CS5100 Spring 2022 Final Project: Cribbage

David Anderson, Yaniv Amiri, Ashwin Sharan, Yian Ding

Project Overview:

Cribbage is a game invented by Sir John Suckling in the early 17th century as an adaption to the far more popular game at the time: Noddy. Since then, however, Noddy has become a game that is mostly out of use while Cribbage has gained popularity. In the present day it's quite popular in both a recreational and competitive setting. The complexities of the game presents an opportunity for application of intelligent concepts. We can study how the concepts of artificial intelligence can create an AI agent that makes use of the concepts that we have learned in class this year, to give rise to better thinking processes, Also explore how it does so. Rather than choosing random cribbage moves, the AI agents can make more optimal choices and hopefully beat a random agent every time. The most efficient of the AI agents should be the best Cribbage player of the models.

We have implemented 5 cribbage playing agents as described below. Briefly, we will implement an agent that plays the game randomly, an agent that greedily tries to increase its score without regard to whether it gives the opponent points, an “expert system” agent that plays according to hard coded rules, a Min/Max agent that enumerates all possible games and attempts to always make the best move given what it knows of the cards in play, and a neural network trained via temporal-difference reinforcement learning.

This project is achieved by using starter code containing implementation of prompt based game of cribbage taken from github authored by Alex Carlin

To understand the rules of the game please visit. How to play a game of cribbage?

Agent Description

Random Agent:

As we understand from the rules of the game the intelligence components lie in how to discard two cards and how to peg a card. The agent solves the discard problem by randomly discarding two cards to the crib as shown in the code below.

```
# Random agent discards the first two cards of the hand.
    def ask_for_discards(self):
        return self.hand[0:2]
```

During pegging, the random agent randomly picks a card to play from its hand.

```
# Random creates a play by randomly choosing two cars from its play.
    def ask_for_play(self, previous_plays):
        shuffle(self.hand) # this handles a case when 0 is not a
legal play
        return self.hand[0]
```

While this behavior is not particularly “intelligent”, it serves as a baseline level of play to evaluate our other agents against. Ideally, all of the other agents should outscore the random agent, and better agents should be expected to outscore the random agent by greater margins.

Greedy agent:

The greedy agent operates by simply making the best choice to push its own score up at every instant, regardless of scoring opportunities that such behavior might open up for the opponent.

For instance, for the discard problem the greedy agent simply computes the value of all the possible resulting hands (plus the score of whatever it puts in the crib if the agent is the dealer) and chooses the option of the cards that create the least value to it.

```
 def ask_for_discards(self):
        """
        For each possible discard, score and select
        highest scoring move. It will discard cards
        that creates the least value.
        """

        print("cribbage: {} is choosing discards".format(self))
        deck = Deck().draw(52)
        potential_cards = [n for n in deck if n not in self.hand]
        bar = tqdm(total=226994)
        discards = []
        mean_scores = []
        for discard in combinations(self.hand, 2):  # 6 choose 2 ==
15
            inner_scores = []
            for pot in combinations(potential_cards, 3):  # 46 choose
3 == 15,180
                inner_scores.append(score_hand([*discard, *pot[:-1]],
pot[-1]))
                bar.update(1)
            inner_scores = np.array(inner_scores)
            discards.append(discard)
            mean_scores.append(inner_scores.mean())
```

```python
        return list(discards[np.argmin(mean_scores)])
```

For the pegging problem, the greedy agent scores whenever possible. Which means it checks the current hand and the cards on the table against possible combinations in cribbage to amass the highest number of points. Otherwise, it simply plays randomly.

```python
def ask_for_play(self, previous_plays):
        """
        Calculate points for each possible play in your hand
        and choose the one that maximizes the points
        """

        scores = []
        plays = []
        for card in self.hand:
            plays.append(card)
            scores.append(score_count(previous_plays + [card]))
        max_index = np.argmax(scores)

        return plays[max_index]
```

To check the possible combinations of hand.

```python
def score_count(plays):
    """Score a play vector"""

    score = 0
    if not plays or len(plays) < 2:
        return score

    count = sum(list(map(lambda x: x.value,plays)))

    if count >= 32:
       return -1
    if count == 15 or count == 31:
        score += 2
    if plays[-1].rank == plays[-2].rank:
        score += 2
    if len(plays) > 2 and plays[-2].rank == plays[-3].rank:
        score += 4
    return score
```

Heuristic Based Agent/Expert System:

This agent will operate as an “expert system” - essentially, it will follow a predefined set of “rules” or “strategies” for discarding and pegging derived from human experience. For instance, several strategies are described here. For discarding, the agent might first check if it has a double run, keeping it if so and discarding the other two cards. Failing that, the agent will take different strategies depending on whose crib it is. For instance, if the agent has the crib, it might first attempt to put a card valued 10 and a 5 in the crib, as this would add two points to the crib. Failing that, it might attempt to put a pair in the crib. Failing that, it might try to put 7s, 8s, and 5s in the crib, as all these cards are likely to lead to good cribs. If it is not the agent’s crib, the agent might instead attempt to put cards that are far apart in value into the crib, as these are less likely to result in scoring opportunities for the opponent.

```
def ask_for_discards(self, dealer=0):
    """
    For each possible discard, score and select
    highest scoring move. Note: this will give opponents
    excellent cribs, needs a flag for minimizing
    """

    #print("cribbage: {} is choosing discards".format(self))
    deck = Deck().draw(52)
    discards = []
```

```
        scores = []
        for discard in combinations(self.hand, 2):  # 6 choose 2 ==
15
            discards.append(discard)
            scores.append(0)
        discards, scores = self.expertStrategyHeuristic(discards,
scores, dealer)
        for discardIdx in range(len(discards)):
            scores[discardIdx] = score_hand_heuristic([x for x in
self.hand if x not in discards[discardIdx]])
        if(len(scores) == 0):
            for discard in combinations(self.hand, 2):  # 6 choose 2
== 15
                discards.append(discard)
                scores.append(0)
            for discardIdx in range(len(discards)):
                scores[discardIdx] = score_hand_heuristic([x for x in
self.hand if x not in discards[discardIdx]])
        return list(discards[np.argmin(scores)])
```
Ask for Discards Heuristic

```
  def expertStrategyHeuristic(self, discards, scores, dealer):
        new_discards = discards.copy()
        new_scores = scores.copy()
        if not dealer:
            for discard in discards:
                if(sum(list(map(lambda d: d.value, discard))) == 5):
                    remIdx = new_discards.index(discard)
                    new_discards.pop(remIdx)
                    new_scores.pop(remIdx)
                elif("J" in list(map(lambda d: d.rank_str, discard))
or "Q" in list(map(lambda d: d.rank_str, discard)) or "3" in
list(map(lambda d: d.rank_str, discard)) or "4" in list(map(lambda d:
d.rank_str, discard)) or "7" in list(map(lambda d: d.rank_str,
discard)) or "8" in list(map(lambda d: d.rank_str, discard))):
                    remIdx = new_discards.index(discard)
                    new_discards.pop(remIdx)
```

```
          new_scores.pop(remIdx)
        elif (abs(discard[0].value - discard[1].value) == 2):
          remIdx = new_discards.index(discard)
          new_discards.pop(remIdx)
          new_scores.pop(remIdx)
        elif (discard[0].suit == discard[1].suit):
          remIdx = new_discards.index(discard)
          new_discards.pop(remIdx)
          new_scores.pop(remIdx)
    else:
      for discard in discards:
        if(not sum(list(map(lambda d: d.value, discard))) ==
5):
          remIdx = new_discards.index(discard)
          new_discards.pop(remIdx)
          new_scores.pop(remIdx)
          continue
        if ("J" in list(map(lambda d: d.rank_str, discard))
or "Q" in list(map(lambda d: d.rank_str, discard)) or "3" in
list(map(lambda d: d.rank_str, discard)) or "4" in list(map(lambda d:
d.rank_str, discard)) or "7" in list(map(lambda d: d.rank_str,
discard)) or "8" in list(map(lambda d: d.rank_str, discard))):
          remIdx = new_discards.index(discard)
          new_discards.pop(remIdx)
          new_scores.pop(remIdx)
          continue
        if (abs(discard[0].value - discard[1].value) == 2):
          remIdx = new_discards.index(discard)
          new_discards.pop(remIdx)
          new_scores.pop(remIdx)
          continue
        if (discard[0].suit == discard[1].suit):
          remIdx = new_discards.index(discard)
          new_discards.pop(remIdx)
          new_scores.pop(remIdx)
          continue
    return new_discards, new_scores
```
Strategy helper function

When it comes to pegging, heuristic uses the same strategy as greedy to get the best score possible. This is because it already pruned out the cards that give the opponent a higher score and so they pick the “safest” option which maximizes their rewards from the remaining cards that they can play.

```python
def ask_for_play(self, previous_plays,turn = 0, count = 0):
      """
      Calculate points for each possible play in your hand
      and choose the one that maximizes the points
      """

      scores = []
      plays = []
      for card in self.hand:
          plays.append(card)
          scores.append(score_count(previous_plays + [card]))
      max_index = np.argmax(scores)

      return plays[max_index]
```

Pegging

Programming this agent amounts to simply defining a bunch of conditional statements describing human-crafted strategies in a hierarchical manner. In a sense, this is akin to hand-crafting a non-probabilistic decision tree for both discards and pegging.

Minimax:

For discards, assess all possible discards that the agent can make plus all the discards the opponent can make. If it is the agent’s crib, maximize hand score + minimum crib score. If it is opponent’s crib, maximize hand score - max scrib score. There are 6 choose 2 = 15 possible discards the agent could make and 46 choose 3 = 15,180 discards the opponent could make plus the turn card, so this amounts to enumerating 226994 total possible cribs and choosing the one that results in the maximum hand value for our agent. Essentially, this method is using the same logic as discards of greedy agent and the code is very similar.

```
    def ask_for_discards(self, my_crib=True):
        """
        For each possible discard, score and select
        highest scoring move.
        """

        print("cribbage: {} is choosing discards".format(self))
        deck = Deck().draw(52)
        potential_cards = [n for n in deck if n not in self.hand]
        bar = tqdm(total=226994)
        discards = []
        mean_scores = []
        for discard in combinations(self.hand, 2):  # 6 choose 2 ==
15
            inner_scores = []
            for pot in combinations(potential_cards, 3):  # 46 choose
3 == 15,180
                inner_scores.append(score_hand([*discard, *pot[:-1]],
pot[-1]))
                bar.update(1)
                inner_scores = np.array(inner_scores)
                discards.append(discard)
             mean_scores.append(inner_scores.mean())

        # return either the best (if my crib) or the worst (if not)
        if my_crib:
            selected = np.argmax(mean_scores)
        else:
            selected = np.argmin(mean_scores)

        return list(discards[selected])
```

For pegging, let the total score for a hand be the agent's score - opponent’s score. The agent, MAX, attempts to maximize this score, while MIN, the opponent, attempts to minimize the score. For the nth play, there are 46-n possible cards the opponent could play: for instance, if our agent leads, the opponent could play any of the 46 cards that the agent hasn't seen to follow up. Our agent has 3 possible responses to each of these, and the opponent has 45 responses to our agent's second card. The resulting minimax tree has 93,990,560 possible orders of play. The minimax agent performs minimax on this tree to decide on an optimal pegging strategy, then proceeds to play. As the minimax tree becomes exponentially larger as the depth increases, we designed it to be of depth 3, that is, we are predicting two moves of the opponent ahead.

```
    def ask_for_play(self, hand, sequence, current_sum):
        tree = minimaxTree(hand, sequence, current_sum, 3)
        return tree.recommendCard(0)
```

We start by constructing a minimax tree with current hands and sequence.

```
    def __init__(self, hand, sequence, current_sum, tree_depth):
        self.root = self.node(None, 0)

        # Consider all cards in hand that are legal moves
        legal_moves = []
        for card in hand:
            # print(card)
            if card.value + current_sum <= 31:
                legal_moves.append(card)
                newNode = self.node(card, self.scorePegging(hand,
sequence, current_sum, card))
                newNode.sumFromPlay = current_sum + card.value
                self.root.addChild(newNode)

        # print("legal moves: ", legal_moves)
        # print("Root's children: ", root.children)


        # Get deck of cards
        whole_deck = []
        deck = Deck()
        for i in deck.cards:
            whole_deck.append(i)
        # Remove known cards from deck
        for i in hand:
            whole_deck.remove(i)
        for i in sequence:
            whole_deck.remove(i)

        # For each level 2 node, put in prob nodes with legal moves
        for cnode in self.root.children:
            # Get rid of illegal or used cards
            card_deck = deepcopy(whole_deck)
            if cnode.getCard() in card_deck:
                card_deck.remove(cnode.getCard())
            for card in card_deck:
                if cnode.getSumFromPlay() + card.value > 31 and card
in card_deck:
                    card_deck.remove(card)

            # Make new nodes in children of current node using prob
nodes
            for card in card_deck:
                newChanceNode = self.chanceNode(card,
1/len(card_deck))
                cnode.addChild(newChanceNode)

        # Go through all prob nodes and add regular nodes for
opponent
        for n in self.root.getChildren(): #cnode, dealer node
            expected_utility = []
            for m in n.getChildren(): #chanceNodes
                testHand = deepcopy(hand)
                testHand.remove(n.getCard())
                testSequence = deepcopy(sequence)
                testSequence.append(n.getCard())
                utility = (self.scorePegging(testHand, testSequence,
current_sum + n.getSumFromPlay(), m.getCard()))
                newNode = self.node(m, utility)
                m.addChild(newNode)
                expected_utility.append(utility)
            n.utility = np.mean(expected_utility)
```

The previous code snippet initializes the minimax tree. After the construction of the tree, the program calls 'Recommend Card' to find the node (move) with minimum gain since the last layer is opponent's layer.

```
   # Searches the expectimax tree and recommends the card to be
played
    def recommendCard(self, risk):
        if len(self.root.getChildren()) == 0:
            return None
        lowestNode = self.root.getChildren()[0]

        lowestNodeScore = float('-inf')
        for cnode in self.root.getChildren():

            if (cnode.getUtility() ) > lowestNodeScore: #+ currMin
                lowestNode = cnode
                lowestNodeScore = cnode.getUtility()

        return lowestNode.getCard()
```

Theoretically, enumerating this tree amounts to "solving" the game - we could include the discard step at the root of this tree (though that would increase the tree size by a factor of 15), and then the agent would simply play its way through the tree, always making the play that lead to the best result. However, this approach is likely to be slow and take up huge amounts of memory as it must compute and store the entire minimax tree for every single hand. While this method of play may be optimal, it will still be interesting to see if our other approaches can approximate its performance while running much more quickly and efficiently.

Reinforcement learning with a Deep Neural Network:

We trained a deep neural network to play cribbage. In the interest of time, we opted to implement a simple multilayer perceptron with one hidden layer. Rather than training on the entire game of cribbage, we opted to first attempt to solve the simpler task of optimal discards and optimal pegging. This essentially amounts to having the network learn to approximate the behavior of our Greedy Agent, but with the advantage of much faster computation time.

To do this, we trained two networks. The first was trained on a "discard game" in which two agents discard cards to the crib, both trying to maximize the value of their hand (if not the dealer) or the value of their hand + the value of the crib (if dealing). We then trained a second network to play the "pegging game" in which each agent is dealt 6 cards, discards two, and then attempts to play their cards in such an order so as to maximize reward with each play.

The network architecture is the same for both of these tasks. The input layer is represented as a one-hot encoded single-dimensional 209 element tensor in which the first value indicates whether or not the player is the dealer, 52 inputs represent the cards that are currently in the agents hand, 52 inputs represent "known" cards, such as cards that have already been played by the agent or its opponent, and the final 104 inputs represent the "state-of-play". The first 13 of these represent the rank of the first card played to the table, the second thirteen the rank of the second, and so on. Agents can get the state given their hand, the previous plays observed, and whether or not they are the dealer using the get_state() method:

```
def get_state(self, in_hand, previous_plays, dealer):
    '''
    Get the state of play as a one-hot encoded np.array.
    One input should represent whether or not the player is the
dealer.
    52 inputs represent 1-hot vector of whether or not card is in
hand.
    52 inputs represent "known" cards - i.e. the cards in the crib
    104 inputs represent the state of play - the first 13 represent
the rank of the first card
    played, the second 13 represent the rank of the second, and so
on.
    '''
    state = np.zeros(209)
    if dealer == 1:
        state[0] = 1

    # Set state of cards in hand:
    for card in in_hand:
        idx = self._get_card_index(card)
        idx = idx + 1  # Add one as we've already set the dealer
state
        state[idx] = 1

    # Set state of "known cards" that were discarded to the crib:
    for card in self.discarded:
        idx = self._get_card_index(card)
        idx = idx + 52 + 1
        state[idx] = 1

    # Set the state of play:
    for i, card in enumerate(previous_plays):
        card_idx = card.rank
        idx = card_idx + (i * 13) + 52 + 52 + 1
        state[idx] = 1

    return state
```

These inputs are then passed through a densely connected 60 unit Linear layer with ReLU activation. Outputs from this layer are then passed to a 52 element output layer, where each element in the output layer represents one possible card to play. The agent then chooses the output with the highest value that is in fact in its hand.

The fully implemented agent loads both of the networks we trained and gets discards by passing its state through the discard network and pegging actions by passing the state through the pegging network. This requires some modifications to the play code and discard code as shown below:

```
def __init__(self, name=None):
    super().__init__(name)
    self.playable_cards = []
    self.discarded = []  # Will represent the discards that the 
player *knows* it discarded to crib
    self.discard_model = LinearQNet(209, 60, 52)
    self.pegging_model = LinearQNet(209, 60, 52)
    discard_path = pkg_resources.resource_filename('cribbage', 
'simple_model.pth')
    peg_path = pkg_resources.resource_filename('cribbage', 
'simple_pegging_model.pth')
    self.discard_model.load_state_dict(torch.load(discard_path))
    self.pegging_model.load_state_dict(torch.load(peg_path))

def play(self, count, previous_plays, turn=0):
    '''
    Overrides parent play method to pass the current count along to 
self.ask_for_play()
    '''
    if not self.hand:
        print('>>> I have no cards', self)
        return "No cards!"
    elif all(count + card.value > 31 for card in self.hand):
        print(">>>", self, self.hand, "I have to say 'go' on that 
one")
```

```
        return "Go!"

    # Only ask for a play if we have cards we can actually play
    card = self.ask_for_play(count, previous_plays, turn)
    self.update_after_play(card)
    return card
```

The code asking for a play is fairly simple, it simply gets the current state, passes it through the trained network, and picks the card with the highest network output that is also in its hand and doesn't result in an illegal play:

```
def ask_for_play(self, count, previous_plays, dealer=0):
    # Limit ourselves to cards we can actually play:
    self.playable_cards = [c for c in self.hand if c.value + count <=
31]
    state = torch.Tensor(self.get_state(self.hand, previous_plays,
dealer))

    output = self.pegging_model(state)

    vals, indices = output.sort(descending=True)

    for idx in indices:
        card = Card(idx)
        if card in self.hand and card in self.playable_cards:
            return card
```

The discard code is quite similar, it simply passes state through the discard network and doesn't have to worry about invalid discards (as you can discard anything from your hand to the crib):

```
def _discard(self, dealer): ''' Private helper method for performing
```

```
a single discard '''

    # Get state appropriate for NN
    state = torch.Tensor(self.get_state(self.hand, [], dealer))

    # Pass input through discard model:
    output = self.discard_model(state)
    vals, indices = output.sort(descending=True)
    for idx in indices:
        card = Card(idx)

        if card in self.hand and card not in self.discarded:
            self.discarded.append(card)
            return card

def ask_for_discards(self, dealer=0):
    self.discarded = []  # Reset the discarded cards
    card1 = self._discard(dealer)
    card2 = self._discard(dealer)
    return [card1, card2]
```

The models loaded for the RLAgent are trained in the file train_models.py, which we opt not to include in this report as it is quite lengthy. The training code essentially implements a small environment simulating only the discard portion of the game to train the discard network and a second environment simulating only the pegging portion of the game to train the pegging network.

Performance Assessment

To access the performance of our agents we use the following metrics:

1. The count of the number of wins subtracted by the number of losses between two of the agents.

2. The time taken by each algorithm on an average between each move.

Performance Observation:

For part 1 we run 1000 games of cribbage between two of the agents to determine the strongest player. To establish a baseline of an effective agent we start making the random agent compete with all other agents to get an idea of their intelligence as shown in the table below.

Player 1	Player 2	Winner	Winning percentage
Random agent	Greedy agent	Greedy agent	98%
Random agent	Heuristic agent	Random agent	60%
Random agent	RL agent	RL agent	86%
Random agent	Minmax agent	MinMax agent	56%

Now for the next step we run the winning agents against each other to see which approach is the best one and find the best cribbage player. As shown in the table below.

Player 1	Player 2	Winner	Winning percentage
Greedy agent	MinMax agent	Greedy agent	97%
MinMax agent	RL agent	RL agent	94%
RL agent	Greedy agent	Greedy agent	61%
Heuristic agent	Greedy agent	Greedy agent	99%
Heuristic agent	Minmax agent	Minmax agent	76%
Heuristic agent	RL agent	RL agent	90%

From the above data the greedy agent is observed to have the highest win rate, defeating every one of the other agents. The RL agent appears to be second best, beating both the Heuristic based agent and the minimax based agent. The heuristic agent performs the worst of the agents we implemented. This speaks to the difficulty of attempting to hard-code reflexive, rule-based agents for complex games.

For part 2, to count how fast each implementation is, we modified the code and ran 10 trials to determine how fast each agent chooses a discard on average and how long it takes each player to peg on average. Here are the results:

Player 1	Player 2	Average Discard Time	Average Peg Time	Number of Trials
Random Agent	Greedy Agent	8.059077188 seconds	6.469E-6 seconds	10
Random Agent	Heuristic Agent	0.000224345 seconds	5.089E-6 seconds	10
Random Agent	RL Agent	0.001530908 seconds	0.0010553 second	10
Random Agent	Minimax Agent	0.000228276 seconds	0.000072 seconds	10

Conclusion

In our trials the GreedyAgent performs quite well, beating all the other agents and winning 98% of its trials against a random agent. This is as we expected, as our Greedy Agent plays as close to the theoretical best player as possible. In terms of timing things are a different story: the greedy agent turns out to be the slowest agent both when pegging and discarding. However, it should be noted that it still loses 2% of its games. This speaks to how difficult a game cribbage is - there is so much random noise that even when you play perfectly you are more or less guaranteed to lose at least 2% of your games simply due to bad luck.

As RLAgent is essentially a fast approximator of the Greedy behavior, it loses more frequently than the Greedy agent due to random chance when playing against the random agent, winning only 86% of those games. This indicates that there may still be room for improvement in the RL network model as it is underfitting the action-value function implicitly defined in the greedy agent's behavior. However, it is much faster than Greedy, and as such might be better to use as an AI for an online game where we want to limit computational expenditure as much as possible so as to support more players.

A limitation for the RL agent is surprisingly the amount of computational time required to simply play simulated cribbage games. In tests, we found that playing the simulated games took twice as long as the actual process of training the network. Improving the cribbage implementation run time would allow us to train the network much faster, and thus presumably improve the performance of the RL agent.

Reflection

When implementing this project we were surprised by many things. First of all, we were surprised to find that with modern computational power it's actually feasible to build an agent that performs near optimally. We had assumed the state-space of possible hands and cribs would be too large to implement our greedy agent in a fully enumerative manner, but as it turned out the agent can even be run on a laptop, though it is rather slow. We were also surprised by the amount of stochasticity in the game of cribbage - we expected all of our models to beat the random player easily, but as it turns out even the best models (such as Greedy and RL) failed to beat a random player due to random chance on several occasions. Finally, we were also highly surprised by how difficult it was to encode common cribbage strategies into an expert system. While our Heuristic agent was able to perform better than chance, it did not perform particularly well compared to the greedy and RL agents. Given the relative ease of programming these agents, this strongly suggests that rule-based expert-systems programming is simply an outmoded method for solving games.

While we show that our RL and Greedy agents perform well, whether or not they are ideal for use in an actual cribbage application depends on your use case. If the purpose is to define an AI that is *fun* to play against, implementing a nearly unbeatable AI such as Greedy is probably not the best course of action. AI for this use case should be tuned to act in ways that people enjoy playing against, presumably sort of like a human player. It is possible this could be achieved with a better heuristic strategy - another possibility would be to train an agent in a supervised manner using human play data to build functions for choosing discards/plays.

It should also be noted that the margins between a “good” and “bad” player in the game of cribbage are quite slim. The best online cribbage players tend to only win about 56% of their games (against other humans). In fact, according to that source the average difference between the number of points scored per hand between the best and worst human cribbage players is less than 0.5. Unfortunately we did not have an easy way to test our agents against human players enough times to get statistics on how well they do against humans, but this could be an interesting project to implement as a web-app.

Overall, this project demonstrates the efficacy (and failures) of enumerative/Greedy, RL/neural network, traditional minimax, and reflex/rule-based approaches for solving a complex, highly stochastic game such as cribbage. These algorithms span the scope of modern AI history, from early rule-based approaches introduced in the ‘50s to the neural network approaches of the last ten years.