an efficient algorithm to test our new evaluation algorithms against.

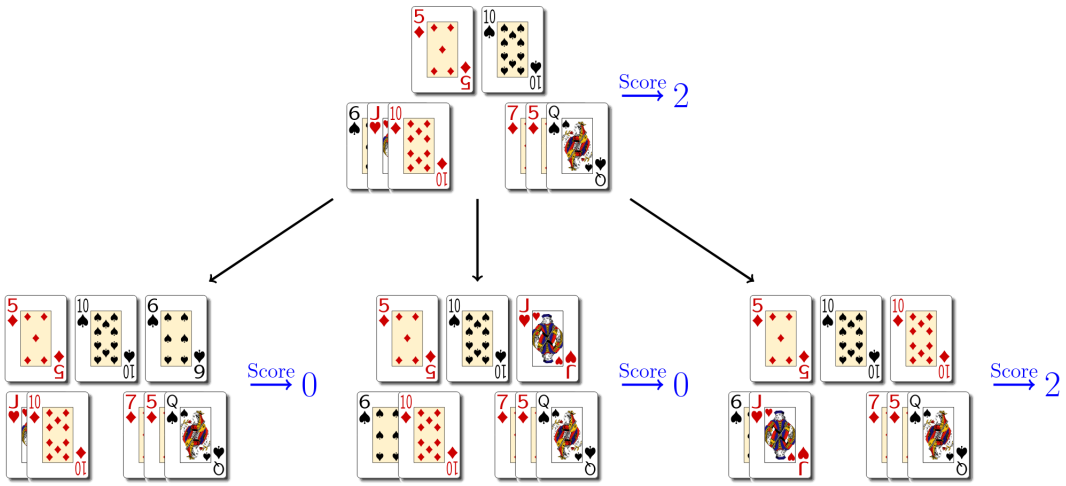

Figure 2: An example of the player's decision tree and their greedy evaluations.

Algorithm 2 A greedy evaluation algorithm

```
function GREEDY(gameState)
    return gameState.mostRecentScore()
end function
```

The next representation we considered is almost as simple to describe as the greedy evaluation but is significantly more difficult to implement. This representation hopes to predict the future by assigning each game state an estimation of the point differential between the player and the opponent after the game is finished. That is given a board state S,

$$\text{eval}(S) = \text{final estimated player score}(S) - \text{final estimated opponent score}(S).$$

If this estimation is close enough to the actual final score differential, using this evaluation along with Algorithm 1 would make Knoddy an optimal agent. If we always knew the final outcome of a given board state, we could just choose whatever actions lead to optimal board states. We tested two different approaches towards calculating this evaluation, a minimax algorithm and a reinforcement learning neural network.

To calculate this estimation, we first used the classic minimax algorithm. See section 5.2.1 of [RN21] for the basics of this algorithm. We chose to implement a minimax algorithm because it guarantees optimal decision making and has a small time complexity for our given problem. Since each player action will reduce their next possible actions by 1 and each player starts with four possible actions (four cards in their hand), there will be $4^2 \cdot 3^2 \cdot 2^2 \cdot 1^2 = 576$ states for our minimax algorithm to search. With modern computing power, this number of states can easily be searched. Algorithm 3 implements the basic minimax algorithm along with our evaluation function as described above.

We next applied a neural network approach to estimate the values of each board state. See [SB15] for a general introduction to neural networks and reinforcement learning. We

supplied the network with our game states as 1×16 arrays. The network consisted of three hidden layers of size 24 using rectified liner unit (ReLU) activation and then converged to a single output node which was activated linearly. We trained the network by playing games against the naive greedy algorithm. We started this process by letting it randomly assign each game state a score. After completing, we fit the network so that all game states that had appeared in the game had their scores shifted closer to the final score differential. By playing games against the greedy algorithm, we were able to generate our own data reducing the need for databases of cribbage games (which we were not able to acquire in a readily available format). The training procedure we used is outlined in Algorithm 4.

2.3 Accounting for imperfect information

So far we've assumed that our algorithms can access all facets of a game state. In a real game of cribbage this is not the case. The players can only see the cards in their hand, the shared card history and the size of their opponents hand but not the actual cards in their opponents hand. To account for this fact, we needed to modify both of our algorithms.

At first, the inability to view the opponent's cards renders the minimax algorithm useless. Without knowing our opponent's hand we're unable to know their possible actions and cannot construct a game tree for our game state. To get around our lack of trees, we decided to grow a forest! Although we don't know our opponent's hand, there are only $\binom{13+4-1}{4} = 1820$ possible hands. By iterating over all possible hands and running our minimax algorithm with the opponent assigned those hands, we can find a game state which maximizes the expected value of our score differential. Although this optimizes our decision making, it is not particularly time efficient. We tried to circumvent this by generating a random sample of size N from the space of all possible hands and running our minimax algorithm against these hands. To determine our choice of N, we needed to experimentally find a size which balances runtime complexity and decision making ability. This random sample approach is given by Algorithm 5.

The minimax algorithm assumes that the opponent will play perfectly and has all of the same information as the player. When Algorithm 3 runs its game simulations, it assumes that the opponent is able to see the player's hand and make decisions using that extra information. This is an incorrect assumption because throughout the game, the player's hand is hidden from the opponent. Figure 3 gives an example of when the ability to see the player's hand changes an opponent's move decision. Without knowing the player's hand, playing either the five or the three will appear as equal choices since both will result in two points for the opponent. However if the opponent can see the player's hand, they will know not to play the five because the player can immediately make a pair afterwards. This lack of information often makes opponents play moves similar to that of the greedy algorithm. We tried accounting for this by having our minimax algorithm simulate its opponent's moves using the greedy algorithm instead of the minimax algorithm. We then used this new greedy minimax algorithm with the sampling algorithm introduced by Algorithm 5. Algorithm 6 highlights this change.

To modify our neural network approach, we only needed some slight modifications. In-

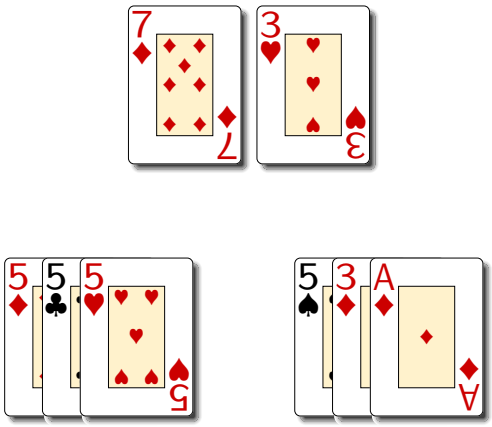

Figure 3: An example where knowing the player's hand changes the optimal move for the opponent.

stead of taking in a 1×16 array with the whole game state, we used only a 1×12 array with the player's hand and shared card history combined together.

3 Analysis

We tested our approach by running simulations of the play phase where both the player and the opponent was given a random four card hand. We had the opponent use the naive greedy algorithm, Algorithm 2, to decide its move. We then had the player use one of our various new algorithms to decide its move. To evaluate our performance after each game, we calculated the final score differential,

$$\text{Score Differential} = \text{Player Score} - \text{Opponent Score}$$

To get a baseline for our upcoming experiments, we tried having the player select random valid cards for its move. Over a simulation of ten thousand games, the random algorithm had an average score differential of -1.14. This score differential confirmed to us that the player's decisions have a significant effect on the outcome of the play phase. Although random chance comes into play, a moderately intelligent player will be able to consistently out-preform a player making random decisions.

We next compared our two minimax algorithms against the greedy algorithm, increasing each algorithm's sample size factor along the way. We did this by playing each algorithm against the greedy algorithm for one thousand random games at each sample size. These thousand games were identical for the first and second minimax algorithms so we were able to accurately compare the two. Figure 4 compares the average score differential of these algorithms at sample sizes between 10 and 400 with a step size of 10. Our data implies that assuming the opponent will play greedily results in better outcomes (average score differential of 0.12) than assuming the opponent will play using the minimax approach (average score differential of -0.11). This is to be expected because we configured our test environment so

that the opponent used a greedy algorithm. Interestingly, the sample size needed for optimal performance is quite low. We arrived at a performance plateau after increasing the sample size to fifty. This leads to an extremely fast and efficient cribbage algorithm.

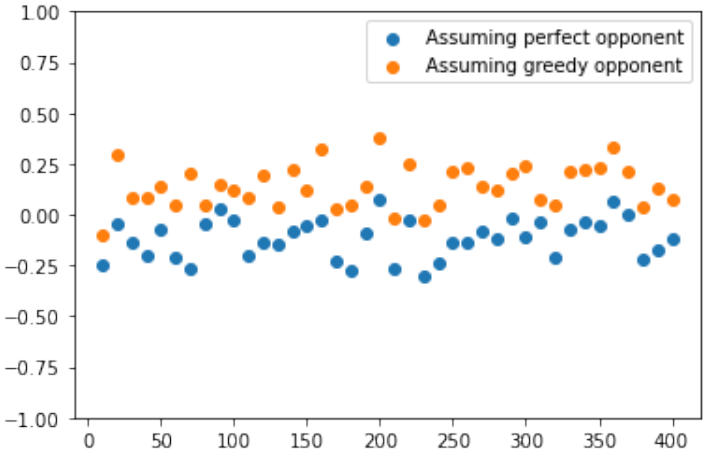

Figure 4: A comparison between Algorithm 5's performance when using Algorithm 3 and Algorithm 6 for its minimax function. The x-axis is the sample size while the y-axis is the average score differential against a greedy opponent.

Our attempt at applying the neural network to the play phase of cribbage was significantly worse than the minimax approach. Figure 5 shows the average score differential between the neural network and the greedy algorithm as we train the network in ten game increments. As expected, the program initially chooses random moves and performs as such. This is then followed by a steep increase in ability followed by a slow die off as the number of training sessions continue. This die off is likely due to over-training. However, even at the neural network's best performance (a score differential of -0.6) it plays worse than the naive greedy algorithm and either of the minimax algorithms.

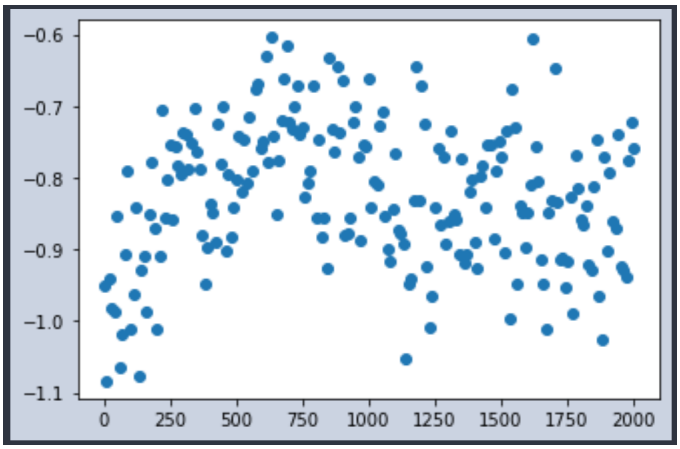

Figure 5: Algorithm 4's performance over 2000 training sessions. The x-axis is the number of training sessions while the y-axis is the average score differential against a greedy opponent.

4 Further Work

We hope to revisit this project to improve upon the neural network approach. The over-training and bad performance should be able to be corrected through tuning hyper-parameters and adjusting the training protocol. Unfortunately, this work was beyond the scope of this research project.

Additionally, we would like to explore a possible improvement to the minimax approach. When choosing sample hands to test our decisions against, we assumed that there is a uniform distribution on the space of all possible hands. However since the play phase of cribbage is preceded by a discard phase where each player prunes their hand, this assumption is not true. For instance since pairs give the player points, they are much more likely to keep pairs in their hand during the discard phase. This would alter the distribution of hands and make hands containing pairs more likely to appear in the play phase. We could account for this different distribution by simulating the discard phase of cribbage when dealing our algorithms their hands for the play phase experiments.

References

[SB15] Sutton and Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2015.

[Con18] American Cribbage Congress. *Cribbage Official Tournament Rules*. 2018. URL: `http://www.cribbage.org/NewSite/rules/default.asp` (visited on 03/05/2020).

[Lan18] Sean R. Lang. *Policy Improvement in Cribbage*. 2018.

[RN21] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. fourth. Pearson, 2021.

Algorithm 3 A basic minimax algorithm for the play phase of cribbage

```
function MINIMAX(gameState, playerHand, opponentHand, isMaximizer)
    if gameState.isDone() then
        return gameState.scoreDifferential()
    end if
    if isMaximizer then
        maxScore = −∞
        for each card in playerHand do
            cardScore = minimax(gameState.playCard(card), playerHand,
                                opponentHand, False)
            if cardScore > maxScore then
                maxScore = cardScore
            end if
        end for
        return maxScore
    else
        minScore = ∞
        for each card in opponentHand do
            cardScore = minimax(gameState.playCard(card), playerHand,
                                opponentHand, True)
            if cardScore < minScore then
                minScore = cardScore
            end if
        end for
        return minScore
    end if
end function
```

Algorithm 4 A neural network approach to the play phase of cribbage

```
function NEURALNET(network, gameState, historyStates)
    if gameState.isDone() then
        score = gameState.scoreDifferential()
        for each state in historyStates do
            network.fit(state, score)
        end for
    else
        historyStates.append(gameState)
        cardWeights = network(gameState)
        bestCard = maxIndex(cardWeights)
        return (gameState.playCard(bestCard), historyStates)
    end if
end function
```

Algorithm 5 An implementation of the minimax algorithm sampling over SampleSize many random opponent hands

```
function minimaxSample(gameState, playerHand, SampleSize)
    Scores = [0, 0, 0, 0]
    for each iteration from 1 to SampleSize do
        opHand = randomHand()
        for each card in playerHand do
            score = minimax(gameState.playCard(card), playerHand, opHand, False)
            Scores[card.index] += score
        end for
    end for
    maxScore = −∞
    maxCard = null
    for each card in playerHand do
        if Scores[card.index] > maxScore then
            maxScore = cardScore
            maxCard = card
        end if
    end for
    return maxCard
end function
```

Algorithm 6 An implementation of the minimax algorithm assuming that the opponent will play greedily.

```
function MINIMAXGREEDY(gameState, playerHand, opponentHand, isMaximizer)
    if gameState.isDone() then
        return gameState.scoreDifferential()
    end if
    if isMaximizer then
        maxScore = −∞
        for each card in playerHand do
            cardScore = minimax(gameState.playCard(card), playerHand,
                                opponentHand, False)
            if cardScore > maxScore then
                maxScore = cardScore
            end if
        end for
        return maxScore
    else
        minScore = ∞
        minCard = null
        for each card in opponentHand do
            cardScore = greedy(gameState.playCard(card))
            if cardScore < minScore then
                minScore = cardScore
                minCard = card
            end if
        end for
        return minimax(gameState.playCard(minCard), playerHand,
                       opponentHand, True)
    end if
end function
```