An Introduction to Counterfactual Regret Minimization

Todd W. Neller[*]   Marc Lanctot[†]

July 9, 2013

1   Motivation

In 2000, Hart and Mas-Colell introduced the important game-theoretic algorithm of *regret matching*. Players reach equilibrium play by tracking regrets for past plays, making future plays proportional to positive regrets. The technique is not only simple and intuitive; it has sparked a revolution in computer game play of some of the most difficult bluffing games, including clear domination of annual computer poker competitions.

Since the algorithm is relatively recent, there are few curricular materials available to introduce regret-based algorithms to the next generation of researchers and practitioners in this area. These materials represent a modest first step towards making recent innovations more accessible to advanced Computer Science undergraduates, graduate students, interested researchers, and ambitious practitioners.

In Section 2, we introduce the concept of player regret, describe the regret-matching algorithm, present a rock-paper-scissors worked example in the literate programming style, and suggest related exercises. Counterfactual Regret Minimization (CFR) is introduced in Section 3 with a worked example solving Kuhn Poker. Supporting code is provided for a substantive CFR exercise computing optimal play for 1-die-versus-1-die Dudo. In Section 4, we briefly mention means of "cleaning" approximately optimal computed policies, which can in many cases improve results. Section 5 covers an advanced application of CFR to games with repeated states (e.g. through imperfect recall abstraction) that can reduce computational complexity of a CFR training iteration from exponential to linear. Here, we use our independently devised game of Liar Die to demonstrate application of the algorithm. We then suggest that the reader apply the technique to 1-die-versus-1-die Dudo with a memory of 3 claims. In Section 6, we briefly discuss an open research problem: Among possible equilibrium strategies, how do we compute one that optimally exploits opponent errors? The reader is invited to modify our Liar Die example code to so as to gain insight to this interesting problem. Finally, in Section 7, we suggest further challenge problems and paths for continued learning.

2   Regret in Games

In this section, we describe a means by which computers may, through self-simulated play, use regrets of past game choices to inform future choices. We begin by introducing the familiar game of Rock-Paper-Scissors (RPS), a.k.a. Roshambo. After defining foundational terms of game theory, we discuss regret matching and present an algorithm computing strategy that minimizes expected regret. Using this algorithm, we present a worked example for learning RPS strategy and associated exercises.

---

[*]tneller@gettysburg.edu, Gettysburg College, Department of Computer Science, Campus Box 402, Gettysburg, PA 17325-1486

[†]marc.lanctot@maastrichtuniversity.nl, Department of Knowledge Engineering, Maastricht University

2.1   Rock-Paper-Scissors

Rock-Scissors-Paper (RPS) is a two-player game where players each simultaneously make one of three gestures: rock (a closed fist), paper (an open face-down palm), or scissors (exactly two fingers extended). With each gesture, there is an opportunity to win, lose, or draw against the other player. Players showing the same gesture draw. A rock wins against scissors, because "rock breaks scissors". Scissors wins against paper, because "scissors cuts paper". Paper wins against rock, because "paper covers rock".

Players will commonly synchronize play by calling out a four-beat chant, "Rock! Paper! Scissors! Shoot!", bobbing an outstretched fist on the first three beats, and committing simultaneously to one of the three gestures on the fourth beat.

2.2   Game Theoretic Definitions

What does it mean to play such a game optimally or perfectly? Does this question itself hold any meaning, given that maximizing wins minus losses depends on how the opponent plays? In this section, we introduce some fundamental terminology and definitions from game theory, and consider *solution concepts* for optimal play. Here, we follow the notation and terminology of [12].

First, let us define a *normal-form game* as a tuple (N, A, u), where:

- $N = \{1, ..., n\}$ is a finite set of n players.

- S_i is a finite set of *actions* or *choices* for player i.

- $A = S_1 \times \ldots \times S_n$ is the set of all possible combination of simultaneous actions of all players. (Each possible combination of simultaneous actions is called an *action profile*.)

- u is a function mapping each action profile to a vector of utilities for each player. We refer to player i's payoff as u_i.

A normal-form game is commonly also called a "one-shot game" since each player only makes a single choice. One can represent such games as an n-dimensional table, where each dimension has rows/columns corresponding to a single player's actions, each table entry corresponds to a single action profile (the intersection of a single action from each player), and the table entry contains a vector of *utilities* (a.k.a. *payoffs* or *rewards*) for each player. The payoff table for RPS is as follows:

	R	P	S
R	$0, 0$	$-1, 1$	$1, -1$
P	$1, -1$	$0, 0$	$-1, 1$
S	$-1, 1$	$1, -1$	$0, 0$

where each entry has the form (u_1, u_2). By convention, the row player is player 1 and the column player is player 2. For example, in RPS, $A = \{(R, R), (R, P), \cdots, (S, P), (S, S)\}$.

A normal-form game is *zero-sum* if the values of each utility vector sum to 0. *Constant-sum* games, where the values of each utility vector sum to a constant, may be reformulated as zero-sum games by adding a dummy player with a single dummy action that always receives the negated constant as a payoff.

A player plays with a *pure strategy* if the player chooses a single action with probability 1. A player plays with a *mixed strategy* if the player has at least two actions that are played with positive probability.

We use σ to refer to a mixed strategy, and define $\sigma_i(s)$ to be the probability of player i chooses action $s \in S_i$. By convention, $-i$ generally refers to player i's opponents, so in a two-player game $S_{-i} = S_{3-i}$. To compute the *expected utility* of the game for an agent, sum over each action profile

the product of each player's probability of playing their action in the action profile, times the player's utility for the action profile:

$$u_i(\sigma_i, \sigma_{-i}) = \sum_{s \in S_i} \sum_{s' \in S_{-i}} \sigma_i(s)\sigma_{-i}(s')u_i(s, s'),$$

in the two-player case. A *best response* strategy for player i is one that, given all other player strategies, maximizes expected utility for player i. When every player is playing with a best response strategy to each of the other player's strategies, the combination of strategies is called a *Nash equilibrium*. No player can expect to improve play by changing strategy alone.

Consider the Battle of the Sexes game:

<div align="center">

Gary

		M	G
	M	2, 1	0, 0
Monica	G	0, 0	1, 2

</div>

Monica is the row player, and Gary is the column player. Suppose Monica and Gary are going out on a date and need to choose an activity (e.g. movie, restaurant, etc.). Gary would like to go to a football game (G) and Monica wants to see a movie (M). They both prefer going together to the same activity, yet each feels less rewarded for choosing the other's preference.

Suppose Monica always chooses M. Gary is better off choosing M and has no incentive to unilaterally deviate from that pure strategy. Likewise, if Gary always chooses G, Monica has no incentive to unilaterally deviate from her pure strategy. The utility is always $(2, 1)$ or $(1, 2)$. So, (M, M) and (G, G) are two pure Nash equilibria profiles. However, there is a mixed strategy Nash equilibrium as well. An equilibrium can be reached when each player, seeing other strategies, is *indifferent* to the choice of action, i.e. all are equally good.

What would have to be the case for Monica to be indifferent to the Gary's choice? Let $\sigma_{Gary}(M) = x$ be the probability of Gary choosing the movie. Then the utility that Monica expects is $2x + 0(1 - x)$ and $0x + 1(1 - x)$ respectively. For Monica to be indifferent between G and M, these two expected utilities would need to be equal. Solving $2x + 0(1 - x) = 0x + 1(1 - x)$ for x, we get $x = \frac{1}{3}$.

With symmetric reasoning, Gary is indifferent when Monica chooses the football game with probability $\sigma_{Monica}(G) = \frac{1}{3}$. Thus, Monica and Gary can use mixed strategies of $(\frac{2}{3}, \frac{1}{3})$ and $(\frac{1}{3}, \frac{2}{3})$, respectively. This pair of mixed strategies forms a Nash equilibrium as neither player can hope to improve their expected utilities through unilateral strategy change. Both players are indifferent to change. Note that these Nash equilibrium strategies yield different expected utility for the players. (What is each player's expected utility for each of the three equilibria?)

The Nash equilibrium is one *solution concept*. Another more general solution concept is that of the *correlated equilibrium*. Now, imagine that both players have access to some type of random signal from a third-party. Players receive information about the signal, but not information about what the signal indicates to other players. If players correlate play with the signals, i.e. each signal corresponds to an action profile, i.e. an action for each player, and each player expects no utility gain from unilaterally changing the player mapping of signals to actions, then the players have reached a *correlated equilibrium*.

Each Nash equilibrium is a correlated equilibrium, but the concept of correlated equilibrium is more general, and permits important solutions. Consider again the Battle of the Sexes. As a simple signal example, imagine a fair coin toss. Players could arrive at a cooperative behavior whereby, for instance, a coin flip of head and tail correspond to both players choosing M and G, respectively. Having reached this equilibrium, neither player has incentive to unilaterally change this mapping of signals to strategies, and both players receive an average utility of 1.5.

2.3   Regret Matching and Minimization

Suppose we are playing RPS for money. Each player places a dollar on a table. If there is a winner, the winner takes both dollars from the table. Otherwise, players retain their dollars. Further suppose that we play rock while our opponent plays paper and wins, causing us to lose our dollar. Let our utility be our net gain/loss in dollars. Then our utility for this play was -1. The utility for having instead played paper and scissors against the opponent's paper would have been 0 and +1, respectively.

We regret not having played paper and drawing, but we regret not having played scissors even more, because our relative gain would have been even greater in retrospect. We here define *regret* of not having chosen an action as the difference between the utility of that action and the utility of the action we actually chose, with respect to the fixed choices of other players.

For action profile $a \in A$ let s_i be player i's action and s_{-i} be the actions of all other players. Further, let $u(s_i', s_{-i})$ be the utility of an action profile with s_i' substituted for s_i, i.e. the utility if player i had played s_i' in place of s_i. Then, after the play, player i's regret for not having played s_i' is $u(s_i', s_{-i}) - u(a)$. Note that this is 0 when $s_i' = s_i$.

For this example, we regret not having played paper $u(\text{paper, paper}) - u(\text{rock, paper}) = 0 - (-1) = 1$, and we regret not having played scissors $u(\text{scissors, paper}) - u(\text{rock, paper}) = +1 - (-1) = 2$.

How might this inform future play? In general, one might prefer to choose the action one regretted most not having chosen in the past, but one wouldn't want to be entirely predictable and thus entirely exploitable. One way of accomplishing this is through *regret matching*, where an agents actions are selected at random with a distribution that is *proportional to positive regrets*. Positive regrets indicate the level of relative losses one has experienced for not having selected the action in the past. In our example, we have no regret for having chosen rock, but we have regrets of 1 and 2 for not having chosen paper and scissors, respectively. With regret matching, we then choose our next action proportionally to the positive regrets, and thus choose rock, paper, and scissors with probabilities 0, $\frac{1}{3}$, and $\frac{2}{3}$, respectively, which are *normalized positive regrets*, i.e. positive regrets divided by their sum.

Now suppose in the next game, we happen to choose scissors (with probability $\frac{2}{3}$) while our opponent chooses rock. For this game, we have regrets 1, 2, and 0 for the respective play of rock, paper, and scissors. Adding these to our previous regrets, we have *cumulative regrets* of 1, 3, and 2, respectively, thus regret-matching for our next game yields a mixed-strategy of $(\frac{1}{6}, \frac{3}{6}, \frac{2}{6})$.

Ideally, we would like to minimize our expected regrets over time. This practice alone, however, is insufficient to minimize our expected regrets. Imagine now that you are the opponent, and you fully understand the regret matching approach that is being used. Then you could perform the same computations, observe any bias we would have towards a play, and exploit that bias. By the time we had learned to regret that bias, the damage would have already been done, and our new dominant regret(s) would be similarly exploited.

However, there is a computational context in which regret matching can be used to minimize expected regret through self-play. The algorithm is as follows:

- For each player, initialize all cumulative regrets to 0.

- For some number of iterations:

    - Compute a regret-matching strategy profile. (If all regrets for a player are non-positive, use a uniform random strategy.)

    - Add the strategy profile to the strategy profile sum.

    - Select each player action profile according the strategy profile.

    - Compute player regrets.

    - Add player regrets to player cumulative regrets.

- Return the average strategy profile, i.e. the strategy profile sum divided by the number of iterations.

Over time, this process converges to a correlated equilibrium [3]. In the next section, we provide a worked example of this algorithm applied to RPS.

2.4   Worked Example: Rock-Paper-Scissors

Now we present a worked example of regret matching for the computation of a best response strategy in Rock, Paper, Scissors (RPS). In RPS, the extension of regret-matching to the two-sided case results in an equilibrium, and is left as an exercise at the end of the section.

We begin with definition of constants and variables that are used throughout the process.

⟨*Definitions*⟩≡

```
public static final int ROCK = 0, PAPER = 1, SCISSORS = 2, NUM_ACTIONS = 3;
public static final Random random = new Random();
double[] regretSum = new double[NUM_ACTIONS],
         strategy = new double[NUM_ACTIONS],
         strategySum = new double[NUM_ACTIONS],
         oppStrategy = { 0.4, 0.3, 0.3 };
```

Although unused in our code, we arbitrarily assign the actions of ROCK, PAPER, and SCISSORS, the zero-based action values of 0, 1, and 2, respectively. Such action indices correspond to indices in any strategy/regret array of length NUM_ACTIONS. We create a random number generator which is used to choose an action from a mixed strategy. Finally, we allocate arrays to hold our accumulated action regrets, a strategy generated through regret-matching, and the sum of all such strategies generated.

Regret-matching selects actions in proportion to positive regrets of not having chosen them in the past. To compute mixed a strategy through regret-matching, we begin by first copying all *positive* regrets and summing them. We then make a second pass through the strategy entries. If there is at least one action with positive regret, we normalize the regrets by dividing by the our normalizing sum of positive regrets. To normalize in this context means that we ensures that array entries sum to 1 and thus represent probabilities of the corresponding actions in the computed mixed strategy.

⟨*Get current mixed strategy through regret-matching*⟩≡

```
private double[] getStrategy() {
    double normalizingSum = 0;
    for (int a = 0; a < NUM_ACTIONS; a++) {
        strategy[a] = regretSum[a] > 0 ? regretSum[a] : 0;
        normalizingSum += strategy[a];
    }
    for (int a = 0; a < NUM_ACTIONS; a++) {
        if (normalizingSum > 0)
          strategy[a] /= normalizingSum;
        else
          strategy[a] = 1.0 / NUM_ACTIONS;
        strategySum[a] += strategy[a];
    }
    return strategy;
}
```

Some readers may be unfamiliar with the selection operator (i.e. *condition* ? *true expression* : *false expression*). It is the expression analogue of an if-else statement. First, the *condition* is evaluated. If the result is true/false, the *true/false expression* is evaluated and the overall expression takes on this value. The selection operator is found in languages such as C, C++, and Java, and behaves as the "if" in functional languages such as LISP and Scheme.

Note that the normalizing sum could be non-positive. In such cases, we make the strategy uniform, giving each action an equal probability (`1.0 / NUM_ACTIONS`).

Once each probability of this mixed strategy is computed, we accumulate that probability to a sum of all probabilities computed for that action across all training iterations.

The strategy is then returned. Given any such strategy, one can then select an action according to such probabilities. Suppose we have a mixed strategy $(.2, .5, .3)$. If one divided the number line from 0 to 1 in these proportions, the divisions would fall at .2 and $.2 + .5 = .7$. The generation of a random number in the range $[0, 1)$ would then fall proportionally into one of the three ranges $[0, .2)$, $[.2, .7)$, or $[.7, 1)$, indicating the probabilistic selection of the corresponding action index.

In general, suppose one has actions $a_0, \ldots, a_i, \ldots, a_n$ with probabilities $p_0, \ldots, p_i, \ldots, p_n$. Let cumulative probability $c_i = \sum_{j=0}^{i} p_j$. (Note that $c_n = 1$ because all probabilities must sum to 1.) A random number r uniformly generated in the range $(0, 1]$ will select action i if and only if for all $j < i$, $r \geq c_j$ and $r < c_i$.

The action is easily computed as follows. First, one generates a random floating-point number in the range $(0, 1]$, initializes the action index `a` to 0, and initializes the cumulative probability to 0. If we were to reach the last action index (`NUM_ACTIONS - 1`), that would necessarily be the action selected, so as long as the action index is not our last, we add the new probability to our cumulative probability, break out of the loop if r is found to be less than the cumulative probability, and otherwise increment the action index.

⟨*Get random action according to mixed-strategy distribution*⟩≡
```
  public int getAction(double[] strategy) {
      double r = random.nextDouble();
      int a = 0;
      double cumulativeProbability =  0;
      while (a < NUM_ACTIONS - 1) {
          cumulativeProbability += strategy[a];
          if (r < cumulativeProbability)
              break;
          a++;
      }
      return a;
  }
```

With these building blocks in place, we can now construct our training algorithm:

⟨*Train*⟩≡
```
  public void train(int iterations) {
      double[] actionUtility = new double[NUM_ACTIONS];
      for (int i = 0; i < iterations; i++) {
          ⟨Get regret-matched mixed-strategy actions⟩
          ⟨Compute action utilities⟩
          ⟨Accumulate action regrets⟩
      }
  }
```

For a given number of iterations, we compute our regret-matched, mixed-strategy actions, compute the respective action utilities, and accumulate regrets with respect to the player action chosen.

To select the actions chosen by the players, we compute the current, regret-matched strategy, and use it to select actions for each player. Because strategies can be mixed, using the same strategy does not imply selecting the same action.

⟨*Get regret-matched mixed-strategy actions*⟩≡
```
  double[] strategy = getStrategy();
  int myAction = getAction(strategy);
  int otherAction = getAction(oppStrategy);
```

Next, we compute the utility of each possible action from the perspective of the player playing `myAction`:

⟨*Compute action utilities*⟩≡
```
  actionUtility[otherAction] = 0;
  actionUtility[otherAction == NUM_ACTIONS - 1 ? 0 : otherAction + 1] = 1;
  actionUtility[otherAction == 0 ? NUM_ACTIONS - 1 : otherAction - 1] = -1;
```

Finally, for each action, we compute the regret, i.e. the difference between the action's expected utility and the utility of the action chosen, and we add it to our cumulative regrets.

⟨*Accumulate action regrets*⟩≡
```
  for (int a = 0; a < NUM_ACTIONS; a++)
      regretSum[a] += actionUtility[a] - actionUtility[myAction];
```

For each individual iteration of our training, the regrets may be temporarily skewed in such a way that an important strategy in the mix has a negative regret sum and would never be chosen. Regret sums and thus individual iteration strategies are highly erratic[1]. What converges to a minimal regret strategy is the *average strategy* across all iterations. This is computed in a manner similar to `getStrategy` above, but without the need to be concerned with negative values.

⟨*Get average mixed strategy across all training iterations*⟩≡
```
  public double[] getAverageStrategy() {
      double[] avgStrategy = new double[NUM_ACTIONS];
      double normalizingSum = 0;
      for (int a = 0; a < NUM_ACTIONS; a++)
          normalizingSum += strategySum[a];
      for (int a = 0; a < NUM_ACTIONS; a++)
          if (normalizingSum > 0)
              avgStrategy[a] = strategySum[a] / normalizingSum;
          else
              avgStrategy[a] = 1.0 / NUM_ACTIONS;
      return avgStrategy;
  }
```

The total computation consists of constructing a trainer object, performing training for a given number of iterations (in this case, 1,000,000), and printing the resulting average strategy.

⟨*Main method initializing computation*⟩≡
```
  public static void main(String[] args) {
      RPSTrainer trainer = new RPSTrainer();
      trainer.train(1000000);
      System.out.println(Arrays.toString(trainer.getAverageStrategy()));
  }
```

---

[1]Add print statements to this code to print the regret sums each iteration.

Putting all of these elements together, we create a Rock Paper Scissors trainer that utilizes regret matching in order to approximately minimize expected regret over time:

⟨*RPSTrainer.java*⟩≡

```
import java.util.Arrays;
import java.util.Random;

public class RPSTrainer {

    ⟨Definitions⟩
    ⟨Get current mixed strategy through regret-matching⟩
    ⟨Get random action according to mixed-strategy distribution⟩
    ⟨Train⟩
    ⟨Get average mixed strategy across all training iterations⟩
    ⟨Main method initializing computation⟩

}
```

The average strategy that is computed by regret matching is the strategy that minimizes regret against the opponent's fixed strategy. In other words, it is a best response to their strategy. In this case, the opponent used a strategy of $(0.4, 0.3, 0.3)$. It might not be obvious, but there is always a pure best response strategy to any mixed strategy. In this case, what pure strategy would be the best response? Does this correspond to the output of RPSTrainer?

2.5   Exercise: RPS Equilibrium

In Rock Paper Scissors and every two-player zero-sum game: when both players use regret-matching to update their strategies, the pair of average strategies converges to a Nash equilibrium as the number of iterations tends to infinity. At each iteration, both players update their regrets as above and then both each player computes their own new strategy based on their own regret tables.

Modify the RPSTrainer program above so that both players use regret matching. Compute and print the resulting unique equilibrium strategy.

2.6   Exercise: Colonel Blotto

Colonel Blotto and his arch-enemy, Boba Fett, are at war. Each commander has S soldiers in total, and each soldier can be assigned to one of $N < S$ battlefields. Naturally, these commanders do not communicate and hence direct their soldiers independently. Any number of soldiers can be allocated to each battlefield, including zero. A commander claims a battlefield if they send more soldiers to the battlefield than their opponent. The commander's job is to break down his pool of soldiers into groups to which he assigned to each battlefield. The winning commander is the one who claims the most battlefields. For example, with $(S, N) = (10, 4)$ a Colonel Blotto may choose to play $(2, 2, 2, 4)$ while Boba Fett may choose to play $(8, 1, 1, 0)$. In this case, Colonel Blotto would win by claiming three of the four battlefields. The war ends in a draw if both commanders claim the same number of battlefields.

Write a program where each player alternately uses regret-matching to find a Nash equilibrium for this game with $S = 5$ and $N = 3$. Some advice: before starting the training iterations, first think about all the valid pure strategies for one player; then, assign each pure strategy an ID number. Pure strategies can be represented as strings, objects, or 3-digit numbers: make a global array of these pure strategies whose indices refer to the ID of the strategy. Then, make a separate function that returns the utility of the one of the players given the IDs of the strategies used by each commander.

3   Counterfactual Regret Minimization

In this section, we see how regret minimization may be extended to sequential games, where players must play a sequence of actions to reach a terminal game state. We begin with definition of terminology regarding extensive game representations, and the counterfactual regret minimization algorithm. We then present a worked example, demonstrating application to Kuhn poker. A 1-die-versus-1-die Dudo exercise concludes the section.

3.1   Kuhn Poker Defined

Kuhn Poker is a simple 3-card poker game by Harold E. Kuhn [8]. Two players each ante 1 chip, i.e. bet 1 chip blind into the pot before the deal. Three cards, marked with numbers 1, 2, and 3, are shuffled, and one card is dealt to each player and held as private information. Play alternates starting with player 1. On a turn, a player may either *pass* or *bet*. A player that bets places an additional chip into the pot. When a player passes after a bet, the opponent takes all chips in the pot. When there are two successive passes or two successive bets, both players reveal their cards, and the player with the higher card takes all chips in the pot.

Here is a summary of possible play sequences with the resulting chip payoffs:

Sequential Actions			Payoff
Player 1	Player 2	Player 1	
pass	pass		+1 to player with higher card
pass	bet	pass	+1 to player 2
pass	bet	bet	+2 to player with higher card
bet	pass		+1 to player 1
bet	bet		+2 to player with higher card

This being a zero-sum game of chips, the losing player loses the number of chips that the winner gains.

3.2   Sequential Games and Extensive Form Representation

Games like Kuhn Poker are *sequential* games, in that play consists of a sequence of actions. Such a game can indeed be reformulated as a one-time-action normal-form game if we imagine that players look at their dealt cards and each choose from among the pure strategies for each possible play situation in advance as a reformulated meta-action. For example, player 1 may look at a 3 in hand and decide in advance, as a single meta-action, to commit to betting on the first round and betting on the third round (if it occurs). Player 2 may look at a 2 in hand and decide to bet if player 1 bets and pass in player 1 passes.

Instead, we will use a different representation. The game tree is formed of states with edges representing transitions from state to state. A state can be a chance node or a decision node. The function of chance nodes is to assign an outcome of a chance event, so each edge represents one possible outcome of that chance event as well as a probability of the event occurring. At a decision node, the edges represent actions and successor states that result from the player taking those actions.

Each decision node in the game tree is contained within an *information set* which (1) contains an active player and all information available to that active player at that decision in the game, and (2) can possibly include more than one game state. For example, after player 1 first acts, player 2 would know two pieces of information: player 1's action (pass or bet), and player 2's card. Player 2 would not know player 1's card, because that is private information. In fact, player 1 could have either of the two cards player 2 is not holding, so the information set contains two possible game states. Player 2

cannot know which game state is the actual game state, and this uncertainty arises from having this game being *partially observable* with private card knowledge, and not knowing the opponent's strategy.

So for Kuhn Poker, there is an information set for each combination of card a player can be holding with each possible non-terminal sequence of actions in the game. Kuhn Poker has 12 information sets. Can you list them? How many possible game states are there in each information set?

3.3   Counterfactual Regret Minimization

Counterfactual regret minimization uses the regret-matching algorithm presented earlier. In addition, (1) one must additionally factor in the probabilities of reaching each information set given the players' strategies, and (2) given that the game is treated sequentially through a sequence of information sets, there is a passing forward of game state information and probabilities of player action sequences, and a passing backward of utility information through these information sets.

We will now summarize the Counterfactual Regret Minimization (CFR) algorithm, directing the reader to [18] and [11] for detailed descriptions and proofs. At each information set recursively visited in a training iteration, a mixed strategy is computed according to the regret-matching equation, for which we now provide notation and define in a manner similar to [11].

Let A denote the set of all game *actions*. Let I denote an *information set*, and $A(I)$ denote the set of legal actions for information set I. Let t and T denote time steps. (Within both algorithms, t is with respect to each information set and is incremented with each visit to the information set.) A *strategy* σ_i^t for player i maps each player i information set I_i and legal player i action $a \in A(I_i)$ to the probability that the player will choose a in I_i at time t. All player strategies together at time t form a *strategy profile* σ^t. We refer to a strategy profile that excludes player i's strategy as σ_{-i}. Let $\sigma_{I \to a}$ denote a profile equivalent to σ, except that action a is always chosen at information set I.

A *history* h is a sequence of actions (included chance outcomes) starting from the root of the game. Let $\pi^\sigma(h)$ be the reach probability of game history h with strategy profile σ. Further, let $\pi^\sigma(I)$ be the probability of reaching information set I through all possible game histories in I, i.e. $\pi^\sigma(I) = \sum_{h \in I} \pi^\sigma(h)$. The counterfactual reach probability of information state I, $\pi_{-i}^\sigma(I)$, is the probability of reaching I with strategy profile σ except that, we treat current player i actions to reach the state as having probability 1. In all situations we refer to as "counterfactual", one treats the computation as if player i's strategy was modified to have intentionally played to information set I_i. Put another way, we *exclude* the probabilities that factually came into player i's play from the computation.

Let Z denote the set of all terminal game histories (sequences from root to leaf). Then proper prefix $h \sqsubset z$ for $z \in Z$ is a nonterminal game history. Let $u_i(z)$ denote the utility to player i of terminal history z. Define the *counterfactual value* at nonterminal history h as:

$$v_i(\sigma, h) = \sum_{z \in Z, h \sqsubset z} \pi_{-i}^\sigma(h) \pi^\sigma(h, z) u_i(z). \tag{1}$$

The *counterfactual regret* of not taking action a at history h is defined as:

$$r(h, a) = v_i(\sigma_{I \to a}, h) - v_i(\sigma, h). \tag{2}$$

The *counterfactual regret* of not taking action a at information set I is then:

$$r(I, a) = \sum_{h \in I} r(h, a) \tag{3}$$

Let $r_i^t(I, a)$ refer to the regret when players use σ^t of not taking action a at information set I belonging to player i. The cumulative counterfactual regret is defined as:

$$R_i^T(I, a) = \sum_{t=1}^{T} r_i^t(I, a) \tag{4}$$

The difference between the value of always choosing action a and the expected value when the players use σ is an action's regret, which is then weighted by the probability that other player(s) (including chance) will play to reach the node. If we define the nonnegative counterfactual regret $R_i^{T,+}(I,a) = \max(R_i^T(I,a), 0)$, then we apply Hart and Mas-Colell's regret-matching from Section 2.3 to the cumulative regrets to obtain the new strategy:

$$\sigma_i^{T+1}(I,a) = \begin{cases} \frac{R_i^{T,+}(I,a)}{\sum_{a \in A(I)} R_i^{T,+}(I,a)} & \text{if } \sum_{a \in A(I)} R_i^{T,+}(I,a) > 0 \\ \frac{1}{|A(I)|} & \text{otherwise.} \end{cases} \tag{5}$$

For each information set, this equation is used to compute action probabilities in proportion to the positive cumulative regrets. For each action, CFR then produces the next state in the game, and computes utilities of each actions through recursively. Regrets are computed from the returned values, and the value of playing to the current node is finally computed and returned.

The CFR algorithm with chance-sampling is presented in detail in Algorithm 1. The parameters to CFR are the history of actions, the learning player, the time step, and the reach probabilities for players 1 and 2, respectively. Variables beginning with v are for local computation and are not computed according to the previous equations for counterfactual value. In line 9, $\sigma_c(h,a)$ refers to the probability distribution of the outcomes at the chance node h. In lines 16, 18, and 23, $P(h)$ is the active player after history h. In lines 10, 17, and 19, ha denotes history h with appended action a. In line 25, π_{-i} refers to the counterfactual reach probability of the node, which in the case of players $\{1, 2\}$ is the same as reach probability π_{3-i}. In line 35, $\emptyset$ refers to the empty history.

The average strategy profile at information set I, $\bar{\sigma}^T$, approaches an equilibrium as $T \to \infty$. The average strategy at information set I, $\bar{\sigma}^T(I)$, is obtained by normalizing s_I over all actions $a \in A(I)$. What is most often misunderstood about CFR is that this *average* strategy profile, and not the final strategy profile, is what converges to a Nash equilibrium [18].

3.4   Worked Example: Kuhn Poker

We begin our application of counterfactual regret minimization (CFR) to Kuhn Poker with a few definitions. We let our 2 actions, `PASS` and `BET` correspond to 0 and 1 respectively. A pseudorandom number generator is defined for Monte Carlo training. We store our information sets in a TreeMap called `nodeMap`, indexed by String representations of all information of the information set[2].

⟨*Kuhn Poker definitions*⟩≡
```
public static final int PASS = 0, BET = 1, NUM_ACTIONS = 2;
public static final Random random = new Random();
public TreeMap<String, Node> nodeMap = new TreeMap<String, Node>();
```

Each information set is represented by an inner class `Node`. Each node has fields corresponding to the regret and strategy variable definitions of `RPSTrainer` with an additional field `infoSet` containing the string representation of the information set:

⟨*Kuhn node definitions*⟩≡
```
String infoSet;
double[] regretSum = new double[NUM_ACTIONS],
         strategy = new double[NUM_ACTIONS],
         strategySum = new double[NUM_ACTIONS];
```

---

[2](This is not the most efficient means of storage and retrieval of information sets, of course. The purpose here, however, is to clarify the core algorithm rather than optimize its application.)

Algorithm 1 Counterfactual Regret Minimization (with chance sampling)

1: Initialize cumulative regret tables: $\forall I, r_I[a] \leftarrow 0$.
2: Initialize cumulative strategy tables: $\forall I, s_I[a] \leftarrow 0$.
3: Initialize initial profile: $\sigma^1(I, a) \leftarrow 1/|A(I)|$
4:
5: **function** CFR(h, i, t, π_1, π_2):
6: **if** h is terminal **then**
7:     **return** $u_i(h)$
8: **else if** h is a chance node **then**
9:     Sample a single outcome $a \sim \sigma_c(h, a)$
10:     **return** CFR(ha, i, t, π_1, π_2)
11: **end if**
12: Let I be the information set containing h.
13: $v_\sigma \leftarrow 0$
14: $v_{\sigma_{I \rightarrow a}}[a] \leftarrow 0$ for all $a \in A(I)$
15: **for** $a \in A(I)$ **do**
16:     **if** $P(h) = 1$ **then**
17:         $v_{\sigma_{I \rightarrow a}}[a] \leftarrow$ CFR(ha, i, t, $\sigma^t(I, a) \cdot \pi_1$, π_2)
18:     **else if** $P(h) = 2$ **then**
19:         $v_{\sigma_{I \rightarrow a}}[a] \leftarrow$ CFR(ha, i, t, π_1, $\sigma^t(I, a) \cdot \pi_2$)
20:     **end if**
21:     $v_\sigma \leftarrow v_\sigma + \sigma^t(I, a) \cdot v_{\sigma_{I \rightarrow a}}[a]$
22: **end for**
23: **if** $P(h) = i$ **then**
24:     **for** $a \in A(I)$ **do**
25:         $r_I[a] \leftarrow r_I[a] + \pi_{-i} \cdot (v_{\sigma_{I \rightarrow a}}[a] - v_\sigma)$
26:         $s_I[a] \leftarrow s_I[a] + \pi_i \cdot \sigma^t(I, a)$
27:     **end for**
28:     $\sigma^{t+1}(I) \leftarrow$ regret-matching values computed using Equation 5 and regret table r_I
29: **end if**
30: **return** v_σ
31:
32: **function** Solve():
33: **for** $t = \{1, 2, 3, \ldots, T\}$ **do**
34:     **for** $i \in \{1, 2\}$ **do**
35:         CFR($\emptyset$, i, t, 1, 1)
36:     **end for**
37: **end for**

Each node also has `getStrategy` and `getAverageStrategy` method just like those of `RPSTrainer`. The following function corresponds to line 28 in Algorithm 1:

⟨*Get current information set mixed strategy through regret-matching*⟩≡
```
private double[] getStrategy(double realizationWeight) {
    double normalizingSum = 0;
    for (int a = 0; a < NUM_ACTIONS; a++) {
        strategy[a] = regretSum[a] > 0 ? regretSum[a] : 0;
        normalizingSum += strategy[a];
    }
    for (int a = 0; a < NUM_ACTIONS; a++) {
        if (normalizingSum > 0)
            strategy[a] /= normalizingSum;
        else
            strategy[a] = 1.0 / NUM_ACTIONS;
        strategySum[a] += realizationWeight * strategy[a];
    }
    return strategy;
}
```

⟨*Get average information set mixed strategy across all training iterations*⟩≡
```
public double[] getAverageStrategy() {
    double[] avgStrategy = new double[NUM_ACTIONS];
    double normalizingSum = 0;
    for (int a = 0; a < NUM_ACTIONS; a++)
        normalizingSum += strategySum[a];
    for (int a = 0; a < NUM_ACTIONS; a++)
        if (normalizingSum > 0)
            avgStrategy[a] = strategySum[a] / normalizingSum;
        else
            avgStrategy[a] = 1.0 / NUM_ACTIONS;
    return avgStrategy;
}
```

Finally, we define the String representation of the information set node as the String representation of the information set followed by the current average node strategy:

⟨*Get information set string representation*⟩≡
```
public String toString() {
        return String.format("%4s: %s", infoSet, Arrays.toString(getAverageStrategy()));
}
```

Putting these together, we thus define the inner `Node` class of our CFR training code whose objects refer to the information sets I from Algorithm 1:

⟨*Information set node class definition*⟩≡

```
class Node {
    ⟨Kuhn node definitions⟩
    ⟨Get current information set mixed strategy through regret-matching⟩
    ⟨Get average information set mixed strategy across all training iterations⟩
    ⟨Get information set string representation⟩
}
```

To train an equilibrium for Kuhn Poker, we first create an integer array containing the cards. We implicitly treat the card at index 0 and 1 as the cards dealt to players 1 and 2, respectively. So at the beginning of each of a given number of training iterations, we simply shuffle these values, which are implicitly "dealt" or not to the players according to their array positions.

After shuffling, we make the initial call to the recursive CFR algorithm with the shuffled cards, an empty action history, and a probability of 1 for each player. (These probabilities are probabilities of player *actions*, rather than the probability of the chance event of receiving the cards dealt.) This function effectively implements the Solve() procedure defined from line 32 in Algorithm 1, with one notable exception below:

⟨*Train Kuhn poker*⟩≡

```
public void train(int iterations) {
    int[] cards = {1, 2, 3};
    double util = 0;
    for (int i = 0; i < iterations; i++) {
        ⟨Shuffle cards⟩
        util += cfr(cards, "", 1, 1);
    }
    System.out.println("Average game value: " + util / iterations);
    for (Node n : nodeMap.values())
        System.out.println(n);
}
```

Note in particular that cards are shuffled before the call to `cfr`. Instead of handling chance events during the recursive calls to CFR, the chance node outcomes can be pre-sampled. Often this is easier and more straight forward, so the shuffling of the cards replaces the if condition on lines 8 to 10. This form of Monte Carlo style sampling is called "chance-sampling", though it is interesting to note that CFR can be implemented without sampling at all ("Vanilla CFR") or with many different forms of sampling schemes [9]. We will assume for the rest of this document that when we use CFR, we specifically refer to chance-sampled CFR. Cards are shuffled according to the Durstenfeld version of the Fisher-Yates shuffle[3] as popularized by Donald Knuth:

⟨*Shuffle cards*⟩≡

```
for (int c1 = cards.length - 1; c1 > 0; c1--) {
    int c2 = random.nextInt(c1 + 1);
    int tmp = cards[c1];
    cards[c1] = cards[c2];
    cards[c2] = tmp;
}
```

---

[3]See URL `http://en.wikipedia.org/wiki/Fisher-Yates_shuffle#The_modern_algorithm`.

The recursive CFR method begins by computing the player and opponent numbers from the history length. As previously mentioned, the zero-based card array holds cards for player 1 and 2 at index 0 and 1, respectively, so we internally represent these players as player 0 and player 1.

We next check if the current state is a terminal state (where the game has ended), as on line 6 of Algorithm 1, and return the appropriate utility for the current player. If it is not a terminal state, execution continues, computing the information set string representation by concatenating the current player card with the history of player actions, a string of "p" and "b" characters for "pass" and "bet", respectively. This String representation is used to retrieve the information set node, or create it if it is nonexistant.

The node strategy is computed through regret-matching as before. For each action, `cfr` is recursively called with additional history and updated probabilities (according the to node strategy), returning utilities for each action. From the utilities, counterfactual regrets are computed and used to update cumulative counterfactual regrets. Finally, the expected node utility is returned.

⟨*Counterfactual regret minimization iteration*⟩≡

```
private double cfr(int[] cards, String history, double p0, double p1) {
    int plays = history.length();
    int player = plays % 2;
    int opponent = 1 - player;
    ⟨Return payoff for terminal states⟩
    String infoSet = cards[player] + history;
    ⟨Get information set node or create it if nonexistant⟩
    ⟨For each action, recursively call cfr with additional history and probability⟩
    ⟨For each action, compute and accumulate counterfactual regret⟩
    return nodeUtil;
}
```

In discerning a terminal state, we first check to see of both players have had at least one action. Given that, we check for the two conditions for a terminal state: a terminal pass after the first action, or a double bet. If there's a terminal pass, then a double terminal pass awards a chip to the player with the higher card. Otherwise, it's a single pass after a bet and the player betting wins a chip. If it's not a terminal pass, but a two consecutive bets have occurred, the player with the higher card gets two chips. Otherwise, the state isn't terminal and computation continues:

⟨*Return payoff for terminal states*⟩≡

```
if (plays > 1) {
    boolean terminalPass = history.charAt(plays - 1) == 'p';
    boolean doubleBet = history.substring(plays - 2, plays).equals("bb");
    boolean isPlayerCardHigher = cards[player] > cards[opponent];
    if (terminalPass)
        if (history.equals("pp"))
            return isPlayerCardHigher ? 1 : -1;
        else
            return 1;
    else if (doubleBet)
        return isPlayerCardHigher ? 2 : -2;
}
```

Not being in a terminate state, we retrieve the node associated with the information set, or create such a node if nonexistant, corresponding to line 12 of Algorithm 1:

⟨*Get information set node or create it if nonexistant*⟩≡
```
Node node = nodeMap.get(infoSet);
if (node == null) {
    node = new Node();
    node.infoSet = infoSet;
    nodeMap.put(infoSet, node);
}
```

Next, we compute the node strategy and prepare space for recursively-computed action utilities. For each action, we append the symbol ("p" or "b") for the action to the action history, and make a recursive call with this augmented history and an update to the current player's probability of playing to that information set in the current training iteration. Each action probability multiplied by the corresponding returned action utility is accumulated to the utility for playing to this node for the current player.

⟨*For each action, recursively call cfr with additional history and probability*⟩≡
```
double[] strategy = node.getStrategy(player == 0 ? p0 : p1);
double[] util = new double[NUM_ACTIONS];
double nodeUtil = 0;
for (int a = 0; a < NUM_ACTIONS; a++) {
    String nextHistory = history + (a == 0 ? "p" : "b");
    util[a] = player == 0
        ? - cfr(cards, nextHistory, p0 * strategy[a], p1)
        : - cfr(cards, nextHistory, p0, p1 * strategy[a]);
    nodeUtil += strategy[a] * util[a];
}
```

Finally, the recursive CFR call concludes with computation of regrets. However, these are not simply accumulated. Cumulative regrets are cumulative *counterfactual* regrets, weighted by the probability that the *opponent* plays to the current information set, as in line 25 of Algorithm 1:

⟨*For each action, compute and accumulate counterfactual regret*⟩≡
```
for (int a = 0; a < NUM_ACTIONS; a++) {
    double regret = util[a] - nodeUtil;
    node.regretSum[a] += (player == 0 ? p1 : p0) * regret;
}
```

CFR training is initialized by creating a new trainer object and initiating training for a given number of iterations. Bear in mind that, as in all applications of Monte Carlo, more iterations lead to closer convergence.

⟨*KuhnTrainer main method*⟩≡
```
public static void main(String[] args) {
    int iterations = 1000000;
    new KuhnTrainer().train(iterations);
}
```

Putting all of these elements together, we thus create a counterfactual regret minimization (CFR) trainer for Kuhn Poker:

⟨*KuhnTrainer.java*⟩≡

```
import java.util.Arrays;
import java.util.Random;
import java.util.TreeMap;

public class KuhnTrainer {
    ⟨Kuhn Poker definitions⟩
    ⟨Information set node class definition⟩
    ⟨Train Kuhn poker⟩
    ⟨Counterfactual regret minimization iteration⟩
    ⟨KuhnTrainer main method⟩
}
```

Food for thought: What values are printed when this program is run? What do they mean? Do you see an opportunity to prune sub-trees for which traversal is provably wasteful? (Hint: What is/are the important operation(s) applied at each information set and under what conditions would these be rendered useless?)

More food (seconds?) for thought: if a subtree would never be visited by an optimal player, is there any reason to compute play for it?

3.5 Exercise: 1-Die-Versus-1-Die Dudo

Dudo is a bluffing dice game thought to originate from the Inca Empire circa 15$^{\text{th}}$ century. Many variations exist in both folk and commercial forms. The ruleset we use from [7] is perhaps the simplest representative form, and is thus most easily accessible to both players and researchers. Liar's Dice, Bluff, Call My Bluff, Perudo, Cacho, Cachito are names of variations[4].

Dudo has been a popular game through the centuries. From the Inca Empire, Dudo spread to a number of Latin American countries, and is thought to have come to Europe via Spanish conquistadors [14]. It is said to have been "big in London in the 18$^{\text{th}}$ century" [4]. Richard Borg's commercial variant, published under the names Call My Bluff, Bluff, and Liar's Dice, won the prestigious Spiel des Jahres (German Game of the Year) in 1993. On BoardGameGeek.com[5], the largest website for board game enthusiasts, Liar's Dice is ranked 270/53298 (i.e. top 0.5%)[6]. Although a single, standard form of the game has not emerged, there is strong evidence of the persistence of the core game mechanics of this favorite bluffing dice game since its creation.

Perudo, a commercial production of the folk game Dudo

3.5.1 Rules:

Each player is seated around a table and begins with five standard six-sided dice and a dice cup. Dice are lost with the play of each round, and the object of the game is to be the last player remaining with dice. At the beginning of each round, all players simultaneously roll their dice once, and carefully view their rolled dice while keeping them concealed from other players. The starting player makes a claim about what the players have *collectively* rolled, and players clockwise in turn each either make

---

[4]In some cases, e.g. Liar's Dice and Cacho, there are different games of the same name.
[5]http://www.boardgamegeek.com
[6]as of August 17$^{\text{th}}$, 2011

a stronger claim or challenge the previous claim, declaring "Dudo" (Spanish for "I doubt it."). A challenge ends the round, players lift their cups, and one of the two players involved in the challenge loses dice. Lost dice are placed in full view of players.

Claims consist of a positive number of dice and a rank of those dice, e.g. two 5's, seven 3's, or two 1's. In Dudo, the rank of 1 is *wild*, meaning that dice rolls of rank 1 are counted in totals for other ranks as well. We will denote a claim of n dice of rank r as $n \times r$. In general, one claim is stronger than another claim if there is an increase in rank and/or number of dice. That is, a claim of 2×4 may, for example, be followed by 2×6 (increase in rank) or 4×3 (increase in number). The exception to this general rule concerns claims of wild rank 1. Since 1's count for other ranks and other ranks do not count for 1's, 1's as a rank occur with half frequency in counts and are thus considered doubly strong in claims. So in the claim ordering, 1×1, 2×1, and 3×1 immediately precede 2×2, 4×2, and 6×2, respectively.

Mathematically, one may enumerate the claims in order of strength by defining $s(n, r)$, the strength of claim $n \times r$, as follows:

$$s(n, r) = \begin{cases} 5n - \lfloor \frac{n}{2} \rfloor - r - 7 & \text{if } r \neq 1 \\ 11n - 6 & \text{if } r = 1 \text{ and } r \leq \lfloor \frac{d_{\text{total}}}{2} \rfloor \\ 5d_{\text{total}} + n - 1 & \text{if } r = 1 \text{ and } r > \lfloor \frac{d_{\text{total}}}{2} \rfloor \end{cases} \tag{6}$$

where d_{total} is the total number of dice in play. Thus for 2 players with 1 die each, the claims would be numbered:

Strength $s(n, r)$	0	1	2	3	4	5	6	7	8	9	10	11
Claim $n \times r$	1×2	1×3	1×4	1×5	1×6	1×1	2×2	2×3	2×4	2×5	2×6	2×1

Play proceeds clockwise from the round-starting player with claims of strictly increasing strength until one player challenges the previous claimant with "Dudo". At this point, all cups are lifted, dice of the claimed rank (including wilds) are counted and compared against the claim. For example, suppose that Ann, Bob and Cal are playing Dudo, and Cal challenges Bob's claim of 7×6. There are three possible outcomes:

- **The actual rank count exceeds the challenged claim.** In this case, the challenger loses a number of dice equal to the difference between the actual rank count and the claim count. Example: Counting 6's and 1's, the actual count is 10. Thus, as an incorrect challenger, Cal loses $10 - 7 = 3$ dice.

- **The actual rank count is less than the challenged claim.** In this case, the challenged player loses a number of dice equal to the difference between the claim count and the actual rank count. Example: Counting 6's and 1's, the actual count is 5. Thus, as a correctly challenged claimant, Bob loses $7 - 5 = 2$ dice.

- **The actual rank count is equal to the challenged claim.** In this case, every player except the challenged player loses a single die. Example: Counting 6's and 1's, the actual count is indeed 7 as Bob claimed. In this special case, Ann and Cal lose 1 die each to reward Bob's exact claim.

In the first round, an arbitrary player makes the first claim. The winner of a challenge makes the first claim of the subsequent round. When a player loses all remaining dice, the player loses and exits the game. The last remaining player is the winner. The following table provides a transcript of an example 2-player game with "1:" and "2:" indicating information relevant to each player:

Round	Actions	Revealed Rolls		Result
1	1:"2×6", 2:"3×6", 1:"5×6", 2:"Dudo"	1:12566	2:23556	1:loses 1 die
2	2:"3×6", 1:"4×5", 2:"5×5", 1:"Dudo"	1:3555	2:23455	1:loses 1 die
3	2:"3×6", 1:"Dudo"	1:356	2:24466	1:loses 1 die
4	2:"2×2", 1:"3×2", 2:"Dudo"	1:12	2:13456	2:loses 1 die
5	1:"2×6", 2:"3×2", 2:"Dudo"	1:26	2:1222	1:loses 2 dice

Exercise

Create a counterfactual regret minimization program `DudoTrainer`, similar to `KuhnTrainer`, that performs counterfactual regret minimization for *1-die-versus-1-die endgame play of Dudo*, computing and printing (1) the average utility for player 1 across all training iterations, and (2) all information sets and associated strategies (with actions ordered by increasing claim strength).

As a test of your code, compute the average player 1 utility across all training runs. Optimal play yields a game value of $-\frac{7}{258} \approx -0.027$.

We supply the following optional code to aid in simplifying representational choice. Let us begin with the assumption that we are limited to two 6-sided dice, yielding 13 possible actions: 12 claims of 1 or 2 of 6 different ranks, plus the doubting "dudo" action. Let us index claims in increasing order of strength starting at 0, and let the "dudo" action have index 12.

We begin with the definition of helpful relevant constants. Note that for any claim number, one can look up the corresponding number and rank of the claim using integer arrays `claimNum` and `claimRank`, respectively.

⟨*Dudo definitions*⟩≡
```
  public static final int NUM_SIDES = 6, NUM_ACTIONS = (2 * NUM_SIDES) + 1,
                          DUDO = NUM_ACTIONS - 1;
  public static final int[] claimNum = {1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2};
  public static final int[] claimRank = {2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 1};
```

Next, we suggest that, for the purposes of this exercise, one simply represents the claim history as a boolean array of length NUM_ACTIONS with a value at index i being true if and only if the corresponding claim/dudo action has been taken. One can convert this to a readable string using the following code:

⟨*Convert Dudo claim history to a String*⟩≡
```
  public static String claimHistoryToString(boolean[] isClaimed) {
      StringBuilder sb = new StringBuilder();
      for (int a = 0; a < NUM_ACTIONS; a++)
          if (isClaimed[a]) {
              if (sb.length() > 0)
                  sb.append(',');
              sb.append(claimNum[a]);
              sb.append('*');
              sb.append(claimRank[a]);
          }
      return sb.toString();
  }
```

Further, if one also supplies the roll of the current player, one can conveniently convert the information set to a unique integer using the following code:

⟨*Convert Dudo information set to an integer*⟩≡

```java
public static int infoSetToInteger(int playerRoll, boolean[] isClaimed) {
    int infoSetNum = playerRoll;
    for (int a = NUM_ACTIONS - 2; a >= 0; a--)
        infoSetNum = 2 * infoSetNum + (isClaimed[a] ? 1 : 0);
    return infoSetNum;
}
```

For simplicity, we recommend retrieving nodes from a `TreeMap` that maps such information set numbers to the information set nodes. The claim history strings are better used in defining the node string representation.

Putting these all together, we offer DudoUtilities.java for your use:

⟨*DudoUtilities.java*⟩≡

```java
public class DudoUtilities {
    ⟨Dudo definitions⟩
    ⟨Convert Dudo claim history to a String⟩
    ⟨Convert Dudo information set to an integer⟩
}
```

4   Interlude: Thresholding Actions and Resetting Strategy Sums

In this section, we break from the normal presentation to discuss means of "cleaning" our results such that they, in most cases, more closely approximate the desired equilibria. Initially, every action has a uniform probability, so there will always be at least some probability of an action in the average strategy. Further, it may take some time for cumulative counterfactual regret to converge, so we often have a small probability of taking any action, even those which are obviously mistaken.

One way to "clean" or *purify* the strategy is to set an action probability *threshold* (e.g. 0.001), and zero the probability of any action that has a probability beneath this threshold. After such thresholding, one must be sure to renormalize all action probabilities of the strategy. Observe the results of applying this thresholding to our worked examples and exercises. In the worst case, this purification and thresholding may remove the guarantee of convergence to an equilibrium, but under certain conditions the payoff when playing a purified strategy against an opponent using an equilibrium strategy can be the same as when using an equilibrium strategy as well [2]. Also, the authors showed that in practice it can increase playing performance in small and large games of Poker.

Another means of getting a better approximation of the equilibrium strategy is to simply not include the strategies of early iterations. For example, one could reset all strategy sums to zero after a given number or fraction of early iterations. Observe the results of resetting strategy sums to out worked examples and exercises.

5   Fixed-Strategy Iteration CFR Minimization

In this section, we discuss computational complexity of CFR, and problem characteristics that allow a dynamic programming structure to offer significant improvement over recursive structure of CFR. We present a dynamic programming algorithm for CFR, and work through a novel example problem. Finally, we present an exercise where one applies the algorithm to 1-die-versus-1-die Dudo with a memory limited to up to three prior claims.

m	Information Sets
1	57626
2	2069336
3	21828536
4	380033636
5	2751474854
all	2.9×10^{20}

(a) No. of abstracted info. sets with varying memory limit m.

Dice	Opponent Dice				
	1	2	3	4	5
1	1794	5928	13950	27156	43056
2	20748	48825	95046	163947	241962
3	130200	253456	437192	693504	971264
4	570276	983682	1560384	2327598	3132108
5	159012	217224	284508	360864	9084852

(b) Information sets for each round with memory limit $m = 3$.

Figure 1: Number of abstracted information sets varying recall limit and number of dice in round.

5.1   Motivation

5.1.1   Imperfect Recall of Actions

The full 5-dice-versus-5-dice, 2-player game of Dudo consists of over 2.9 quintillion information sets. For common modern machines, 2.9×10^{20} information sets is too large to iterate over for convergence of mixed strategies. As with successful computational approaches to Texas Hold'em Poker, we can abstract information sets in order to reduce the problem size. We can then solve the abstraction and apply the abstracted policy to the original game. For Dudo, the growth rate of possible claim sequences is most responsible for the overall growth of the extensive game-tree. We also note that the later claims of a round are less-easily supported and thus more often contain reliable information.

Since more recent claims tend to be more important to the decision at hand, our chosen means of abstraction is to form abstract information sets that recall up to m previous claims. For example, consider this 5-vs.-5 round claim sequence: 1×5, 2×5, 4×2, 5×4, 3×1, 6×4, 7×2. For a claim memory limit of $m = 3$, the 5-vs.-5 round information set for each of these decision would be enumerated according to the current player dice roll enumeration and the enumeration of the up-to-3 most recent claims: $\{\}, \{1 \times 5\}, \{1 \times 5; 2 \times 5\}, \{1 \times 5; 2 \times 5; 4 \times 2\}, \ldots, \{3 \times 1; 6 \times 4; 7 \times 2\}$.

Imperfect recall of actions allows us to trade off fine distinction of judgment for computational space and time requirements. Figure 1 shows how the number of abstract information sets varies according to the memory limit m and the number of dice for each player. There are safe abstraction techniques to transform a game in this way while conserving the guarantee of convergence to an equilibrium in CFR [10], but unfortunately the abstraction above does not satisfy the required conditions. Therefore, convergence to a Nash equilibrium may no longer be possible. However, experiments with Dudo, Poker, and other games have shown that one can apply imperfect recall of actions with only a minor penalty (compared to the memory savings) in convergence rate and play performance [10, 16, 15].

We conclude this section by highlighting an important and interesting consequence of abstracting with imperfect recall of actions: *For any given training example, one can have different paths to the same abstracted information set.* Thus, recursive CFR calls revisit the same information set within the same training iteration. For example, suppose we play two-player Dudo with a claim memory limit of $m = 3$. The following two claim sequences

$$s_1 : \quad 1 \times 2, \quad 1 \times 3, \quad 1 \times 5, \quad 1 \times 6, \quad 2 \times 2$$
$$s_2 : \qquad\qquad\qquad\qquad\quad 1 \times 5, \quad 1 \times 6, \quad 2 \times 2$$

will be part of the same information set since the $1 \times 2, 1 \times 3$ is forgotten. Furthermore, assuming the same roll outcomes in both cases, the expected utilities from this point on given a strategy profile used by both players will be the the same in both cases, since the only thing that matters for determining the payoff is the most recent claim of 2×1. CFR would enumerate both sequences. FSICFR avoids the redundant computation by restructuring the tree traversal and computation of counterfactual regrets.

5.1.2   Liar Die

In this section, we consider the rules of another sequential game that also allows different histories that naturally lead to the same abstracted information sets. The game of Liar Die is a creation of the author that is a simplification of bluffing dice games such as Mäxchen (a.k.a. Little Max) and Liar Dice[7] (a.k.a. Schummeln) [7].

A player begins by rolling an s-sided die secretly (e.g. behind a hand/barrier, under a cup, etc.), observing the roll, and claiming a roll rank. The opponent then has two possible actions:

- **Doubt:** The opponent doubts the claim, the roll is revealed and compared against the rank claimed. If the roll rank is greater than or equal to the claim rank, the player wins. Otherwise, the opponent wins.

- **Accept:** The opponent accepts the claim. Without revealing the prior roll rank, the die is given to the opponent, who now takes on the same role as the initial player with one exception. After the die is again secretly rolled and observed, *one must make a claim that is higher than the previous claim.* Thus, players will take turns secretly rolling the die and making successively higher claims until one player doubts their opponent's claim, and the roll is checked to see who wins the challenge.

Note that the only relevant information to the doubt/accept decision is the past two claims. No claim prior to those have any bearing on the current decision, as the die has been rerolled and is without memory. All a player needs to know is the current claim and the constraints on that claim. Note also that the only relevant information to a new claim decision is the roll the claim concerns and the constraints on the claim, i.e. the previous claim if existent.

We can then think of there being two types of information set nodes: response nodes and claim nodes. Response nodes may be indexed by the two most recent claims (using 0 as a null value if there was only one claim). Response nodes have two actions: doubt or accept. Claim nodes may be indexed by the prior claim (constraining the next claim) and the roll. Claim nodes have a number of actions equal to the number of sides of the die minus the prior claim rank. Our information sets are thus abstracted by removal of prior information that has become irrelevant to future game decisions.

We conclude this section by highlighting that we again are presented with a game where, for any given sequence of rolls for a training example, one can have different paths to the same abstracted information set. Again, recursive CFR calls revisit the same information set within the same training iteration.

5.1.3   Structural Motivation

In general, suppose we can fix all chance events of a training iteration in advance. For finite games, this then induces a directed acyclic graph on all possible paths through the information sets. Consider Figure 2, a simple directed acyclic graph (DAG). With increasing depth, the number of possible paths to a node grows exponentially. If we continue the DAG pattern to greater depth, we have a linear growth of nodes and exponential growth of paths.

This exponential growth of paths occurs in both Dudo and Liar Die if we apply recursive CFR. With each CFR visit to an information set node, we perform a regret update, which can impact the possibly mixed strategy of the next CFR visit to the node. On the surface, it seems that we are constrained to make these exponentially growing visits to the same node. Further, unless we deliberately introduce randomization to the ordering of action selection, the ordering of the histories of these visits may create interesting ebbs and flows to accumulated counterfactual regrets.

Most importantly, we would like some means of training that would avoid exponentially growing visits to nodes, while retaining the benefits of counterfactual regret minimization.

---

[7]Liar Dice is not to be confused with Liar's Dice, another name for Dudo.

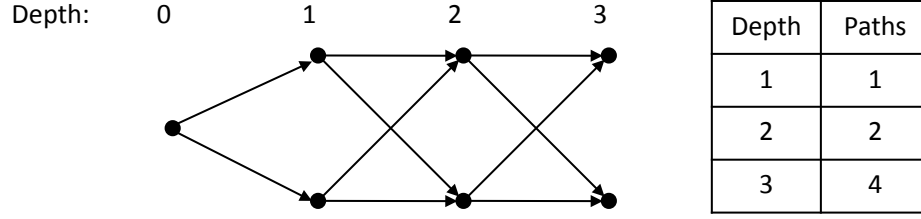| Depth	Paths
1	1
2	2
3	4

Figure 2: Example directed acyclic graph.

5.2 Algorithm

With such games, it is indeed possible to perform counterfactual regret minimization with a single forward and backward visit to each node in a topologically-sorted directed acyclic graph. Recursive CFR is restructured as a dynamic programming algorithm where cumulative counterfactual regrets and thus strategies are updated once per training iteration. Since the node strategies remain fixed throughout the forward propagation of player reach/realization probability information, we refer to the algorithm as Fixed-Strategy Iteration Counterfactual Regret Minimization (FSICFR). Like CFR, it relies on the regret-matching equation (5) for convergence to a Nash equilibrium.

Essentially, CFR traverses extensive game subtrees, recursing forward with reach probabilities that each player will play to each node (i.e. information set) while maintaining history, and backpropagating utilities used to update parent node action regrets and thus future strategy.

Fixed-Strategy Iteration CFR (FSICFR) [15] divides the recursive CFR algorithm into two iterative passes, one forward and one backward, through a DAG of nodes. On the forward pass, visit counts and reach probabilities of each player are accumulated, yet all strategies remain fixed. (By contrast, in CFR, the strategy at a node is updated with each CFR visit.) After all visits are counted and probabilities are accumulated, a backward pass computes utilities and updates regrets. FSICFR computational time complexity is proportional to the number of *nodes* times the average node outdegree, whereas CFR complexity is proportional to the number of *node visits* times the average node outdegree. Since the number of node visits grows exponentially in such abstracted problems, FSICFR has exponential savings in computational time per training iteration relative to CFR.

We now present the FSICFR algorithm for two-player, zero-sum games in detail as Algorithm 2. Each player node n consists of a number of fields:

visits - the sum of possible paths to this node during a single training iteration.

$pSum_i$ - the sum of the probabilities that player i would play to this node along each possible path during a single training iteration.

r - a mapping from each possible action to the average counterfactual regret for not choosing that action.

σ - a node strategy, i.e. a probability distribution function mapping each possible action to the probability of choosing that action.

σSum - the sum of the strategies used over each training iteration the node is visited.

player - the player currently taking action.

T - the sum of *visits* across all training iterations.

v - the expected game value for the current player at that node.

Algorithm 2 FSICFR(L)

Require: A topologically sorted list L of extensive game DAG nodes.
Ensure: The normalized strategy sum $n.\sigma Sum$, i.e. the average strategy for each player node n, approximates a Nash equilibrium.

1: **for** each training iteration **do**
2:     Predetermine the chance outcomes at all reachable chance nodes in L.
3:     **for** each node n in order of topologically-sorted, reachable subset $L' \subset L$ **do**
4:         **if** n is the initial node **then**
5:             $n.visits \leftarrow n.pSum_1 \leftarrow n.pSum_2 \leftarrow 1$
6:         **end if**
7:         **if** n is a player node **then**
8:             Compute strategy $n.\sigma$ according to Equation 5.
9:             **if** $n.r$ has no positive components **then**
10:                Let $n.\sigma$ be the uniform distribution.
11:             **end if**
12:             $n.\sigma Sum \leftarrow n.\sigma Sum + (n.\sigma \cdot n.pSum_1$ **if** $n.player = 1$ **else** $n.\sigma \cdot n.pSum_2)$
13:             **for** each action a **do**
14:                Let c be the associated child of taking action a in n with probability $n.\sigma(a)$.
15:                $c.visits \leftarrow c.visits + n.visits$
16:                $c.pSum_1 \leftarrow c.pSum_1 + (n.\sigma(a) \cdot n.pSum_1$ **if** $n.player = 1$ **else** $n.pSum_1)$
17:                $c.pSum_2 \leftarrow c.pSum_2 + (n.pSum_2$ **if** $n.player = 1$ **else** $n.\sigma(a) \cdot n.pSum_2)$
18:             **end for**
19:         **else if** n is a chance node **then**
20:             Let c be the associated child of the predetermined chance outcome for n.
21:             $c.visits \leftarrow c.visits + n.visits$
22:             $c.pSum_1 \leftarrow c.pSum_1 + n.pSum_1$
23:             $c.pSum_2 \leftarrow c.pSum_2 + n.pSum_2$
24:          **end if**
25:     **end for**
26:     **for** each node n in reverse order of topologically-sorted, reachable subset $L' \subset L$ **do**
27:         **if** n is a player node **then**
28:             $n.v \leftarrow 0$
29:             **for** each action a **do**
30:                Let c be the associated child of taking action a in n with probability $n.\sigma(a)$.
31:                $n.v(a) \leftarrow (c.v$ **if** $n.player = c.player$ **else** $-c.v)$
32:                $n.v \leftarrow n.v + n.\sigma(a) \cdot n.v(a)$
33:             **end for**
34:             Counterfactual probability $cfp \leftarrow (n.pSum_2$ **if** $n.player = 1$ **else** $n.pSum_1)$
35:             **for** each action a **do**
36:                $n.r(a) \leftarrow \frac{1}{n.T + n.visits}(n.T \cdot n.r(a) + n.visits \cdot cfp \cdot (n.v(a) - n.v))$
37:             **end for**
38:             $n.T \leftarrow n.T + n.visits$
39:         **else if** n is a chance node **then**
40:             Let $[n.player, n.v] \leftarrow [c.player, c.v]$, where c is the predetermined child of n.
41:         **else if** n is a terminal node **then**
42:             $n.v \leftarrow$ the utility of n for current player $n.player$.
43:         **end if**
44:         $n.visits \leftarrow n.pSum_1 \leftarrow n.pSum_2 \leftarrow 0$
45:     **end for**
46: **end for**

A significant, domain-specific assumption is required for this simple, zero-sum form of FSICFR: There is a one-to-one correspondence between player nodes and abstracted information sets, and our visits to nodes must contain enough state information (e.g. predetermined public and private information for both players) such that the appropriate successor nodes may be chosen. In the case of Dudo, this means that the algorithm, having predetermined player rolls, knows which player information sets are legal successors.

We note that the predetermination of chance node outcomes may present difficulties for some games or game abstractions where constraints on a chance node are dependent on the path by which it is reached (e.g. drawing the next card without knowing how many and thus which cards have been previously drawn). This does not pose a problem for Dudo or non-draw forms of Poker. Observe that such predetermination should proceed in topological order, as predetermination of chance nodes affects reachability of later chance nodes.

Also, we note that if, as in the case of Dudo, the number of visits to a node will be constant across all training iterations, then one can eliminate the *n.visits* variable and replace it with the value 1 in equations.

In summary, consider FSICFR as being similar to the case where we hold the strategy before a CFR training iteration fixed and execute an entire iteration of regret updates without changing strategy until after the iteration. In such a case, all operations at a node are the same, and we transform the exponential tree-recursion of CFR into a linear dynamic-programming graph-traversal for FSICFR. As a consequence, node update frequency is equalized rather than exponentially proportional to depth.

5.3   Worked Example: Liar Die

We now apply FSICFR to the game of Liar Die. We begin by defining constants for the actions `DOUBT` and `ACCEPT` and creating a pseudorandom number generator `random`. Each `LiarDieTrainer` will be constructed with a parameter for the number of `sides` of the die[8]. We will here focus on the computation of strategy with a standard 6-sided die, but the game is arguably more fun to play with a 20-sided die. Finally, we have two 2D arrays of `Node` objects representing abstracted information sets without irrelevant play history.

The `responseNodes` array is a 2D array of response nodes indexed by the claim before the current claim and the current claim. Claims are assumed to range from 1 through `sides`, so claim number 0 is used when a claim does not exist. For example, the response node for an initial claim of c would be at index `[0][c]`.

The `claimNodes` array is a 2D array of claim nodes indexed by the accepted prior claim and the current die roll the claim decision concerns. Again, claim number 0 is used when a claim does not exist. For example, the claim node for the initial claim of a game given a roll of r would be at index `[0][r]`.

⟨*Liar Die definitions*⟩≡
```
static final int DOUBT = 0, ACCEPT = 1;
static Random random = new Random();
int sides;
Node[][] responseNodes;
Node[][] claimNodes;
```

---

[8]Dice may be easily obtained with a variety of numbers of sides. Role-playing games commonly use a set of dice with 4, 6, 8, 10, 12, and 20 sides.

Each Liar Die information set node contains a number of familiar fields, but must also contain additional information to allow the dynamic programming computation:

⟨*Liar Die node definitions*⟩≡
```
public double[] regretSum, strategy, strategySum;
public double u, pPlayer, pOpponent;
```

In addition to the familiar `regretSum`, `strategy`, and `strategySum` fields, we now have 3 additional fields. The field `u` allows each node to hold its utility value in order to allow backpropagation of the utility to all predecessor nodes in the dynamic programming algorithm. In forward propagation of reach probabilities (i.e. realization weights), these are accumulated in the `pPlayer`, and `pOpponent` fields. These are not probabilities. Rather, they are a sum of probabilities. (They represent average probabilities if we divide them by the number of node visits.)

The node constructor takes the number of possible actions as a parameter and allocates field arrays accordingly:

⟨*Liar Die node constructor*⟩≡
```
public Node(int numActions) {
    regretSum = new double[numActions];
    strategy = new double[numActions];
    strategySum = new double[numActions];
}
```

The node methods `getStrategy` and `getAverageStrategy` are defined as before:

⟨*Get Liar Die node current mixed strategy through regret-matching*⟩≡
```
public double[] getStrategy() {
    double normalizingSum = 0;
    for (int a = 0; a < strategy.length; a++) {
        strategy[a] = Math.max(regretSum[a], 0);
        normalizingSum += strategy[a];
    }
    for (int a = 0; a < strategy.length; a++) {
        if (normalizingSum > 0)
            strategy[a] /= normalizingSum;
        else
            strategy[a] = 1.0/strategy.length;
    }
    for (int a = 0; a < strategy.length; a++)
        strategySum[a] += pPlayer * strategy[a];
    return strategy;
}
```

⟨*Get Liar Die node average mixed strategy*⟩≡
```
public double[] getAverageStrategy() {
    double normalizingSum = 0;
    for (int a = 0; a < strategySum.length; a++)
        normalizingSum += strategySum[a];
    for (int a = 0; a < strategySum.length; a++)
        if (normalizingSum > 0)
            strategySum[a] /= normalizingSum;
        else
            strategySum[a] = 1.0 / strategySum.length;
    return strategySum;
}
```

Thus, a Liar Die player decision `Node` is defined with the following components:

⟨*Liar Die player decision node*⟩≡
```
class Node {
    ⟨Liar Die node definitions⟩
    ⟨Liar Die node constructor⟩
    ⟨Get Liar Die node current mixed strategy through regret-matching⟩
    ⟨Get Liar Die node average mixed strategy⟩
}
```

The main task of the `LiarDieTrainer` constructor is to allocate these nodes. After storing the number of die sides in the trainer field, the response and claim node arrays are allocated and the nodes are constructed.

⟨*Construct trainer and allocate player decision nodes*⟩≡
```
public LiarDieTrainer(int sides) {
    this.sides = sides;
    responseNodes = new Node[sides][sides + 1];
    for (int myClaim = 0; myClaim <= sides; myClaim++)
        for (int oppClaim = myClaim + 1; oppClaim <= sides; oppClaim++)
            responseNodes[myClaim][oppClaim]
                = new Node((oppClaim == 0 || oppClaim == sides) ? 1 : 2);
    claimNodes = new Node[sides][sides + 1];
    for (int oppClaim = 0; oppClaim < sides; oppClaim++)
        for (int roll = 1; roll <= sides; roll++)
            claimNodes[oppClaim][roll] = new Node(sides - oppClaim);
}
```

When allocating response nodes, note that there is only one possible action when there is no prior claim (i.e. `ACCEPT`) and only one possible action when the opponent claims the highest rank equal to the number of sides (i.e. `DOUBT`). When allocating claim nodes, the number of legal claims remaining are the number of sides minus the previous opponent claim.

The training procedure itself will be presented top-down. A regret vector is allocated with a capacity equal to the largest number of actions for any node. In FSICFR, we topologically sort the information set nodes that can be visited in a training iteration, so we precompute our chance events. In the context of Liar Die, we precompute each roll that is made after the acceptance of each possible non-terminal claim number, so an array is allocated to store these rolls. Then, for the given number of iterations, we (1) initialize these rolls and starting probabilities, (2) accumulate realization weights forward, and (3) backpropagate resulting utilities, adjusting regrets (and thus strategies) once per node. For better results, we add an optional step that, when training is half done, resets all node strategy sums.

⟨*Train with FSICFR*⟩≡
```
public void train(int iterations) {
    double[] regret = new double[sides];
    int[] rollAfterAcceptingClaim = new int[sides];
    for (int iter = 0; iter < iterations; iter++) {
        ⟨Initialize rolls and starting probabilities⟩
        ⟨Accumulate realization weights forward⟩
        ⟨Backpropagate utilities, adjusting regrets and strategies⟩
        ⟨Reset strategy sums after half of training⟩
    }
    ⟨Print resulting strategy⟩
}
```

First, we precompute the possible rolls, and set the initial reach probabilities (i.e. realization weights) to 1:

⟨*Initialize rolls and starting probabilities*⟩≡
```
for (int i = 0; i < rollAfterAcceptingClaim.length; i++)
        rollAfterAcceptingClaim[i] = random.nextInt(sides) + 1;
claimNodes[0][rollAfterAcceptingClaim[0]].pPlayer = 1;
claimNodes[0][rollAfterAcceptingClaim[0]].pOpponent = 1;
```

Next, we need to visit each possible node *ordered according to the forward order of possible visits*. Our array structure allows us to do this easily, iterating through opponent claims, first visiting response nodes and then visiting claim nodes:

⟨*Accumulate realization weights forward*⟩≡
```
for (int oppClaim = 0; oppClaim <= sides; oppClaim++) {
    ⟨Visit response nodes forward⟩
    ⟨Visit claim nodes forward⟩
}
```

Visiting response nodes forward, we iterator through the previous possible claims the player could make prior to the opponent claim. The corresponding node is accessed and the strategy for that node is computed with counterfactual regret matching. If the opponent claim allows the player to increase the claim, i.e. the opponent claim is less than the number of sides, we use our precomputed roll to look up the claim node that would come next after an *ACCEPT* action. To this node, we *accumulate* our reach/realization probabiltiies. Note that, for FSICFR, we will have visited all possible predecessor nodes and accumulated all such probabilities before visiting a node. It is as if we are performing the recursive CFR algorithm, but holding all strategies fixed and later performing a single final regret computation and update per node.

⟨*Visit response nodes forward*⟩≡
```
if (oppClaim > 0)
    for (int myClaim = 0; myClaim < oppClaim; myClaim++) {
        Node node = responseNodes[myClaim][oppClaim];
        double[] actionProb = node.getStrategy();
        if (oppClaim < sides) {
            Node nextNode = claimNodes[oppClaim][rollAfterAcceptingClaim[oppClaim]];
            nextNode.pPlayer += actionProb[1] * node.pPlayer;
            nextNode.pOpponent += node.pOpponent;
        }
    }
```

Similarly, we next visit each possible claim nodes to which we have just propagated probabilities. For each of these, for each legal claim, we similarly accumulate probabilities forward to successor response nodes. In this way, we ensure proper visiting order. In general, FSICFR requires that forward visiting occur in topologically sorted order according to the subgraph formed by precomputed chance events.

⟨*Visit claim nodes forward*⟩≡
```
if (oppClaim < sides) {
    Node node = claimNodes[oppClaim][rollAfterAcceptingClaim[oppClaim]];
    double[] actionProb = node.getStrategy();
    for (int myClaim = oppClaim + 1; myClaim <= sides; myClaim++) {
        double nextClaimProb = actionProb[myClaim - oppClaim - 1];
        if (nextClaimProb > 0) {
            Node nextNode = responseNodes[oppClaim][myClaim];
            nextNode.pPlayer += node.pOpponent;
            nextNode.pOpponent += nextClaimProb * node.pPlayer;
        }
    }
}
```

Having completed this computation we know the relative frequencies for reaching each reachable node. We next visit these nodes a second time *in reverse order*, propagating the resulting utilities backwards from game end to game beginning, computing and accumulating counterfactual regrets. Note that, whereas we visited response nodes before claim nodes in forward propagation, here we reverse the order in backward propagation:

⟨*Backpropagate utilities, adjusting regrets and strategies*⟩≡
```
for (int oppClaim = sides; oppClaim >= 0; oppClaim--) {
    ⟨Visit claim nodes backward⟩
    ⟨Visit response nodes backward⟩
}
```

After retrieve each node and its strategy, we reset the node utility from its prior computation, and iterate through each legal claim action. For each legal claim action greater than the opponent's claim, we retrieve the corresponding node and, given that this is a zero-sum game, the node player's utility is the negation of the next node's utility for the opponent playing it. We initialize the regret values to each of these negated successor node utilities, and add the utility to the node utility, weighted by the probability of choosing that action.

Once we have completed computation of the node utility, we can then subtract these values from the individual action utilities to compute action regrets. Multiplying these by the opponent reach probability sums, we thus accumulate counterfactual regret for the node.

Finally, we reset these reach probabilities.

⟨*Visit claim nodes backward*⟩≡

```
if (oppClaim < sides) {
    Node node = claimNodes[oppClaim][rollAfterAcceptingClaim[oppClaim]];
    double[] actionProb = node.strategy;
    node.u = 0.0;
    for (int myClaim = oppClaim + 1; myClaim <= sides; myClaim++) {
        int actionIndex = myClaim - oppClaim - 1;
        Node nextNode = responseNodes[oppClaim][myClaim];
        double childUtil = - nextNode.u;
        regret[actionIndex] = childUtil;
        node.u += actionProb[actionIndex] * childUtil;
    }
    for (int a = 0; a < actionProb.length; a++) {
        regret[a] -= node.u;
        node.regretSum[a] += node.pOpponent * regret[a];
    }
    node.pPlayer = node.pOpponent = 0;
}
```

The computation for visiting response nodes is very similar, except that these are the only nodes leading to terminal game situations. As such, regrets for the `DOUBT` action compute utility as 1 or -1 depending on whether or not the opponent's claim was unsupported or supported, respectively.

⟨*Visit response nodes backward*⟩≡

```
if (oppClaim > 0)
    for (int myClaim = 0; myClaim < oppClaim; myClaim++) {
        Node node = responseNodes[myClaim][oppClaim];
        double[] actionProb = node.strategy;
        node.u = 0.0;
        double doubtUtil = (oppClaim > rollAfterAcceptingClaim[myClaim]) ? 1 : -1;
        regret[DOUBT] = doubtUtil;
        node.u += actionProb[DOUBT] * doubtUtil;
        if (oppClaim < sides) {
            Node nextNode = claimNodes[oppClaim][rollAfterAcceptingClaim[oppClaim]];
            regret[ACCEPT] = nextNode.u;
            node.u += actionProb[ACCEPT] * nextNode.u;
        }
        for (int a = 0; a < actionProb.length; a++) {
            regret[a] -= node.u;
            node.regretSum[a] += node.pOpponent * regret[a];
        }
        node.pPlayer = node.pOpponent = 0;
    }
```

At the end of each training iteration, we check if training is halfway complete. If so, we iterate through all nodes, resetting all strategy sums to 0. As mentioned in the previous section, this is one means of eliminating the strategy errors accumulated in the early iterations of training.

⟨*Reset strategy sums after half of training*⟩≡

```
if (iter == iterations / 2) {
    for (Node[] nodes : responseNodes)
        for (Node node : nodes)
            if (node != null)
                for (int a = 0; a < node.strategySum.length; a++)
                    node.strategySum[a] = 0;
    for (Node[] nodes : claimNodes)
        for (Node node : nodes)
            if (node != null)
                for (int a = 0; a < node.strategySum.length; a++)
                    node.strategySum[a] = 0;
}
```

After training is complete, we print a summary of initial claim actions given initial rolls. This is followed by a complete output of the entire computed strategy for each node.

⟨*Print resulting strategy*⟩≡

```
for (int initialRoll = 1; initialRoll <= sides; initialRoll++) {
    System.out.printf("Initial claim policy with roll %d: ", initialRoll);
    for (double prob : claimNodes[0][initialRoll].getAverageStrategy())
        System.out.printf("%.2f ", prob);
    System.out.println();
}
System.out.println("\nOld Claim\tNew Claim\tAction Probabilities");
for (int myClaim = 0; myClaim <= sides; myClaim++)
    for (int oppClaim = myClaim + 1; oppClaim <= sides; oppClaim++)
        System.out.printf("\t%d\t%d\t%s\n", myClaim, oppClaim,
            Arrays.toString(responseNodes[myClaim][oppClaim].getAverageStrategy()));
System.out.println("\nOld Claim\tRoll\tAction Probabilities");
for (int oppClaim = 0; oppClaim < sides; oppClaim++)
    for (int roll = 1; roll <= sides; roll++)
        System.out.printf("%d\t%d\t%s\n", oppClaim, roll,
            Arrays.toString(claimNodes[oppClaim][roll].getAverageStrategy()));
```

The trainer main method specifies the number of die sides, creates the trainer, and executes training for a given number of iterations.

⟨*LiarDieTrainer main method*⟩≡

```
public static void main(String[] args) {
    LiarDieTrainer trainer = new LiarDieTrainer(6);
    trainer.train(1000000);
}
```

Combining these definitions, we form the complete `LiarDieTrainer`:

⟨*LiarDieTrainer.java*⟩≡

```
import java.util.Arrays;
import java.util.Random;

public class LiarDieTrainer {
    ⟨Liar Die definitions⟩
    ⟨Liar Die player decision node⟩
    ⟨Construct trainer and allocate player decision nodes⟩
    ⟨Train with FSICFR⟩
    ⟨LiarDieTrainer main method⟩
}
```

5.4   Exercise: 1-Die-Versus-1-Die Dudo with Memory of 3 Claims

Review the 1-die-versus-1-die Dudo CFR exercise. In this exercise, we abstract the prior problem making use of imperfect recall of actions. In particular, we allow the player to make decisions remembering at most the three most recent claims. Create `Dudo3Trainer.java`, applying FSICFR to this abstracted Dudo exercise as we did with Liar Die.

6   Exploiting Mistakes

In such computation of optimal play for Liar Die, we may sometimes observe odd play behavior. For example, suppose you claim 4 against the computer's strategy, and the computer accepts. The computer may reply with a claim of 6, forcing you to doubt and reveal a computer roll of 4 or less. Why does this appear to be an error in play?

    We reason that the computer should have claimed 5 with a roll of 4 or less because, had we accepted the claim, the computer would be better off. By claiming 6, the computer denies us the ability to trust a 5 claim in error. Is this rational?

    Yes, it is rational: The computer has determined that the rational player will doubt any claim that follows a 4 claim. A claim of 5 or 6 after a 4 makes no difference to a rational player, because a rational response is to always doubt. However, this does not maximally take account of and take advantage of potential player mistakes. This point is related to a joke about two economists walking down a street. One economist notices a twenty-dollar bill on the sidewalk and leans to pick it up. The other economist lays a hand the other's shoulder and explains "It can't be a twenty-dollar bill. Otherwise, someone would have picked it up by now." In a rational world, no one would leave $20 on the ground, so why consider the possibility? A rational player in the Nash-equilibrium sense would not stoop to pick up $20, because it's inconceivable that a rational opponent would leave it there. Such a player would also not consider playing in such a way as to hope for a future dropped $20.

    There are a variety of solution concepts related to this point, including perturbed equilibria, sequential equilibria, and (quasi-)perfect equilibria. For these exercises, we will not go into depth theoretically, but rather encourage the learner to experiment empirically with means of consideration of possible errors.

6.1   Epsilon Error Modeling

One assumption that can be added to these means of computation is an assumption that players will play with a fixed bounded error policy. One simple model assumes that, with a small probability epsilon (ϵ), a player will choose a move at random. At all other times, the player will follow our usual regret-matching strategy. This essentially represents a mix of uniform and regret-matching strategy.

    One may model this algebraically, for a given node, is by computing the uniform and regret-matching utilities separately. The algebraic utilities with ϵ as a variable would then be $(1 - \epsilon)$ multiplied by the regret-matching strategy utility plus ϵ multiplied by the uniform strategy utility. More formally, this model redefines the utility for player i at history h to be $u_{i,\epsilon}(h) = (1 - \epsilon)u_{i,\sigma}(h) + \epsilon u_{i,r}(h)$, where the inner utilities are defined in terms of $u_{i,\epsilon}$ (recursively) as:

$$u_{i,\sigma}(h) = \begin{cases} u_i(h) & \text{if } h \in Z; \\ \sum_{a \in A(h)} \sigma(h,a)u_{i,\epsilon}(ha) & \text{otherwise,} \end{cases} \quad \text{and } u_r(h) = \begin{cases} u_i(h) & \text{if } h \in Z; \\ \sum_{a \in A(h)} \frac{1}{|A(h)|}u_{i,\epsilon}(ha) & \text{otherwise.} \end{cases}$$

The problem is that the number of terms grows exponentially with the number of actions remaining from h to a leaf. To see this more clearly, imagine a fixed-depth horizon version of the recursive definition above which falls back back to the old utilities after some depth d. Then, assuming a fixed branching factor of $|A|$: when $d = 1$, there would be $2|A|$ child terms. When $d = 2$, there would be $4|A|^2$ terms. At depth d, there would be $(2|A|)^d$ terms. This places a significant computational burden

for algebraic representation since it accounts for every possible perturbation in the future. Not only are utilities potentially polynomials of degree d, but we need to store more regrets for these newly defined utilities as well, which in practice is prohibitive. One possible means of addressing this problem (which to the author's knowledge has not been tried, i.e. would be interesting research) would be to truncate such representations to a small number of terms as a form of approximate analysis. For example, we could ignore terms for $d > 3$.

A much simpler approach that requires very little in the way of modification to the algorithms above is to not treat the errors algebraically, but rather introduce them with some probability. For example, one could have an `epsilon` variable set to a small value (e.g. 0.001), that is directly factored into the computation of strategies. Similarly to the algorithms of Johanson et. al. in [6, 5], each learning agent can train against a *restricted* opponent that is forced to play a random move once every thousand moves but otherwise learns in the usual way.

Miltersen & Sørensen applied the ϵ-error modeling approach by constructing a perturbed sequence-form representation of the game [13]. In their work, they show that optimal solutions can be obtained for arbitrarily small $\epsilon > 0$ in polynomial time, and hence result in a quasi-perfect equilibrium (which are sequentially rational). In CFR, one can consider what happens in the limit as $\epsilon \to 0$; one might gradually reduce the value of the `epsilon` variable over time, e.g. by multiplying it by a decay factor (e.g. .9999) for each training iteration. In this way, one would build an initial regret bias towards behaviors exploiting error, but an open question is how much such bias would remain through continued chance sampling with low/zero `epsilon`. In other words, how could such an `epsilon` variable be best put to use in order to create policies that closely approximate a Nash equilibrium while maximally exploiting past and potential future player mistakes?

6.2   Exercise: Perturbed Liar Die

For this exercise, we invite the reader to experiment with the use of an `epsilon` error variable in calls to `getStrategy` for `LiarDieTrainer`. Use all that has been learned so far in order to form the highest quality approximations of optimal play that maximally exploit player error.

As a test of your code, observe the claims made for various rolls after the acceptance of an opponent claim of 4. One should see the consistent response of 5 for rolls less than or equal to 5 after the acceptance of an opponent claim of 4..

7   Further Challenges

Of course, these techniques are applicable to the many hundreds of poker variations that exist. Rules to over 200 Poker variations may be found in Ernest, Foglio, and Selinker's *Dealer's Choice: the complete handbook of Saturday night poker* [1].

However, it is often advisable to start with simpler games and work towards greater complexity. Many simpler bluffing dice games offer good opportunities for increasing your skill in applying these techniques. Reiner Knizia's *Dice Games Properly Explained* [7] has an excellent chapter surveying a variety of bluffing dice games, many of which have not yet had equilibrium play computed at this time.

7.1   Exercise: Minimum Unique Fingers

When one of the authors was young, children in his neighborhood would engage in complex and lengthy processes for determining who was "it" in games of tag, etc. Whereas Rock-Paper-Scissors suffices for two-player games in selecting a first player, children would commonly turn to Josephus-Problem[9]-like counting-out games[10] such as one-potato-two-potato, eeny-meeny-miny-moe, ink-a-dink,

---

[9]See URL `http://en.wikipedia.org/wiki/Josephus_problem`.
[10]See URL `http://en.wikipedia.org/wiki/Counting-out_game`.

and bubblegum.

In retrospect, these methods of choice were overly complex, unfair, and not particularly fun. What he has passed on to his own children is the Minimum Unique Fingers game, which is his practical specialization of the *lowest unique positive integer game* [17].

Players synchronize action by chant the four-beat "One, two, three, go!". On calling "go!", each player simultaneously extends one hand, and with that hand extends some number of fingers (up to 5 and including the possibility of extending 0 fingers, i.e. a fist). The player with the lowest number of fingers extended, whose number is not duplicated by any other player, wins. In other words, the player showing the minimum unique number of fingers wins. If there is no player with a unique number of fingers, there is no winner.

For example, if Ann, Bob, and Cal extend 0, 0, and 5 fingers, respectively, Cal wins. If they extend 0, 1, and 2 fingers, respectively, Ann wins. If they extend 1, 1, and 1 fingers, respectively, no one wins.

Let p be the number of players. To formulate this as a zero-sum game, all payoffs/utilities are 0 when no one wins, and when one wins, the winner gets a payoff of $p - 1$ while the losers get a payoff of -1.

As it turns out, regret matching is more generally capable of computing *correlated equilibria* for non-zero-sum games and games with more than two players. Study the correlated equilibria solution concept, starting with [3].

Create a regret minimization program `MUFTrainer` that is parameterized for the number of players p and a number of training iterations. Compute and print a correlated equilibrium strategy for 5 children. For this to work, the way the average strategy is computed has to change. Instead, the algorithm must compute the average joint strategy by normalizing the frequency of observed tuples in the joint action space $(0, 1, 2, 3, 4, 5)^5$. Is the correlated equilibrium computed by your program also a Nash equilibrium? How does one show this?

8   Acknowledgements

References

[1] James Ernest, Phil Foglio, and Mike Selinker. *Dealer's Choice: the complete handbook of Saturday night poker*. Overlook Duckworth, Peter Mayer Publishers, Inc., New York, 2005.

[2] Sam Ganzfried, Tuomas Sandholm, and Kevin Waugh. Strategy purification and thresholding: Effective non-equilibrium approaches for playing large games. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2012.

[3] Sergiu Hart and Andreu Mas-Colell. A simple adaptive procedure leading to correlated equilibrium. *Econometrica*, 68(5):1127–1150, September 2000.

[4] Gil Jacobs. *The World's Best Dice Games, new edition*. John N. Hansen Co., Inc., Milbrae, California, 1993.

[5] Michael Johanson and Michael Bowling. Data biased robust counter strategies. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 264–271, 2009.

[6] Michael Johanson, Martin Zinkevich, and Michael Bowling. Computing robust counter-strategies. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20 (NIPS)*, pages 721–728, Cambridge, MA, 2008. MIT Press.

[7] Reiner Knizia. *Dice Games Properly Explained*. Elliot Right-Way Books, Brighton Road, Lower Kingswood, Tadworth, Surrey, KT20 6TD U.K., 1999.

[8] Harold W. Kuhn. Simplified two-person poker. In Harold W. Kuhn and Albert W. Tucker, editors, *Contributions to the Theory of Games*, volume 1, pages 97–103. Princeton University Press, 1950.

[9] Marc Lanctot. *Monte Carlo Sampling and Regret Minimization for Equilibrium Computation and Decision-Making in Large Extensive Form Games*. PhD thesis, University of Alberta, University of Alberta, Computing Science, 116 St. and 85 Ave., Edmonton, Alberta T6G 2R3, January 2013.

[10] Marc Lanctot, Richard Gibson, Neil Burch, and Michael Bowling. No-regret learning in extensive-form games with imperfect recall. In *Proceedings of the Twenty-Ninth International Conference on Machine Learning (ICML 2012)*, 2012.

[11] Marc Lanctot, Kevin Waugh, Martin Zinkevich, and Michael Bowling. Monte carlo sampling for regret minimization in extensive games. In Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1078–1086. MIT Press, 2009.

[12] Kevin Layton-Brown and Yoav Shoham. *Essentials of Game Theory: A Concise, Multidisciplinary Introduction*. Morgan and Claypool Publishers, 2008.

[13] Peter Bro Miltersen and Troels Bjerre Sørensen. Computing a quasi-perfect equilibrium of a two-player game. *Economic Theory*, 42(1):175–192, 2010.

[14] Merilyn Simonds Mohr. *The New Games Treasury*. Houghton Mifflin, Boston, Massachusetts, 1993.

[15] Todd W. Neller and Steven Hnath. Approximating optimal Dudo play with fixed-strategy iteration counterfactual regret minimization. In H. Jaap van den Herik and Aske Plaat, editors, *LNCS 7168: Advances in Computer Games, 13th International Conference, ACG 2011, Tilburg, The Netherlands, November 20-22, 2011, Revised Selected Papers*, pages 169–182. Springer, 2012.

[16] Kevin Waugh, Martin Zinkevich, Michael Johanson, Morgan Kan, David Schnizlein, and Michael H. Bowling. A practical use of imperfect recall. In Vadim Bulitko and J. Christopher Beck, editors, *SARA*. AAAI, 2009.

[17] Qi Zeng, Bruce R. Davis, and Derek Abbott. Reverse auction: The lowest unique positive integer game. *Fluctuation and Noise Letters*, 7(4):L439–L447, 2007.

[18] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1729–1736. MIT Press, Cambridge, MA, 2008.