Regret minimisation learning

01872353

March 8, 2024

Abstract

The report emphasizes on numerical simulations of regret minimising learning. It contains various examples, from simple to challenging. Firstly, it includes two classic games, *Rock, Paper, Scissors* and *Blotto* game, helping to build up basic ideas of regret minimising learning. Then *Liar Die* example provides insights of fixed-strategy iteration counterfactual regret minimization, a type of regret minimising learning. Lastly, a significant and highly related algorithm, q-learning, has been researched and implemented for *1 dice versus 1 die Dudo*. Interesting figures and findings are presented in the main context for all the games above.

1 Regret Minimisation Learning

1.1 What is Regret Learning

1.1.1 Rock, Paper, Scissors

Rock, Paper, Scissors(RPS): A classic two-player game. Each player can choose to present rock, paper, scissors by gesture. Rock beats scissors, scissors cut papers, and paper wins rock. Same gesture is a draw.

	rock	paper	scissor
rock	0,0	-1,1	1,-1
paper	1,-1	0,0	-1,1
scissor	-1,1	1,-1	0,0

Table 1: payoff table of RPS

This game is usually played for N times and the player who wins more would be the final winner. It is a really good example for explaining what regret learning is. Suppose we have player A,B, and the payoff of winning one round is $+1$, while losing is -1 and a draw is 0. If player A plays scissor, B plays paper, then B would regret not showing scissors to draw and regret more not playing rock to win. Then player B would probably have higher probability to show scissor or rock in the next round. Continuing playing RPS, the regret of not choosing strategies would accumulate and yield a mixed strategy.

1.1.2 Definitions

Now we formalise the concepts:
N: total of number of rounds played, maximum iterations
$S_A = s_i^A, i = 1, 2, ..., N$, $S_B = s_i^B, i = 1, 2, ..., N$ are sets of actions/moves of player A, B could play.
u is payoff function or called utility, $u(s_i^A, s_j^B)$ is the payoff when A choose strategy i, B choose strategy j.

Definition: *Regret* of not having chosen an action is the difference between the payoff of that action and the payoff of the action actually chose, with respect to the fixed choices of other players.

Cumulative Regret: The sum of regrets up to certain round. R_j^i is the cumulative regret of player i to j iteration.

Regret Matching: A player choose an action at random with a distribution that is proportional to the positive regrets. Normalised positive regrets are positive regrets divided by their sum. Denote the probability distribution after jth iteration for player i as r_j^i.

Example:

	round 1	round 2	round 3	...
player A	scissors	paper	rock	...
player B	rock	paper	scissors	...
payoff	-1,1	0,0	1,-1	...

Table 2: example of RPS

Initially, the regrets of A, B are both 0. After first play, the cumulative regrets of player A for rock, paper, scissors is 1,2,0 respectively, then we have regret-matching $(\frac{1}{3}, \frac{2}{3}, 0)$

	R^A	r^A	R^B	r^B
round 1	(1,2,0)	$(\frac{1}{3}, \frac{2}{3}, 0)$	(0,0,0)	(0,0,0)
round 2	(1,2,2)	$(\frac{1}{5}, \frac{2}{5}, \frac{2}{5})$	(0,0,2)	(0,0,1)
round 3	(1,2,2)	$(\frac{1}{5}, \frac{2}{5}, \frac{2}{5})$	(1,2,2)	$(\frac{1}{5}, \frac{2}{5}, \frac{2}{5})$

Table 3: regret of each player

1.2 Numerical Implement of RPS

1.2.1 Convergence

According to [3], the convergence of the algorithm can be guaranteed. A regret-learning algorithm would converge to correlated equilibrium.

Definition:
Player A, B has m, n actions, the matrix $(p_{ij}), i = 1, \ldots, m$ and $j = 1, \ldots, n$ is a probability distribution if all its entries are ≥ 0 and $\sum_{ij} p_{ij} = 1$. A joint distribution (p_{ij}) is a correlated equilibrium (CE) for the bimatrix game (A, B) if

$$\sum_k a_{i'k} p_{ik} \leq \sum_k a_{ik} p_{ik} \quad \text{and} \quad \sum_l b_{lj'} p_{lj} \leq \sum_l b_{lj} p_{lj}$$

NE is a subset of CE.

1.2.2 Algorithm

I followed the procedure in [4] and wrote a python scheme by myself (see appendix) to implement "Rock, Paper, Scissors" game.

Algorithm 1 Algorithm

```
1. Initialize all cumulative regrets to 0
2. During the iterations:
- compute the regret-matching strategy profile
- add the strategy profile to the strategy profile sum
- randomly give each player's action according the strategy profile
- compute players' regrets and add to the cumulative regrets
3. Return the probability distribution, i.e. the strategy profile sum divided by the number of iterations.
```

1.2.3 Convergence and Results

By looking at figure 1, we can see all lines of the strategies converge to approximately $\frac{1}{3}$. So initial conditions do not influence the result of the scheme.

Print-Out Strategy Profile

		10000 iterations	100000 iterations
player 1	rock	0.33435895	0.33362983
	paper	0.33286919	0.3331664
	scissors	0.33277186	0.33320377
player 2	rock	0.33439215	0.3338807
	paper	0.33200418	0.3318921
	scissors	0.33360367	0.3342272

Table 4: Action Probabilities

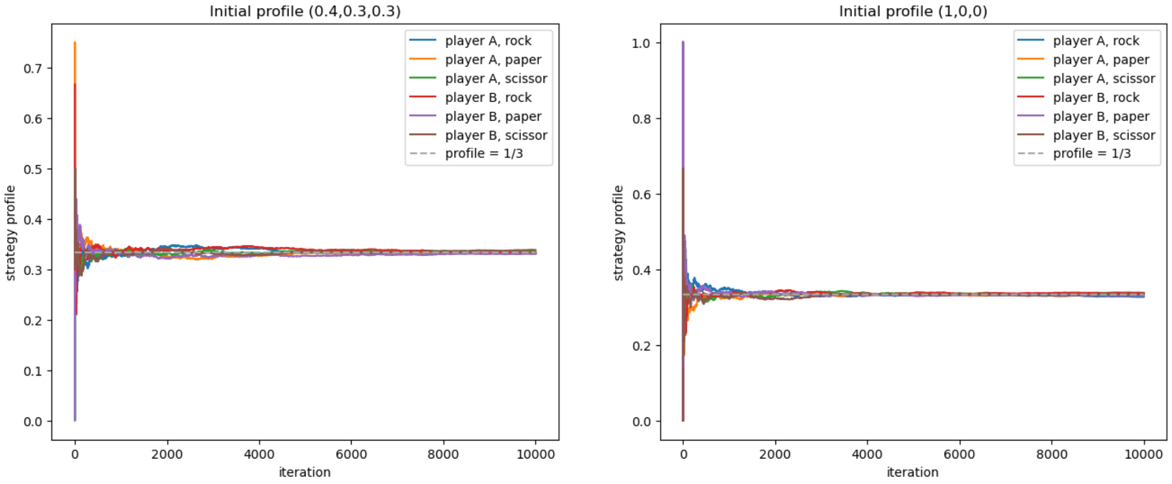


Figure 1: strategy profile of 10000 iterations for two initial value

Figure 2 is a zoom-in figure of initial profile $(0.4, 0.3, 0.3)$ and 10000 iterations. The probability values are all around $\frac{1}{3}$, which coincides with our expectations. Comparing the print-out results of fig 2,3, 4, with more iterations, the probability profile is getting closer to $[0.3333..., 0.3333..., 0.3333...]$. We can conclude that our numerical computed equilibrium, which is the correlated equilibrium, is close to the Nash Equilibrium $\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$.

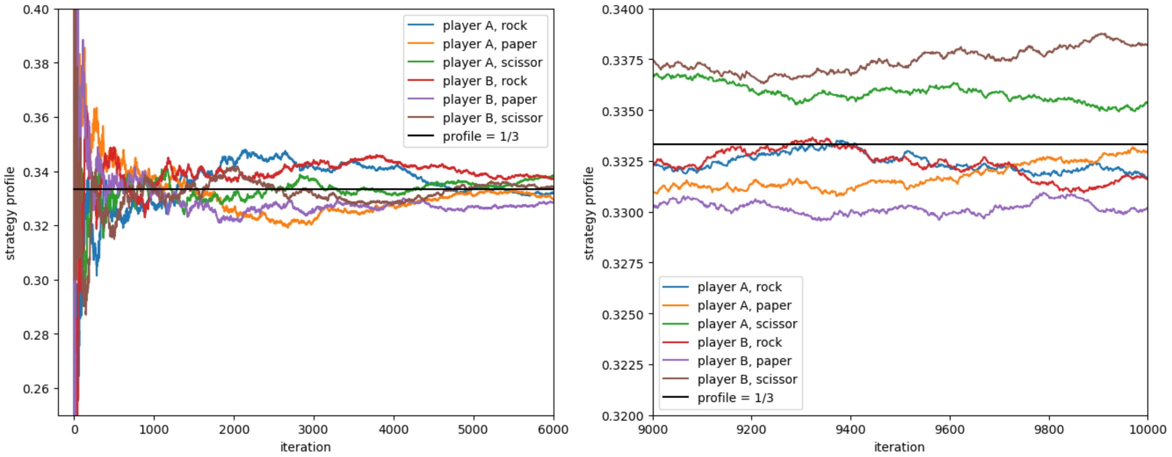


Figure 2: A zoom-in graph for initial profile (0.4,0.3,0.3)

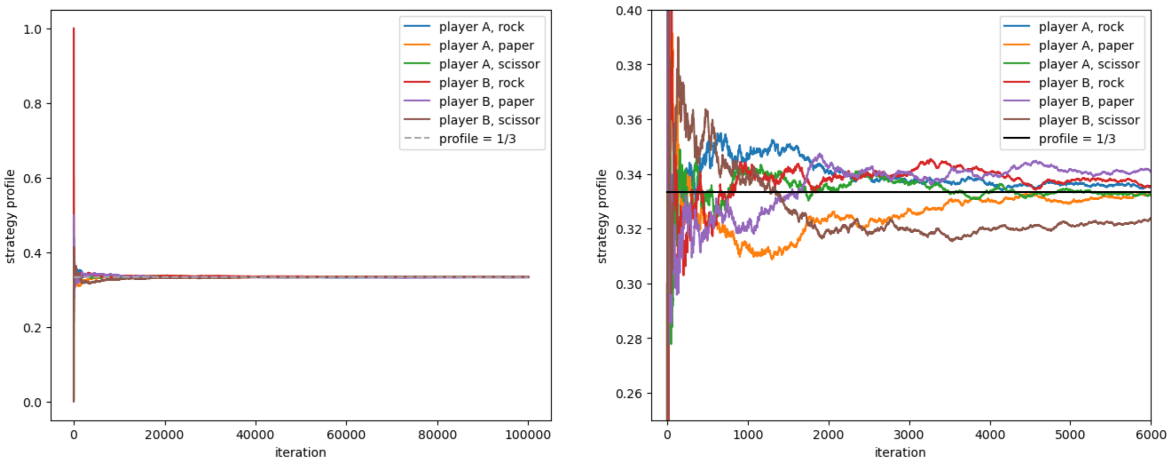


Figure 3: 100000 iterations with initial profile (0.4,0.3,0.3)

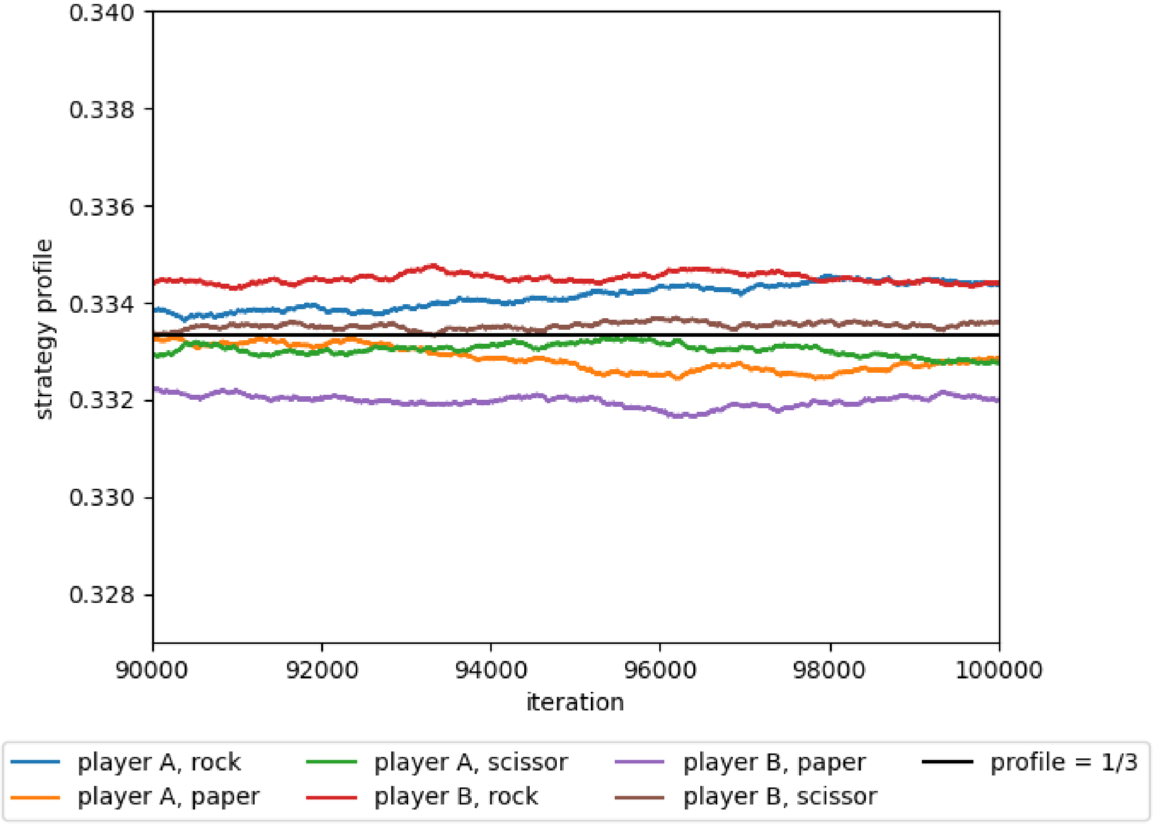


Figure 4: zoom-in

2 Colonel Blotto Game

2.1 Game Setup

Colonel Blotto Game is a constant-sum game with two players and they are asked to allocate limited resources simultaneously over several objects (called battlefields). Let's first look at the classical version discussed in [4]:

There are two commanders in one war, namely Colonel Blotto and Boba Fett. Each of them has S soldiers and needs to distribute the soldiers into N battlefields, where $N < S$. No communication is allowed between the two commanders. If more soldiers assigned by one commander in the battlefield, the commander claims this battlefield. The commander wins more battlefield wins the war.

Blotto game also has many different variations. Instead of simply winning or losing, the battlefields can have different values of scores, when the game is non-homogeneous. If all players have same budget, it is symmetry. The way of distributing also classifies the game:

Discrete: allocations are non-negative integers
Continuous: allocations are non-negative real numbers
Boolean: only 1 or 0 can be assigned to the battlefields.
Here the classical version is a discrete, homogeneous, symmetry Blotto game.

2.2 Simple Example of Classical Version

According to [4], I use $S = 5, N = 3$ as a simple example for discussion. Let's say we have player X, Y. Denote the battlefield as B_i, $i = 1, 2, 3$, and $(a, b, c)_j$ shows how many soldiers are assigned by commander j, $j = X, Y$, to battlefield 1, 2, 3 $(a, b, c \geq 0, a+b+c = S)$.For instance, $(0, 0, 5)_x$ means 5 soldiers are assigned by player X in B_3 and 0 in B_1, B_2. The payoff is the number of battles won.

Considering being player X, we can find all our strategies: Firstly, I separate the $S = 5$ soldiers into $N = 3$ pieces and assign them to each battlefield. The process of assigning is just permutation. Here we can have $(0, 0, 5), (0, 1, 4), (0, 2, 3); (1, 1, 3), (1, 2, 2)$, so there are $3 \cdot 3!/2! + 2 \cdot 3! = 21$ different strategies.

Then we can have the payoff matrix:

		player Y					
		(0,0,5)	(0,1,4)	(0,2,3)	(1,1,3)	(1,2,2)	...
player X	(0,0,5)	1.5,1.5	1.5,1.5	1.5,1.5	1,2	1,2	...
	(0,1,4)	1.5,1.5	1.5,1.5	1.5,1.5	1.5,1.5	1,2	...
	(0,2,3)	1.5,1.5	1.5,1.5	1.5,1.5	1.5,1.5	1.5,1.5	...
	(1,1,3)	2,1	1.5,1.5	1.5,1.5	1.5,1.5	1.5,1.5	...
	(1,2,2)	2,1	2,1	1.5,1.5	1.5,1.5	1.5,1.5	...
	⋮	⋮	⋮	⋮	⋮	⋮	

Table 5: size 21 × 21 payoff matrix

Generally, Given N battlefields, S soldiers of each commander ($N < S$), we can think of assigning S balls into N different boxes. The number of ways of distributing should be

$$\binom{S+N-1}{N-1} = \frac{(S+N-1)!}{S!(N-1)!}$$

Thus, for each player they have $\binom{S+N-1}{N-1} = K$ strategies. In principle, they would have a $K \times K$ payoff matrix.

2.3 Nash Equilibrium

2.3.1 Existence

The most important problem for players is that spending more resources in one battlefield to win results in less contribution in other fields. From table1, we can see there is no strictly dominated strategy for X, Y, thus no pure strategy Nash equilibrium exists. We can prove there must exist Nash Equilibrium when the allocation to each battlefield is integer.

Definition: A game is finite if all players in the game have a finite number of pure strategies. If at least one player has an infinite number of pure strategies then the game is an infinite game.

Theorem(Nash (1951)): Every finite game has at least one equilibrium.

proof[1]: This theorem can be applied to any general games. Here I only prove for two-player games. Say a finite ($M \times N$) two-player game, with player X, player Y. Let $C = \mathbb{X} \times \mathbb{Y}$ be the product of the mixed strategy sets. $\mathbb{X}$ and $\mathbb{Y}$ are convex compact sets by definition. Since product of convex and compact sets is convex and compact, C is also convex and compact.

Denote the mixed strategy of player X, Y as $x = (p_1, p_2, \cdots, p_n)$, $y = (q_1, q_2, \cdots, q_n)$, payoff function $g_X(x, y)$, $g_Y(x, y)$. Now define a function $f : C \mapsto C$. if the pure strategy x_i has better expected payoff than the mixed strategy, it maps each pair of mixed strategies (x, y) to another pair $f(x, y) = (x^*, y^*)$ where each $p_i^* > p_i$, $q_i^* > q_i$. If worse, it maps to (x^*, y^*) where each $p_i^* < p_i$, $q_i^* < q_i$ and zero if $p_i, q_i = 0$.

When x and y are best response to each other at the equilibrium, there will be only worse strategies and no better pure strategies. By comparing and adjusting the probabilities here, the mixed strategy x will not change with fixed y, and same for y. Thus, $f(x, y) = (x^*, y^*) = (x, y)$, (x, y) is a fixed point of function f. Now we can formalise the function:

$$\begin{aligned} \phi &: \mathbb{X} \times \mathbb{Y} \mapsto \mathbb{R}^n, \\ \psi &: \mathbb{X} \times \mathbb{Y} \mapsto \mathbb{R}^m. \end{aligned}$$

The component functions $\phi_i(x, y)$, $\psi_j(x, y)$, where $i = 1, 2, \cdots, n$, $j = 1, 2, \cdots, m$

$$\begin{aligned} \phi_i(x, y) &= \max\left\{0, g_X\left(a_i, y\right) - g_X(x, y)\right\}, \\ \psi_j(x, y) &= \max\left\{0, g_Y\left(x, b_j\right) - g_Y(x, y)\right\} \end{aligned}$$

If $g_X\left(a_i, y\right) \geq g_X(x, y)$, $g_Y\left(x, b_j\right) \geq g_Y(x, y)$, $\phi_i(x, y)$ and $\psi_j(x, y)$ are positive respectively and equal to the differences of payoffs between the pure strategy and the mixed strategy. If $g_X\left(a_i, y\right) < g_X(x, y)$, $g_Y\left(x, b_j\right) < g_Y(x, y)$, $\phi_i(x, y)$ and $\psi_j(x, y)$ are 0. Thus, $\phi(x, y)$ is a nonnegative vector in $\mathbb{R}^n$ and $\psi(x, y)$ is a nonnegative vector in $\mathbb{R}^m$. And these functions are continuous.

Using ϕ and ψ, function f would replace the vectors x with $x + \phi(x, y)$ to achieve x^* and replace y with $y + \psi(x, y)$ to achieve y^*. The sums are non-negative, but the elements in the vectors do not sum up to one, meaning that they are not mixed strategies anymore. We need to normalise them:

$$\begin{aligned} p_i^* &= \frac{p_i + \phi_i(x, y)}{\sum_i p_i + \sum_i \phi_i(x, y)} = \frac{p_i + \phi_i(x, y)}{1 + \sum_i \phi_i(x, y)}, \\ q_j^* &= \frac{q_j + \psi_j(x, y)}{\sum_j q_j + \sum_j \psi_j(x, y)} = \frac{q_j + \psi_j(x, y)}{1 + \sum_j \psi_j(x, y)}, \end{aligned}$$

We define $x^* = (p_1^*, p_2^*, \cdots, p_n^*)$ and $y^* = (q_1^*, q_2^*, \cdots, q_m^*)$. Now x^* and y^* are valid mixed strategies. We now can define the function $f : C \mapsto C$ as

$$f(x, y) = (x^*, y^*)$$

Next we want to show that when our constructed function f has a fixed point, we have an equilibrium of the game.

Claim:

$$(x, y) \text{ is an equilibrium } \Longleftrightarrow f(x, y) = (x, y)$$

Proof of claim:
$\Rightarrow$: Suppose (x, y) is an equilibrium. By definition

$$g_X\left(a_i, y\right) \leq g_X(x, y)$$

for all $i = 1, 2, \cdots, n$. This means that $\phi_i(x, y) = 0$ for all $i = 1, 2, \cdots, n$ and it follows that $p_i = p_i^*$ and hence $x = x^*$. Similarly, we can show $y = y^*$.

$\Leftarrow$: prove by contradiction. Assume $x^* = x$, $y^* = y$ but (x, y) is not an equilibrium. Then either $\exists$ x' such that $g_X(x', y) > g_X(x, y)$, or $\exists$ y' such that $g_Y(x, y') > g_Y(x, y)$. Without loss of generality, assume

$$\exists x' \textit{ such that } g_X(x', y) > g_X(x, y)$$

(The other one is similar) Let $x' = (p_1', p_2', \cdots, p_n')$, then by definition

$$g_X(x', y) = \sum_{i=1}^{n} p_i' g_X(a_i, y)$$

Thus, there must exist an i for which $g_X(a_i, y) > g_X(x, y)$. Otherwise, if this were not the case, then we contradict $g_A(x', y) > g_X(x, y)$ because

$$g_X(x', y) \le \sum_{i=1}^{n} p_i' g_X(x, y) = g_X(x, y)$$

For each i,

$$\phi_i(x, y) = \max\{0, g_X(a_i, y) - g_X(x, y)\} > 0$$

and thus $\sum_i \phi_i(x, y) > 0$. Consider

$$g_X(x, y) = \sum_{i=1}^{n} p_i g_X(a_i, y)$$

so $g_X(x, y) \ge g_X(a_k, y)$, for some k, with $p_k > 0$. Otherwise,

$$\begin{aligned} g_X(x, y) &= \sum_k p_k g_X(a_k, y) \ > \ g_X(x, y) \sum_k p_k \\ &= g_X(x, y) \end{aligned}$$

which does not make sense. For this value of k,

$$\phi_k(x, y) = \max\{0, g_X(a_k, y) - g_X(x, y)\} = 0$$

$$p_k^* = \frac{p_k + \phi_k(x, y)}{1 + \sum_i \phi_i(x, y)} = \frac{p_k}{1 + \sum_i \phi_i(x, y)} < p_k$$

As p_i's are not equal, $x^* \neq x$. Contradiction raises because we assumed $x^* \neq x$. Hence (x, y) must be an equilibrium. The claim holds.

At last, we show our function always has a fixed point. By Brouwer's fixed point theorem: a continuous function from a closed, bounded, convex set C to itself always has at least one fixed point. Our function satisfies the requirements of the theorem, so we always have at least one fixed point. Hence, the game always has at least one equilibrium. $\square$

Claim: The Blotto game must have a Nash equilibrium if the allocation to each battle field is an integer

proof: Given $N < \infty$ battlefields, $S < \infty$ soldiers of each commander ($N < S$), since our allocations are integers, the number of strategies for each player should be $\binom{S+N-1}{N-1}$, which is a finite number. Thus, it is a finite game. Using the theorem above, we can conclude that for discrete Blotto game, a Nash equilibrium must exist.

2.3.2 Finding the NE

I found the Nash equilibrium of $N = 3, S = 5$ case by following [2]. The paper introduced discrete Colonel Blotto and general Lotto games and proved they have same solutions, which means we can obtain the equilibrium of Blotto game via finding the equilibrium of the Lotto game. Firstly, it defined the expectation of how many soldiers assigned to each field,

$$E(X) = \frac{budget\ of\ player\ X}{number\ of\ battlefields} = a, \quad E(Y) = \frac{budget\ of\ player\ Y}{number\ of\ battlefields} = b$$

and we can classify into different cases depending on the value of a, b. Here $a = b = \frac{5}{3}$, we can use following arguments to seek for our Nash equilibrium:

Theorem 3 Let $a = m + \alpha$ and $b = m + \beta$ where $m \geq 0$ is an integer and $0 < \alpha, \beta < 1$. Then the value of the General Lotto game $\Gamma(a, b)$ is

$$\operatorname{val}\Lambda(a, b) = \frac{a - b}{\lceil a \rceil} = \frac{\alpha - \beta}{m + 1}$$

and the unique optimal strategies are $X^* = (1 - \alpha)U_{\mathrm{E}}^m + \alpha U_{\mathrm{O}}^{m+1}$ for Player A and $Y^* = (1 - \beta)U_{\mathrm{E}}^m + \beta U_{\mathrm{O}}^{m+1}$ for Player B.

Where U are defined as uniform distribution for every positive integer m:

$$U^m := U(\{0, 1, \ldots, 2m\}) = \sum_{i=0}^{2m} \left(\frac{1}{2m+1}\right) \mathbf{1}_i$$

$$U_{\mathrm{O}}^m := U(\{1, 3, \ldots, 2m - 1\}) = \sum_{i=1}^{m} \left(\frac{1}{m}\right) \mathbf{1}_{2i-1}$$

$$U_{\mathrm{E}}^m := U(\{0, 2, \ldots, 2m\}) = \sum_{i=0}^{m} \left(\frac{1}{m+1}\right) \mathbf{1}_{2i}$$

When $N = 3, S = 5$, $a = b = \frac{5}{3}$, $\operatorname{val}\Lambda(a, b) = 0$, and the unique optimal strategies are

$$X^* = \frac{1}{3}U_{\mathrm{E}}^1 + \frac{2}{3}U_0^2 = \frac{1}{3}U(\{0, 2\}) + \frac{2}{3}U(\{1, 3\})$$

$$Y^* = \frac{1}{3}U_{\mathrm{E}}^1 + \frac{2}{3}U_0^2 = \frac{1}{3}U(\{0,2\}) + \frac{2}{3}U(\{1,3\})$$

At the same time, we can derive a more explicit form. In the proof of *proposition 6*[2], we can write a matrix form S^r, where m is odd, $m = 2n-1$, and $1 \le r \le K-1$. $S^1 = \left(s_{ij}^1\right)$, $i = 0,1,\dots,4n-1$, $j = 1,\dots,2k+1$ is a $(2m+2) \times K$ matrix (as below). There are $k-1$ columns in the first block k in the second. The last two columns do not change. For S^r, $2 \le r \le K-1$, we just add 1 to all entries in the first $r-1$ columns. The columns would indicate our equilibrium.

$$S^1 = \left(\begin{array}{ccc|ccc|c|c}
0 & \cdots & 0 & 4n-2 & \cdots & 4n-2 & 0 & 2n \\
2 & \cdots & 2 & 4n-4 & \cdots & 4n-4 & 1 & 2n+1 \\
\vdots & & \vdots & \vdots & & \vdots & \vdots & \vdots \\
4n-2 & \cdots & 4n-2 & 0 & \cdots & 0 & 2n-1 & 4n-1 \\
0 & \cdots & 0 & 4n-2 & \cdots & 4n-2 & 2n & 0 \\
\vdots & & \vdots & \vdots & & \vdots & \vdots & \vdots \\
4n-2 & \cdots & 4n-2 & 0 & \cdots & 0 & 4n-1 & 2n-1
\end{array}\right)$$

In our case, $n = 1$, $k = 1$, S^1 is a 4×3 matrix:

$$S^1 = \begin{pmatrix} 2 & 0 & 2 \\ 0 & 1 & 3 \\ 2 & 2 & 0 \\ 0 & 3 & 1 \end{pmatrix}, \qquad S^2 = \begin{pmatrix} 3 & 0 & 2 \\ 1 & 1 & 3 \\ 3 & 2 & 0 \\ 1 & 3 & 1 \end{pmatrix}$$

Thus, the optimal strategy is

$$\frac{1}{2}\langle 0,2,3\rangle + \frac{1}{2}\langle 1,1,3\rangle$$

where $\langle a,b,c\rangle$ denotes the partitions It generates

$$\frac{1}{3}\left(\mathbf{1}_1 + \mathbf{1}_3\right) + \frac{1}{6}\left(\mathbf{1}_0 + \mathbf{1}_2\right) = \frac{1}{3}U_{\mathrm{E}}^1 + \frac{2}{3}U_0^2$$

.

2.4 Numerical Implement of Blotto Game

Here I used the game setup in previous and in [4]: Each player is using regret-matching alternatively. In the figures, We can see the strategy profiles are converging and allocations with same number of soldiers but different orders can be considered together as a group. Comparing fig 5 and 6, more iterations show better convergence and no chaotic patterns can be observed.

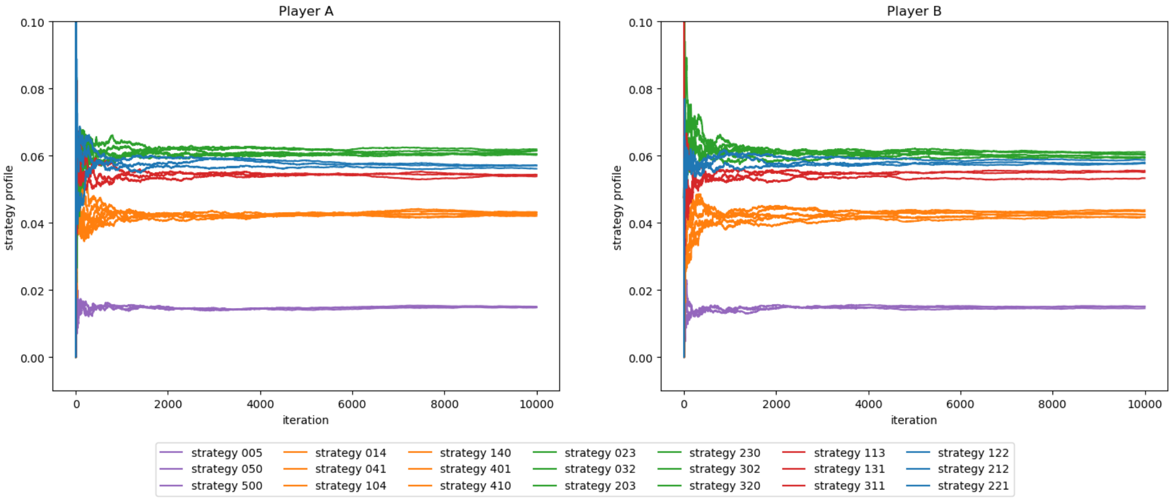


Figure 5: 10000 iterations

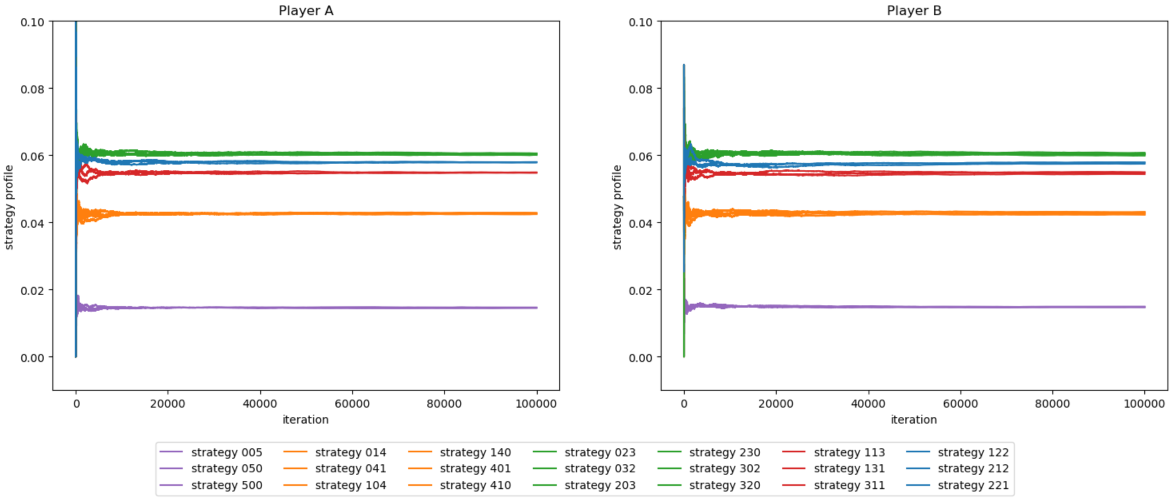


Figure 6: 100000 iterations

By numerical calculation, (up to 8 significant number) the correlated equilibrium of each permutations is

$\langle 0,0,5\rangle$	$\langle 0,1,4\rangle$	$\langle 0,2,3\rangle$	$\langle 1,1,3\rangle$	$\langle 1,2,2\rangle$
0.015221416	0.042718941	0.060107203	0.054736100	0.057723529

For histogram (fig 7), it is more clear to see $\langle 0,2,3\rangle$, $\langle 1,1,3\rangle$, $\langle 1,2,2\rangle$ are better strategies to win. Although $\langle 0,0,5\rangle$, $\langle 0,1,4\rangle$ hardly ever win, they still exhibit little probability. Possibly it is because assigning a 4 or 5 in one battlefield would claim that field in most of the time. And then they can obtain a tie, which are kind of safe choices.

This could be a surprising result, as our numerical equilibrium is quite different from analytical Nash equilibrium. [3] indicates it is reasonable to happen that numerical computations does not necessarily converge to Nash equilibrium, and the numerical results can still demonstrates valuable points. It is undeniable $\langle 0,2,3\rangle, \langle 1,1,3\rangle$ are good strategies. $\langle 0,2,3\rangle$ can win or get tie in most of cases, and $\langle 1,1,3\rangle$ is the only one can win $\langle 0,2,3\rangle$. Moreover, $\langle 1,2,2\rangle$ can be perceived as a moderate choice. It can beat $\langle 1,1,3\rangle$ and get a tie with $\langle 0,2,3\rangle$.

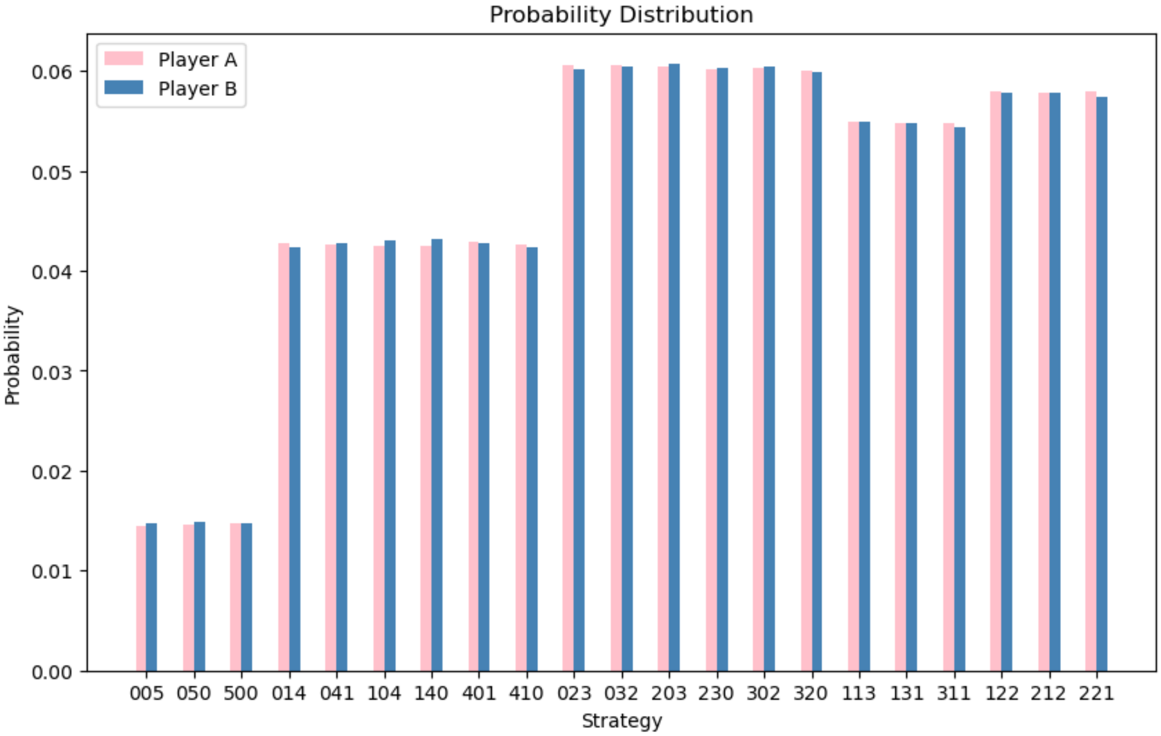


Figure 7: Histogram

3 Fixed-Strategy Iteration Counterfactual Regret Minimization (FSICFR)

3.1 Counterfactual Regret Minimization (CFR)

Counterfactual regret minimization (CFR) is an iterative algorithm, usually used for finding approximate Nash equilibrium in games. The word counterfactual is counter to the facts, i.e. imagine "what if...". Here the counterfactual regret means during calculating the regrets, the difference between current outcomes and actions we did not take is considered as well as the payoff difference between current and future decisions. This is a useful algorithm especially in imperfect information games, such as pokers and chess. Among the plays, the algorithm can dynamically adjust the probability of choosing actions based on new information and cumulative regrets.

3.2 Fixed-Strategy Iteration CFR Minimization

In CFR, strategies are updated iteratively based on regret minimization. The "Fixed-Strategy Iteration" (FSI) refers to that one player's strategy is considered fixed while the other players update their strategies. The process involves repeatedly playing the game, computing regrets, and adjusting strategies' profiles to reduce regrets over time.

The combination of these concepts, Fixed-Strategy Iteration Counterfactual Regret Minimization, suggests a specific application or variant of CFR algorithms where strategies are updated iteratively with one player's strategy being treated as fixed during each iteration.

3.3 Liar Die

Liar Die is a simplified-version game of all kinds of die-bluffing games. This section would introduce the important ideas in the game and numerical simulations would be presented in the next section.

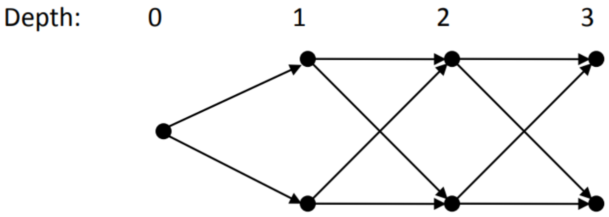
Game setup: Two players are called the player and the opponent. Each one has a s-sided die, with integer numbers one to s. At the beginning, the player roll the die under a cup and claim what the die shows (he or she is free to give a true, or smaller or bluffing number). Now the opponent has two choices:

Accept: The opponent thinks the player is not bluffing, so just accept it. Then it is opponent's turn to roll and claim, where he or she must claim a larger number than before. The game would stop until someone choose to doubt.

Doubt: The opponent doubts about it. The player remove the cup and show the true value of the die. If the real number is greater or equal to the claimed number, the player wins. Otherwise, the opponent wins.

3.4 FSICFR Algorithm

We can formalise the game in a directed acyclic graph (DAG). Acyclic means graph that does not contain any cycles. Each node represents a claimed number. Since we can only make higher and higher claims each time, so the graph is directed and never has closed loops.



Depth	Paths
1	1
2	2
3	4

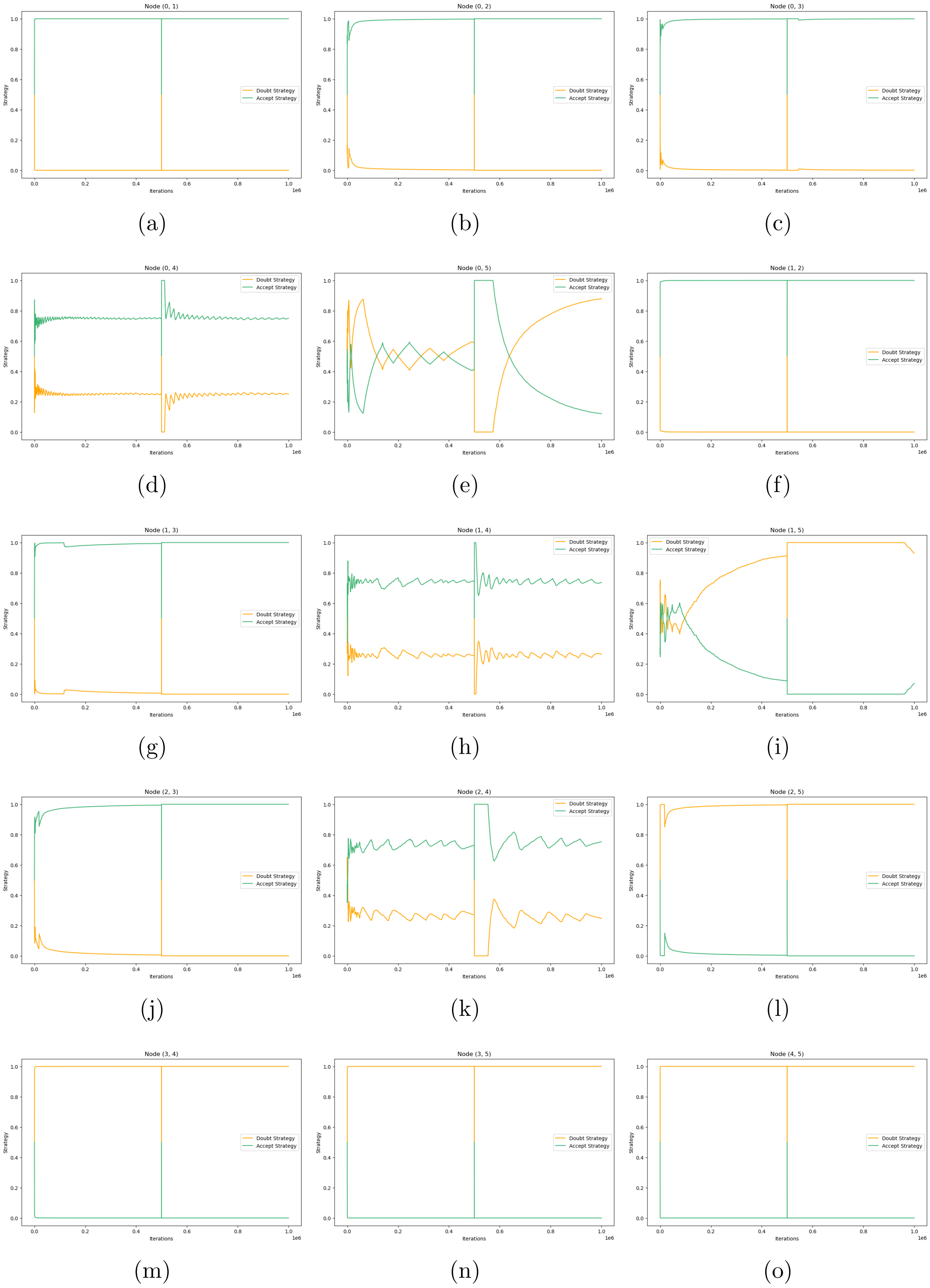
Figure 8: directed acyclic graph[4]

Followed the numerical structures in [4], we consider two sets: response nodes and claim nodes. Each node is defined as class or dictionaries, and it contains the all the information we need:
(i) all the possible paths through the node
(ii) the probabilities of each path for the players to choose, and their sum
(iii) strategy profile according to the probabilities
(iv) expected payoff at the node
(v) player actions at the node
For a more complicated directed acyclic graph, more information, such as the visits, are needed. Here we only need the datas stated above.

Node 0 means the start of the game. No one has made a claim. For two players, we have 7 × 7 nodes (0,1,2,3,4,5,6). For each iteration, we randomly generated new rolls, go through the path and adapt the probability of making different decisions.

3.5 Numerical Simulation

I wrote a numerical scheme based on [8] to simulate two-players Liar Die. Here I use a most common die, six-sided fair die. The code can be adapted to any other number of sides. (See Appendix)



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

(m) (n) (o)

Figure 9: plots for showing convergence

3.5.1 Convergence

Firstly, I plotted some graphs for testing convergence. Some numerical schemes of simple game would converge, such as Rock, Paper, Scissors, but some are not. [3] In the figure, although there is no sign of absolute convergence, we can observe no chaos. Thus, the numerical scheme can be trust.

3.5.2 Results

The table 6 shows the action probability of the opponent when knowing last claim and the new claim. Higher probability indicates higher chances for the opponent to win. Some entries are left blank as they are not allowed by the game rules. For instance, if last claim is 1, and the next claim is 2, the opponent chooses to accept would have 88.15% chances to win. This coincides with the reality, since as long as the player rolls a number larger or equal to two, he or she would be safe. For any s-sided fair die, $P(\mathit{rolled\ number} \geq 2) = \frac{s-1}{s}$ is quite large. When the new claims are 5 or 6, the only choice left for opponent is to doubt. If the opponent accepts 6, there is no larger possible number to claim. If 5 is accepted, the opponent must roll a six to win. Thus, in both cases, the winning probability of doubt converges to 1.

Opponent's Table

		New Claims					
		1	2	3	4	5	6
Old Claims	0	[0.0, 1.0]	[0.0, 1.0]	[0.0, 1.0]	[0.2516, 0.7484]	[1.0, 0.0]	[1.0, 0.0]
	1		[0.1185, 0.8815]	[0.0, 1.0]	[0.2525, 0.7475]	[1.0, 0.0]	[1.0, 0.0]
	2			[0.0, 1.0]	[0.2998, 0.7002]	[1.0, 0.0]	[1.0, 0.0]
	3				[1.0, 0.0]	[1.0, 0.0]	[1.0, 0.0]
	4					[1.0, 0.0]	[1.0, 0.0]
	5						[1.0, 0.0]

Table 6: Action Probabilities [*Doubt*, *Accept*] for Old and New Claims

The player's table 7 demonstrates given the previous claim and the number the player just rolled, which number to claim gives higher probability to win. Entries are left blank as they are not allowed by the game rules. Entries of 0.0 means too little probability to win, but not necessarily no probability to win. For instance, at the beginning of the game, if the player rolls a 4 but claims 1, no matter the opponent choose to doubt or accept, the player is safe. But claiming 1 leaves many numbers for later, which gives large uncertainty to the final result. The player could have claim 3 or 4, which is also safe and makes the opponent hard to decide.

Player's Table

Last Claim	Roll	1	2	3	4	5	6
0	1	0.0962	0.0962	0.4241	0.1911	0.0962	0.0962
	2	0.0981	0.0981	0.2077	0.3997	0.0981	0.0981
	3	0.0828	0.0828	0.4793	0.1894	0.0828	0.0828
	4	0.0	0.0	0.0059	0.9941	0.0	0.0
	5	0.0	0.0	0.0031	0.3524	0.6446	0.0
	6	0.0	0.0	0.0016	0.2064	0.3798	0.4122
1	1		0.1994	0.1994	0.2026	0.1994	0.1994
	2		0.1380	0.2660	0.4304	0.0828	0.0828
	3		0.0300	0.8154	0.1546	0.0	0.0
	4		0.0037	0.0001	0.9963	0.0	0.0
	5		0.0004	0.0084	0.3451	0.6461	0.0
	6		0.0004	0.0045	0.2060	0.3844	0.4048
2	1			0.2291	0.3127	0.2291	0.2291
	2			0.2250	0.3249	0.2250	0.2250
	3			0.8942	0.1058	0.0	0.0
	4			0.0	1.0	0.0	0.0
	5			0.0012	0.3344	0.6644	0.0
	6			0.0031	0.2032	0.3969	0.3969
3	1				0.3333	0.3333	0.3333
	2				0.3333	0.3333	0.3333
	3				0.3333	0.3333	0.3333
	4				1.0	0.0	0.0
	5				0.5	0.5	0.0
	6				0.3333	0.3333	0.3333
4	1					0.5	0.5
	2					0.5	0.5
	3					0.5	0.5
	4					0.5	0.5
	5					1.0	0.0
	6					0.5	0.5
5	1						1.0
	2						1.0
	3						1.0
	4						1.0
	5						1.0
	6						1.0

Table 7: Action Probabilities for Old and New Claims

4 Reinforcement Learning

4.1 Reinforcement Learning in Game Theory

Reinforcement learning is a popular topic in machine learning, game theory, optimization and so on. It has many real-world applications, such as autonomous driving systems and energy storage operation. With careful training, we can even teach a robot to walk. More importantly, reinforcement learning is significantly useful in long-term games with short-term trade-off. A famous example is the AlphaGo in chess. Reinforcement learning concerns about how an agent reacts and adapts its strategies in a dynamical environment. Generally, reinforcement learning scheme involves agent, environment, state, action and reward.

4.2 Q-Learning

Q-learning is a environment-free reinforcement learning. It does not require an environment and plays a dominate role in stochastic transitions. Applying it to a finite Markov decision process, it can discover the optimal strategy which maximises the rewards. "Q" is normally a function of expected rewards of actions at a state. The function values at different states and actions form a Q table, or called Q matrix.

In our games, the agent can be considered as a brainless player (a child), who has no thinking and takes actions only according to the memory. The memory would be used to update Q table. The idea is quite similar to cumulative regrets. The action profile change as updating the values of Q matrix. If no memory has presented, the action would be totally random. Action refers to strategy, and reward is our payoff.

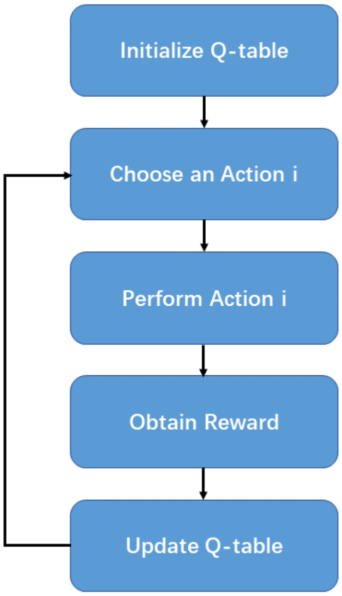


Figure 10: process of Q learning

4.3 1 Die Versus 1 Die Dudo

In this project, I have trained an agent to play 1 Die Versus 1 Die Dudo against some dummy agents. 1 Die Versus 1 Die Dudo is a classic party game. It is a multiple-players version of the Liar Die. The environment is dynamical, and during the game, it is hard for human to calculate the probabilities and find optimal strategy. So I choose this game for an agent to play.[6]

4.3.1 Game Rules

There are N players on the table and each player has S six-sided dice and a cup. In each round, every player on the table uses the cup to cover the dice they have and then rolls at the same time. Then they carefully and secretly read what they have rolled. One player starts to make a claim about the quantity of numbers on the table, such as four 3s, five 2s, six 6s, where 3,2,6 are the numbers, four, five six are called rank of number. For representing, I use (n, r) for (rank, number), i.e. $(4, 3), (5, 2), (6, 6)$.

The next player chooses between accept or doubt it. If the second player accepts it, he or she must make a stronger claim – either adds at least one on the rank or adds at least one on the number. If this second player chooses to doubt, declaring ”Dudo!” (meaning ”I doubt it” in Spanish), this round ends. All players on the table remove the cup and show the numbers they have. If the actual rank is equal to or more then the claim, that is when the claimer is not bluffing, the challenger loses a dice. Otherwise, the challenged player would lose a dice.

It is worth noting that number 1 can be played as ‘wild’, which means 1 can be counted as any other numbers while others are not. If I have $5, 5, 4, 3, 2, 4, 6, 1, 1$, then I have $(3, 6), (4, 5), (4, 4), (3, 3), (3, 2), (2, 1)$.

In the real life, the game rules may have some variations. In some region, the challenger would only lose dice if the actual rank is more then the claim. When actual total rank is equal the claim, all other players lose a dice (except the claimer and the challenger). Additionally, some people would set the rule of losing number of dice equal to the difference between the rank and the claim. For instance, if the claim is four 3s and there are actually seven 3s, the player who claimed would lose $7 - 4 = 3$ dice in one round. It speeds up the game. Moreover, some people would prefer everyone has rights to doubt once a claim is made, while here only the next player can choose to doubt.

How to set up the rules is purely depending on the taste. Here in the simulation, I only use the simplest version. Number 1 is not counted repeatedly, the claim is reacted by the next player, and each round only one player loses one dice.

4.4 Numerical Implementation

With consulting R code in [7], I implemented the reinforcement learning algorithm by myself in python. Firstly, a bunch of small functions need to be defined for later use.

I chose to store list of numbers a player has into a frequency table. It is clear and can easily use the number as index to search. *Combinations* can generates all the states we need. The *prob* function is used when dummy agents decide to doubt based on probability. As each of them know how many of the number claimed they have, it can be considered as a Binomial distribution. Suppose the claim is n of number a, one player has s of number a, then there should be at least $n - s$ of number a for the claim to be true. The probability of obtaining a certain number on a fair dice is $\frac{1}{6}$. Thus,

$$P(x \geq n - s) = \sum_{i=n-s}^{n} \binom{n}{i} \left(\frac{1}{6}\right)^{i} \left(\frac{5}{6}\right)^{n-i}$$

```
Useful Functions
  function ROLL_TABLE($rolls, Deep = false$)
      Generate a frequency table of rolled dice values. deep indicates if a list of list presents
      Input: List of rolled dice values.
      Returns: list, frequency table of rolled dice values.
  end function

  function PROB($x$)
      Calculate the probability that there is at least the bid quantity on the table. use later in dummy agent deciding
      Returns: probability that the claim is not bluffing
  end function

  function COMBINATIONS($ndice, length = 6$)
      Generate all combinations of values in a sequence of length n values, where the sum of each combination is number of dices. (for building memory structures)
      Returns: list of tuples
  end function
```

4.4.1 Non-agent Player

In order to train the agent properly, the dummy players should be dummy enough and have random behaviors. Our agent needs multiples steps of training to be smart enough to cope with smart players. Random behaviors ensure the agent encounters all states.

Make a Claim: When making claims, the dummy players have 3 choices, bluffing, honest and random. They will randomly choose one strategy to play.
1. Honest: The player would choose to claim about the number he or she has the most

and add 1 to the previously claimed rank. If 2 is the number one player has the most and the previous claim is $(4, 2)$, then the new claim is $(5, 2)$.
2. Bluffing: Adding one on the rank or number or both of the previous claim, without considering what they actually have. Equal chances are assigned to these three cases. If the previous claim is $(4, 3)$, bluffing can give $(5, 3), (4, 4), (5, 4)$.
3. Random: randomly giving a claim which obeys the rule.

Accept or Doubt: Dummy players have equal chance to play randomly or based on the probability. Random choice sets $(0.5, 0.5)$ probability of accept or doubt. If using the probability to decide, $P(Accept) = P(x \geq n - s)$, $P(doubt) = 1 - P(x \geq n - s)$.

Important Functions for each Round

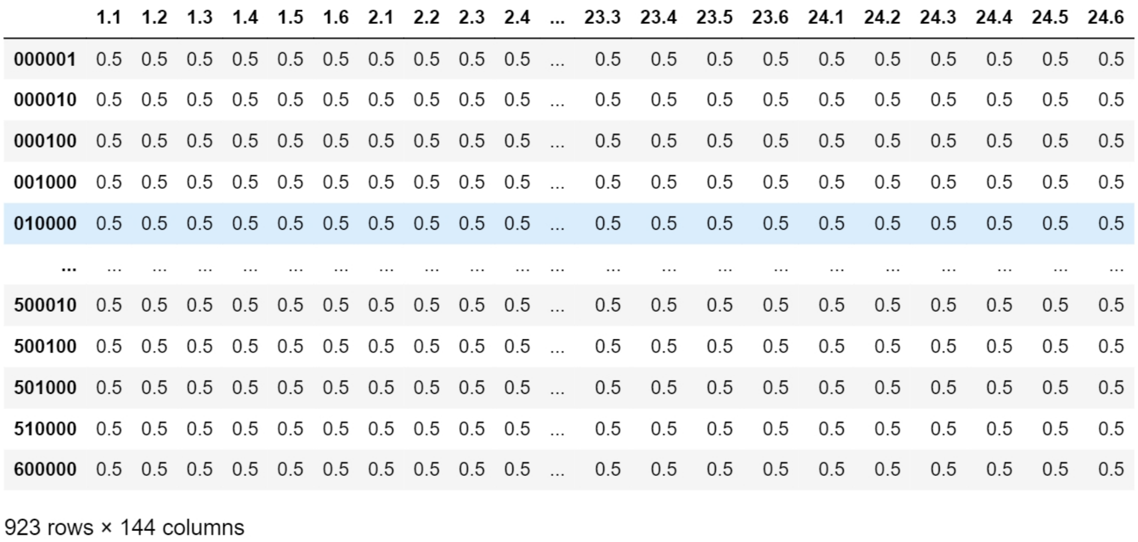
```
function playerClaim($dice\_truth, total, last\_claim = None$)
    Dummy agents make claim
    Input: $dice\_truth$: List, the numbers the player has
        $total$: total number of dices on the table
        $last\_claim$: [rank, number]
    Returns: A new claim, [rank, number]
end function

function decide($rolls, claim, total$)
    Dummy agents decide to accept or doubt. Firstly choose to randomly doubt or
based on probability.
    If random: equal probability of doubting and accepting;
    Else: based on probability
    Returns: True or False
end function

function Judge($Roll, claim$)
    judge if the challenge succeeds or not
    Input: $Roll$: all the rolled numbers on the table
        $claim$: the claim doubted
    Returns: True or False
end function
```

4.4.2 Q table

The Q table is initialised as pandas dataframe. Each row represents a state, where each state is the frequency table of numbers the agent has. For example, in fig 11, there are 4 players and each has 6 dice. 000100 means the agent only has one dice and it is 4. 301011 means the agent has not lost dice and holds numbers $1, 1, 1, 3, 5, 6$. The columns are corresponding to the claims. 17.3 means claimed rank is 17, number is 3, i.e. $(17, 3)$. Each entry $q_{i,j}$ of the Q table is the probability of doubting at state i when a claim j is made. In fact, each entry should be a list of $[P(accept),\ P(doubt)]$. Since $P(accept) = 1 - P(doubt)$, we can save computational costs by only updating the probability of $P(doubt)$.



	1.1	1.2	1.3	1.4	1.5	1.6	2.1	2.2	2.3	2.4	...	23.3	23.4	23.5	23.6	24.1	24.2	24.3	24.4	24.5	24.6
000001	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	...	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
000010	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	...	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
000100	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	...	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
001000	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	...	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
010000	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	...	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
500010	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	...	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
500100	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	...	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
501000	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	...	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
510000	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	...	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
600000	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	...	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

923 rows × 144 columns

Figure 11: Example of initial Q table (4 players, 6 dice each)

The entries in Q table is updated by Q function. The general Q function is

$$Q_{i+1}(s, a) = (1 - \alpha) Q_i(s, a) + \alpha \left(R\left(s' \mid s, a\right) + \gamma \max_a Q_i\left(s', a\right) \right)$$

s is the state, a is the action, $R\left(s' \mid s, a\right)$ is the reward of changing from s to s' given action a. α is the learning rate. It shows the importance of new messages. γ is discount factor, indicating how importance the future possible payoff is.

In our case, the column of Q table is not action but the claim j. The discount factor is dropped as it does not influence the learning much (by experiments). We can remind that each entry of Q matrix is the probability of doubting, so we want to emphasize on training when to doubt. Thus, the Q function is modified as

$$Q_{new}(s, j) = (1 - \alpha R\left(s' \mid s, a\right)) Q_{old}(s, j) + \alpha R\left(s' \mid s, a\right)$$

$$\Rightarrow \quad Q_{new}(s, j) = Q_{old}(s, j) + \alpha R\left(s' \mid s, a\right) \left(1 - Q_{old}(s, j)\right)$$

When the agent accept or doubt, it make decisions randomly with q_{ij} probability of doubting and $(1 - q_{ij})$ probability of accepting.

Functions about Q table

```
function Q_INITIALISE($ndice, nplayer$)
    build up initial Q table, each entry is filled as 0.5
    Returns: initial Q table
end function

function UPDATE($play\_memory, Q\_matrix, reward, alpha = 0.1$)
    use the records to update Q matrix each round and each play
    Input: $play\_memory$: play records
      $Q\_matrix$: current Q table
      $reward$: payoffs
      $alpha = 0.1$: learning rate
    Returns: updated Q table
end function

function AGENTDECIDE($Q\_mat, state, claim, total$)
    the smart agent chooses to doubt or accept
    Returns: True or False (doubt or not)
end function
```

With all these small functions, we can build up the whole algorithm. The rewards are set up in the functions of each round and play (see appendix). To win the whole play, the agent needs to challenge at proper time step in order to reduce the number of dice other players have, while the more important aim is to win. Consequently, the rewards are defined as: +10 if winning the game, -10 if losing the game; +1 if winning one round, -1 if losing one round.

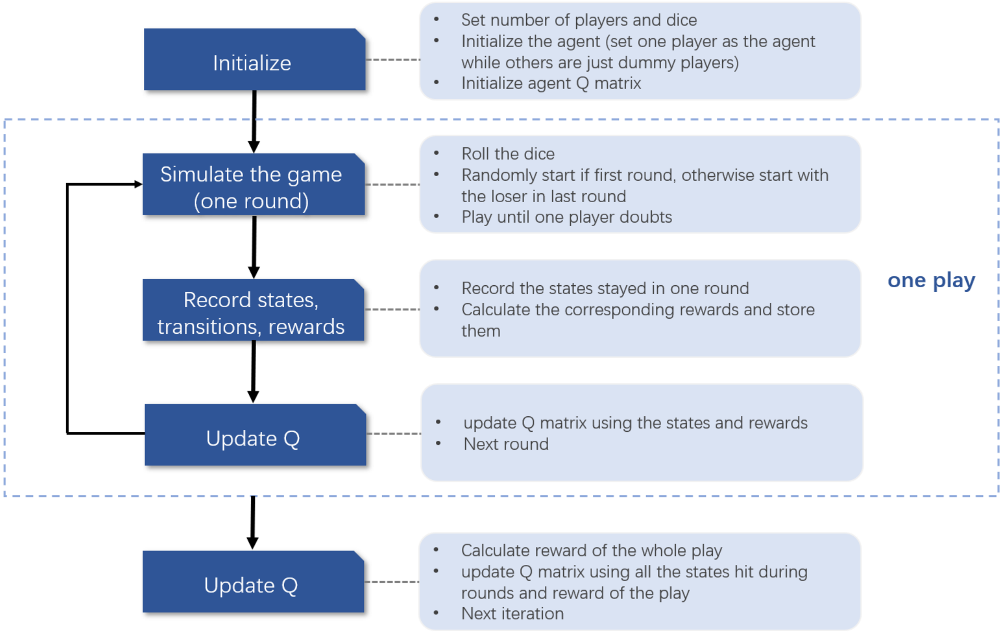

Initialize
Set number of players and dice
Initialize the agent (set one player as the agent while others are just dummy players)
Initialize agent Q matrix
Simulate the game (one round)
Roll the dice
Randomly start if first round, otherwise start with the loser in last round
Play until one player doubts
Record states, transitions, rewards
Record the states stayed in one round
Calculate the corresponding rewards and store them
one play
Update Q
update Q matrix using the states and rewards
Next round
Update Q
Calculate reward of the whole play
update Q matrix using all the states hit during rounds and reward of the play
Next iteration

Figure 12: process of whole algorithm

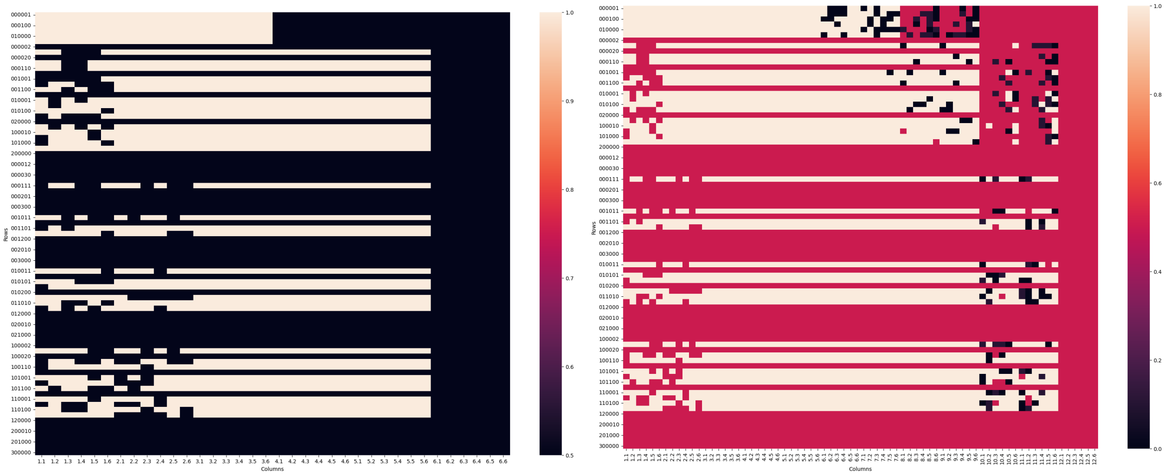
4.5 Numerical Results

4.5.1 Q Table

I ran a lot of simulations with the algorithm. The first simple case is 2 players with 3 dice each and the second case is 4 players with 3 dice each–different number of players with same total dice number. Fig 13 is contour graph of Q tables. The x axis and y axis corresponds to the states and claims. Although they are discrete, we can still observe some patterns here.

A good sign to notice is when the claimed rank reaches total dice number minus 1, the agent would doubt unless agent itself has 2 or 3 of the same number. Consequently, there was not enough training claimed rank equals total dice number. In 13a, where the players and number of dice are small, accepting is not a strategy we considered. Because if we accept for one or two times, the new claims we can made are often outrageous bluffing, which is too risky. Smart action is to doubt frequently. As described in 13b, when the agent has little dice in hand, it tends to be more careful and accept more, since it is harder to make a decision when less information in hand.

The graphs looks reasonable but incomplete. In next section, we will discuss the limitations and improvements.



(a) 2 players, each has 3 dice

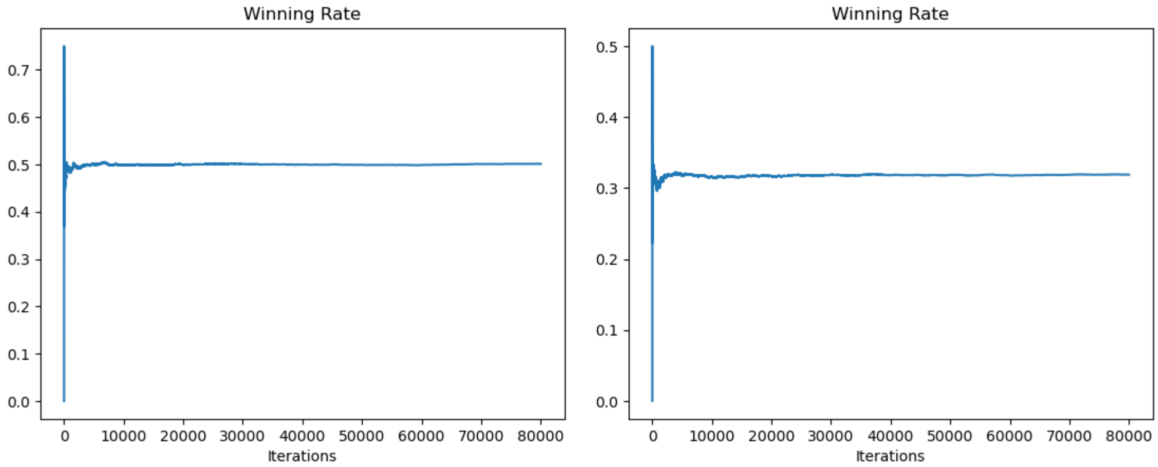
(b) 4 players, each has 3 dice

Figure 13

4.5.2 Winning Rate

The algorithm shows excellent winning rates. Initially, for a brainless child like our agent, there is no chances to win. If all players are equally dummy, the winning rate should be even. In fig 14b, the average winning rate of 4 players should be only 25%, but our agent achieves around $\frac{1}{3}$. The two-players game is quite tricky, it is easy to successfully challenge as little good claims are producing in the scheme. Comparing fig 14a and 15a, when more dice exist on the table, the agent has more information to adapt, it illustrates a better winning rate.

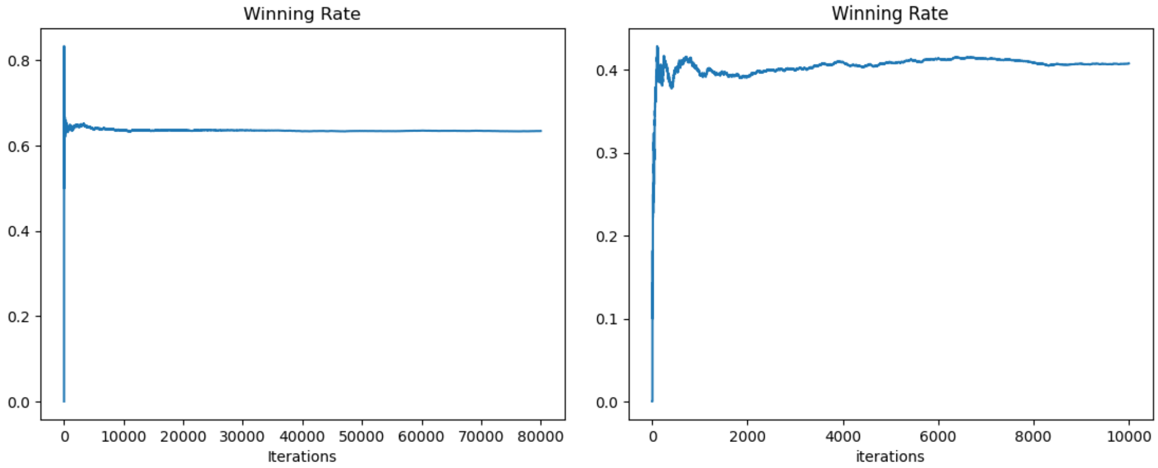
At last, I ran a massive system of 4 players, 6 dice each for 1000000 iterations (15b). With increasing number of total of dice and iterations, the winning proportion ascends $\frac{1}{3}$ than before (fig 14b and 13b). Overall, it is a successful training.



(a) 2 players, each has 3 dice

(b) 4 players, each has 3 dice

Figure 14



(a) 2 players, each has 6 dice

(b) 4 players, each has 6 dice

Figure 15

4.6 Limitations and Outlook

Although this project successfully trained the agent to play against dummy players, there could be some improvements for the algorithm and much more steps to train a mature agent. Fig 16 is a plot of 4 players, 6 dice for each game with running 1000000 iterations, the graph is still incomplete, many entries of the Q table have updated too little to be observed in the graph.

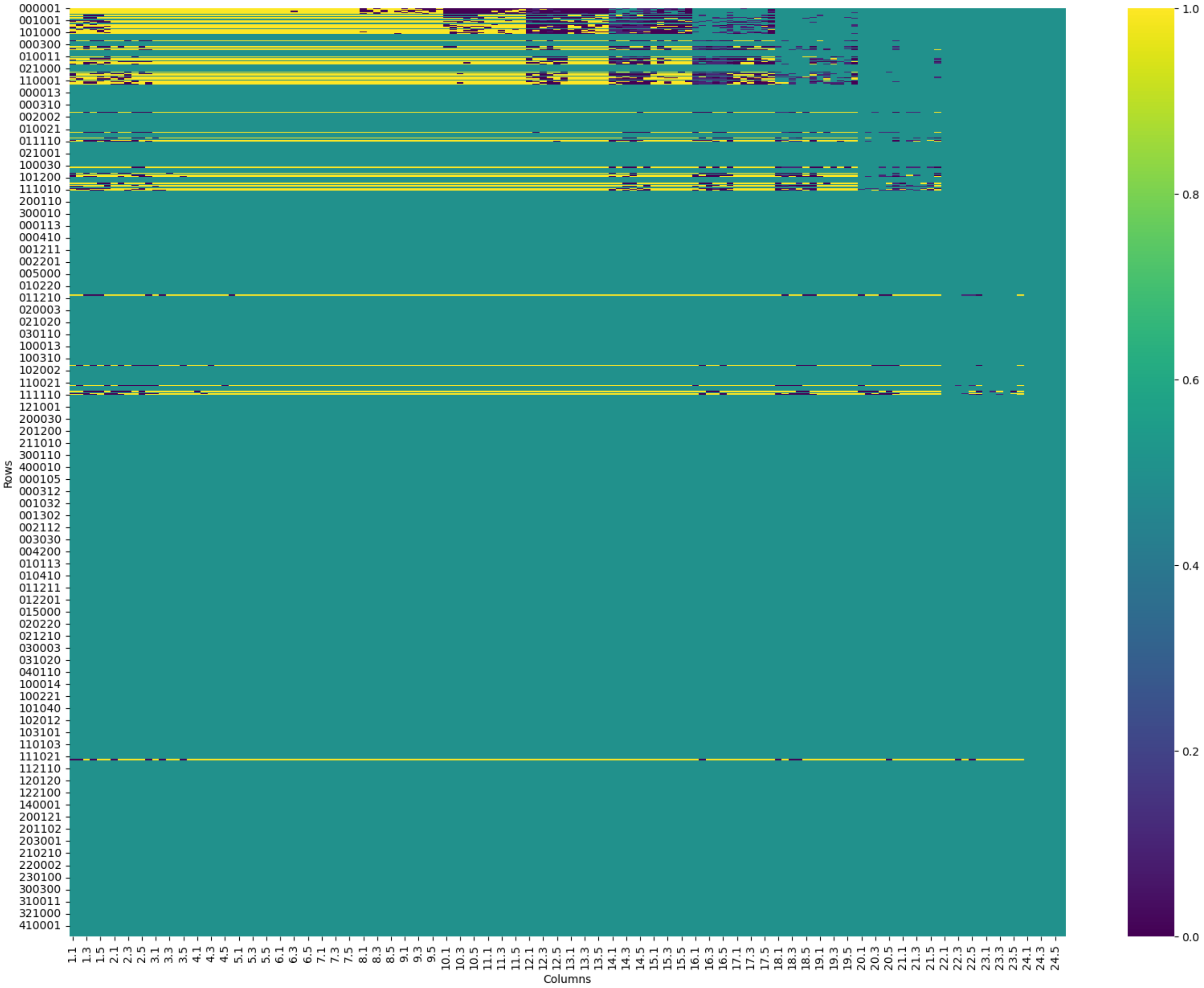


Figure 16

Computational Cost: As a matter of fact, the algorithm can be really time-consuming and of high computing costs. For a four-players, six dice game, there are $\left(\!\binom{4+6-1}{6-1}\!\right) \times 24 \times 6 = 132912$ entries in the Q table. To update all the entries in the table, we need at least $1e^7$ iterations. By numerical measuring, it requires days. The fig 16 took hours to run but there are lots of states that have not been touched. If we can come up with some methods to reduce the number of entries in the Q table, we can save a lot of computational costs. One idea is to only update probability of doubt as in the previous, but this is not enough. Another method can be considered is to add probability buckets[5]. According to the probability when the claim is true, we classify the table columns into certain number of buckets. It can reduce the number of entries largely.

Higher-Level Training: As the winning rate graphs show, we can observe the agent is doing a good job. If we are still greedy to improve the winning rate, the other players should be adapted smarter. Then the agent would obtain more valuable information to learn. More importantly, 1 dice Versus 1 dice Dudo is also a game about making claims. We can train the agent to make claims and decide alternatively. In this case, the agent would definitely be more wise.

Complicated Game Rules: The previous simulations only take simplest rules into account. That is because the agent needs to start with the easiest. If we take a further step, more and more complicated game rules can be applied when updating the Q table. We can consider number 1 as wild. This introduces uncertainty in doubting or accepting.

Acknowledgments

I would like to sincerely express my appreciation to my classmate 01705202 for the insightful discussions. The ideas shared proved to be inspiring for my independent work.

Appendix: Python Code

Rock, Paper, Scissors

In [ ]:

```
import random
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib.cm as cm
```

In [ ]:

```
def regret(a,b):
    '''
    a,b: choice of player A,B
    p: payoff
    return two lists of regret of A,B
    '''
    if a=='r':
        if b=='p':
            return np.array([0,1,2]),np.array([0,0,0])
        elif b=='s':
            return np.array([0,0,0]),np.array([1,2,0])
        else:
            return np.array([0,1,0]),np.array([0,1,0])

    if a=='p':
        if b=='s':
            return np.array([2,0,1]),np.array([0,0,0])
        elif b=='r':
            return np.array([0,0,0]),np.array([0,1,2])
        else:
            return np.array([0,0,1]),np.array([0,0,1])

    if a=='s':
        if b=='r':
            return np.array([1,2,0]),np.array([0,0,0])
        if b=='p':
            return np.array([0,0,0]),np.array([2,0,1])
        else:
            return np.array([1,0,0]),np.array([1,0,0])
```

In [ ]:

```
def RPS(initial, N, Memory = True):
    '''
    tol: tolerence
    initial: initial probability distribution
    '''
    # initial play
    strategy = ['r','p','s']
    R_A = np.zeros(3)  # cumulative regret of A
    R_B = np.zeros(3)  # cumulative regret of B
    Ap = initial     # initialise
    Bp = initial
    MemoryA = np.zeros((N,3))
    MemoryB = np.zeros((N,3))

    for i in range(N):
        MemoryA[i] = np.array(Ap)
        MemoryB[i] = np.array(Bp)
        Sa = random.choices(strategy, weights=Ap)[0]  # return a list, select first element
        Sb = random.choices(strategy, weights=Bp)[0]
        rA, rB = regret(Sa,Sb)
        R_A += rA
        R_B += rB
        if R_A.any() != 0.:
            Ap = R_A / np.sum(R_A)
        if R_B.any() != 0.:
            Bp = R_B / np.sum(R_B)
    if Memory:
        return MemoryA, MemoryB
    else:
        return Ap, Bp
```

In [ ]:

```
M = 100000
p1, p2= RPS([0.4,0.3,0.3],M)
P1, P2 = RPS([1,0,0],M)
print(p1[-1,:],p2[-1,:])
print(P1[-1,:],P2[-1,:])
```

Colonel Blotto

In [ ]:

```
Strategy = ['005','050','500',
            '014','041','104','140','401','410',
            '023','032','203','230','302','320',
            '113','131','311',
            '122','212','221']
```

In [ ]:

```
def CBpayoff(Sa,Sb):
    pa, pb = 0, 0   # initialise payoff
    for soldier1, soldier2 in zip(Sa,Sb):
        if soldier1 > soldier2:
            pa += 1
            # pb -= 0
        elif soldier1 == soldier2:
            pa += 0.5 # 0.5
            pb += 0.5 # 0.5
        else:
            # pa -= 0
            pb += 1

    return pa, pb
```

In [ ]:

```
def RegretV(sa, sb, n=21):
    '''
    sa, sb: strings of strategy
    strategy: all strategy
    return: two length n vectors, regret of A and regret of B
    '''
    # define
    strategy = ['005','050','500','014','041','104','140','401','410','023','032','203','230','302','320',
                '113','131','311','122','212','221']
    # returns regret vector
    regretA = np.zeros(n)
    regretB = np.zeros(n)
    # turn string to list
    S_a = [int(x) for x in sa]
    S_b = [int(x) for x in sb]
    payA, payB = CBpayoff(S_a, S_b)
    if payA > payB:
        for j, b in enumerate(strategy):
            Alter_b = [int(x) for x in b]
            new_pa, new_pb = CBpayoff(S_a, Alter_b)
            if new_pb > payB:
                regretB[j] = new_pb - payB
    elif payB > payA:
        for i, a in enumerate(strategy):
            Alter_a = [int(x) for x in a]
            new_pa, new_pb = CBpayoff(Alter_a, S_b)
            if new_pa > payA:
                regretA[i] = new_pa - payA
    else:
        for i, a in enumerate(strategy):
            Alter_a = [int(x) for x in a]
            new_pa, new_pb = CBpayoff(Alter_a, S_b)
            if new_pa > payA:
                regretA[i] = new_pa - payA
        for j, b in enumerate(strategy):
            Alter_b = [int(x) for x in b]
            new_pa, new_pb = CBpayoff(S_a, Alter_b)
            if new_pb > payB:
                regretB[j] = new_pb - payB

    return regretA, regretB
```

In [ ]:

```
def Blotto(initial, N, Memory = True):
    '''
    tol: tolerence
    initial: initial probability distribution
    '''
    # initial play
    strategy = ['005','050','500',
            '014','041','104','140','401','410',
            '023','032','203','230','302','320',
            '113','131','311',
            '122','212','221']
    R_A = np.zeros(21)  # cumulative regret of A
    R_B = np.zeros(21)  # cumulative regret of B
    Ap = initial
    Bp = initial
    MemoryA = np.zeros((N,21))
    MemoryB = np.zeros((N,21))

    for i in range(0,N,2):
        MemoryA[i] = np.array(Ap)
        MemoryB[i] = np.array(Bp)
        Sa = random.choices(strategy, weights=Ap)[0]  # return a list, select first element
        Sb = random.choices(strategy, weights=Bp)[0]
        rA, rB = RegretV(Sa,Sb)
        R_A += rA
        R_B += rB
        if R_A.any() != 0.:
            Ap = R_A / np.sum(R_A)
        if R_B.any() != 0.:
            Bp = R_B / np.sum(R_B)

        MemoryA[i+1] = np.array(Ap)
        MemoryB[i+1] = np.array(Bp)
        Sa = random.choices(strategy, weights=Ap)[0]  # return a list, select first element
        Sb = random.choices(strategy, weights=Bp)[0]
        rA, rB = RegretV(Sa,Sb)
        R_A += rA
        R_B += rB
        if R_A.any() != 0.:
            Ap = R_A / np.sum(R_A)
        if R_B.any() != 0.:
            Bp = R_B / np.sum(R_B)

    if Memory:
        return MemoryA, MemoryB
    else:
        return Ap, Bp
```

Liar Die

Modified on: \ SamFurman. Exploring counterfactual regret minimization for small imperfect information games. https://github.com/SamFurman/counterfactual_regret_minimization/pulse, 2019.

In [ ]:

```
import random
```

In [ ]:

```
class Node:
    def __init__(self,nactions):
        self.regretSum = [0.0] * nactions
        self.strategy = [0.0] * nactions
        self.strategySum  = [0.0] * nactions
        self.nactions = nactions
        self.pPlayer = 0.0
        self.pOpponent = 0.0
        self.utility = 0.0

    #get liar die node current mixed strategy with regret-matching
    def Strategy(self):
        normalizingSum = 0.0
        for a in range(self.nactions):
            self.strategy[a] = max(self.regretSum[a],0)
            normalizingSum += self.strategy[a]

        for a in range(self.nactions):
            if normalizingSum > 0:
                self.strategy[a] /= normalizingSum
            else:
                self.strategy[a] = 1.0 / self.nactions

        for a in range(self.nactions):
                self.strategySum[a] += self.pPlayer * self.strategy[a]

        return self.strategy

    #get liar die node average mixed strategy
    def AverageStrategy(self):
        average_strategy = [0.0] * self.nactions
        normalizingSum = sum(self.strategySum)
        for a in range(self.nactions):
            if normalizingSum > 0:
                average_strategy[a] = self.strategySum[a] / normalizingSum
            else:
                average_strategy[a] = 1.0 / self.nactions
        return average_strategy
```

In [ ]:

```
def Initialisation(sides):
    responseNodes = [[Node(2) for _ in range(sides + 1)] for _ in range(sides)]
    for myClaim in range(sides + 1):
        for oppClaim in range(myClaim + 1,sides + 1):
            responseNodes[myClaim][oppClaim] = Node(
                1 if ((oppClaim == 0) or (oppClaim == sides))
                else 2)
        claimNodes = [[Node(2) for _ in range(sides + 1)] for _ in range(sides)]
        for oppClaim in range(sides):
            for roll in range(1,sides + 1):
                claimNodes[oppClaim][roll] = Node(sides - oppClaim)
    return responseNodes, claimNodes
```

In [ ]:

```
def train(iterations, record_interval=100, sides=6, DOUBT = 0, ACCEPT = 1, Actions = 2):
    responseNodes, claimNodes = Initialisation(sides)
    strategy_history = []

    for iter in range(iterations):
        #Initialize rolls with starting probabilities
        rollAfterAcceptingClaim = [random.randint(1, sides) for _ in range(sides)]

        claimNodes[0][rollAfterAcceptingClaim[0]].pPlayer = 1
        claimNodes[0][rollAfterAcceptingClaim[0]].pOpponent = 1

        #accumulate realization weights forward
        for oppClaim in range(sides + 1):
            #Visit response nodes forward
            if oppClaim > 0:
                for myClaim in range(oppClaim):
                    node = responseNodes[myClaim][oppClaim]
                    actionProb = node.Strategy()
                    if oppClaim < sides:
                        nextNode = claimNodes[oppClaim][rollAfterAcceptingClaim[oppClaim]]
                        nextNode.pPlayer += actionProb[ACCEPT] * node.pPlayer
                        nextNode.pOpponent += node.pOpponent

            #Visit claim node forward
            if oppClaim < sides:
                node = claimNodes[oppClaim][rollAfterAcceptingClaim[oppClaim]]
                actionProb = node.Strategy()
                for myClaim in range(oppClaim+1,sides + 1):
                    nextClaimProb = actionProb[myClaim - oppClaim - 1]
                    if nextClaimProb > 0:
                        nextNode = responseNodes[oppClaim][myClaim]
                        nextNode.pPlayer += node.pOpponent
                        nextNode.pOpponent += nextClaimProb * node.pPlayer

        #Backpropagate utilities, adjust regrets and strategies
        for oppClaim in range(sides, -1, -1):
            regret = [0.0] * sides
            #visit claim nodes backward
            if oppClaim < sides:
                node = claimNodes[oppClaim][rollAfterAcceptingClaim[oppClaim]]
                actionProb = node.strategy.copy() # copy?
                node.u = 0.0
                for myClaim in range(oppClaim+1,sides + 1):
                    actionIndex = myClaim - oppClaim - 1
                    nextNode = responseNodes[oppClaim][myClaim]
                    childUtil = -nextNode.u
                    regret[actionIndex] = childUtil
                    node.u += actionProb[actionIndex] * childUtil
                for a in range(len(actionProb)):
                    node.regretSum[a] += node.pOpponent * regret[a]
                node.pPlayer = node.pOpponent = 0

            #visit response nodes backward
            if oppClaim > 0:
                for myClaim in range(oppClaim):
                    node = responseNodes[myClaim][oppClaim]
                    actionProb = node.strategy # copy?
                    node.u = 0.0
                    doubtUtil = 1 if oppClaim > rollAfterAcceptingClaim[myClaim] else -1
                    regret[DOUBT] = doubtUtil
                    node.u += actionProb[DOUBT] * doubtUtil
                    if oppClaim < sides:
                        nextNode = claimNodes[oppClaim][rollAfterAcceptingClaim[oppClaim]]
                        accept_util = nextNode.u
                        regret[ACCEPT] = accept_util
                        node.u += actionProb[ACCEPT] * accept_util
                    for a in range(len(actionProb)):
                        regret[a] -= node.u
                        node.regretSum[a] += node.pOpponent * regret[a]
                    node.pPlayer = node.pOpponent = 0
        #reset strategy sums after half of training
        if iter == iterations //2:
            for nodes in responseNodes:
                for node in nodes:
                        for a in range(len(node.strategySum)):
                            node.strategySum[a] = 0
            for nodes in claimNodes:
                for node in nodes:
                        for a in range(len(node.strategySum)):
                            node.strategySum[a] = 0

        # Record strategies at specified intervals
        if iter % record_interval == 0 or iter == iterations - 1:
            snapshot = {}
            for my_claim in range(sides + 1):
                for opp_claim in range(my_claim + 1, sides + 1):
                    snapshot[(my_claim, opp_claim)] = responseNodes[my_claim][opp_claim].AverageStrategy()
            strategy_history.append((iter, snapshot))


    #Print strategy
    for initialRoll in range(1,sides+1):
        print('Initial claim policy with role {0}'.format(initialRoll))
        print([f'{prob:0.2f}' for prob in claimNodes[0][initialRoll].AverageStrategy()])
    print('\nOld Claim\tNew Claim\tActionProbabilities')
    for myClaim in range(sides + 1):
        for oppClaim in range(myClaim + 1, sides + 1):
            print(f'\t{myClaim}\t{oppClaim}\t\t{responseNodes[myClaim][oppClaim].AverageStrategy()}')

    print('\nOld Claim\tRoll\tActionProbabilities')
    for oppClaim in range(sides):
        for roll in range(1,sides+1):
            print(f'{oppClaim}\t\t{roll}\t{claimNodes[oppClaim][roll].AverageStrategy()}')

    return responseNodes, claimNodes, strategy_history
```

1 Die Versus 1 Die Dudo

In [ ]:

```
import pandas as pd
import math
from itertools import product

# set random seed
random.seed(42)
```

In [ ]:

```
def roll_table(rolls, Deep=False):
    """
    Generate a table of rolled dice values.
    Args: rolls (list): List of rolled dice values.
    Returns: list, Table of rolled dice values.
    """
    if Deep == False:
        rt = np.bincount(rolls, minlength=7)[1:]
        return rt.tolist()
    else:
        m = len(rolls)
        rt = 0
        for i in range(m):
            rt += np.bincount(rolls[i], minlength=7)[1:]
        return rt.tolist()

def prob(x, bin_size=20):
    """
    Calculate the probability that there is at least the bid quantity on the table.
    Args:
    x (list): List of values [total dice number, player dice number, claim quantity, player has].
    bin_size (int): Size of the probability bucket (default is 20).

    Returns:
    int: Calculated probability.
    """
    if x[2] <= x[3]:
        return 1 * bin_size
    else:
        n = x[0] - x[1]
        prob = 0
        for k in range(min(x[2] - x[3], n), n + 1):
            prob += math.comb(n, k) * (1/6)**k * (5/6)**(n-k)
        return int(np.floor(prob*bin_size))

def Combinations(ndice, length = 6):
    """
    Generate all combinations of values in a sequence of length n_values
    where the sum of each combination is number of dices.
    return list of tuples
    """
    final = []
    for n in range(1,ndice+1):
        all_combinations = list(product(range(n + 1), repeat=length))
        result = [combo for combo in all_combinations if sum(combo) == n]
        final = final + result

    return final
```

In [ ]:

```
def playerClaim(dice_truth, total, last_claim=None):
    '''
    dice_truth: List, the numbers the player has
    total: total number of dices on the table
    last_claim: [amount, number]
    '''
    s = random.randint(1,3)
    t = random.randint(1,3)
    rt = roll_table(dice_truth)

    if last_claim == None:
        if s==1:  # bluffing
            number = random.randint(1,6)
            size = rt[number-1] + 1
        elif s==2:   # honest
            number = np.max(dice_truth)  # index
            size = rt[number-1]
        else:   # random
            number = random.randint(1,6)
            size = random.randint(1,total)

    else:
        A, B = last_claim
        if s==1:  # bluffing
            if B == 6:
                number = B
                size = A+1
            elif t == 1:
                number = B
                size = rt[number-1] + 1
            elif t == 2:
                number = B + 1
                size = A
            else:
                number = B + 1
                size = A + 1
        elif s==2:   # honest
            own_most = np.argmax(dice_truth)  # index
            number = own_most+1
            size = A + 1
        else:   # random
            if B == 6:
                number = B
                size = random.randint(A+1,total)
            elif t==1:
                number = random.randint(B+1,6)
                size = random.randint(A,total)
            elif t==2:
                number = random.randint(B,6)
                size = random.randint(A+1,total)
            else:
                number = random.randint(B+1,6)
                size = random.randint(A+1,total)

    return [size,number]
```

In [ ]:

```
def decide(rolls, claim, total):
    '''
    ft: numbers the player has
    '''
    ft = roll_table(rolls)
    startegy = random.randint(1,2)
    A, B = claim
    doubt = False
    probability = prob([total,sum(ft), A, ft[B-1]], bin_size=1)

    if startegy == 1:
        doubt = random.sample([True, False],1)
    else:
        doubt = np.random.choice([True, False], p=[1-probability,probability])

    return doubt

def Judge(Roll, claim):
    '''
    Roll: all the rolled numbers on the table
    claim: the claim doubted
    '''
    A, N = claim
    ft = roll_table(Roll, Deep=True)
    if A <= ft[N-1]:
        return False
    else:
        return True
```

In [ ]:

```
def Q_initialise(ndice, nplayer):
    '''
    simple Q table
    ndice: number of dice
    nplayer: number of players
    column: last_claim
    '''
    total_claim_number = ndice * nplayer
    row_label = Combinations(ndice)
    state_number = len(row_label)
    column_label = [f"{i}.{j}" for i in range(1, total_claim_number+1) for j in range(1, 7)]

    Q = np.zeros((state_number, total_claim_number*6)) + 0.5
    Q_df = pd.DataFrame(data=Q, index=row_label, columns=column_label)

    Q_df.index = pd.Index([''.join(map(str, x)) for x in row_label], dtype='object')
    return Q_df
```

In [ ]:

```
def update(play_memory, Q_matrix, reward, alpha=0.1):
    '''
    play_memory: "State", "Last claim"
    Q_matrix
    reward:
    '''
    Qcopy = Q_matrix.copy()
    length = len(play_memory)
    for i in range(length):
        s = play_memory[i][0]
        last_c = play_memory[i][1]
        l_c = last_c[0] + 0.1 * last_c[1]     # last claim
        sft = roll_table(s)         # state frequency table
        index = ''.join(map(str, sft))
        val = Qcopy.loc[index, "{}".format(l_c)]
        new_prob = val + alpha * reward * (1 - val)
        new_prob = new_prob.clip(0, 1)
        Qcopy.loc[index, "{}".format(l_c)] = new_prob

    return Qcopy
```

In [ ]:

```
def AgentClaim(dice_truth, total, last_claim):
    return playerClaim(dice_truth, total, last_claim)

def AgentDecide(Q_mat, state, claim, total):
    '''
    state: list of numbers
    claim: [amount, number]
    '''
    state_ft = roll_table(state)
    claim_label = claim[0] + 0.1 * claim[1]
    Dprob = Q_mat.loc[''.join(map(str, state_ft)), "{}".format(claim_label)]

    decision = np.random.choice([True, False], p=[Dprob, 1-Dprob])
    if claim[0] >= total:
        decision = True

    return decision
```

In [ ]:

```
def Round(nplayer, dice_count, control, a=1):
    '''
    When someone doubts, a new round starts
    control: input the number of initial player
    nplayers: number of players
    dice_counts: list of how many dice each player has
    '''
    # initialisation
    # roll the dice
    Roll = [sorted(random.sample(range(1, 7), dice_count[x])) if dice_count[x] != 0 else [0]*6 for x in range(n

    total = sum(dice_count)
    Doubt = False
    ctrl = []
    state = []
    action = []
    claim = None

    # no dice no play
    while dice_count[control-1] == 0:
        control = control % nplayer + 1

    # Make a Claim (Initial)
    if control != a:
        claim = playerClaim(Roll[control-1], total, claim)
    else:  # agent play
        claim = AgentClaim(Roll[a-1], total, claim)

    while Doubt == False:
        # no dice no play
        control = control % nplayer + 1
        while dice_count[control-1] == 0:
            control = control % nplayer + 1

        # Next player decide
        if control != a:
            Doubt = decide(Roll[control-1], claim, total)
            if Doubt == False:
                claim = playerClaim(Roll[control-1], total, claim)
                control = control % nplayer + 1
        else:  # agent decide
            Doubt = AgentDecide(Q, Roll[a-1], claim, total)
            if Doubt == False:
                claim = AgentClaim(Roll[control-1], total, claim)
                control = control % nplayer + 1
    return control, Roll, claim
```

In [ ]:

```
def Play(nplayer, ndice, Q_mat, control=1, a=1):
    # initiate
    dice_counts = [ndice] * nplayer
    # record for learning
    Agent_lose = 'unknown'
    columns = ["State", "Last claim"]  # Replace with your column names
    plays_record = []

    while any(dice_counts) != 0:
        # terminate and remove players
        if dice_counts[0] == 0:
            Agent_lose = True
            break

        # each round
        new_control, Roll_for_judge, claim_for_judge = Round(nplayer, dice_counts, control=control, a=a)
        control = new_control
        challenge = Judge(Roll_for_judge, claim_for_judge)
        if new_control == a:
            # setting reward
            if challenge == True:
                reward = 1
            else:
                reward = -1
            # new_data = pd.DataFrame({"State": [Roll_for_judge[a-1]], "Last claim": [claim_for_judge]})
            new_data = [[Roll_for_judge[a-1],claim_for_judge]]
            plays_record += new_data
            # update for once
            Q_mat = update(new_data, Q_mat, reward)
        if challenge == True:
            if dice_counts[new_control-2] != 0:
                dice_counts[new_control-2] -= 1
        else:
            if dice_counts[new_control-1] != 0:
                dice_counts[new_control-1] -= 1

    # Update Q for whole play
    if Agent_lose == True:
        reward = -10
        Q_mat = update(plays_record, Q_mat, reward)
    else:
        reward = 10
        Q_mat = update(plays_record, Q_mat, reward)

    if Agent_lose == True:
        return 0, Q_mat
    else:
        return 1, Q_mat
```

In [ ]:

```
Q = Q_initialise(6,4)
L = []
M = []

# iterate
for i in range(1000000):
    a, Q = Play(4,6,Q)
    L.append(a)

# win rate
Rate = []
tol = len(L)
for i in range(tol):
    r = sum(L[:i])/(i+1)
    Rate.append(r)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

References

[1] Dr Sam Brzezicki. Introduction to game theory math60141/math70141. Lecture Notes, Autumn Term 2023. Imperial College London.

[2] Sergiu Hart. Discrete colonel blotto and general lotto games. *International Journal of Game Theory*, 36(3):441–460, 2008.

[3] Sergiu Hart and Andreu Mas-Colell. A simple adaptive procedure leading to correlated equilibrium, Jan 1, 2016.

[4] Todd W. Neller and Marc Lanctot. An introduction to counterfactual regret minimization.

[5] Daniel Oehm. Liar's dice in r. Nov 24, 2018.

[6] Daniel Oehm. Q-learning example with liar's dice in r, Nov 20 2018.

[7] Daniel Oehm. liars-dice. https://github.com/doehm/liars-dice, 2020.

[8] SamFurman. Exploring counterfactual regret minimization for small imperfect information games. https://github.com/SamFurman/counterfactual_regret_minimization/pulse, 2019.