Dynamics of Games Project
Regret Minimalization of Colonel Blotto Game

Pobpawat Pordi
CID: 01192761

January 9, 2020

Contents

1 Introduction 4

2 Colonel Blotto 5
2.1 The Colonel Blotto Game 5
2.2 Symmetric Colonel Blotto 5
2.2.1 Nash Equilibria 6

3 Regret Minimalization 8
3.1 Motivation 8
3.2 Regret 8
3.3 Regret Matching 8
3.4 Minimal Regret Strategy is the average historical strategy . . 9

4 Implementation of Regret Minimalization Algorithm and Other Utilities 10
4.1 Differences to the original paper, design choices and extensibility 10
4.2 Game Class 10
4.3 Player Class 10
4.4 Trainer Class 11
4.4.1 Graphs and Animations 11
4.5 Testing the Implementation with Rock Paper Scissors 11
4.5.1 Finding best response against a fixed strategy 11
4.5.2 Finding Nash Equilibrium via self-play 12
4.5.3 Convergence of Strategies, Utilities 13

5 Results of Algorithm 15
5.1 Symmetric Blotto Game 15
5.1.1 Two-Player Regret Matching towards specific Equilibria 15
5.1.2 Evolutionary Stable Strategies of Symmetric Colonel Blotto 15
5.2 Original Colonel Blotto Game 17
5.2.1 Nudging Colonel Blotto towards predicted Equilibria . 17
5.2.2 Asymmetric probabilities between symmetric actions . 17

6 Arguments behind the Algorithm 17

A Graphs and Plots 18
A.1 Symmetric Blotto 18
A.2 Colonel Blotto 19
A.2.1 Nudging towards predicted equilibria 20

B Code 21
B.1 Games 21
B.1.1 Colonel Blotto 21
B.1.2 Symmeteric Blotto 22
B.1.3 Rock Paper Scissors 23

B.2 Players 24
B.2.1 OneShotPlayer 24
B.2.2 Regret Matching Player 24
B.2.3 Fixed Strategy Player 27
B.3 Trainer 28
B.4 Example of main program 33

1 Introduction

This paper will focus on the task described by Project 3 of the Dynamics of Games offered by the Department of Mathematics at Imperial College.

First, we will take a brief look at the game of Colonel Blotto described in TODO (specifically, the case of 5 armies and 3 battlefields) and form some predictions about the any equilibria the game may have. In section 4 we will then describe our customized implementation of the regret-matching algorithm from the Neller-Lancot paper, as well as other tools and utilities that are of use in analysis of iterated one-shot games. Section 6 will describe the results of the Colonel Blotto game achieved by the algorithm.

2 Colonel Blotto

2.1 The Colonel Blotto Game

The Colonel Blotto game is a two-player, one-shot, zero-sum game revolving around the idea of two opposing players each having a discrete armies and having to split them up to fight on b different battlegrounds. Each battlegrounds is won by a player if and only if they have sent a larger force to that battleground, a draw if both players have sent the same number of armies, and a loss otherwise. The player who has won more battlegrounds wins the game, and the other loses. A draw is possible if neither player has won more battlegrounds than the other. The utility for a win, draw, and loss is 1, 0, and -1 respectively.

For example, in the case of each player having 4 armies and 3 battlefields, player I may choose to split their armies as $(2,2,0)$ - meaning 2 armies to the firt battlefield, 2 armies to the 2nd battlefield, and no armies to the third battlefield. Player II may choose to utilise their army via $(0,1,3)$. The result of the game is that player I's armies are victorious in the first two battlefields, and player II only wins the third battleground, therefore player I wins and is awarded a utility of 1, and player II gains a utility of -1.

Analysis of this game is a tricky, for mainly for two reasons:

1. The number of possible actions for a armies and b battlefields is $\frac{(a+b-1)!}{a!(b-1)!}$, meaning the size of the payoff matrix grows very quickly with a and b , and even for small cases such as $a = 4, b = 2$ have a $10x10$ payoff matrix.

2. Small cases which are feasable to analyse by hand are uninteresting. For example, with $a = 1, b = 2$, the payoff matrix is actually a 2x2 matrix of zeroes, as neither player can win. In fact for all cases of $b = 2$ neither player can ever win.

2.2 Symmetric Colonel Blotto

Note that the actions in any Colonel Blotto game are *symmetric.* That is, all actions have corresponding actions such that the payoffs are equivalent. For example, (a_1, a_2, a_3) has the same payoff against (b_1, b_2, b_3) as (a_2, a_1, a_3) has to (b_2, b_1, b_3), and so forth.

We may then first attempt to analyze a simpler, symmetric version of the Colonel Blotto game, called the *Symmetric* Colonel Blotto, and hope that the results translate to the original version. Here, we group actions of the original Colonel Blotto game together by their permutative equivalences, e.g. an action $a_1a_2a_3$ represents all of its possible permutations in the original Colonel Blotto game. We assume each permutation of an action is equally likely due to symmetry, therefore we can calculate the payoff matrix of the Symmetric Colonel Blotto game with 3 armies and 5 battlefields.

	005	014	023	113	122
005	0	$-\frac{1}{3}$	$-\frac{1}{3}$	-1	- 1
014	$\frac{1}{3}$	0	0	$-\frac{1}{3}$	$-\frac{2}{3}$
023	$\frac{1}{3}$	0	0	0	$\frac{1}{3}$
113	1	$\frac{1}{3}$	0	0	$-\frac{1}{3}$
122	1	$\frac{2}{3}$	$-\frac{1}{3}$	$-\frac{1}{3}$	0

Figure 1: Payoff for Symmetric Colonel Blotto

Each element of the matrix is the expected payoff of any (uniform) random permutation of the row action against any (uniform) random permutation of the column action. In comparison to the 21x21 matrix of the (5, 3) instance of the original game, this 5x5 matrix is much simpler to analyze.

2.2.1 Nash Equilibria

Recall the definition of Nash Equilibria in a 2-player game:

$$\begin{aligned} x \cdot A\hat{y} &\leq \hat{x} \cdot A\hat{y} \\ y \cdot B\hat{x} &\leq \hat{y} \cdot B\hat{x} \end{aligned} \tag{1}$$

or equivlalently in terms of *best responses*

$$\begin{aligned} \hat{x} &\in \mathcal{BR}_A(\hat{y}) \\ \hat{y} &\in \mathcal{BR}_B(\hat{x}) \end{aligned} \tag{2}$$

where $\mathcal{BR}_A(y) = \{x^* \in \Delta \mid x^* \cdot Ay \geq x \cdot Ay \ \forall x \in \Delta\}$.

As we have a symmetric, zero-sum game here, we use only one payoff matrix A.

Note that in the payoff matrix, the first two rows always perform worse than the bottom 3 rows, regardless of the column - i.e. actions `005` and `014` are dominated by the other actions. Hence no Nash Equilibria would involve those two actions, and we can consider the sub-matrix of the last 3 actions (with the payoffs normalized).

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ -1 & 1 & 0 \end{bmatrix} \tag{3}$$

For the above matrix, there are 4 Nash Equilibria:

$$\hat{x} = (1,0,0)^T,\ \hat{y} = (1,0,0)^T \tag{4}$$

$$\hat{x} = (1,0,0)^T,\ \hat{y} = (\tfrac{1}{2},\tfrac{1}{2},0)^T \tag{5}$$

$$\hat{x} = (\tfrac{1}{2},\tfrac{1}{2},0)^T,\ \hat{y} = (1,0,0)^T \tag{6}$$

$$\hat{x} = (\tfrac{1}{2},\tfrac{1}{2},0)^T,\ \hat{y} = (\tfrac{1}{2},\tfrac{1}{2},0)^T \tag{7}$$

To show that the above are indeed Nash Equilibria, we show that each $\hat{x}$ we and $\hat{y}$ are each others' best reponses.

$$(1,0,0)A(1,0,0)^T = 0 = \max_x\ x \cdot A(1,0,0)^T \tag{8}$$

$$\begin{aligned}(1,0,0)A(\tfrac{1}{2},\tfrac{1}{2},0)^T &= 0 = \max_x\ x \cdot A(\tfrac{1}{2},\tfrac{1}{2},0)^T \\ (\tfrac{1}{2},\tfrac{1}{2},0)A(1,0,0)^T &= 0 = \max_x\ x \cdot A(1,0,0)^T\end{aligned} \tag{9}$$

$$(\tfrac{1}{2},\tfrac{1}{2},0)A(\tfrac{1}{2},\tfrac{1}{2},0)^T = 0 = \max_x\ x \cdot A(\tfrac{1}{2},\tfrac{1}{2},0)^T \tag{10}$$

Hence we may speculate that in the original Colonel Blotto game, there may be Nash Equilibria where each player has either one of the following mixed strategies:

1. All permutations of `023` are equally likely with probability 1/6, and all other actions have 0 probability (corresponding to $(1,0,0)$ of the symmetric version).
2. All permutations of `023` are equally likely with probability 1/12, and all permutations of `113` are equally likely with probability 1/6.

We can also be confident that any equilibria will not involve non-zero probabilities for any actions that are not permutations of `023` or `113`

3 Regret Minimalization

3.1 Motivation

Consider an iterative series of plays of a *one shot* game. Suppose the game has utility function u and that there are n_1 actions $a_1, a_2 ... a_{n_1}$ for player I, and similarly n_2 actions $b_1, b_2 ... b_{n_1}$ for player II. Player I picks action a_i and player II picks b_j. After the game, as both players are aware of the utility function, player I can calculate their utilities had they picked any other action $a_{i'}, i' \neq i$. If this alternative utility is greater than the utility actually achieved by playing then we can say that player I *regrets* not having played $a_{i'}$ instead of a_i.

3.2 Regret

More concisely, we define the *regret* of not having chosen an action the to be the difference in utilities of that action against the opposite player's already-chosen action, and the utility of the action that was chosen.

$$r(a, t) = u(a, b^t) - u(a^t, b^t) \tag{11}$$

Where a is an action, a^t and b^t are the actions chosen by players I and II at time t. Positive regret can be interpreted as "I should have played that instead", negative regret as "I'm glad I didn't play that", and 0 regret as there being no change in outcome had that action been picked instead.

3.3 Regret Matching

This forms the basis of the regret-matching algorithm that we will extensively use in this paper - the idea that a player can select future moves by drawing from a probability distribution proportional to their all their *positive* regrets (moves which they wish they had previously played).

The basic `regret matching` algorithm can be described as follows:

- Initialize the vector `regret_sum` (of length equal to the number of actions) of regrets to 0.
- After one iteration of the game, the player has picked action a_i, against the opponent's action b. The player calculates the regret of *all* their actions as a vector, and adds it to their `regret_sum`.
- In order to pick the player's next action, they sample from the discrete probability distribution that is proportional to their positive regrets. For example, a `regret_sum` of $(-1, 1, 2)$ would yield discrete probabilities of $(0, 1/3, 2/3)$. Note that the choice of only using positive regrets can be conceptualized as not wanting to pick actions that the player has negative regret i.e. is happy they never picked it.

3.4 Minimal Regret Strategy is the average historical strategy

However, probability distributions of actions at any time t may be highly erratic and/or skewed. For instance, regrets of $(0, -1, 1)$ automatically imply that the third action is chosen with 100% certainty the next game, despite their regrets being so close. In 4.5.2 we will further illustrate that this issue can happen at any time in the game, not necessarily just at the first few iterations.

The minimal regret strategy can instead be found in the average strategy across all iterations:

$$P(t) = \frac{1}{t}\sum_{i=1}^{t+1} p(i) \tag{12}$$

which is merely the mean of all the action probability distributions at each iteration of the game.

To keep track of this, we add some modifications to the previously described algorithm in a few ways:

- We keep track of the sum of all previous probability distributions by using a vector variable `strategy_sum`, initialized to all 0s.
- At every iteration when we compute a new probability distribution of actions, we add that to `strategy_sum`.
- At each iteration, the added probability distribution sums to 1, therefore the minimal regret strategy is the normalization of `strategy_sum`.

For a simple but fully implemented example please refer to TODO for a complete walkthrough of a Java-style example of Rock Paper Scissors.

4 Implementation of Regret Minimalization Algorithm and Other Utilities

4.1 Differences to the original paper, design choices and extensibility

The underlying logic of the main algorithm described in the previous section and the original Neller-Lancot paper remain the same. However, plenty of changes have been made in order to facilitate more utilities and usages outside of the particular task stated in the project description of (paraphrasing) having 2 regret-matching players against each other on the (5,3) instance of Colonel Blotto. Great attention was paid to maximizing the usefulness of this code for more general cases of iterative one shot games.

4.2 Game Class

The `game` interface is for classes representing games, and holds two pieces of information:

- The possible actions of each player. Called by `get_actions` method.
- The utility function, to calculate the utilities of each player given their moves. Called by `get_utility` function.

All `game` classes are stateless with respect to any games played and do not keep any history whatsoever.

4.3 Player Class

The `player` interface is for classes representing players/agents, and holds the following functionalities:

- `initialize` is a method called to initialize the player with a specific `game`.
- `get_action` selects an action for the game.
- `post_play` informs the player of the outcome of a game so their history and/or strategy can be updated accordingly.
- `get_strategy` returns the player's current mixed strategy.
- `get_average_strategy` returns the player's average strategy over their lifespan.

All player implementations are also subclasses of `OneShotPlayer`, which keeps track of the total number of games played, as well as cumulative and average utilities.

There are two `player` classes, `FixedPlayer` and `RegretMinPlayer`. As one may deduce from the names the former is a player adhering to a fixed strategy - by default it is all actions equally, but the strategy can be changed simply by setting the player's `strategy` field. The latter is the implementation of the Regret Matching Algorithm described in the previous section, being very similiar to the Rock-Paper-Scissors example from the TODO paper, but 1.) not being directly coupled with any particular game and 2.) taking advantage of existing `numpy` features.

4.4 Trainer Class

The `trainer` is what glues everything together: given two `player` objects and an instantiated `game` object, we can run the iterative series of plays between both players.

The most simple way to instantiate everything and run the algorithm is as follows:

```
game = RockPaperScissorsGame()

player_a = RegretMinPlayer()
player_b = RegretMinPlayer()

trainer = OneShotTrainer(player_a, player_b, game)

trainer.train(1000)
trainer.report()
```

Other examples and usages may be found in the Appendix.

4.4.1 Graphs and Animations

Included in this class are also many utilities for plotting various graphs and animations to visualize strategies, average strategies, regrets, and convergence over time.

4.5 Testing the Implementation with Rock Paper Scissors

We gather some confidence that our implementation of the algorithm works by testing it with an already well-known game: Rock Paper Scissors.

The `main` scripts used to generate all the results below can be found in the Appendix.

4.5.1 Finding best response against a fixed strategy

First, we check that Regret Matching player can successfully minimise regret against a player with a fixed strategy - specifically, the fixed pure strategy of only playing *rock* (therefore, the Regret Matching player should have an average strategy converging to only *paper*).

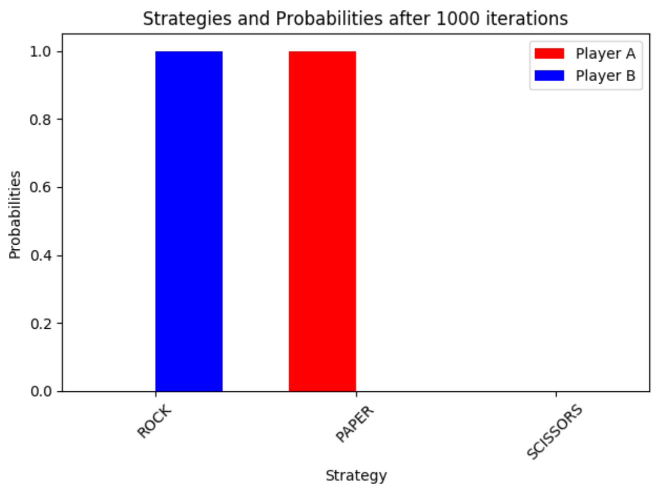
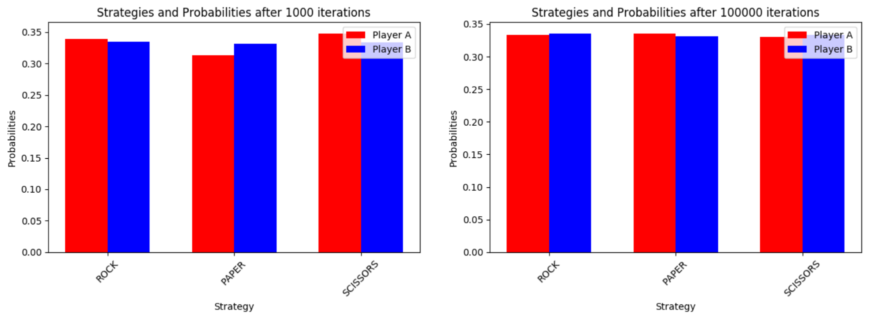


Figure 2: Regret Matching against a fixed pure Rock strategy

The regret matching player (Red / Player A) successfully identifies the best response is $(0, 1, 0)$ within a 1000 iterations.

4.5.2 Finding Nash Equilibrium via self-play

We can now try Exercise 2.5 from TODO: having both players use regret matching against each other in the game of Rock Paper Scissors.



(a) 1k iterations (b) 100k iterations

Figure 3: Regret Matching player vs Regret Matching player in RPS

After 1000 iterations, we can see that the average strategies starts to look like the expected Nash Equilibrium. However, we can also see both strategies are still rather rough-looking and not quite there yet. After 100k iterations, the average strategies starts looking much smoother and closer to $(1/3, 1/3, 1/3)$.

After the end of 100k iterations, both players report:

```
Player A:
Average Utility: -0.00043
Cumulative Utility: -86
Games Played: 200000
```

```
Final Regrets: [245, 105, -92]

Player B:
Average Utility: 0.00043
Cumulative Utility: 86
Games Played: 200000
Final Regrets: [-498, 163, 77]
```

Note that both players' regrets are not remotely near being uniform across all the actions - player A is currently skewed towards picking *rock* for the near future, whilst player B is biased towards *paper*. This illustrates the idea that the per-iteration strategies are highly volatile and erratic from 3.4.

Also note that the `Average Utility` is ≈ 0 for both players - **an important characteristic of symmetric, zero-sum games is that any equilibria they have must have expected payoff of 0 for both players.**

4.5.3 Convergence of Strategies, Utilities

How do we know that a strategy has converged? In the Rock-Paper-Scissors, we *assumed* that the Regret Matching algorithm was converging because we knew $(1/3, 1/3, 1/3)$ is a Nash Equilibrium. For games where we are unsure of their NE, this isn't good enough - if we didn't know the NE of RPS we would have had no way of knowing that the algorithm has converged, or that it wouldn't later diverge or converge to something else given more iterations.

We need a way of visualizing the change of average strategy (and regrets, utilities, etc) with respect to *time*, hence a few different methods were developed

4.5.3.1 Plotting Average Strategy Deltas between Intervals of Iterations

We can calculate the differences in average strategy every `n` iterations and plot them on a graph, like the following:

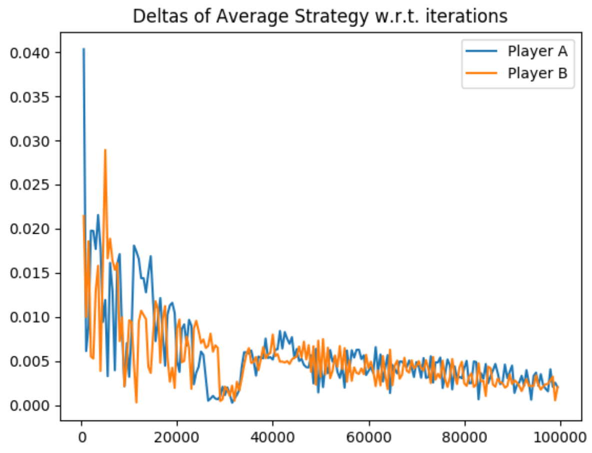


Figure 4: Average Strategy Deltas Every 1000 iterations for RPS

4.5.3.2 Plotting Norm of Average Regret Vector

To achieve the minimum-regret set in equilibria, the average sum of positive regrets across all iterations must converge to a minimum.

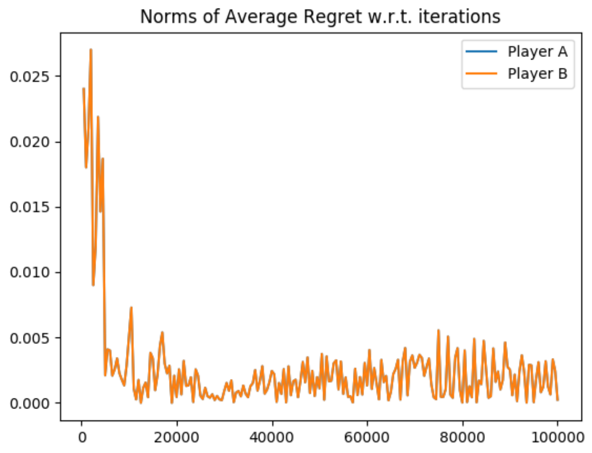


Figure 5: Average Strategy Deltas Every 1000 iterations for RPS

4.5.3.3 Average Strategy Animations

For a more visually appealing approach, we can generate animations of how average strategy changes as the number of iterations increases. Links to GIFs of these animations will be provided in the Appendix.

5 Results of Algorithm

5.1 Symmetric Blotto Game

We run two-player Regret Matching on the Symmetric Blotto Game to test our predictions from 2.2.

Our first run yields the graph in 7 (here, we include all 5 possible actions, so $(1, 0, 0)$ from 2.2 becomes $(0, 0, 1, 0, 0)$. Unsurprisingly, we get the both-player $(0, 0, 1, 0, 0)$ (henceforth called NE_1) strategy equilibrium quite easily.

Running the same same algorithm multiple times (in this instance) always yields this same equilibrium. Is there any way to get the other equilibria as well?

5.1.1 Two-Player Regret Matching towards specific Equilibria

The first idea might be to *"nudge"* both players' regrets (and therefore strategies) towards what we know/predict to be an equilibrium. To do this, we can manually set each players' `regret_sum` fields prior to training to be strongly favoured towards that certain equilibrium. Here we use regrets of `[-10, -10, 10, 10, -10]` for both players. The results of this can be seen in the **??**.

Unfortunately, our idea does not quite seem to work. Even trying with some other starting regrets (and similar numbers of iterations), the two players either:

- Converge to $(0, 0, 1, 0, 0)$
- Do not converge but behave erratically/slows down/cycles around $(0, 0, 1/2, 1/2, 0)$

Convergence to the desired $(0, 0, 1/2, 1/2, 0)$ NE (now called NE_2) is achievable only via *very* heavily skewing the initial regrets (to magnitudes of 1000+) and allowing for very large number of iterations (10-100k+ compared to the first NE's 1k). Compared to NE_1, even though the algorithm runs non-deterministically, it is able to achieve NE_1 most of the time with just 1k runs.

5.1.2 Evolutionary Stable Strategies of Symmetric Colonel Blotto

Recall the notions of *Evolutionary Stable Strategies* for 1-player games. $\hat{x}$ is said to be an ESS if:

$$x \cdot A(\epsilon x + (1 - \epsilon)\hat{x}) < \hat{x} \cdot A(\epsilon x + (1 - \epsilon)\hat{x}) \tag{13}$$

The intuition behind ESS is that if $\hat{x}$ experiences a small pertubation (e.g. a small portion of the population adopting the new strategy), then $\hat{x}$ will still be the preferable strategy for any individual to have against the pertubed entire population. In simpler terms, *is $\hat{x}$ the uniquely optimal strategy against other strategies near itself?*

5.1.2.1 2-Player Evolutionary Stable Strategies

We extend the concept of ESS to 2-player games in order to analyze the difficulty in converge of other equilibrium that aren't NE_1.

We define the evolutionary stable strategies $\hat{x}$, $\hat{y}$ of a 2-player game to be:

$$x \cdot A(\epsilon y + (1-\epsilon)\hat{y}) < \hat{x} \cdot A(\epsilon y + (1-\epsilon)\hat{y}) \tag{14}$$

$$y \cdot B(\epsilon x + (1-\epsilon)\hat{x}) < \hat{y} \cdot A(\epsilon x + (1-\epsilon)\hat{x}) \tag{15}$$

for $x \neq \hat{x}$ and $\epsilon > 0$ small enough.

This can be interpreted as $\hat{x}$ still being the uniquely optimal strategy against a small change in $\hat{y}$, and vice versa. If an NE is not an ESS, then a small perturbation in one player's strategy will also cause the other player to move away from the NE - and given our algorithm works via lots of random perturbations, this would explain the behaviour seen in the previous section.

5.1.2.2 ESS of the Symmetric Blotto Game

Predictably, NE_1 is an ESS. Proof:

$$\begin{gathered} A(\epsilon y + (1-\epsilon)\begin{bmatrix}1\\0\\0\end{bmatrix}) = \begin{bmatrix}\epsilon y_3\\ -\epsilon y_3\\ -1+\epsilon-\epsilon(y_1-y_2)\end{bmatrix} \\ \arg\max_{x\in\Delta} x \cdot \begin{bmatrix}\epsilon y_3\\ -\epsilon y_3\\ -1+\epsilon-\epsilon(y_1-y_2)\end{bmatrix} = \begin{bmatrix}1\\0\\0\end{bmatrix} \end{gathered} \tag{16}$$

NE_2 is not an ESS:

$$\begin{gathered} A(\epsilon y + (1-\epsilon)\begin{bmatrix}1/2\\1/2\\0\end{bmatrix}) = \begin{bmatrix}\epsilon y_3\\ -\epsilon y_3\\ -\epsilon(y_1-y_2)\end{bmatrix} \\ \arg\max_{x\in\Delta} x \cdot \begin{bmatrix}\epsilon y_3\\ -\epsilon y_3\\ -\epsilon(y_1-y_2)\end{bmatrix} \neq \begin{bmatrix}1/2\\1/2\\0\end{bmatrix} \end{gathered} \tag{17}$$

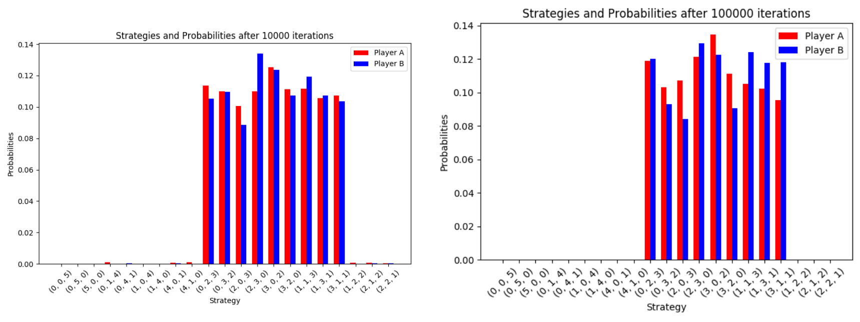
And from the last line of our NE_1 proof we also have

$$\arg\max_{x\in\Delta} x \cdot \begin{bmatrix}\epsilon y_3\\ -\epsilon y_3\\ -1+\epsilon-\epsilon(y_1-y_2)\end{bmatrix} \neq \begin{bmatrix}1/2\\1/2\\0\end{bmatrix} \tag{18}$$

Therefore only NE_1 is our only ESS - which explains why our algorithm seems to have great preference to converging to that equilibrium!

5.2 Original Colonel Blotto Game

Now, we *finally* get around to running the two-player regret matching algorithm with the original Colonel Blotto Game. Core, important graphs and data will be shown here whilst supplementary graphs can be found in the Appendix.



(a) 10k iterations (b) 100k iterations

Figure 6: Regret Matching player vs Regret Matching player in Colonel Blotto

On first look, it appears as though our algorithms are not converging. However, the convergence graphs generated seem to show a good convergence rate - and running it for 10 million iterations (see appendix) gets similar results.

These results are neither of the equilibria we expected from our predictions in 2.2. However, using similar "nudging" techniques from previous sections, we can try to achieve those two equilibria.

5.2.1 Nudging Colonel Blotto towards predicted Equilibria

Using initial regret weights of 1000 for permutations of `023` and -100 for all other actions (which is small relative to 100k iterations), we successfully achieve *bot* NE we predicted. Refer to A.2.1 for graphs. Therefore, our original predictions from the symmetric version were indeed correct.

5.2.2 Asymmetric probabilities between symmetric actions

The remaining question is how are there asymmetric probabilities between symmetric (permutative) actions? As we saw, these asymmetric results are indeed most likely NE (due to their decreasing regrets and deltas between strategies).

6 Arguments behind the Algorithm

Blah blah

Appendices

A Graphs and Plots

A.1 Symmetric Blotto

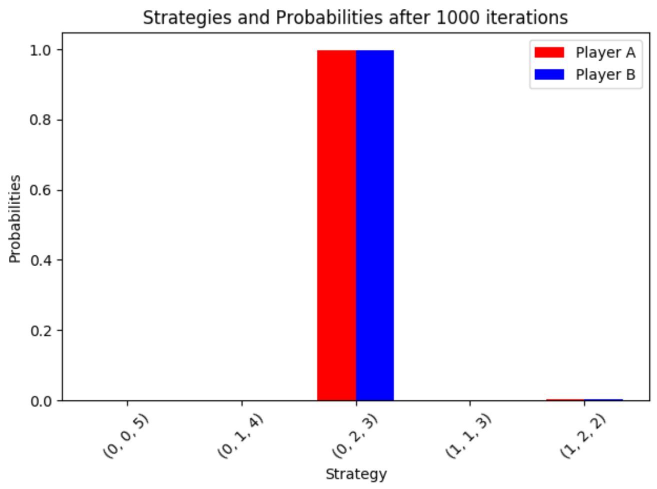


Figure 7: Symmetric Blotto Convergence to NE after 1k iterations

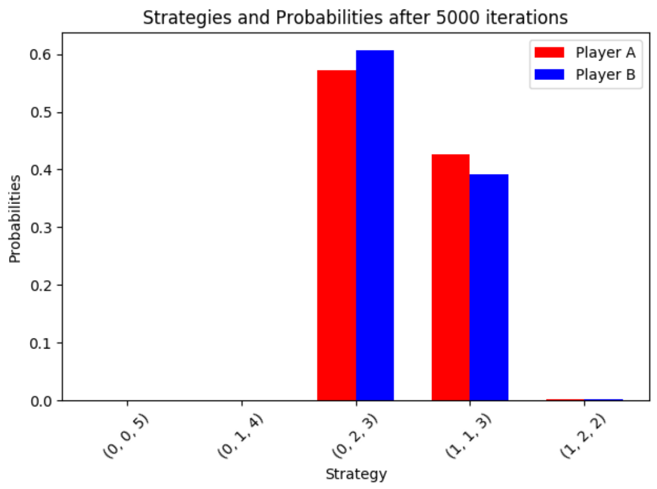


Figure 8: Nudging Symmetric Blotto to NE_2 using different starting regrets do not converge

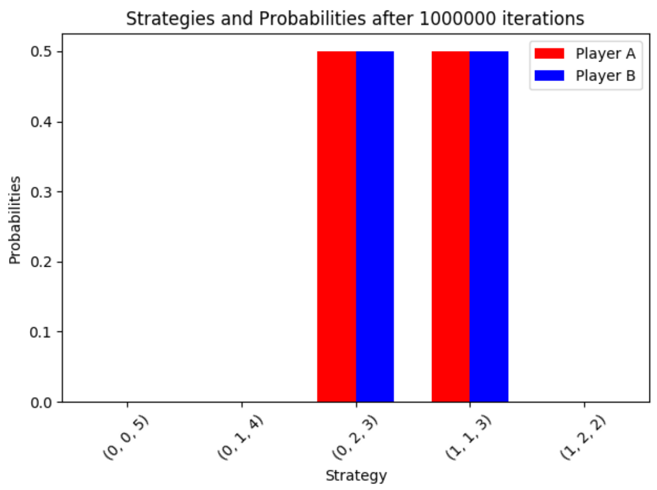
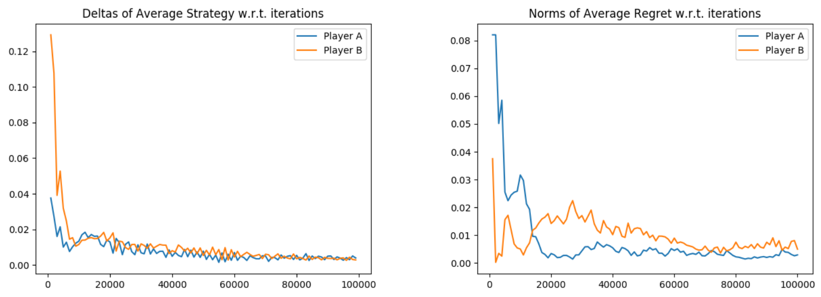


Figure 9: Extreme Nudging of Symmetric Blotto to NE_2 using different starting regrets

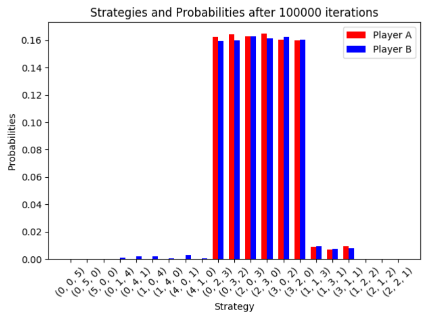
A.2 Colonel Blotto



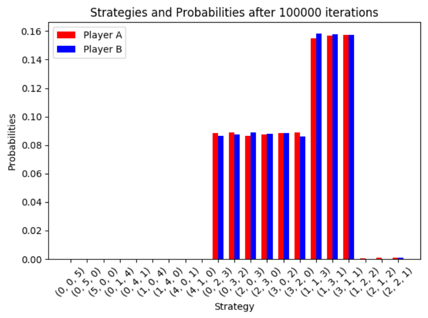
(a) Average Straegy Deltas (b) Average Total Positive Regrets

Figure 10: Deltas and Average Total Positive Regrets for Colonel Blotto 100k iterations

A.2.1 Nudging towards predicted equilibria



(a) Towards Predicted Equilibrium 1



(b) Towards Predicted Equilibrium 2

Figure 11: Nudging towards Colonel Blotto's Predicted Equilibria using starting conditions

B Code

All of the below code can also be found on `github.com/jackel119/regret_matching`. Readers may prefer to go there for a superior code-viewing experience.

B.1 Games

B.1.1 Colonel Blotto

```
import itertools


class ColonelBlottoGame:

    def __init__(self, no_soldiers, no_battlefields):
        assert no_soldiers > no_battlefields
        self.no_soldiers = no_soldiers
        self.no_battlefields = no_battlefields

        self._init_actions()

    # Plays the game and returns the utility for player A
    def get_utility(self, a_action, b_action):

        # Assert same number of battlefields as initialized game
        assert len(a_action) == self.no_battlefields
        assert len(b_action) == self.no_battlefields

        # Asserts each player are using the defined no. of soldiers
        assert sum(a_action) == self.no_soldiers
        assert sum(b_action) == self.no_soldiers

        a_score = 0
        b_score = 0

        for i in range(self.no_battlefields):
            if a_action[i] > b_action[i]:
                a_score = a_score + 1
            elif a_action[i] < b_action[i]:
                b_score = b_score + 1

        if a_score == b_score:
            return 0
        elif a_score > b_score:
            return 1
        else:
            return -1
    def get_actions(self):
        # Returns a list of all possible actions
        # i.e. ways to partition s elements into b sequential boxes
        return self._actions

    def _init_actions(self):
        # Initializes list of all actions

        rng = [i for i in range(self.no_soldiers + 1)] * self.no_battlefields

        permutations = \
            filter(lambda x: sum(x) == self.no_soldiers,
                   list(itertools.permutations(rng, self.no_battlefields)))

        permutations = list(set(permutations))
        permutations.sort()

        self._actions = permutations
        self._actions.sort(key=sorted)
```

B.1.2 Symmeteric Blotto

```
from colonel_blotto import ColonelBlottoGame

import itertools

class SymmetricBlotto:

    def __init__(self, no_soldiers, no_battlefields):

        self._blotto_instance = ColonelBlottoGame(no_soldiers, no_battlefields)
        self._init_actions()

    def get_utility(self, a_action, b_action):
        aps = list(itertools.permutations(a_action))
        bps = list(itertools.permutations(b_action))

        pairs = list(itertools.product(aps, bps))

        util = 0

        for x, y in pairs:
            util += self._blotto_instance.get_utility(x, y)

        util = util / len(pairs)

        return util
    def get_actions(self):
        return self._actions


    def _init_actions(self):
        # Initializes list of all actions

        actions = self._blotto_instance.get_actions()
        self._actions = sorted(list(set([tuple(sorted(a)) for a in actions])))

if __name__ == "__main__":

    g = ColonelBlottoGame(5, 3)

    actions = g.get_actions()
    actions = set([tuple(sorted(a)) for a in actions])

    actions = sorted(list(actions))

    for i in range(len(actions)):
        for j in range(len(actions)):
            a = actions[i]
            b = actions[j]

            aps = list(itertools.permutations(a))
            bps = list(itertools.permutations(b))

            pairs = list(itertools.product(aps, bps))

            util = 0

            for x, y in pairs:
                util += g.get_utility(x, y)

            util = util / len(pairs)

            print(f"Utility for {a} against {b}: {util}")

        print()


    print(actions)
```

B.1.3 Rock Paper Scissors

```
class RockPaperScissorsGame:

    _actions = ("ROCK", "PAPER", "SCISSORS")
    _name = "Rock, Paper Scissors"

    # Plays the game and returns the utility for player A
    def get_utility(self, a_action, b_action):

        assert a_action in RockPaperScissorsGame._actions
        assert b_action in RockPaperScissorsGame._actions

        a_index = RockPaperScissorsGame._actions.index(a_action)
        b_index = RockPaperScissorsGame._actions.index(b_action)
        diff = a_index - b_index

        utility = diff if abs(diff) == 1 else int(diff // -2)

        return utility

    def get_actions(self):
        return list(RockPaperScissorsGame._actions)
```

B.2 Players

B.2.1 OneShotPlayer

```
class OneShotPlayer(object):

    def __init__(self):
        self.reset_stats()

    def inform(self, utility):
        self.games_played += 1
        self.total_util += utility

    def report(self):
        avg = self.total_util / self.games_played
        return f"Average Utility: {avg} \n" + \
            f"Cumulative Utility: {self.total_util} \n" + \
            f"Games Played: {self.games_played}"

    def games_played(self):
        return self.games_played

    def reset_stats(self):
        self.games_played = 0
        self.total_util = 0
```

B.2.2 Regret Matching Player

```
import numpy as np
```

```python
from one_shot_player import OneShotPlayer


class RegretMinPlayer(OneShotPlayer):

    def __init__(self):
        super(RegretMinPlayer, self).__init__()
        self.initialized = False

    def initialize(self, game):
        self.game = game
        self.action_list = game.get_actions()
        self.num_actions = len(self.action_list)
        self.regret_sum = [0 for _ in self.action_list]
        self.strategy_sum = [0 for _ in self.action_list]
        self.games_played = 0
        self.initialized = True

    def check_init(self):
        if not self.initialized:
            raise Exception('Player not initialized!')


    def post_play(self, other_player_action):

        self.check_init()

        game_result = self.game.get_utility(self.last_action,
                                            other_player_action)

        super().inform(game_result)

        self._update_regrets(other_player_action)

        self.games_played += 1

        # if self.games_played == 10000:
        #     super(RegretMinPlayer, self).reset_stats()

    def report(self):

        self.check_init()

        print(super(RegretMinPlayer, self).report())

        print(f"Final Regrets: {self.regret_sum}")
        print(f"Final Average Cumulative Regret: {self.get_cumulative_regret()}")
```

```
def _update_regrets(self, other_player_action):

    utilities = [self.game.get_utility(a_action, other_player_action)
                 for a_action in self.action_list]

    played_utility = self.game.get_utility(self.last_action,
                                           other_player_action)

    regrets = [i - played_utility for i in utilities]

    self.regret_sum = [sum(x) for x in zip(self.regret_sum, regrets)]

def get_action(self):

    self.check_init()

    strategy = self.get_strategy()
    action = self.action_list[np.random.choice(
        len(self.action_list), p=strategy)]
    self.last_action = action

    return action

def get_strategy(self):


    strategy = [i if i > 0 else 0 for i in self.regret_sum]
    normalizing_sum = sum(strategy)

    if normalizing_sum > 0:
        normalized_strategy = [i / normalizing_sum for i in strategy]
    else:
        normalized_strategy = [1 / self.num_actions for i in strategy]

    self.strategy_sum = [sum(x) for x in
                         zip(self.strategy_sum, normalized_strategy)]

    return normalized_strategy

def get_average_strategy(self):
    # Returns the average strategy throughout the entire training process

    self.check_init()

    normalizing_sum = sum(self.strategy_sum)

    if normalizing_sum > 0:
        average_strategy = [i / normalizing_sum
```

```
                                for i in self.strategy_sum]
        else:
            import pdb; pdb.set_trace()
            average_strategy = [1 / normalizing_sum for
                                _ in range(self.num_actions)]

        return average_strategy

    def get_cumulative_regret(self):
        return sum([i if i > 0 else 0 for i in self.regret_sum]) / super().games_play
```

B.2.3 Fixed Strategy Player

```
import numpy as np

from one_shot_player import OneShotPlayer


class FixedPlayer(OneShotPlayer):

    def __init__(self):
        super(FixedPlayer, self).__init__()
        self.initialized = False

    def initialize(self, game):
        self.game = game
        self.action_list = game.get_actions()
        self.num_actions = len(self.action_list)
        self.strategy = [ 1 / self.num_actions for _ in self.action_list ]
        self.games_played = 0

        self.initialized = True

    def check_init(self):
        if not self.initialized:
            raise Exception('Player not initialized!')

    def post_play(self, other_player_action):

        self.check_init()

        game_result = self.game.get_utility(self.last_action,
                                            other_player_action)
        super().inform(game_result)


        self.games_played += 1
    def report(self):
        print(super(FixedPlayer, self).report())

    def get_action(self):

        strategy = self.strategy
        action = self.action_list[np.random.choice(
            len(self.action_list), p=strategy)]
        self.last_action = action

        return action

    def get_average_strategy(self):
        return self.strategy
```

B.3 Trainer

```
from matplotlib import pyplot as plt
from celluloid import Camera
from tqdm import trange

import imageio
import numpy as np


class OneShotTrainer:

    def __init__(self,
                 player_a,
                 player_b,
                 game_instance,
                 strat_delta=False,
                 plot_regret=False,
                 interval=1000,
                 animate=False):

        self.player_a = player_a
        self.player_b = player_b

        # Initialize players to know their games
        self.player_a.initialize(game_instance)
        self.player_b.initialize(game_instance)

        self.game_instance = game_instance

        # Animate Strats or not
        self.animate = animate
        if animate:
            self._fig, self._ax = plt.subplots()
            self._camera = Camera(self._fig)
            # Initialize animation graphs
            self._ax.set_xlabel('Strategy')
            self._ax.set_ylabel('Probabilities')
            self._ax.set_title('Strategies by Probability')

        # Plot Regrets or not
        self.plot_regret = plot_regret
        if self.plot_regret:
            self.a_avg_regrets = []
            self.b_avg_regrets = []


        self.games_played = 0
        self.interval = interval


        self.strat_delta = strat_delta
        if strat_delta:

            self.a_last_strat = None
            self.b_last_strat = None

            self.a_delta_history  = []
            self.b_delta_history  = []

    def train(self, iterations):

        for i in trange(iterations):
            self.train_once()

    def train_once(self):
        player_a_strategy = self.player_a.get_action()
        player_b_strategy = self.player_b.get_action()

        result = self.game_instance.get_utility(player_a_strategy,
                                                player_b_strategy)

        self.player_a.post_play(player_b_strategy)
        self.player_b.post_play(player_a_strategy)

        self.games_played += 1



        if self.games_played % self.interval == 0:
```

```python
# a_average_strat = self.player_a.get_strategy()
# b_average_strat = self.player_b.get_strategy()

a_average_strat = self.player_a.get_average_strategy()
b_average_strat = self.player_b.get_average_strategy()

if self.plot_regret:
    a_regret = self.player_a.get_cumulative_regret()
    b_regret = self.player_b.get_cumulative_regret()

    self.a_avg_regrets.append(np.linalg.norm(a_regret))
    self.b_avg_regrets.append(np.linalg.norm(b_regret))

if self.strat_delta:


    # Plot delta of A strat
    a_strat = np.array(a_average_strat)

    if self.a_last_strat is not None:
        a_delta = np.linalg.norm(a_strat
                                 - self.a_last_strat)
        self.a_delta_history.append(a_delta)

    self.a_last_strat = a_strat

    # Plot delta of B strat
    b_strat = np.array(b_average_strat)

    if self.b_last_strat is not None:
        b_delta = np.linalg.norm(b_strat
                                 - self.b_last_strat)
        self.b_delta_history.append(b_delta)

    self.b_last_strat = b_strat



if self.animate:


    index = np.arange(len(a_average_strat))
    width = np.min(np.diff(index))/3

    self._ax.bar(index, a_average_strat, width,
                 color='r',
                 label='Player A')
```

```
                self._ax.bar(index + width, b_average_strat, width,
                             color='b',
                             label='Player B')

                self._ax.set_xticks(index + width / 2)
                self._ax.set_xticklabels(self.game_instance.get_actions(), rotation=4

                self._ax.text(0.05, 0.95, f"Iterations: {self.games_played}", horizon
                     verticalalignment='center', transform=self._ax.transAxes)



                self._fig.tight_layout()

                self._camera.snap()


    def save_animation(self, filename, FPS=20):
        if not self.animate:
            raise Exception("Animation parameter not set!")

        tmp = 'tmp.gif'

        self._camera.animate().save(tmp, writer = 'imagemagick')

        gif = imageio.mimread(tmp, memtest='2GiB')

        imageio.mimsave(filename, gif, fps=FPS)



    def report(self):

        print("Player A:")
        self.player_a.report()
        a_average_strat = self.player_a.get_average_strategy()
        print("Average Strategy:", a_average_strat)
        print()
        print("Player B:")
        self.player_b.report()
        b_average_strat = self.player_b.get_average_strategy()
        print("Average Strategy:", b_average_strat)

        plt.clf()
        fig, ax = plt.subplots()

        index = np.arange(len(a_average_strat))
```

```
width = np.min(np.diff(index))/3

ax.bar(index, a_average_strat, width,
       color='r',
       label='Player A')

ax.bar(index + width, b_average_strat, width,
       color='b',
       label='Player B')

ax.set_xlabel(f'Strategy')
ax.set_ylabel('Probabilities')
ax.set_title(f'Strategies and Probabilities after {self.games_played} iterati
ax.set_xticks(index + width / 2)
ax.set_xticklabels(self.game_instance.get_actions(), rotation=45)
ax.legend()

fig.tight_layout()
plt.show()

if self.strat_delta:

    plt.clf()

    index = np.arange(1, len(self.a_delta_history) + 1) * self.interval

    fig, ax = plt.subplots()

    ax.plot(index, self.a_delta_history, label="Player A")
    ax.plot(index, self.b_delta_history, label="Player B")
    ax.set_title("Deltas of Average Strategy w.r.t. iterations")

    ax.legend()

    plt.show()

if self.plot_regret:

    plt.clf()

    index = np.arange(1, len(self.a_avg_regrets) + 1) * self.interval
    fig, ax = plt.subplots()

    ax.plot(index, self.a_avg_regrets, label="Player A")
    ax.plot(index, self.b_avg_regrets, label="Player B")
    ax.legend()

    ax.set_title("Norms of Average Regret w.r.t. iterations")
```

```
            plt.show()
```

B.4 Example of main program

```
from colonel_blotto import ColonelBlottoGame

from regret_minimization import RegretMinPlayer
from fixed import FixedPlayer

from rock_paper_scissors import RockPaperScissorsGame

from one_shot_trainer import OneShotTrainer


game = ColonelBlottoGame(5, 3)

p = RegretMinPlayer()

trainer = OneShotTrainer(p, p,
                         game, strat_delta=True, plot_regret=True, interval=1000, ani


trainer.train(1000 * 1000)
# trainer.save_animation("self_play.gif")
trainer.report()
```