Texas Hold’em AI

Anna Branam[1] and Jonathon Cooke-Akaiwa[1]

[1] *University of Indiana at Bloomington, Indiana, United States of America*

May 5, 2017

The goal of this project was to create an intelligent agent that could play the game of Texas Hold‘em. Furthermore, our goal included the statement that this agent should be able to consistently beat a player with no strategy. To accomplish this goal, we employed the use of calculating possible better hands that the player could obtain. These calculations were then used in conjunction with probabilities to determine an optimal line of play.

1 Introduction

“For AI researchers, the abstract nature of games makes them an appealing subject for study. The state of a game is easy to represent, and agents are usually restricted to a small number of actions whose outcomes are defined by precise rules.” (i,i,Norvigi,i, and Russell, 2010)

Texas Holdem is one such game. The deck and cards are easily translated into data structures and the game has a specific set of rules that must be followed. In this instance, the rules are slightly different than a game of Texas Hold’em that might be played in a Casino. However, this helps to reduce the search space of an agent.

There are only five possible types of actions in Texas Holdem, significantly reducing state space on that front. However, the real work involves determining current hand status and ranking as well as possible better hands. With this aspect in mind, the game of Texas Holdem becomes vastly more complicated than before.

2 Strategy

Contrary to the way in which Texas Holdem is typically played, this AI was designed under the assumption that it can only bet once. This assumption greatly reduces the number of computations that must be performed by a player and therefore significantly reduces the search space as well. In addition to changing the number of calculations, many traditional strategies are no longer valid. Many traditional strategies take into account betting history for a player where frequency and amount in a round are critical to deciding a player’s action. With this change in rules we have had to modify our strategy accordingly.

Hands are evaluated numerically with a larger number corresponding to a higher quality hand. Hands of equal value are equivalent but may not consist of the same cards. Possessing a hand above a threshold of a straight is our baseline of hands we are comfortable betting on. Any hand of a value lower than this is bet is upon based on the chances of that hand improving as discussed.

Our player will sometimes make the non-optimal choice in a hand. This is based on our bluff factor. Based on how aggressive we want our player to play it will sometimes be told to ignore some of our logic, thus effectively telling our player that its hand is of a higher quality than it actually is. This makes for more aggressive plays that will allow us to win some hands due to presenting a stronger play than we should normally.

2.1 Betting Amounts

The maximum amount we will bet is as follows (if the maximum allowed bet is lower, that bet will be made instead):

Any hands of a lower quality will not be bet on in order to conserve our funds. These hands will will produce an action of “check”.

Table 1: *Amount of Money to Bet per Hand*

Hand	Amount to Bet
Royal Flush	All In
Straight Flush	50% of current chips
Four of a Kind	30% of current chips
Full House	20% of current chips
Flush	10% of current chips
Straight	5% of current chips

2.2 Calling VS. Folding

Calling is determined by looking at the amount needed to call relative to the current amount of money in the pot. This method is based on (*How to Calculate Pot and Hand Odds in Limit Hold 'Em Poker*). This produces a ratio in the form call amount / pot. This is then compared to the number of outs a hand possesses. An "out" is any card that will change a hand from its current type to one of a greater type. The ratio of money needed to call versus the total money in the pot must be greater than our number of outs minus 1. This allows us to make a calculated risk in our play where the winnings outweigh our odds of losing. For hands of flush or higher, we will call based on the idea above, but instead of using the number of outs, instead have a static value. If the ratio is smaller than our calculated number of outs, we will fold the hand as our odds of having a winning hand is not worth the investment. Before the flop, the number of hands that must be calculated is too large to done in an efficient amount of time. To deal with this time we have adapted the 2 card power rating as shown in (*Wizard of Odds - Texas Hold'em*). This ranks two card hands with a numerical value. While this was not the original use of these value, we use them as a baseline when evaluating if a call should be made.

2.3 Raising

Unlike a human player, we are not concerned with the speed of play, thus our raising strategy is limited. Just as before, we check the ratio of the amount needed to call a bet with the pot total. If this ratio is below a preset amount, we will increase the bet by a set percentage of our current chips. This percentage is determined based on the value of our current hand. This allows us to raise the amount of money in the pot by an amount we are comfortable risking, and hopefully either draw in more money from our opponents, or scare off potentially stronger opponents.

3 Code

3.1 Data Structures

The code in this project was designed to be as flexible as possible, thus an object oriented approach was taken. All of the players created are derived from a parent class of Player that allows for common actions to be easily updated in one location. Much of the game has been abstracted to classes such as each card being an individual class, while a deck of cards is another class that encapsulates these cards. This allowed for more flexibility in displaying and manipulating each object to facilitate better development. Many of the other structures in the code are dictionaries and lists to allow for easy lookup of information to keep the player from being slowed down.

3.2 Players

In order to fully test our AI, it was necessary to develop multiple classes of Texas Holdem players to run in our simulations. The Player class provided a base for every other player class that was created for the game simulation.

The RandomPlayerClass was created as for the purpose of testing the AI's ability to play poker. This class randomly chooses an action based on the current legal actions of the game and, if the chosen action is 'bet' or 'raise', this player will randomly choose an amount to bet based on the maximum bet allowed.

The HumanPlayer Class was also created to allow better interaction with and more testing for our AI. This class allows a human player to input actions in a textual version of texas holdem.

The NoBluffPlayerClass was the first class developed for our agent. Again, as this is a player class, it returns one of the legal actions for current game state. Unlike the other classes, however, this class is designed to generate decisions based on a slightly more complex strategy and requiring more use of the provided information than that of the RandomPlayerClass.

The BluffPlayer contains the refined version of our agent. This class is extremely similar to the NoBluffPlayer class with one distinct difference. The former class possesses the ability to bluff on a given hand of cards.

3.3 Analysis

The code we constructed for this project also involved multiple analysis programs. These programs were developed to analyse a given hand of cards and predict the possibility of receiving a better hand.

The Hand class was created to analyse and categorize the various types of hands that a player could receive. The Hand class takes an array of integers representing cards and will perform various calculations on those cards. A global numeric classification system is used to differentiate the value of a given hand. For example, a hand classified as a straight will return a higher numeric value than that of one classified as high and lower than that of a full house hand. Furthermore, the numeric system extends to the cards within a specific hand. Therefore, a hand with three twos will have a lower numeric value than a hand with three nines. The code developed for this project also includes probability functionality. These calculations are a major part of the agent's strategy and involve calculation of the possible better hands that could arise from any given hand of cards. This particular set of functions is contained in the $EV_script.py$ file. This file will take in a set of cards and from there calculate the current best hand a player possesses. From here it will determine which unique cards will take the current best hand and improve it to a hand of a higher value. This calculation is returned as a percentage of cards that improve the hand, over the number of cards left in the deck.

3.4 Simulation

The Game class is used to test the AI. This class, taking a list of players and a starting amount of chips, simulates the game of poker. It includes legality checks on each player's actions as well as a textual output for the game.

The Deck is used within the Game class to simulate a poker deck of fifty-two cards. It's main methods are dealing and shuffling.

The Card class simulates cards for the deck class and allows the game's cards to be printed out more clearly.

3.5 Running

Finally, we developed two separate files to run our code. The first was created for testing purposes: $test\_game.py$. This file was used to run multiple quick simulations on the agent without the textual display that a full simulation of the game would generate. The second file was created for the purpose of interaction with the agent: $play\_game.py$. Specifically, this file allows a user to select a number of players and player types as well as a chip amount and run a full simulation of the game, complete with textual display of the game.

4 Results

Ultimately, we succeeded in attaining our goal of winning at least 60

Throughout the process of creating this AI, we encountered issues with the number of poker chips in the game becoming inconsistent as the game progressed. Eventually, we managed to deal with this problem, however, the occasional rogue chip seems to appear in the game from time to time (about 1/50 full games). While this particular problem is more involved with the simulation of the game rather than the actual agent itself, it is still an important aspect of the game that would be worth pursuing further.

Initially, we incorporated a pre-flop fold factor in our agent. This, however, proved to be more of a detriment than an aid to the agent's strategy. Instead of filtering out the worst hands that we had no chance of winning, we instead chose to play these hands, just at a more conservative rate. Incorporating a more refined version of this pre-flop fold into the agent would be an interesting avenue to explore. These hands would also benefit from a more effective bluffing strategy.

The agent will still sometimes attempt to submit an action that is not legal at the current game state. This causes to the game to force a fold on the player. This is another area wherein it would be helpful to continue refining the agent.

While raising and calling are implemented, with more time and testing it would have been possible to narrow down an optimal threshold for each hand, thus optimizing how much money will be invested in a hand. To to develop an effective set of threshold values it would benefit our player to play against other AI rather than just itself or a random player. The final goal we had in this respect is related to evaluating what type of hand our opponent has to implement the strategy as described in (*How Thinking in Ranges Drastically Improves Your Poker Game*). To effectively use this strategy we would need to train against a wide variety of opponents to determine what characteristics a player shows with a certain type of hand. Unfortunately due to the custom set of rules for this version of the game, any opponent would also have to be modified. This severely limits the ability to evaluate our player against known and established players.

Bibliography

How Thinking in Ranges Drastically Improves Your Poker Game. URL: `http://www.pokerlistings.com/why-and-how-thinking-in-ranges-drastically-improves-your-poker-game`.

How to Calculate Pot and Hand Odds in Limit Hold 'Em Poker. URL: `http://www.wikihow.com/Calculate-Pot-and-Hand-Odds-in-Limit-Hold-%27Em-Poker`.

i,i,Norvig, Peteri,i, and Stuart J. Russell (2010). *Artificial Intelligence A Modern Approach. Third ed.* P. 161.

Wizard of Odds - Texas Hold'em. URL: `https://wizardofodds.com/games/texas-hold-em/`.

A Player Base Class

```
#from game import *

# Player default class
import re
import random

class Player(object):
  def __init__(self):
    self.name = "default"
    self.hand = [-1, -1]
    self.betFlag = 0
    self.round = 0
    self.current_bet = 0
    self.chips = 0
    self.bluff_factor = 0

  def handToValue(self):
    self.tempHand = []
    for x in self.hand:
      self.tempHand.append(x.toValue())
    return self.tempHand

  def setChips(self,chips):
    self.chips = chips

  def addChips(self, amount):
    self.chips += amount

  def subChips(self, amount):
    self.chips -= amount

  def setActiveGame(self, game):
    self.game = game

  def setName(self, name):
    self.name = name

  def getName(self):
    return self.name

  def resetFlag(self):
    self.betFlag = 0
    self.round = 0
    self.current_bet = 0
    bluff_factor = random.randrange(10)

  def setHand(self, card1, card2):
    self.hand = [card1, card2]

  def getHand(self):
    return self.hand

  def action(self, maxbet):
    # bet
    # raise
    # call
    # check
    # fold
    return(["fold", 0])

  def collectAnti(self, amount):
    self.chips -= amount
    return amount

  def legal_moves(self, history, maxbet):
      moves = set()
      if self.haveBet(history): moves.add("call")#bet has occured
      else: moves= set(["check"])#bet has not occured
      if (not self.betFlag) and maxbet>0:
          if "check" in moves: moves.add("bet")
          else: moves.add("raise")
      if self.chips: moves.add("fold")#check for all in, don't fold if no chips are left
      return moves

  def haveBet(self, history):#TODO: check for hand end, check for round end
      for i in range(-1, -1*len(history), -1):

        if ("Round" in history[i]): return False
        #parsing issue
        if ("bet" in history[i]) or ("raise" in history[i]) or ("call" in history[i]):return True
      return False

  def chipAmount(self):
    return self.chips

  def callAmount(self, history):
    return self.game.callAmount(self)

    """
                sum = 0
                b = re.compile("bet*")
                r = re.compile("raise*")
                s = re.compile("\d+")
                for i in range(-1,-1*len(history), -1):
                        if("Round" in history[i]):
                                return sum
                        if b.search(history[i][0]):
                                result = s.search(history[i][0])
                                start = result.start()
                                end = result.end()
                                sum += int(history[i][0][start:end])
                        if r.search(history[i][0]):
                                result = s.search(history[i][0])
                                start = result.start()
                                end = result.end(sum)
                                sum += int(history[i][0][start:end])
                return False
"""
```

B BluffPlayer Class

```
# no bluff player

"""
Last modified: 5/2/17
Last modified by: Anna

Things to fix:
searching history
check bet ammount for legality
"""


from player import *
from hand_classification.texas_holdem_hand import *
import random
import re
from EV_script import *

MIN_BLUFF = 5

class BluffPlayer(Player):
        def __init__(self, history = [], chips=0, af=0, verbose = False):
                Player.__init__(self)
                self.verbose = verbose
                self.agression_factor = af#work with this later
                self.chips = chips#add initial chip set option
                self.current_bet = 0#keep track of how much agent has in the pot
                self.round= 0
                self.history = history
```

```
            self.bestHand =[]

            #keep track of private vs public cards?

    # TODO
    # -- take into account aggression factor (reduce threshold by ~1)


    #fundction that determines if agent should fold initially


    def getBestHand(self):
        return self.bestHand

    def setHistory(self, history):
            self.history = history

    def checkFoldPreFlop(self): #false - don't fold | true - fold
            self.checkHand = Hand(self.handToValue())
            self.suitFlag = self.checkHand.flush()
            self.suitedThreshold = {12: 0, 11: 0, 10: 5, 9: 5, 8: 5, 7: 5, 6: 4, 5: 13, 4:13,
3:13, 2:13, 1:13, 0:13}
            self.unsuitedThreshold = {12:7, 11:7, 10:7, 9:6, 8:6, 7:6, 6:4, 5:13, 4:13, 3:13,
2:13, 1:13, 0:13 }
            # pre flop check
            # check for pairs
            if len(self.checkHand.pairs()[0]) == 0:
                    self.temp = self.checkHand.sortByValue(self.handToValue())
                    if self.temp[1]%13 > 6:
                            if self.suitFlag:#check if suit matches
                                    if self.temp[0]%13 > self.suitedThreshold[self.temp[0]%13]:#
matching suit
                                            return False #good starting hand, don't fold
                                    else:
                                            return True #bad starting and, agent should fold
                            else:
                                    if self.temp[0]%13 > self.unsuitedThreshold[self.temp
[0]%13]:#non-matching suiit
                                            return False #good starting hand, don't fold
                                    else:
                                            return True #bad starting and, agent should fold
                    else:
                            return True #bad starting and, agent should fold
            else:
                    return False #have a pair in opening hand (don't fold)


    def action(self, maxbet):
            #print(self.round)#for testing
            # pre flop check
            isCallRound = self.callRound(self.history)
            if isCallRound:
                return self.callRoundAction(self.game.callAmount(self), True, maxbet)
            self.checkHand = Hand(self.handToValue())
            self.temp = self.checkHand.sortByValue(self.handToValue())
            self.bestHand = self.checkHand.best_hand()

            # if self.round==0 and self.checkFoldPreFlop():#should probably only check this at
the begining of a hand
            #         return (["fold", 0])

            #betting strategy, go for a check unless we like our hand.

            # didn't fold before flop so need to check or call
            moves = self.legal_moves(self.history, maxbet)#get possible moves
            if self.round == 0 and self.bluff_factor<MIN_BLUFF:# don't want to bet or raise
first round
                    if "bet" in moves: moves.remove("bet")
                    if "raise" in moves: moves.remove("raise")

            #self.round +=1#number after 1 doesn't matter; just need to differentiate the pre-
flop
            #randomly choose for now
            num = random.randrange(len(moves))
            move = list(moves)[num]
            if self.verbose: print(str(self.name) + " moves: " + str(moves))

            if "raise" in moves:
                if self.bluff_factor<MIN_BLUFF:
                    return self.callRoundAction(self.callAmount(self.history), False, maxbet)
                else:
                    self.chips-=self.game.callAmount(self)
                    if self.chips>0:
                        val =  math.floor(min(maxbet, (.05 * self.chips)))
                        if val <= 0: val = 1
                        self.chips -= val
                        self.betFlag = 1
                        return(["raise", val])
                    else:return["call", val]

            if "call" in moves:
                return self.callRoundAction(self.callAmount(self.history), False, maxbet)
            self.round+=1

            if (self.betFlag == 0 and "bet" in moves):#don't need to check bet flag twice
                    field_cards = []
                    for x in self.game.field:
                            field_cards.append(x.val)
                    check_hand = Hand([self.hand[0].val, self.hand[1].val], field_cards)
                    if check_hand.better_hand_check([True,"straight_flush"], check_hand.is_hand()):
                            # we have a royal flush so let's bet higher.
                            self.chips -= maxbet
                            self.betFlag = 1
                            return ["bet", maxbet]
                    elif check_hand.better_hand_check([True,"four"], check_hand.is_hand()):
                            val =  math.floor(min(maxbet, (.5 * self.chips)))
                            if val <= 0: val = 1
                            self.chips -= val
                            self.betFlag = 1
                            return ["bet", val]
                    elif check_hand.better_hand_check([True,"full"], check_hand.is_hand()):
                            val =  math.floor(min(maxbet, (.3 * self.chips)))
                            if val <= 0: val = 1
                            self.chips -= val
                            self.betFlag = 1
                            return ["bet", val]
                    elif check_hand.better_hand_check([True,"flush"], check_hand.is_hand()):
                            val =  math.floor(min(maxbet, (.2 * self.chips)))
                            if val <= 0: val = 1
                            self.chips -= val
                            self.betFlag = 1
                            return ["bet", val]
                    elif check_hand.better_hand_check([True,"straight"], check_hand.is_hand()):
                            val =  math.floor(min(maxbet, (0.1 * self.chips)))
                            if val <= 0: val = 1
                            self.chips -= val
                            self.betFlag = 1
                            return ["bet", val]
                    elif check_hand.better_hand_check([True,"three"], check_hand.is_hand()):
                            val =  math.floor(min(maxbet, (.05 * self.chips)))
                            if val <= 0: val = 1
                            self.chips -= val
                            self.betFlag = 1
                            return ["bet", val]
                    else: return ["check", 0]

            return ["check", 0]




            #print(move)#for testing

            # #if move == "fold": self.round = 0
            # if move == "call": self.chips-=self.game.callAmount(self)
            # if move == "call" or move =="check": return [move, self.game.callAmount(self)]
            # if move == "raise" or move =="bet": #bet randomly for now
```

```
            #         if maxbet>1:bet = random.randrange(1, maxbet+1)
            #         else: bet = maxbet
            #         self.chips -= bet + self.game.callAmount(self)
            #         self.betFlag = 1
            #         return [move, bet]

            # return [move, 0]



    def callRoundAction(self, needBet, isCallRound, maxbet):
            if(needBet == 0):
                return ["call", 0]
            if (self.round == 0):
                    if needBet/self.chips >= (0.1 * self.preflop_call_percent()) or self.
bluff_factor>=MIN_BLUFF:
                            bet = self.game.callAmount(self)#current_bet is not reliable
                            #needBet - self.current_bet
                            self.chips -= bet#subtract bet from chips
                            return ["call", bet]
                    else:
                            return ["fold", 0]
            else:
                    field_cards = []
                    for x in self.game.field:
                            field_cards.append(x.val)
                    check_hand = Hand([self.hand[0].val, self.hand[1].val], field_cards)
                    if check_hand.better_hand_check([True,"straight_flush"], check_hand.is_hand
()) or self.bluff_factor>=MIN_BLUFF+4:
                            if (self.betFlag == 0 and (self.game.pot/needBet >= 0)):
                                bet = maxbet
                                self.chips -= bet
                                return ["raise", bet]
                            if (self.game.pot/needBet) >=  0:
                                    bet = self.game.callAmount(self)
                                    self.chips -= bet
                                    return ["call", bet]
                            else:
                                    return ["fold", 0]
                    elif check_hand.better_hand_check([True,"four"], check_hand.is_hand()) or
self.bluff_factor>=MIN_BLUFF+3:
                            if (self.betFlag == 0 and (self.game.pot/needBet >= .5)):
                                bet = min(maxbet, round(self.chips * .8))
                                self.chips -= bet
                                return ["raise", bet]
                            if (self.game.pot/needBet) >=  1:
                                    bet = self.game.callAmount(self)
                                    self.chips -= bet
                                    return ["call", bet]
                            else:
                                    return ["fold", 0]
                    elif check_hand.better_hand_check([True,"full"], check_hand.is_hand()) or
self.bluff_factor>=MIN_BLUFF+2:
                            if (self.betFlag == 0 and (self.game.pot/needBet >= 0)):
                                bet = min(maxbet, round(self.chips * .4))
                                self.chips -= bet
                                return ["raise", bet]
                            if (self.game.pot/needBet) >=  2:
                                    bet = self.game.callAmount(self)
                                    self.chips -= bet
                                    return ["call", bet]
                            else:
                                    return ["fold", 0]
                    elif check_hand.better_hand_check([True,"flush"], check_hand.is_hand()) or
self.bluff_factor>=MIN_BLUFF+1:
                            if (self.betFlag == 0 and (self.game.pot/needBet >= 0)):
                                bet = min(maxbet, round(self.chips * .2))
                                self.chips -= bet
                                return ["raise", bet]
                            if (self.game.pot/needBet) >=  2:
                                    bet = self.game.callAmount(self)
                                    self.chips -= bet
                                    return ["call", bet]
                            else:
```

```
                    return ["fold", 0]
            elif check_hand.better_hand_check([True,"straight"], check_hand.is_hand())
or self.bluff_factor>=MIN_BLUFF:
                if (self.game.pot/needBet) >=  3:
                    bet = self.game.callAmount(self)
                    self.chips -= bet
                    return ["call", bet]
                else:
                    return ["fold", 0]

            elif (self.game.pot/needBet >= (better_hand_outs(self.hand + self.game.field
)[0] - 1)) or self.bluff_factor>=MIN_BLUFF:
                bet = self.game.callAmount(self)#current_bet is not reliable
                #needBet - self.current_bet
                self.chips -= bet#subtract bet from chips
                return ["call", bet]
        return ["fold", 0]

    def preflop_call_percent(self):
        c1 = self.hand[0].val % 13
        c2 = self.hand[1].val % 13 #val
        suit_flag = False
        if (abs(self.hand[0].val - self.hand[1].val) <= 13):
            suit_flag = True
        pairs = {0:11, 1:11, 2:11, 3: 12, 4:13, 5:13, 6: 15, 7: 16, 8:19, 9:22, 10:27,
11:32, 12:40}
        suited = {1:[7],2:[7,8],3:[8,9,10],4:[6,8,10,11],5:[6,7,9,10,11],
        6:[6,6,8,9,11,12],7:[7,7,7,8,10,12,13],
8:[8,8,8,8,10,11,13,15],9:[8,9,9,9,9,11,13,15,18],
        10:[9,10,10,10,11,11,13,15,18,19], 11:[11,11,11,12,12,13,13,15,18,20,21],
        12:[13,14,14,15,14,14,15,16,19,20,22,24]}
        unsuited = {1:[1], 2:[2,3],3:[2,4,5], 4:[1,3,4,6], 5:[0,1,1,3,5,6],
        6:[0,1,2,4,5,7],7:[1,1,1,3,4,6,8],8:[2,2,2,2,4,6,8,10],
        9:[2,2,3,3,3,5,7,9,13], 10:[3,3,4,4,4,5,7,9,12,14],11:[4,5,5,5,6,6,7,9,13,14,16],
        12:[7,7,8,8,7,8,9,10,13,14,16,19]}

        if (c2 > c1):
            tempc = c1
            c1 = c2
            c2 = tempc

        if (c1 == c2):
            return pairs[c1]

        if (suit_flag):
            return suited[c1][c2]
        else:
            return unsuited[c1][c2]

    def callRound(self, history):
        for i in range(-1, -1*len(history), -1):
            if ("Round" in history[i]):
                break
            if ("Call Round" in history[i]):
                needBet = history[i][1]
                return needBet
        return False
```