IB Subject: Mathematics

Extended Essay

Probability and Game Theory in the Card Game Pairs

What strategies achieve Subgame Perfect Nash Equilibrium in the card game Pairs?

Word Count: 3111

Table of Contents

Introduction

Research Question:

What strategies achieve Subgame Perfect Nash Equilibrium in the card game Pairs?

Rules of the Game:

Pairs is a classic pub game. The deck is composed of only the numbers 1 through 10. The deck is also known as a triangular deck since it has 1 card with the value 1, 2 cards with the value 2, 3 cards with the value 3, and so on until 10 cards with the value 10. The goal of the game is to end up with the least amount of points. When a player reaches the target score, they lose and everyone else wins. The target score for 2-6 players is calculated by $\frac{60}{n} + 1$ where n is the number of players.

Five cards are burned in the beginning, meaning left out of the game, to make card counting more difficult. First, everyone is dealt one card. You can take a card, known as a hit, or fold. If you hit, then you receive another card. If you do not make a pair, meaning having another of the same card, with your cards, then it continues to the next player's turn. If you do make a pair, you score the value of points of the card you made a pair with and the round ends. If you fold instead, then you score the lowest card value on the field, and the round ends. When the round ends, all the players' faceup cards are reshuffled into the deck. Then, cards are burnt and dealt again. (PAIRS)

Personal Attachment:

My family and I are a fan of card games. We are competitive with each other. The use of mathematics can greatly assist one's chances of winning. Thus, I wanted to explore the cases of different players and how to minimize the chance of losing in the form of an Extended Essay.

The goal of this paper is to find the optimal strategy for a player by comparing several different strategies by analyzing where they deviate and where they perform better. I will build off the base game Pairs and analyze smaller cases.

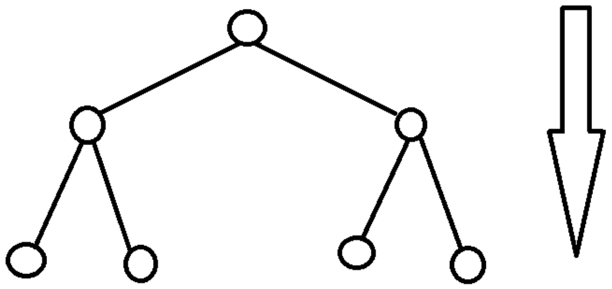
Extension of the Game Pairs to Game Theory

Game Theory Definitions:

Game theory is the analysis of the ways in which interacting choices produce outcomes with different payoffs.

Payoffs are the utility or the preference ranking of a choice. There are two kinds of payoffs, ordinal and cardinal. Ordinal is a ranking system of payoffs. Cardinal assigns a numerical value. Since the game Pairs employs a point system, cardinal payoffs will be used. An outcome is a set of payoffs.

The game Pairs can be illustrated as a tree with nodes. Game trees are directed graphs meaning that each node has an edge with a direction.



Above is an example of an unlabeled graph. The top node is considered the initial node. In the case of Pairs, that is the point where cards have been shuffled and five cards have burnt. The initial card has been dealt, as well. Each node is where a player must make a decision. Terminal nodes end the game if reached. Subgames are sets of nodes and edges that branch off from another node. An example of a subgame in the game of Pairs would be Player 1 having the cards 1, 2, 3, 4, and 5 face up and Player 2 having the cards 4, 5, 7, 8, and 9 face up and the set of decision possibilities that follow. The players' setups branched off a node somewhere in the middle of the tree. Analyzing subgames will greatly simplify the process of analyzing the game as a whole.

A strategy in mathematics is a program instructing what a player should decide at every node in the game tree when he or she can take an action.

Nash equilibrium is a set of strategies where changing one's strategy will not yield a better payoff.

Subgame Perfect Nash Equilibrium is when each subgame achieves Nash equilibrium.

(Ross, Don)

Lemma 1:

□All these principles of game theory and Nash Equilibrium can be applied to Pairs since Pairs is an acyclic game (Jost 31). Here is a proof by contradiction as to why Pairs is an acyclic game:

Let us assume that Pairs is a cyclic game. Let us define each transition to the next game state or node as a hit or a fold. Let each game state be denoted as S_i. Game states change by an increase in player score or change in player hand state. They are both weakly increasing since a player can either gain cards or gain points. Since both are weakly increasing, each game state for m game states must look like this:

$$S_1 \leq S_2 \leq \cdots \leq S_m \leq S_1$$

The mth game state cycles back to the first game state. However, for an arbitrary x and y, if

$$x \leq y \; and \; y \leq x, then \; x = y.$$

This can be extended to the game states.

$$S_i \leq S_j and \; S_j \leq S_i \;, then \; S_i = S_j$$

This means a player cannot have increase in hand state nor player score. This can be extended to the rest of the players in the game. However, we defined each transition to the next game state or node as a hit or fold. If the players are doing neither, the game cannot transition to the next game state. Thus, this is a contradiction, and the game must be acyclic. If the game is acyclic, it must be a directed graph with a base node as shown in the diagram before.

□

Heuristic Methods

Heuristic Method 1 – 50% Rule:

Of course, there is the strategy that can extend to any scenario. The player can calculate the probability of forming a pair in his or her hand. He or she would fold if the probability of taking a hit would yield 50% or more. Since there are total 55 cards in the game, that is the initial denominator. The number of cards that are face up of all players subtract from 55. Then, the numerator would be the remaining number of cards available of what is currently in the player's hand. An example is the scenario:

Player 1 has the cards 1, 2, 3, 4, and 5 face up.

Player 2 has the cards 4, 5, 7, 8, and 9 face up.

To calculate the probability that Player 1 hits and forms a pair:

55 is the initial denominator. 10 cards have been used. The new denominator is 45. There are no more 1s left. There is one more 2 left. There are 2 more 3s left. There are 2 more 4s left since both players have played a 4. There are 3 more 5s left since both players have played a 5.

$$\frac{1+2+2+3}{45} = \frac{8}{45} \approx 17.8\%$$

17.8% is less than 50%. Thus, it would be wise to hit.

Likewise, to calculate the probability that Player 2 hits and forms a pair:

55 is the initial denominator. 10 cards have been used. The new denominator is 45. There are 2 more 4s left since both players have played a 4. There are 3 more 5s left since both players have played a 5. There are 6 more 7s left. There are 7 more 8s left. There are 8 more 9s left.

$$\frac{2+3+6+7+8}{45} = \frac{26}{45} \approx 57.8\%$$

57.8% is more than 50%. Thus, it would be wise to fold.

This strategy is relatively simple. However, this is not always wise to follow. Since the game ends and a loser is decided when one player reaches the target point value, it would not be wise to fold if Player 2 was at 30 points. The target score for 2 players is $\frac{60}{n} + 1 = \frac{60}{2} + 1 = 31$ points. If Player 2 were to fold, then he or she would receive 1 point since that is the lowest card in play. However, that would make Player 2 lose the game. If Player 2 were at 5 points, then it would be fine to fold. Thus, in the case of 30 points for Player 2, it is better to hit.

Heuristic Method 2 – Expected Point Value Rule:

This method requires the player to calculate the expected value of folding and hitting. The player chooses the action that yields the lower value in points.

An example is the scenario:

Player 1 has the cards 1, 2, 3, 4, and 5 face up.

Player 2 has the cards 4, 5, 7, 8, and 9 face up.

To calculate the expected value from Player 1 hitting:

Player 1 already has a 1 and there are no more 1s in the deck, so that is 0. The probability of hitting and receiving a 2 is 1/45 since there are 45 cards left in the deck and only one 2 has been played by all players. The probability of hitting and receiving a 3 is 2/45 since there are only two 3s left in the deck. The probability of hitting and receiving a 4 is 2/45 since there have two 4s already played in the round. The probability of hitting and receiving a 5 is 3/45 since two 5s have been already played.

$$2\left(\frac{1}{45}\right) + 3\left(\frac{2}{45}\right) + 4\left(\frac{2}{45}\right) + 5\left(\frac{3}{45}\right) \approx 0.689$$

So, the expected value of hitting for Player 1 is 0.689 points. If Player 1 were to fold, the expected value would be 1. 0.689 is less than 1, so it is wise for Player 1 to hit according to this method.

Similarly, to calculate the expected value from Player 2 hitting:

Player 2's probability of hitting and receiving a 4 is 2/45 since there have two 4s already played in the round, and there are 45 cards left in the deck. The probability of hitting and receiving a 5 is 3/45 since two 5s have been already played. The probability of hitting and receiving a 7 is 6/45. The probability of hitting and receiving an 8 is 7/45. The probability of hitting a receiving a 9 is 8/45.

$$4\left(\frac{2}{45}\right) + 5\left(\frac{3}{45}\right) + 7\left(\frac{6}{45}\right) + 8\left(\frac{7}{45}\right) + 9(\frac{8}{45}) \approx 4.29$$

So, the expected value of hitting for Player 2 is 4.29 points. If Player 2 were to fold, the expected value would be 1. 4.29 is more than 1, so it is wise for Player 2 to fold according to this method.

Reminiscent of Heuristic Method 1, the expected value heuristic fails if Player 2 were at 30 points hypothetically. Player 2 would fold according to the strategy and reach the target point value, ending the game. It would be wiser to hit as that does not guarantee a loss.

Optimized Strategy

To devise the optimized strategy, I developed a computer program to calculate a table of expected probabilities of winning for each player at every game state with dynamic programming. Then, an action table was made to determine whether the player should hit or fold depending on their expected probability of winning. If hitting had a strictly higher expected probability of winning, then the action player would display True for that index to indicate that the player should hit. Else, the player should fold which is signified by a False value. The following will discuss the math used to develop the program. This will achieve Perfect Subgame Nash Equilibrium since it solves the expected probability of winning for each action of folding and hitting for each subgame. Thus, deviating from this optimized strategy will not provide a better payoff.

First, it can be established that a player can have or not have a card. The total states for a player's hand state are 2^k where k is the number of card types possible in the deck. In this case, k equals 10. So, there are 2^{10} total states of a player's hand, he or she either has each card or not. The player's state for each card can be denoted as False or 0 if the player does not contain the card and True or 1 if the player does contain the card. The player's score is also a factor for their state. For the game Pairs, the player has $2^{10} * T$ states where T is the target score. The game state includes the player state and the deck state. For each card type in the deck, there are k + 1 possibilities where k is the number of cards for a specific card type. The +1 comes from not having the card at all. To be more general, for a triangular deck, there are (k + 1)! configurations of deck states. So, the total number of game states for a 2-player game of Pairs would be

$$T = \frac{60}{2} + 1 = 31$$

$$11 * 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 2^{10} * T = 11! * 2^{10} * 31$$

However, storing each game state in a table would make the table extremely large, so bit masking can be used to turn each game state into a single value. To simplify, let us consider the scenario that there are only 3 card types, 3 threes, 2 twos, and 1 one.

Now, here are the possible configurations for one hand:

3 2 1	Value
(0, 0, 0)	0
(0, 0, 1)	1
(0, 1, 0)	2
(0, 1, 1)	3
(1, 0, 0)	4
(1, 0, 1)	5
(1, 1, 0)	6

(1, 1, 1)	7

The pattern is that values of 3 have the most priority or value, values of 2 have the second most value, and values of 1 have the least value. Now given any configuration and the number of card types, how do we find the value corresponding to it? In this case since the player can only hold one of each card, the value corresponds to the binary value or base 2 conversion. But what if we took the case the players could not just hold one card of each card type but as many possible?

c_2 c_1

2 1	Value
(0, 0)	0
(0, 1)	1
(1, 0)	2
(1, 1)	3
(2, 0)	4
(2, 1)	5

We consider only 2 card types for simplicity. We can notice a pattern with factorials. For instance, $2!\,c_2 + c_1 = 2! * 2 + 1 = 5$ where c_i is the card type for a particular card value. If there were 4 card types, the corresponding value would be:

$$4!\,c_4 + 3!\,c_3 + 2!\,c_2 + c_1$$

Thus, we can now correspond each game state which is composed of deck state and player state to a single value to simplify the table easily.

In the dynamic programming table of expected probabilities of winning, every entry represented a tuple of probabilities of winning for each player. The expected probabilities are conditional probabilities since they depend on their parent nodes (Bonanno, 300). Whenever a player scored the target score or higher, the element at that index contained an ending probability of 0s and 1s. In fact, all the ending probabilities can be expressed as an n-dimensional vector.

For instance, (1, 1, 1, 0, 1, 1, 1) represents a probability tuple where every player has won, and the fourth player has lost. By the rules of the game Pairs, one person hits or folds which causes an increase in points if they made a pair. In this case, the fourth player did either and exceeded or achieved the target score causing them to lose. Their loss is deterministic and results in a probability of winning of 0. The other players' probabilities become 1 since they win.

<u>Lemma 2</u>

□Now, I will be using structural induction to prove that the sum of the probabilities of winning for all the players at a node is n-1. Since Pairs is an acyclic game and starts with a base node, this proof can be extended for all subsequent nodes. For folding, it is deterministic. There will be a definite loser and definite winners for the endgame. Thus, there would be one 0 probability and n-1 1 probabilities which evaluates to a sum of n-1 for the sum of probabilities. The base case is a vector of just 0s and 1. However, drawing is not deterministic since there are many different cards possible to draw. Each draw has a probability associated with it. Thus, the program takes the expected probability of winning of all the draw states possible from a single node (game state).

Let $v, w \in [0,1]^n$ be a vector representing the probabilities of winning for each player at a particular game state node. For a vector v, let $v^{(j)}$ represent the jth element and the probability of player j winning the game from the game state node.

We consider a game state node and w, its vector of winning probabilities of each of the players. Suppose at the game state node under consideration, there are k possible cards that may be drawn, resulting in k possible game state nodes. We shall index the k possible draw from 1 to k and let v_i represent the vector of probabilities of winning for all the players if the ith indexed card is drawn.

Given a particular node where the player decides to draw as their action, by structural induction, we shall assume that the hypothesis is true for all of the potential resulting nodes from the draw action:

$$\sum_{i=1}^{k} v_i^{(j)} = n - 1$$

The probability of winning for a particular player at this game state node can be determined with the law of total probability. Consider the k possible draw cards, they all have a probability associated with drawing each one of them and each one has some probability of winning.

$$w = \sum_{i=1}^{k} p_i v_i$$

w is the sum of probabilities which are $p_i v_i$ for each player by Law of Total Probability which states $P(A) = \sum_n P(A \mid B_n) P(B_n)$ for event A and event B

p_i is the probability of drawing of v_i

$$\sum_{i=1}^{k} p_i = 1$$

The sum of all the probabilities of drawing cards is 1

$$\sum_{j=1}^{n} w^{(j)} = \sum_{j=1}^{n} \left(\sum_{i=1}^{k} p_i v_i \right)^{(j)}$$

Switching the order of summation

$$\sum_{j=1}^{n} w^{(j)} = \sum_{i=1}^{k} \left(\sum_{j=1}^{n} p_i v_i \right)^{(j)}$$

Factoring out the p_i since it is not dependent on j

$$\sum_{j=1}^{n} w^{(j)} = \sum_{i=1}^{k} p_i \left(\sum_{j=1}^{n} v_i \right)^{(j)}$$

Substituting $\sum_{i=1}^{k} v_i^{(j)} = n - 1$

$$\sum_{j=1}^{n} w^{(j)} = \sum_{j=1}^{k} p_i (n-1)$$

Substituting $\sum_{i=1}^{k} p_i = 1$

$$\sum_{j=1}^{n} w^{(j)} = (n-1) \sum_{j=1}^{k} p_i$$

We arrive at our hypothesis

$$\sum_{j=1}^{n} w^{(j)} = (n-1)$$

$\square$

Comparing the Strategies (Results)

Given each game state, the computer program I developed made an action table for each heuristic method and the optimized method. I compared the optimized method to each of the heuristic methods. I counted the differences for several game configurations of players and cards. Most of these are smaller configurations to accommodate for memory allocation. An example of a game configuration could be for a target score of 5, 2 players, and a triangular deck of up to 3.

Number of Players	Target Score	Triangular Deck Size	Number of Differences Between Optimized and Heuristic 1 (50% Rule)	Number of Differences Between Optimized and Heuristic 2 (Expected Point Value)
2	5	3	1916	315
2	6	3	3199	558
2	5	4	22476	5554
2	6	4	39208	10410
3	5	4	330128	61158
3	6	4	685838	120727

Conclusion

Consistently, the expected point value heuristic yielded fewer differences than the 50% rule heuristic. But of course, as the target score, number of players, and triangular deck size increased, the differences increased exponentially. The optimized computer program achieves Subgame Perfect Nash Equilibrium since it calculates the expected value of the probability of winning for every game state and determines whether the player should hit or fold. If the expected probability of winning if the player had hit is strictly larger than the expected probability of winning if the player had folded, then the program would have told the player to hit. Else, the player should fold. The program calculates the probability and correct action for each subgame. It finds the Nash Equilibrium strategy for each subgame where Nash Equilibrium is the situation where changing the player's strategy will not yield a better payoff. For extension, I would optimize the computer program more to analyze even more cases. One such case would be an arbitrary deck of difference values of different occurrences in the deck, not necessarily triangular. I have made the program able to analyze such cases, but due to memory capabilities, it currently has a limited range.

Bibliography

Bonanno, Giacomo. *Game Theory*. Davis, CreateSpace Independent Publishing Platform, 2018.

Jost, Jurgen. "Game Theory. Mathematical and Conceptual Aspects." *Max Planck Institute for Mathematics in the Sciences,* 19 July 2013.

"PAIRS." *Cheapass.com*, cheapass.com/pairs/.

Ross, Don. "Game Theory." *Stanford Encyclopedia of Philosophy*, Stanford University, 8 Mar. 2019, plato.stanford.edu/archives/win2019/entries/game-theory/. Accessed 1 Aug. 2020.

Appendix

Code:

Deck.py

```
from numpy.random import choice
import random
import copy

class Deck:
    # How storing the cards structure? Is it a list? Is it a dict?
    # Dictionary: Key (Card type) -> Value (Number of how many cards there
are)
    # How do you randomly generate a card?
    # Use numpy.choice and pass in a probability distribution

    def __init__(self, original_deck, curr_deck=None, discards=None,
shuffling_limit=0):
        self.original_deck = original_deck
        if curr_deck is None:
            curr_deck = original_deck.copy()
        self.curr_deck = curr_deck
        if discards is None:
            discards = {card_type: 0 for card_type in original_deck}
        self.discards = discards
        self.num_distinct_cards = len(self.original_deck)
        self.num_total_cards = sum(self.original_deck.values())
        self.shuffling_limit = shuffling_limit

    def draw_specific_card(self, draw_card):
        if self.curr_deck[draw_card] <= 0:
            raise ValueError("Draw card {} is not present in the current deck
{}".format(draw_card, self.curr_deck))
        self.curr_deck[draw_card] -= 1
        if sum(self.curr_deck.values()) <= self.shuffling_limit:
            self.shuffle_discards()

    def draw_random_card(self):
        probabilities = [float(card_frequency) / sum(self.curr_deck.values())
                         for card_type, card_frequency in
self.curr_deck.items()]
        draw = choice(list(self.curr_deck.keys()), 1, False, probabilities)
        self.curr_deck.draw_specific_card(draw)

    def shuffle_discards(self):
        for card_type in self.curr_deck:
            self.curr_deck[card_type] += self.discards[card_type]
        for card_type in self.discards:
            self.discards[card_type] = 0

    def add_single_discard(self, card):
        self.discards[card] += 1

    def add_dict_discards(self, cards):
        for card_type, freq in cards.items():
            self.discards[card_type] += freq
    def deepcopy(self):
        c = Deck(copy.deepcopy(self.original_deck),
copy.deepcopy(self.curr_deck), copy.deepcopy(self.discards),
                 self.shuffling_limit)
        return c
```

TriangularDeck.py

```
from src.pairs.Deck import Deck


class TriangularDeck(Deck):
    def __init__(self, num_distinct):
        original_deck = {i: i for i in range(1, num_distinct+1)}
        super().__init__(original_deck)
```

Player.py

```
import copy

class Player:
    def __init__(self, original_deck, points=0, hand_state=None,
hand_capacity=None):
        self.original_deck = original_deck
        self.points = points
        if hand_state is None:
            hand_state = self.hand_state = {card_type: 0 for card_type in
original_deck}
        self.hand_state = hand_state
        if hand_capacity is None:
            hand_capacity = {card_type: 1 for card_type in original_deck}
        self.hand_capacity = hand_capacity

    def increase_points(self, n):
        self.points += n

    def add_card(self, card):
        self.hand_state[card] += 1

    def reset(self):
        for card_type in self.hand_state:
            self.hand_state[card_type] = 0

    def deepcopy(self):
        c = Player(copy.deepcopy(self.original_deck), self.points,
copy.deepcopy(self.hand_state),
                   copy.deepcopy(self.hand_capacity))
        return c
```

Game State.py

```
from src.pairs.Player import Player
from src.pairs.util import intMapper
from collections import deque, Iterable
import numpy as np
import copy


class GameState:
    def __init__(self, deck, target_score, players):
        self.deck = deck
        self.target_score = target_score
        if isinstance(players, int):
            self.num_players = players
            self.players = deque(Player(deck.original_deck) for _ in
range(self.num_players))
        elif isinstance(players, Iterable):
            self.players = deque(players)
            self.num_players = len(self.players)

    def deepcopy(self):
        c = GameState(copy.deepcopy(self.deck), self.target_score,
                      deque(copy.deepcopy(player) for player in
self.players))
        return c

    def next_draw_states(self):
        states = []
        for card_type in self.deck.curr_deck:
            total_cards = sum(self.deck.curr_deck.values())
            if self.deck.curr_deck[card_type] > 0:
                new_state = copy.deepcopy(self)
                new_state.deck.draw_specific_card(card_type)
                new_state.players[0].add_card(card_type)
                if new_state.players[0].hand_state[card_type] >
new_state.players[0].hand_capacity[card_type]:
                    new_state.player_scored(card_type)
                new_state.players.rotate(-1)
                states.append((new_state, self.deck.curr_deck[card_type] /
total_cards))
        return states

    def next_fold_state(self):
        next_fold_state = copy.deepcopy(self)
        minimum = min((min(
            (card_type for card_type, freq in player.hand_state.items() if
freq > 0), default=float('inf'))
            for player in next_fold_state.players), default=float('inf'))
        if minimum == float('inf'):
            return None
        next_fold_state.player_scored(minimum)
        next_fold_state.players.rotate(-1)
        return next_fold_state

    def player_scored(self, points):
        self.players[0].points += points
        for player in self.players:
            self.deck.add_dict_discards(player.hand_state)
            player.reset()
    # ending_state returns None if the game state is not an ending state
    # otherwise, it returns a numpy array representing the probabilities of 
winning
    # for each of the players in the game
    def ending_probabilities(self):
        if any(player.points >= self.target_score for player in 
self.players):
            return np.array([float(player.points < self.target_score) for 
player in self.players])
        return None

    def game_to_tuple(self):
        curr_deck_state = [intMapper(self.deck.original_deck.values(), 
self.deck.curr_deck.values())]
        curr_player_states = [intMapper(list(player.hand_capacity.values()) + 
[self.target_score], 
                                        list(player.hand_state.values()) + 
[player.points])
                              for player in self.players]
        curr_state = tuple(curr_deck_state + curr_player_states)
        return curr_state
```

util.py

```
import numpy
# Capacity list will be a list of capacities, e.g. [72, 46, 23]
# and value list will be a list of particular values, e.g. [36, 12, 17]
# and converts it to an integer using the base system
def intMapper(capacity_list, value_list):
    value = 0
    list_convert = [1]
    list_convert.extend(c + 1 for c in capacity_list)
    list_products = list_convert.copy()
    values_list = list(value_list)
    for i in range(1, len(list_products)):
        list_products[i] = list_products[i] * list_products[i-1]
    for i in range(len(capacity_list)):
        value += values_list[i] * list_products[i]
    return value
```

DPSolver.py

```
from src.pairs.util import intMapper
import numpy as np


class DPSolver:
    def __init__(self, game):
        deck_states = np.prod([card + 1 for card in 
game.deck.original_deck.values()])
        player_states = np.prod([value + 1 for value in 
game.players[0].hand_capacity.values()]) * game.target_score
        dimensions = [deck_states] + [player_states] * game.num_players
        self.action_table = np.zeros(tuple(dimensions), dtype=bool)
        self.fifty_percent_action_table = np.zeros(tuple(dimensions),
dtype=bool)
        self.expected_point_action_table = np.zeros(tuple(dimensions),
dtype=bool)
        dimensions.append(game.num_players)
        self.dp_table = np.empty(tuple(dimensions))
        self.dp_table[:] = np.nan
        self.game = game

    # Solve will return a 1-dimensional numpy array
    # representing the probabilities of winning for each of the players
    def solve(self, game_state):
        # 1. Check if game state is an ending game state. If it is, return
that probability array
        # 2. Check if game state already exists in the dp_table, if it
doesn't, solve for the
        #       fold probabilities and hit probabilities and update action
table and dp table
        # 3. Return answer
        ending_probabilities = game_state.ending_probabilities()
        if ending_probabilities is not None:
            return ending_probabilities
        table_index = game_state.game_to_tuple()
        if np.isnan(self.dp_table[table_index][0]):
            fold_state = game_state.next_fold_state()
            fold_probabilities = np.roll(self.solve(fold_state), 1) if
fold_state is not None else None
            draw_probabilities = np.roll(
                sum(prob * self.solve(draw_state) for draw_state, prob in
game_state.next_draw_states()), 1)
            self.fifty_percent(game_state, table_index)
            self.expected_point(game_state, table_index)
            if fold_probabilities is None or draw_probabilities[0] >
fold_probabilities[0]:
                self.action_table[table_index] = True
                self.dp_table[table_index] = draw_probabilities
            else:
                self.action_table[table_index] = False
                self.dp_table[table_index] = fold_probabilities
        return self.dp_table[table_index]

    def fifty_percent(self, game_state, table_index):
        hit_fifty_probability = 0
        for card_type in game_state.players[0].hand_state:
            if game_state.players[0].hand_state[card_type] > 0:
                hit_fifty_probability += game_state.deck.curr_deck[card_type]
/ sum(game_state.deck.curr_deck.values())
        if hit_fifty_probability > .5:
            self.fifty_percent_action_table[table_index] = True
        else:
            self.fifty_percent_action_table[table_index] = False

    def expected_point(self, game_state, table_index):
        hit_expected_point = 0
        fold_expected_point = min((min(
            (card_type for card_type, freq in player.hand_state.items() if
freq > 0), default=float('inf'))
            for player in game_state.players), default=float('inf'))
        for card_type in game_state.players[0].hand_state:
            if game_state.players[0].hand_state[card_type] > 0:
                hit_expected_point += card_type * \

(game_state.deck.curr_deck[card_type] /

sum(game_state.deck.curr_deck.values()))
        if hit_expected_point < fold_expected_point:
            self.expected_point_action_table[table_index] = True
        else:
            self.expected_point_action_table[table_index] = False

    def get_dp_table(self):
        return self.dp_table

    def get_action_table(self):
        return self.action_table

    def get_fifty_percent_table(self):
        return self.fifty_percent_action_table

    def get_expected_point_action_table(self):
        return self.expected_point_action_table
```

Main.py

```
from src.pairs.TriangularDeck import TriangularDeck
from src.pairs.util import intMapper
from src.pairs.GameState import GameState
from src.pairs.DPSolver import DPSolver
from src.pairs.Deck import Deck
import numpy as np


def main():
    deck = TriangularDeck(4)
    game = GameState(deck, 3, 4)
    solver = DPSolver(game)
    solver.solve(game)
    diff_dp_fifty = solver.get_action_table() ==
solver.get_fifty_percent_table()
    print(np.count_nonzero(diff_dp_fifty == 0))
    diff_dp_exp = solver.get_action_table() ==
solver.get_expected_point_action_table()
    print(np.count_nonzero(diff_dp_exp == 0))


if __name__ == "__main__":
    main()
```