B351 AI Project: Texas Hold 'em

Adam Hurm[1] and Ari Korin[1]

[1] *Indiana University, Bloomington, IN, USA*

May 5, 2017

This paper explores the design and implementation of a Texas Hold'em artificial intelligence agent. The agent capable of placing bets, handling the procedure of the Texas Hold'em game, recognizing valuable hand combinations, and acting on the information presented to it. The implementation explained in this paper may be used as a framework for future Texas Hold'em agents.

1 Introduction

Games are a series of states. In each state of the game, there are actions that can be taken in order to move into another state. These actions often take the form of turns, physical movement of pieces, the picking of cards, or other factors that change the current state of affairs. In a given state, players will want to choose the optimal state to move to such as the state in which a player wins chips or an opponent is forced into a difficult position.

Games are fascinating from an artificial intelligence perspective because they are difficult to completely "solve". There are many different types of games. Some games involve unpredictability and random events, these games are known as stochastic games (Norvig and Russell, 2009). Others games are deterministic, meaning all states of the game have been reached through the actions the player has made without external influence. Many games restrict the information that each player has on the actions of opponent players. As such, in some games, agents are required to make decisions based on partial information or assumptions about how an opponent would act in a given state. Texas Hold'em is an interesting to explore because of its deterministic nature and the fact that players are not able to see other players' hands; Texas Hold'em is only partially observable.

2 Playing Texas Hold'em as an AI Problem

Our initial plan for the agent relied on calculating probabilities and basing our actions on those calculations. We would have a matrix of probabilities of achieving a good hand given our own cards and the cards on the table. As we progressed, we found that this method made it difficult to calculate multiple hands without the array blowing out of proportion, having wasted space, or being generally difficult to access components within the structure. We refreshed our approach by instead calculating what we call a PotentialHand and keeping an array of them. Each turn the cards on the table and in hand would be analyzed, and any set of cards within those that made significant progress toward a completed hand would create a PotentialHand. The PotentialHand created would contain the percentage of completion of the hand, the score of the hand as it stands, which cards the agent has that partially complete the hand, and which cards it would need to complete the hand. More simply put: completion, score, have, need. Then when our agent has to return our best hand, it can easily check the array for a PotentialHand with 100 percent completion and the highest score and return the cards. The PotentialHands structure also provides helpful turn-by-turn information for checking how many partially completed hands are over a certain completion and score threshold. That feedback can be used to play more aggressively or passively depending on how good the situation appears. The need array can be used to determine which cards are needed to complete the hand and check them against the game history to see which of the cards are potentially still available. For example, if the agent is one card away from a royal flush but it finds that the card needed to complete it has already been taken out of the game, then the hand is no longer possible and it should look at other options. Otherwise, the agent would likely go all in or play very aggressively for a hand that could never be completed. The completion percentage is also used to fold in a case of not having any completed hand when nearing the end of the game to avoid unnecessary involvement that will likely only result in chip loss.

The actual building process of PotentialHands uses lookup tables to find hands matching criteria and individual checks of matching card numbers and suits by looping through and counting occurrences. The same set of cards may end up generating multiple PotentialHands. For example, encountering a pair will create PotentialHands for a pair with full completion, a three of a kind with almost full completion, and a four of a kind with half completion. It provides useful feedback for checking for flushes and matching card numbers, so the agent can then base its actions on the number of PotentialHands and their respective scores and completion percentages. Tracking and using our completed PotentialHands results in a much more conservative agent that only makes decisions based off of concrete hands with good levels of success. The agent is much more concerned with its own success than that of its opponents, only competing if it stands a good chance of success and for the most part ignoring others that might bluff. Unless the pot is raised to too high of a level for its current hand to be likely to win, it plays through and over multiple iterations should hopefully slowly collect chips by its more conservative play style.

3 Code

Our PotentialHand concept is implemented as a class, with variables complete, sum, have, and need. The Hand class holds the cards in our hand, those on the table, and an array of PotentialHands. Its generatePotential method does the bulk of the calculations, using the deucify method and dictionary to convert hands to deuces format and check their score. In generatePotential, a loop runs and puts groups of cards of matching suits and numbers in their own array. These are used later to calculate progress or completion for pairs, three of a kinds, four of a kinds, and flushes. Then cards are checked against dictionaries for royal flushes and straight flushes and modulo'd cards are checked against a dictionary for a flush, with PotentialHands being generated accordingly. The dictionaries save computational work if we already have a full hand. In the next step, the matching number array is checked and pair, three of a kind, and four of a kind PotentialHands are generated. Following that is a check of the matching suit array for partially completed flushes and PotentialHand generation. Pairs and three of a kinds are then checked for a full house and the best pair and three of a kind are chosen if there are multiple. It then returns an array of the PotentialHands generated, giving the agent the data it needs to make decisions.

The Player class tracks all the different components of the game such as chips, table cards, cards in hand, round number, and PotentialHands, performing actions with the data. It checks the completion percentage and scores for our PotentialHands to determine the level of game involvement needed. The check procedure makes sure that we have at least a mostly completed hand before being involved past the "turn" round, otherwise it folds. The raise procedure will be constrained by the maximum raise size given to it by the action object, raising it when we have a completed hand.

4 Results

It is difficult to measure the success of our implementation because it has not been completely unit tested and simulated. Although as it stands, our Texas Hold'em agent is a conservative player. As a result of our agent's conservative player style, discussed in the prior section, our agent will most likely not make significant winnings. This is not to suggest that our agent does not function well but we have included strategies that prevent significant loss as well limiting risky moves that have the potential for big pay-offs.

The most impressive feature of the agent is its PotentialHands class. PotentialHands is a clever way to keep track of all possible winning card combinations as well as how complete each combination is. While PotentialHands is immensely useful in deciding which final hand to evaluate, it has room for increased functionality. A significant improvement in PotentialHands would be a better way evaluate the "type" a given hand in the list of potential hands. Currently we give each potential hand a score based on the deucify module. In addition to the deucify score, additional information such as the type of hand (i.e. is the hand a straight, a flush, a pair?) would be helpful in calculating which hand to use in the end. For example if the agent only needs one more card to complete a straight but already has a pair, the agent should prioritize completing the straight. Priority of hands in potential hands is based on the score given by deucify, but the type of hand should also be taken into account.

In addition to added functionality in PotentialHands, improvements in the agent's action procedures would make the agent perform better. Currently, the procedure for betting, folding, and checking is rather simplistic. The procedure relies on whether there are any potential hands that are "completed". While this may be sufficient to make the agent work, a more sophisticated suite of action procedures would improve the agent's ability to win chips. An example of an additional procedure would be the use of bluffing by the agent. During development, the use of bluffing by the agent was not fully fleshed out. If further development were to occur, a bluffing procedure would be implemented as bluffing has the potential to be a powerful procedure in Texas Hold'em.

There were very few things that went wrong during development. The main issue when developing the agent was understanding what data structures would need to be passed to the agent in order for it to act and meta-issues that involved small inconveniences that occurred as a result of two developers working on the same code-base. If we were to develop this agent from scratch again, we would focus on maintaining separate git branches as well as working on the structure of the agent before delving into the AI of the agent. Regardless of the small inconveniences that both of these caused, our agent is successful and has provided a rich learning experience.

Bibliography

Norvig, Peter and Stuart Russell (2009). “Artificial Intelligence: A Modern Approach, 3rd Ed.” In: *Pearson*.

5 Code Appendix

```
#Ari Korin and Adam Hurm
#B351 Final Project
#Spring 2017


from deuces import Evaluator
from deuces import Card

#Calculating Hands:
#for every card c in Hand H, x = c % 13
#royal_flush = [12, 11, 10, 9, 8] and same_suit
#straight_flush = straight and same_suit
#four_of_kind = [x, x, x, x, y]
#full_house = three_of_kind and pair
#flush = same_suit
#straight = [x, x+1, x+2, x+3, x+4]
#three_of_kind = [x, x, x, y, z]
#pair = [x, x, y, z, a]
#same_suit = (c / 13) == y
#high_card = max(H)

hands = {'royal_flush' : [[51, 50, 49, 48, 47], [39, 38, 37, 36, 35, 34], [25,
    24, 23, 22, 21], [12, 11, 10, 9, 8]], \
          'straight_flush' : [[0, 1, 2, 3, 4], [1, 2, 3, 4, 5], [2, 3, 4, 5, 6],
              [3, 4, 5, 6, 7], [4, 5, 6, 7, 8], [5, 6, 7, 8, 9], \
               [6, 7, 8, 9, 10], [7, 8, 9, 10, 11], [8, 9, 10, 11, 12], [13, 14,
                   15, 16, 17], [14, 15, 16, 17, 18], [15, 16, 17, 18, 19], \
               [16, 17, 18, 19, 20], [17, 18, 19, 20, 21], [18, 19, 20, 21, 22],
                   [19, 20, 21, 22, 23], [20, 21, 22, 23, 24], [21, 22, 23, 24, 25],
                   \
               [26, 27, 28, 29, 30], [27, 28, 29, 30, 31], [28, 29, 30, 31, 32],
                   [29, 30, 31, 32, 33], [30, 31, 32, 33, 34], [31, 32, 33, 34, 35],
                   \
               [32, 33, 34, 35, 36], [33, 34, 35, 36, 37], [34, 35, 36, 37, 38],
                   [39, 40, 41, 42, 43], [40, 41, 42, 43, 44], [41, 42, 43, 44, 45],
                   \
               [42, 43, 44, 45, 46], [43, 44, 45, 46, 47], [44, 45, 46, 47, 48],
                   [45, 46, 47, 48, 49], [46, 47, 48, 49, 50], [47, 48, 49, 50,
                   51]], \
          'flush' : [[0, 1, 2, 3, 4], [1, 2, 3, 4, 5], [2, 3, 4, 5, 6], [3, 4, 5,
              6, 7], [4, 5, 6, 7, 8], [5, 6, 7, 8, 9], \
               [6, 7, 8, 9, 10], [7, 8, 9, 10, 11], [8, 9, 10, 11, 12]]}
#MUST DO (X % 13) BEFORE USING FLUSH IN DICTIONARY

classes_need = {'royal_flush' : 5, 'straight_flush' : 5, "four_kind" : 4, '
    full_house' : 5, \
                'flush' : 5, 'straight' : 5, 'three_kind' : 3, 'two_pair': 4, '
                    pair': 2} #number of cards needed for a class

deucify_dict = ['Ac', '2c', '3c', '4c', '5c', '6c', '7c', '8c', '9c', '10c', 'Jc
    ', 'Qc', 'Kc', \
                'Ad', '2d', '3d', '4d', '5d', '6d', '7d', '8d', '9d', '10d', 'Jd
                    ', 'Qd', 'Kd', \
                'Ah', '2h', '3h', '4h', '5h', '6h', '7h', '8h', '9h', '10h', 'Jh
                    ', 'Qh', 'Kh', \
                'As', '2s', '3s', '4s', '5s', '6s', '7s', '8s', '9s', '10s', 'Js
```

```
                 ', 'Qs', 'Ks']


class Card(object):
    def __init__(self, number, in_hand):
        self.number = number #number of card 0-51 (int)
        #suit determined by floor(number / 13)
        #Club,Diamond,Heart,Spade
        self.in_hand = in_hand #bool

# PotentialHand are the hands that we can possibly have. This means that if we
    have cards
#  [2Hearts, 3Hearts, 4Heart, 5Hearts, 2Spades] we can potentially have a
    straight with Heart cards but we also
#   Have a pair with the 2s
class PotentialHand(object):
    def __init__(self, complete, sum, have, need):
        self.complete = complete
        self.sum = sum
        self.have = have
        self.need = need

# Hand models the cards that we can use as a "Hand". This includes the cards
    dealt directly to use as well as
# the cards on the table
class Hand(object):
    def __init__(self, cards, table, potential_hands):
        self.cards = cards #array of cards (int[])
        self.board = table.cards
        self.potential_hands = potential_hands

    def deucify(int[] cards):
        newcards = []
        for card in cards:
            newcards.append(Card.new(deucify[card]))
        return newcards

    def generatePotential(self):
        matching_suit_hands = []
        matching_number_hands = []
        moduloCards = []
        cards = [] #will replace self.cards for poker.py
        cards.append(self.cards)
        cards.append(self.board)

        for card in cards:
            #keep track of initial card suit
            sameSuitTest = card / 13
            #holders for checking matching suit and number
            matching_suit_hand = []
            matching_numbers = []
            #disregard suit for checking flush
            moduloCards.append(card % 13)
            for card2 in cards:
                if (card2 / 13) == sameSuitTest:
                    matching_suit_hand.append(card2)
                if (card2 % 13) == (card % 13):
                    matching_numbers.append(card2)
            if len(matching_suit_hand) >= 5:
```

```
            matching_suit_hands.append(matching_suit_hand)
        if len(matching_numbers) > 1:
            matching_number_hands.append(matching_numbers)

    score = evaluator.evaluate(self.cards, self.table)

    #check for royal flush
    if self.cards in hands['royal_flush']: #if cards match list from dict
        key, search for index of matching list and get list[index]
        keep = hands['royal_flush'][hands['royal_flush'].index(self.cards)]
        self.potential_hands.append(PotentialHand(100, score, keep, []))

    #check for straight flush
    if self.cards in hands['straight_flush']:
        keep = hands['straight_flush'][hands['royal_flush'].index(self.cards
            )]
        self.potential_hands.append(PotentialHand(100, score, keep, []))

    #check for flush
    if moduloCards in hands['flush']:
        keep = hands['flush'][hands['flush'].index(self.cards)]
        self.potential_hands.append(PotentialHand(100, score, keep, []))

    #check for matching number
    for numlist in matching_number_hands:
        #find need by checking which cards of the number are not in the
            current list
        need = []
        cardnum = numlist[0] % 13
        for num in [cardnum, cardnum + 13, cardnum + 26, cardnum + 39]:
            if num not in numlist:
                need.append(num)
        if len(numlist) == 4:
            potential_hands.append(PotentialHand(100, score, numlist, [])) #
                4 of a kind
        if len(numlist) == 3:
            potential_hands.append(PotentialHand(100, score, numlist, [])) #
                3 of a kind
            potential_hands.append(PotentialHand(75, score, numlist, need))
                #partial 4 of a kind
        if len(numlist) == 2:
            potential_hands.append(PotentialHand(100, score, numlist, [])) #
                pair
            potential_hands.append(PotentialHand(75, score, numlist, need))
                #partial 3 of a kind
            potential_hands.append(PotentialHand(50, score, numlist, need))
                #partial 4 of a kind

    #check for matching suit
    for numlist in matching_suit_hands:
        #find need by checking which cards of the suit are not in the
            current list
        cardsuit = numlist[0] / 13
        for num in range(0+13(cardsuit),(12+13(cardsuit))+1):
            if num not in numlist:
                need.append(num)
        if len(numlist) == 4:
            potential_hands.append(PotentialHand(80, score, numlist, need))
                #4 of matching suit
            if len(numlist) == 3:
                potential_hands.append(PotentialHand(60, score, numlist, need))
                    #3 of matching suit
            if len(numlist) == 2:
                potential_hands.append(PotentialHand(40, score, numlist, need))
                    #2 of matching suit

        pairs = []
        threes = []
        #gather complete pairs and 3ok's
        for hand in potential_hands:
            if hand.complete == 100:
                if len(hand.have) == 3:
                    threes.append(hand)
                if len(hand.have == 2):
                    pairs.append(hand)
        #grab best pair and 3ok
        if pairs or threes:
            maxpair = []
            for pair in pairs:
                if (pair[0] % 13) > (maxpair[0] % 13):
                    maxpair = pair
            maxthree = []
            for three in threes:
                if (three[0] % 13) > (maxthree[0] % 13):
                    maxthree = three
            if pairs and threes:
                potential_hands.append(PotentialHand(100, score, maxpair.extend(
                    maxthree)))
            if threes:
                potential_hands.append(PotentialHand(100, score, maxthree)
            if len(pairs) >= 2:
                potential_hands.append(PotentialHand(100, score, pairs[0].extend
                    (pairs[1])))
            if len(pairs) == 1:
                potential_hands.append(PotentialHand(100, score, maxpair))
                potential_hands.append(PotentialHand(50, score, maxpair))

# Table class models that cards in the river as well as the chips being played
class Table(object):
    def __init__(self, cards, pot):
        self.cards = cards #array of cards (int[])
        self.pot = pot #number of chips (int)

# Simple class to model our chips and maintain history of our moves
class Chips(object):
    def __init__(self, chips, history):
        self.chips = chips #number of chips (int)
        self.history = history #array of chip count for past moves (int)

    def add(self, chips):
        self.history.append(chips)
        self.chips += chips

    def remove(self, chips):
        self.history.append(chips)
        self.chips -= chips

    def getHistory(self, move):
        return self.history[move]


# Player is the main class that acts. It receives chips, a table,
    potential_hands, and an ActionObject then acts
#   in regards to the actions requested by the ActionObject
class Player(object):

    def __init__(self, chips, table, potential_hand, ActionObject):
        self.chips = chips
        self.table = table
        self.hand = potential_hand
        self.round_number = ActionObject.round_number
        self.potential_hands = self.hand.potential_hands
        self.has_completed = False
        self.has_partial_complete = False

        for potential in self.potential_hands:
            if potential.complete == 100:
                self.has_completed = True
            if potential.complete > 75:
                self.has_partial_complete = True

    # Receives an ActionObject and returns a procedure based on the request of
        the ActionObject
    def act(self, action_object):
        if action_object.action == "Check":  # We can add a condition later that
             bets if we have a hand of a given score
            return self.check_procedure()

        if ActionObject.action == "Bet":
            return self.raise_procedure(action_object.raise_size)

    def check_procedure(self):
        # If it is above the 3rd round and we have a hand that is more than 75%
            complete then check, else fold
        if self.round_number >= 3:
            if self.hand[1].complete > 75:
                return "Check"
            else:
                return "Fold"

        #If it is not greater than the third round, just check
        return "Check"

    def raise_procedure(self, raise_size):
        # Betting procedure for the first round, if we have completed stay in
            else fold
        if self.round_number <= 1:
            if self.has_completed:
                return "Raise", raise_size
            else:
                return "Fold"

        if self.has_completed or self.has_partial_complete:
            return "Raise", raise_size


# A dummy class used to simulate the data structure given to us by the
```

```
    simulation
# The ActionObject is used to model what the simulation would ask us.
class ActionObject(object):

    def __init__(self, round_number, action_needed, max_bet_size, raise_size,
        cards_on_table):
        self.round_number = round_number
        self.action = action_needed
        self.max_bet_size = max_bet_size
        self.raise_size = raise_size
        self.cards_on_table = cards_on_table
```