Regret Minimisation Learning and Equilibria in Games

jyl20@ic.ac.uk, CID:01857602

Year 3

January 12, 2023

Contents

1 Notation **4**

2 Introduction **4**

3 Equilibria in games **5**

3.1 Correlated equilibrium 5

3.2 Geometry of correlated equilibria and relation to Nash equilibria 6

4 Regret minimisation learning **8**

4.1 Regret matching 8

4.2 Discussion 9

4.2.1 Importance 9

4.2.2 Friction 10

4.2.3 Generalisation 10

4.3 Alternative definition of regret 11

5 Blotto game **12**

5.1 Labelling the pure strategies 12

6 Solving Blotto games **13**

6.1 Regret minimisation learning 13

6.1.1 A simple example: solving $\mathcal{B}(3, 5, 5)$ 13

6.1.2 Speed of convergence 14

6.2 The Collapsed Blotto Game and abstraction 15

6.2.1 The Collapsed Blotto Game 16

6.2.2 Labelling the pure strategies 16

6.2.3 Comparing results for the Blotto and Collapsed Blotto games 17

6.2.4 Abstraction 18

6.3 An alternative approach 19

6.3.1 Alternative definition for Blotto games 19

6.3.2 General Lotto Games and Colonel Blotto Games 20

6.3.3 $\Gamma(a, a)$ for integer a 20

6.3.4 $\Gamma(a, a)$ for non-integer a 21

7 Dynamics of learning agents **22**

A Github **24**

B Core infrastructure **24**

C Infrastructure for Blotto games **31**

D Actual code **34**

D.1 Finding strategies for Blotto games 34

D.2 Timing training iterations 35

1 Notation

A *game* consists of

1. a finite set of players $N = \{1, \ldots, n\}$
2. for each $i \in N$, a set S^i of pure strategies of player i
3. for each $i \in N$, a payoff function $u^i : S \to \mathbb{R}$, where $S \coloneqq \prod_{i \in N} S^i$

A generic element s of S is $s = (s^i)_{i \in N}$; $s^{-i} \coloneqq (s^{i'})_{i' \neq i}$ denotes the combination of pure strategies of all players other than i.

A *strategy* x^i for player i is a probability distribution over S^i.

2 Introduction

In this project, we first give a brief overview of equilibria in games and motivate the study of *correlated equilibria.*

Then, we introduce the notion of *regret minimisation learning* and give a brief discussion on the topic.

Next, we implement regret minimisation learning in the context of Blotto games and explore how *abstraction through considering a simpler game*, the Collapsed Blotto Game, can help us find optimal strategies more efficiently. We also look at how Hart used the related General Lotto Game to derive optimal strategies for the Blotto game.

Finally, we look at how the actions of learning agents evolve with time.

It is difficult for me to say which parts of the project I am most proud of, but I would say I tried to make section 3 and section 6.2 the most unique. For the former, I tried to present the notion of equilibria in games from the different perspectives of different authors, as opposed to simply listing out the definitions. For the latter, I tried to explore on my own how abstraction can be very useful for simplifying games, but in hindsight I realised that this is a lot less original than I thought it was.

3 Equilibria in games

It is natural to study equilibria in games. Aumann 1974 [1] views equilibria as self-enforcing agreements in the sense that if all players agree to play a certain strategy before the game is played, then players will want to renege with probability 0.

The concept of Nash equilibrium is commonly studied in game theory. Informally, a collection of strategies, one for each player, is a Nash equilibrium if, given that other players play their strategy in the collection, no player can gain anything by unilaterally changing their strategy. Nash equilibrium assumes that, for some reason, each player knows which strategies the other players are using.[2]

This can be a reasonable assumption for simple games where it is easy to play optimally. That said, it is in general hard to find Nash equilibria. As Hart puts it [7], there are no general, natural dynamics leading to Nash equilibrium. It is not possible to find an adaptive procedure which is simple and efficient- in terms of the computations involved, time for convergence and information of each player- which will eventually arrive and stay at a close neighbourhood of the Nash equilibrium. It might not be reasonable to assume that other players are playing at a Nash equilibrium since such points can be very hard to find.

It might be instructive to look at the set of correlated equilibria if one thinks of an opponent's play as the outcome of some learning process. This is because a number of learning rules converge to the set of correlated equilibria of a game. Examples include calibrated learning described by Foster and Vohra in [4] and regret matching described by Hart and Mas-Colell in [8].

3.1 Correlated equilibrium

Assume that before each round, each player i receives a random signal θ^i. The combination of signals $\theta = (\theta^1, \theta^2, \ldots, \theta^n)$ follow a joint probability distribution F which is known by all players. Moreover, the signals do not affect the payoff of the game.

A canonical way to view correlated equilibria is think consider a 'referee' who chooses a set of strategies $(\theta^1, \theta^2, \ldots, \theta^n) \in S$ according to the distribution F and recommends each player i to play θ^i. Player i does not observe θ^{-i}, the recommendation to other players.

A distributions of signals F is a correlated equilibrium if no player has the incentive to unilaterally deviate from the suggested action. Formally, we require

$$\sum_{s^{-i}} F(j, s^{-i}) u^i(j, s^{-i}) \geq \sum_{s^{-i}} F(j, s^{-i}) u^i(k, s^{-i})$$

for all players i and actions $j, k \in S^i$

The interaction between signals give us some information about the set of correlated equilibria (CE). If each component θ^i of the signal is independent, then the players have no information about the actions of other players and the correlated equilibrium is just a Nash equilibrium. On the other hand, if the signals are fully correlated and each player knows what others will play, then a signal is followed by a Nash equilibrium play and the CE corresponds to convex combinations of Nash equilibria. [5]

In real life scenarios, such a 'referee' may not exist. There are however other ways of motivating correlated equilibria without using this notion. One approach through evolutionary games is outlined in [3]. Suppose nature selects two players u and v according to some symmetric probability measure $(p(u, v) = p(v, u))$ and each player knows the conditional distribution of the other player given its own. Then nature acts as a correlation device and the strict correlated equilibrium correspond to an outcome distribution of an evolutionary stable strategy for a 'simple contest'

Alternatively, [2] frames correlated equilibria as an expression of Bayesian rationality. Under the assumption that players have a common prior, playing a correlated equilibrium is equivalent to being Bayesian rational.

3.2 Geometry of correlated equilibria and relation to Nash equilibria

The correlated equilibria is defined by solutions to systems of linear inequalities which are not strict. Note that such solutions are convex and closed. Furthermore, the set of correlated equilibria is bounded. Hence, we conclude that the CE is convex and compact.

As mentioned above, by considering independent signals one can show that the CE contains the set of Nash equilibria. The existence of Nash equilibria is proven by Nash [10], and it follows that the CE is non-empty.

There turns out to be a stronger relationship between correlated equilibria and Nash equilibria in some classes of two-person games. It is shown in [12] that games which are 'best-response equivalent' to a two-person zero-sum game contain no correlated equilibria where for any Nash equilibrium, there is a player who prefers the correlated equilibrium over the Nash equilibrium. In other words, thinking of equilibria as self-enforcing agreements as mentioned above, there is no reason to make an agreement which is a correlated equilibrium but not a Nash equilibrium since no player would gain benefit from this.

4 Regret minimisation learning

4.1 Regret matching

Consider player i who has a set of pure strategies S^i. To adopt regret matching, the player looks at the pure strategy j which they played in the previous round.

Let $j, k \in S^i$ and s_t be the action of all the players at time t.

$$R^i(j,k,t) := (u^i(k, s_t^{-i}) - u^i(s_t^i))1_{s_t^i=j}(s_t)$$

denotes the regret of not having played k instead of j at time t.

$$R_T^i(j,k) := \max\left(\frac{1}{T}\sum_{t=1}^{T} R^i(j,k,t), 0\right)$$

denotes the corresponding average regret up to time t.

At round $T+1$, the player looks at the regrets $R_T^i(j,k)$, $k \neq j$ and decides whether to switch to another action or to stick with playing j again, with the probability of switching proportional to the regrets relative to the current strategy. The factor of proportionality determined by some constant c and is stationary. Given that the player switches, the probability of playing action k, $k \neq j$, is proportional to $R_t^i(j,k)$.

Algorithm 1 Regret matching

```
Require: Player index $i$, action count $|S^i|$, payoff function $u^i$, current time $T$, scaling constant $c$ (with restrictions given in 4.2.2)
1: D $\leftarrow$ zeros($|S^i|, |S^i|$)    ▷ $i, j$th entry is the gain in total utility from using $i$ to replace all occurrences of $j$
2:
3: function INCREMENT($s_T^i$, chosenAction)
4:     chosenUtility $\leftarrow u^i(s_T^i)$    ▷ Utility of chosen action
5:     for $k = 1, 2, ..., |S^i|$ do
6:         kUtility $= u^i(k, s_T^{-i})$    ▷ Utility had the player chosen $k$ this round
7:         D[k, chosenAction] += kUtility − chosenUtility
8:     end for
9: end function
10:
11: function GETPROBABILITY(chosenAction)
12:     row = max(0, row)    ▷ Convert negative entries to 0
13:     row = D[:,chosenAction]    ▷ Get row of D corresponding to last action
14:     row[chosenAction] = 0
15:     row = $\frac{\text{row}}{ct}$    ▷ Scale the vector so the sum is less than 1
16:     pStay = 1 − sum(row)
17:     row[chosenAction] = pStay
18:     return row
19: end function
```

After each action, *Increment* is called to update the matrix of cumulative regrets. On each round, *GetProbability* is called to return the probability vector.

4.2 Discussion

4.2.1 Importance

The importance of this algorithm is given by the following theorem from [8]

Theorem 4.1. *Let each player play regret matching. Then the joint distribution of play converges to the set of correlated equilibria of the stage game.*

Let s_t be the random variable corresponding to the action of players at time t. The joint distribution of play $z_T(s) := \frac{1}{T}\sum_{t=1}^{T} 1_{s_t=s}(s)$ is defined to be the empirical frequency of the each N-tuple of strategy.

It is worth noting that while, conditional on the history of previous plays, the actions of players on each round are independent, the long run statistics of play converge to the *correlated* equilibria. [5]

4.2.2 Friction

Under this algorithm, there is a uniform lower bound on the probability of sticking with the action played at the last round.

Lemma 4.1.1. Let

$$c > \max_{l \in N}(|S^l| - 1) \max_{j,k \in S^l, s^{-l} \in S^{-l}, l \in N} |u^l(k, s^{-l}) - u^l(j, s^{-l})|$$

Then there is some ϵ_c such that

$$p^{stay}_{j,t} := P(\text{play action } j \text{ at round } t+1 \mid \text{played } j \text{ at round } t) > \epsilon_c \qquad \forall t$$

Proof. Using the notation in the pseudo-code, it is straightforward to see that for any $k \in S^i$

$$R^i_T(j,k) = \frac{\texttt{D[j,k]}}{t} \leq \max_{j',k' \in S^l, s^{-l} \in S^{-l}, l \in N} |u^l(k', s^{-l}) - u^l(j', s^{-l})|$$

since at each time step t, the increase in regret is bounded above by $|u^i(k, s^{-i}_t) - u^i(j, s^{-i}_t)|$.

Let $p^{switch}_{j,t} := 1 - p^{stay}_{j,t}$ so $p^{switch}_{j,t} = \sum_{l \neq k} \frac{\texttt{D[l,k]}}{ct}$. We have

$$\frac{\texttt{D[l,k]}}{ct} < \frac{1}{\max_{l \in N}(|S^l| - 1)}$$

so

$$p^{switch}_{j,t} < \frac{|S^i| - 1}{\max_{l \in N}(|S^l| - 1)} < 1$$

as desired. □

This idea seems a bit counter-intuitive: no matter how bad the previous action is in hindsight, there is still a positive probability of picking the action again. It turns out friction is crucial for the above theorem to hold. Intuitively, it prevents the players from being locked into a 'bad' cycle, an example of which is given in [8].

4.2.3 Generalisation

In the regret matching strategy described above, given that the player switches, the action chosen is proportional to the (average) regret. What if a player wants to rate the actions with a different metric? For example, could the players play proportional to the square of the regret, in which case actions with higher regrets will be played disproportionately more often, or play proportional to the square root of regret, in which case actions with high regrets will be played less often in comparison?

In [5], Hart generalises regret matching to scenarios where, given the player switches, instead of playing action k with probability proportional to $R_T^i(j,k)$ in the above strategy, the player plays proportional to $f(R_T^i(j,k))$, where f is some Lipschitz continuous sign-preserving real function. Note that the functions given above are Lipschitz since the average regret of games with finite actions is bounded, so f is restricted to some compact set. In the paper, Hart noted that if f grows quickly, such as $f(x) = x^r$ with $r \to \infty$, then the probability (given the player switches) will tend to become a uniform distribution over the actions with maximal regret, which resembles the rules in best-response dynamics and fictions play.

Under this generalisation, the conclusion that joint distribution of play converges to the set of correlated equilibria still holds. [5]

4.3 Alternative definition of regret

In [11], Lanctot and Neller gave a different formulation of regret. Instead of keeping track an $|S^i| \times |S^i|$ matrix of regrets for player i as we have done above, we keep track an $|S^i|$-vector of regrets and on each round, increment entry j by the change in utility had the player chosen to play action j instead of their chosen action.

5 Blotto game

Consider two players, Blotto and Enemy who are at a war where there are K battlefields. Blotto and Enemy have S_B and S_E soldiers respectively, where K, S_B, S_E are positive integers. Each player needs to distribute the soldiers into the K battlefields simultaneously. For each battlefield, if the number of soldiers distributed by Blotto and Enemy are unequal, then the player who distributed more soldiers gets $+1$ while the other gets -1; if an equal number of soldiers is distributed, both players get 0. We will denote such a game by $\mathcal{B}(K, S_B, S_E)$.

5.1 Labelling the pure strategies

We now assign to each pure strategy a natural number as an index. For the case with K battlefields and T soldiers, the set of pure strategies is $S_{K,T}\{(x_1, \dots, x_K) \mid \sum_{i=1}^{K} x_i = T, x_i \in \mathbb{N} \cup \{0\}\}$ which has size $\binom{K+T-1}{K-1}$ by the stars and bars argument. By conditioning on the number of soldiers in the first battlefield,

$$|S_{K,T}| = \sum_{t=0}^{T} \binom{K+t-2}{K-2} = \sum_{t=0}^{T} |S_{K-1,t}|$$

Hence, for any $0 \le i < |S_{K-1,T}|$, we can find $0 \le t \le T$ such that $i \le \sum_{t'=0}^{t} |S_{-1K,t'}|$ but $i > \sum_{t'=0}^{t-1} |S_{K-1,t'}|$. The value t is the number of soldiers reserved for the remaining $K-1$ battlefields. Hence, this procedure tells us that for a given index i, the first battlefield has $T-t$ soldiers. We then replace the index i with $i - \sum_{t'=0}^{t-1} |S_{K-1,t'}|$, which allows us to find the number of soldiers in the second battlefield by ignoring the first battlefield and applying the procedure to the remaining $K-1$ battlefields. Doing this recursively gives us a mapping from the indices to the pure strategies.

6 Solving Blotto games

6.1 Regret minimisation learning

6.1.1 A simple example: solving $\mathcal{B}(3, 5, 5)$

We now use the algorithm proposed in Neller and Lanctot in [11] to find strategies for this game. To ease notation, we denote with $\langle x \rangle$, where x is some vector, the set of distinct permutations on x. For example,

$$\langle 1, 1, 3 \rangle = \{(1, 1, 3), (1, 3, 1), (3, 1, 1)\}$$

A trial of 100,000 rounds gave a strategy which was approximately the uniform distribution over the set

$$\langle 1, 1, 3 \rangle \cup \langle 0, 2, 3 \rangle$$

It is worth noting that the probability of choosing each distriubtion in each $\langle x \rangle$ seems to be roughly the same. This idea will be explored in more detail in the next section.

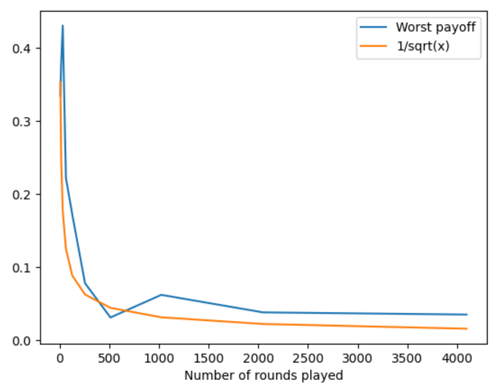
How can we evaluate this strategy? One way to do so is to compare it against the class of pure strategies. Since the Blotto game is a two player zero-sum game, the payoff of the opponent is a good indicator of the payoff of the player. The expected payoff of the opponent is a convex combination of the payoff of the pure strategies. Hence, an appropriate metric for our strategy x would be

$$M_x = \max_{s^j \in S^j} u^j(x, s^j)$$

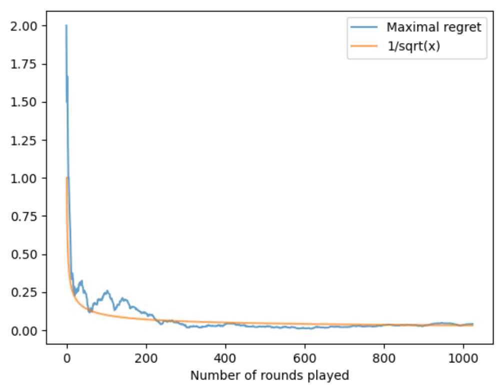
where j is the index of the opponent. The higher the number, the worse our strategy is. This metric can be interpreted as the worst case exploitability of our strategy.

Using the strategy $\hat{x}$ found by the algorithm, $M_{\hat{x}} \approx 0.01$. In other words, we could expect to lose at most 0.01 battlefields on average using this strategy no matter what our opponent does.

6.1.2 Speed of convergence



(a) Worst payoff against time



(b) Maximal regret against time

We investigate the speed of convergence by looking at the graph of worst payoff M_{x_t}, where x_t is the strategy obtained by training an agent for t rounds, and the graph of maximal average regret.

From the literature [13], the average overall regret is $O\left(\frac{1}{\sqrt{t}}\right)$. Also, suppose the players' average overall regret is less than ϵ, then the strategy is a 2ϵ Nash equilibrium so the worse payoff is at most 2ϵ. The graphs seem to match this result.

6.2 The Collapsed Blotto Game and abstraction

The number of possible strategies increase quickly for Blotto games. As mentioned above, there are $N_{K,S_B} := \binom{K+S_B-1}{K-1}$ strategies in the above formulation for Blotto games for K battlefields and S_B soldiers. To initialise the game, one needs to populate a $N_{K,S_B} \times N_{K,S_B}$ payoff matrix; on each round each player needs to update a N_{K,S_B}-vector of regrets. Consider the case where K is fixed and S_B varies. It is elementary to see that

$$\frac{N_{K,S_B+1}}{N_{K,S_B}} = \frac{K+S_B}{S_B+1}$$

which means that the growth rate is quite fast. Below, we plot the times required for the game $\mathcal{B}(4, S_B, S_B)$. The graph on the left focuses on the cases where S_B are smaller so the time taken is dominated by the agents playing the game; the graph on the right includes cases where S_B is larger so the time is mainly used to initialise the large matrices.

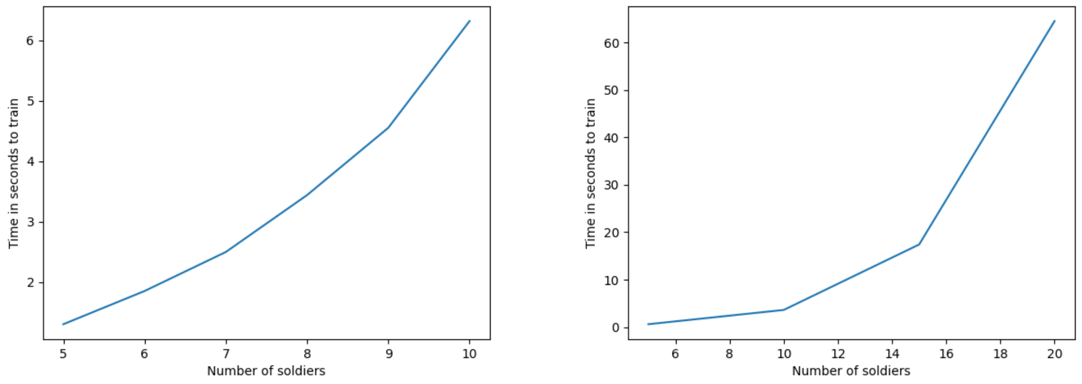


Figure 1: Time needed to train agents for 4 battlefields

In the following section, we look at a simplified version of the game called the Collapsed Blotto Game

6.2.1 The Collapsed Blotto Game

In 6.1.1, we made the observation that the probability of choosing each strategy in $\langle x\rangle$ seems to be roughly the same. This makes a lot of sense- the battlefields are symmetric, hence for any k, $P(\text{Battlefield } j \text{ has } k \text{ soldiers})$ should be the same over all j. This motivates our definition for $\langle x\rangle$: we can view of it as an equivalence class where given that the player chose a certain class, the player chooses evenly among the actions within the class. We call this game where the pure strategies of the players are the set of equivalence classes the Collapsed Blotto Game. The payoff of one class against another is simply the expected payoff of the original game over all possible combinations of pure strategies within each class.

Blotto Game	Collapsed Blotto Game
(0,2)	$\langle 0,2\rangle$
(1,1)	$\langle 1,1\rangle$
(2,0)	

Table 1: Pure strategies for $K = 2$, $S_B = S_E = 2$

6.2.2 Labelling the pure strategies

In the Blotto game $\mathcal{B}(K, S_B, S_E)$, we considered the partitions of S_B into K non-negative integers; in the Collapsed Blotto Game $\mathcal{CB}(K, S_B, S_E)$, we consider the same partitions without order, so $(0, 2)$ and $(2, 0)$ are the same. [9] gives us a way to count such partitions using generating functions.

Let $P_n(M)$ be the number of unordered non-negative integer partitions of size n for the integer M. By considering the smallest element of the partition, we get

$$P_n(M) = \sum_{j=0}^{\infty} P_{n-1}(M - jn)$$

where each term corresponds to the case where there the smallest element is j. We define $P_n(0) = 1$ and $P_n(M) = 0$ for $M < 0$. Next, we consider the generating function for P_n:

$$g_n(x) = \sum_{M=0}^{\infty} P_n(M)x^M$$

One can show that

$$g_n(x) = \frac{g_{n-1}(x)}{1 - x^n}$$

Using the recurrence relation and comparing coefficients, one can recover $P_n(M)$. This allows us to map the numbers $\{1, 2, \ldots, P_K(S_B)\}$ to the pure strategies using the same method in 5.1.

Finally, we note that the number of pure strategies $P_K(S_B)$ is approximately $\frac{1}{K!}\binom{K+S_B-1}{K-1}$ for fixed K and as $S_B \to \infty$. This is intuitively clear, since for S_B large the K numbers in the partitions would be unique with high probability and we reduce the number of strategies by a factor $\frac{1}{K!}$.

6.2.3 Comparing results for the Blotto and Collapsed Blotto games

We now compare the performances of the two games by training agents to play 5000 rounds.

(K, S_B, S_E)	No. of pure strategies	Initialise game (s)	Train (s)	$M_{\hat{x}}$
$(5, 6, 6)$	210	0.7	2.1	0.02
$(5, 7, 7)$	330	1.7	3.2	0.03
$(5, 8, 8)$	495	4.9	4.5	0.05
$(5, 9, 9)$	715	8.3	6.5	0.04
$(5, 15, 15)$	3876	233	34.4	0.05

Table 2: Computational performance for Blotto Game

We note that for each unit of increase in the soldier count, the number of pure strategies, time for initialisation and training increase by around 50%, 100% and 50% respectively. Note that the algorithm is already struggling with $(5, 15, 15)$.

(K, S_B, S_E)	No. of pure strategies	Initialise game (s)	Train (s)	$M_{\hat{x}}$
$(5, 9, 9)$	23	0.9	0.4	0.005
$(5, 15, 15)$	84	0.6	0.8	0.01
$(5, 20, 20)$	192	3.2	1.8	0.005
$(5, 25, 25)$	377	12.1	3.2	0.03

Table 3: Computational performance for Collapsed Blotto Game

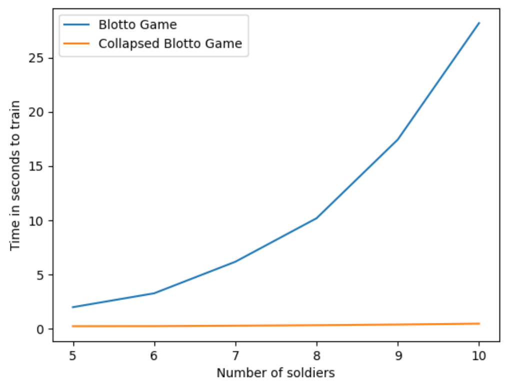


Figure 2: Comparison between Blotto game and abstracted version (5 battlefields)

We see that the performance for the Collapsed Blotto game is a lot better.

6.2.4 Abstraction

Abstraction refers to the act of restricting players' strategy spaces. This is essential for games with a large amount of possible actions/outcomes. This technique has been used to find strategies in games like Poker.

6.3 An alternative approach

In [6], Hart gave an alternative definition of Blotto games, introduced a related class of games called General Lotto games and showed that the results in these games can be applied to the original Blotto games.

6.3.1 Alternative definition for Blotto games

The definition used by Hart is similar to the one given above, with the exception that instead of looking at all K battlefields, we look at one battlefield drawn at random. In other words, using X_i, Y_i to denote the solders distributed by Blotto and Enemy at battlefield i respectively for $i = 1, 2, \ldots, K$ the payoff for Blotto in the original formulation is

$$U = \sum_{i=1}^{K} (1_{X_i > Y_i} - 1_{X_i < Y_i})$$

For Hart's formulation, the payoff is

$$U' = 1_{X_I > Y_I} - 1_{X_I < Y_I}$$

where I is a random variable (independent from the X_i and Y_i) drawn uniformly from the index set $\{1, 2, \ldots, K\}$.

We will show that these two formulations are equivalent using the well known result that positive affine transformations preserve the structure of games in the sense that the best response sets and hence Nash equilibria are unchanged. This is intuitively clear: the addition of constants can be thought of as an entry fee/compensation which is independent of the game, while (positive) scaling can be thought of as a change in currency through which the game is played. None of these should change how players act. Hence, it is sufficient to show that $\frac{E(U)}{K} = E(U')$. Let $\vec{X}$, $\vec{Y}$ denote $(X_1, X_2, \ldots, X_K)$ and $(Y_1, Y_2, \ldots, Y_K)$ respectively.

Proof.

$$\begin{aligned} E(U') =& E_{\vec{X},\vec{Y}} \left[E_I \left(1_{X_i > Y_i} - 1_{X_i < Y_i} \mid \vec{X}, \vec{Y} \right) \right] && \text{(Law of iterated expectations)} \\ =& E_{\vec{X},\vec{Y}} \left[\frac{\sum_{i=1}^{K} (1_{X_i > Y_i} - 1_{X_i < Y_i})}{K} \right] && \text{(By definition)} \\ =& \frac{E(U)}{K} \end{aligned}$$

□

This formulation is useful for two reasons. Computationally, this removes the need to iterate over an array of length K to compare the distributions across all battlefields; conceptually, this allows us to think about a single battlefield only and makes the problem more tractable.

6.3.2 General Lotto Games and Colonel Blotto Games

Let $a, b > 0$ and X, Y be integer-valued non-negative random variables satisfying $E(X) = a$ and $E(Y) = b$. Hart defines a General Lotto game $\Gamma(a, b)$ as a zero-sum game where player A chooses a distribution for X and player B chooses a distribution for Y. The payoff of A is $E(1_{X>Y} - 1_{X<Y})$ where X, Y are independent. Intuitively, we think of $a = \frac{S_B}{K}$ and $b = \frac{S_E}{K}$.

Note that *any strategy for the Colonel Blotto game $\mathcal{B}(K, S_B, S_E)$ is a strategy for the General Lotto game $\Gamma(\frac{S_B}{K}, \frac{S_E}{K})$ but not vice versa.* In other words, suppose we have found an optimal strategy for $\Gamma(\frac{S_B}{K}, \frac{S_E}{K})$ which can be implemented in $\mathcal{B}(K, S_B, S_E)$, then the implementation is also optimal for the corresponding Blotto game.

In [6], Hart solves the General Lotto game for different values of a and b then gives conditions for when such strategies can be implemented for the Blotto game. We will look at some special cases below.

6.3.3 $\Gamma(a, a)$ for integer a

We call a strategy X for the General Lotto game (T, K)-feasible if X can be obtained from a non-negative integer partition of K-partition of T. In other words, X is (T, K)-feasible if it can be implemented by a player with T soldiers on K battlefields.

Suppose $a = \frac{T}{K}$. We have the following results from the paper:

1. A strategy X (for the General Lotto game) is optimal if and only if

$$X \in \text{conv}\{U_O^a, U_E^a\} \qquad (*)$$

   where U_O^a is the uniform distribution over the odd numbers $\{1, 3, \ldots, 2a-1\}$ and U_E^a is the uniform distribution over the even numbers $\{0, 2, \ldots, 2a\}$. .

2. If K is even, both U_O^a, U_E^a are (T, K)-feasible.

3. Suppose K is odd. If T is odd, U_O^a is (T, K)-feasible while U_E^a is not; if T is even, U_E^a is (T, K)-feasible while U_O^a is not.

Next, we give some of the strategies found by the regret minimisation algorithm.

(K, S_B, S_E)	Strategy
$(3, 6, 6)$	$0.056\langle 0, 2, 4\rangle + 0.395\langle 0, 3, 3\rangle + 0.395\langle 1, 1, 4\rangle + 0.034\langle 1, 2, 3\rangle + 0.12\langle 2, 2, 2\rangle$
$(4, 4, 4)$	$\frac{1}{3}\langle 0, 1, 1, 2\rangle + \frac{2}{3}\langle 1, 1, 1, 1\rangle$
$(4, 4, 4)$	$\frac{1}{4}\langle 0, 0, 2, 2\rangle + \frac{1}{3}\langle 0, 1, 1, 2\rangle + \frac{5}{12}\langle 1, 1, 1, 1\rangle$
$(4, 4, 4)$	$\langle 1, 1, 1, 1\rangle$

It is elementary to see that the strategies found match the form stated in $(*)$ as they are convex combinations of U_O^a, U_E^a for the corresponding a.

6.3.4 $\Gamma(a, a)$ for non-integer a

Let $a = m + \alpha$ for integer m and $\alpha \in (0, 1)$. Suppose $a = \frac{T}{K}$. We have the following results from the paper:

1. The only unique optimal strategy X^* (for the General Lotto game) is
$$X^* = (1 - \alpha)U_E^m + \alpha U_O^{m+1} \tag{†}$$
2. If $T = mK + r$ where $M \geq 0$ and $1 \leq r \leq K - 1$ are integers, then
$$(1 - \frac{r}{K})U_E^m + \frac{r}{K}U_O^{m+1}$$
is (T, K)-feasible

Next, we give some of the strategies found by the regret minimisation algorithm.

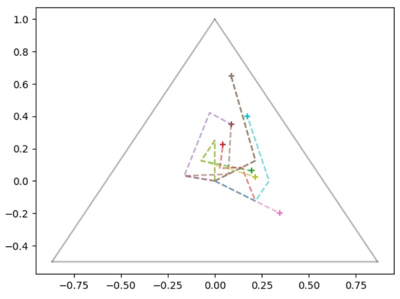
(K, S_B, S_E)	Strategy
$(3, 4, 4)$	$\langle 0, 2, 2\rangle$
$(3, 4, 4)$	$\frac{1}{4}\langle 0, 1, 3\rangle + \frac{3}{4}\langle 0, 2, 2\rangle$
$(3, 5, 5)$	$\langle 0, 2, 3\rangle$
$(3, 5, 5)$	$\frac{3}{4}\langle 0, 2, 3\rangle + \frac{1}{4}\langle 1, 1, 3\rangle$

Contrary to the previous section, the strategies found here are not of the form given in (†). Should we be concerned, given that (†) is supposed to be the unique solution?

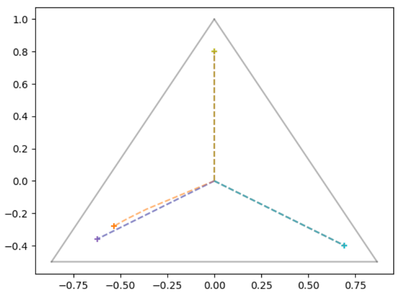
We believe that the answer is no. What (†) suggests is that there exists strategies *for the General Lotto game* which beats the strategies we found. However, it does not say that these strategies are feasible to implement *in the context of Blotto games*. The strategies which we found might be optimal against the class of feasible solutions but not the full class of solutions. The empirical values of $M_{\hat{x}}$ found for these strategies are quite small, which supports our claim.

7 Dynamics of learning agents

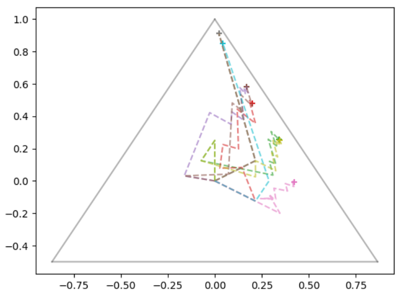
In this section, we compare the dynamics for different regret minimisation agents in the context of the Collapsed Blotto game $\mathcal{CB}(3,3,3)$. On the left, we have the implementation from Neller and Lanclot [11]; on the right we have the implementation from Hart and Mas-Colell [8]. The plots show the empirical distribution of actions of learners. Bottom left, bottom right and top correspond to the actions $\langle 0,0,3\rangle$, $\langle 0,1,2\rangle$ and $\langle 1,1,1\rangle$ respectively. The code used to produce the plots were produced by Aamal Hussain and found on the blackboard page of the course.



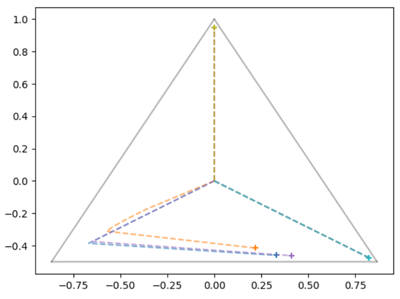
(a) $t = 5$



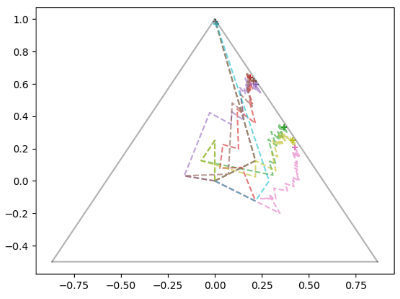
(b) $t = 5$



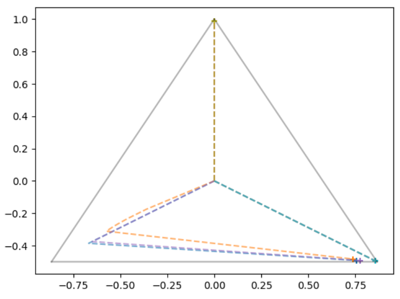
(c) $t = 20$



(d) $t = 20$



(e) $t = 100$



(f) $t = 100$

Firstly, we note that the behaviour on the right is much more regular, in the sense that the learner tends to move in the same direction. This is an evidence of the notion of ‘friction’ discussed before.

Next, we note that some agents on the right spend quite a high portion of their actions playing bottom left at first, which is not optimal. Again, this is due to friction.

It is interesting to note that on the left, some agents converge to certain points corresponding to mixed strategies but on the right agents favor the pure strategies. However, it should be the case that any point on the edge connecting the top and bottom right is a Nash equilibrium. Unfortunately, I am unable to provide a good explanation as to why agents on the right prefer pure strategies (possibly because of friction again?) or why agents on the left prefer some points over other points on the edge.

Finally, one can observe that agents on the left tend to prefer the top over bottom right. I believe this is because $\langle 1, 1, 1\rangle$ performs better on average against $\langle 0, 0, 3\rangle$ when compared to $\langle 0, 1, 2\rangle$.

A Github

The code can be found here. `https://github.com/justinlau20/Regret-learning`

B Core infrastructure

Contains classes for general games and agents; also has some useful functions.

```
from abc import ABC, abstractmethod
import numpy as np
import random
from numpy import cumsum
from numpy.random import rand
from collections.abc import Iterable
from typing import List
import matplotlib.pyplot as plt
from plotting import *

class Game(ABC):
    player_count = None
    action_counts = None
    strategy_maps = None
    """
    Encodes information about the game.

    Attributes:
    ---------------
        player_count:
            number of players in the game
        action_count:
            number of possible actions
    """
    @abstractmethod
    def __init__(self):
        pass

    def _setup(self):
        self.utility = np.vectorize(self._utility, excluded=['self', 'index'])
        self.payoff_matrix:Iterable = np.array([self.utility(i, *np.meshgrid(
                                                *(range(self.action_counts[j])
                                                for j in range(self.player_count)),
                                                indexing='ij'))
                                                 for i in range(self.player_count)])

    def _get_utility(self, index, actions):
        return self.payoff_matrix[index][actions]

    def _utility(self, index: int, *actions: Iterable) -> float:
        """
        Args:
        ----------------------
        action:
            action[i] = action of player i
        index:
            index of player

        Output:
        ----------------------
            Utility of player

        """
        pass


class BimatrixGame(Game):
    def __init__(self, A, B):
        self.player_count = 2
        self.payoff_matrix = [A, B]
        self.action_counts = np.shape(A)


class Agent(ABC):
    def __init__(self, game: Game, index: int, prior=None, save_strat_hist=False):
        """
        Encodes information about a player.

        Attributes:
        --------------
        game:
            Game object repersenting what game the agent is playing
        index:
            A label for the agent
        strategy:
            A vector of weights for each action. The agent will act according to this
                                                             array on each round
        total_utility:
            Total utility for the agent throughout the game.
        """
        self.t = 0
        self.game: Game = game
        self.action_count = game.action_counts[index]
        self.index:int = index
        if prior is None:
            self.strategy: Iterable = np.ones(self.action_count) / self.action_count
        else:
            self.strategy = prior
        self.strategy_sum: np.array = np.copy(self.strategy)
        self.strategy_cumsum = np.cumsum(self.strategy)
        self.total_utility: float = 0
        self.cum_strat = None
        self.avg_overall_regrets = []
        self.strategy_sums = np.empty((self.action_count, 0))
        self.save_strat_hist = save_strat_hist

    def action(self) -> int:
        """
        Returns the action of the agent.
        """
        return random.choices(range(self.action_count), cum_weights=self.strategy_cumsum)[
                                                                     0]

    def update(self, actions: Iterable):
        """
        Updates the
        """
        chosen_utility = self.game._get_utility(self.index, actions)
        self.total_utility += chosen_utility
        # print(chosen_utility)

        self.strategy_sum += self.strategy
        if self.save_strat_hist:
            self.strategy_sums = np.c_[self.strategy_sums, self.strategy_sum / sum(self.
                                                                 strategy_sum)]

        self.t += 1
        return chosen_utility


class Regret_Minimisation_Agent(Agent):
    def __init__(self, game: Game, index: int, prior=None):
        super().__init__(game, index, prior)
        self.regrets:np.array = np.zeros(self.action_count)
        self.strategy_sum: np.array = np.copy(self.strategy)
    def update(self, actions: tuple):
        chosen_utility = super().update(actions)

        for i in range(self.action_count):
            temp = list(actions)
            temp[self.index] = i
            temp = tuple(temp)
            action_i_utility: float = self.game._get_utility(self.index, temp)
            self.regrets[i] += action_i_utility - chosen_utility

        self.strategy = np.maximum(0, self.regrets)
        normalising_const = sum(self.strategy)

        if normalising_const <=0:
            self.strategy = np.ones(self.action_count) / self.action_count
        else:
            self.strategy /= normalising_const
        self.strategy_sum += self.strategy
        self.strategy_cumsum = np.cumsum(self.strategy)
        self.avg_overall_regrets.append(max(self.regrets) / self.t)


class Swap_Regret_Agent(Agent):
    def __init__(self, game: Game, index: int, prior=None):
        super().__init__(game, index, prior)
        self.swap_regret = np.zeros((self.action_count, self.action_count))
        self.cum_swap_diff = np.zeros((self.action_count, self.action_count))
        self.prev_action = None
        self.mu =  2 * (max(game.action_counts) - 1) * abs(game.payoff_matrix).max() + 1

    def update(self, actions: Iterable):
        chosen_utility = super().update(actions)
        chosen = actions[self.index]
        self.prev_action = chosen
        for i in range(self.action_count):
            temp = list(actions)
            temp[self.index] = i
            temp = tuple(temp)
            action_i_utility: float = self.game._get_utility(self.index, temp)
            self.cum_swap_diff[i][chosen] += action_i_utility - chosen_utility

        row = self.cum_swap_diff[:, chosen].clip(min=0)
        row[self.prev_action] = 0
        row = row / (self.t * self.mu)
        p_stay = 1 - sum(row)
```

```python
        row[self.prev_action] = p_stay
        self.strategy = row
        self.strategy_sum[chosen] += 1
        self.strategy_cumsum = np.cumsum(self.strategy)
        self.avg_overall_regrets.append(self.cum_swap_diff.max() / self.t)


class Trainer:
    def __init__(self, game, agents):
        self.game = game
        self.agents = agents

    def train(self, n, out=True):
        for i in range(n):
            action = tuple(agent.action() for agent in self.agents)
            for agent in self.agents:
                agent.update(action)
        strats = []
        for agent in self.agents:
            temp = agent.strategy_sum / sum(agent.strategy_sum)
            agent.cum_strat = temp
            strats.append(temp)
            agent.avg_utility = agent.total_utility / n
            if out:
                print(f"Utility of Agent {agent.index} is {agent.total_utility}")
                print(np.round(temp, 3))
        return self.agents


class Evaluation():
    def __init__(self, game: Game, rounds, sample_size) -> None:
        self.sample_size = sample_size
        self.rounds = rounds
        self.agents = [find_CE(game, iterations=rounds) for i in range(sample_size)]
        self.cum_probs = np.array([np.array([agent.cum_strat for agent in agent_arr])
                                   for agent_arr in self.agents] )
        self.average_strategy = np.average(self.cum_probs, axis= (0, 1))
        self.sds = np.std(self.cum_probs, axis=0)
        self.game = game


class Single_Evaluation(Evaluation):
    def __init__(self, game: Game, rounds=100000, agent = Regret_Minimisation_Agent,
                                                 priors=None, save_hist=False) -> None:
        self.rounds = rounds
        self.game = game
        self.agents: List[Agent] = find_CE(game, agent=agent, iterations=rounds, priors =
                                             priors, save_hist=save_hist)
        self.agent_count = len(self.agents)

    def viable_strategies(self, eps=0.001, dps=3):
        for index, agent in enumerate(self.agents):
            print(f"Viable strategies for agent {index}:")
            for action_index, prob in enumerate(np.round(agent.cum_strat, dps)):
                if prob >= eps:
                    if self.game.strategy_maps:
                        print(f"\t Probability of playing {self.game.strategy_maps[index][
                                                              action_index] } (with
                                                               index {action_index}
                                                              ) is {prob}.")
                    else:
                        print(f"\t Probability of playing {action_index} is {prob}.")
            print("\n\n\n")

    def get_strategies(self):
        return [agent.cum_strat for agent in self.agents]

    def plot_regrets(self, index=0):
        x_arr = np.arange(1, self.rounds)
        plt.plot(self.agents[index].avg_overall_regrets, label="Maximal regret", alpha=0.7
                                             )
        plt.plot(x_arr, 1 / np.sqrt(x_arr), label="1/sqrt(x)", alpha=0.7)
        plt.xlabel("Number of rounds played")

        plt.legend()
        plt.savefig("./plots/simple game/maximal_regret")
        plt.show()



def train_repeatedly(game: Game, each_train, sample_size):
    strats = np.empty((0, game.player_count, game.action_count))

    for i in range(sample_size):
        strats = np.r_[strats, [find_CE(game, iterations=each_train)]]

    for j in range(game.player_count):
        print(f"Standard deviation of strategy of player {j} is {np.sqrt(strats[:, j].var(
                                             axis=0))}")
```

```
def find_CE(game: Game, agent= Regret_Minimisation_Agent, iterations=100000, priors=None,
                                              save_hist=False):
    if priors is None:
        agents = [agent(game, i) for i in range(game.player_count)]
    else:
        agents = [agent(game, i, priors[i]) for i in range(game.player_count)]
    if save_hist:
        for agent in agents:
            agent.save_strat_hist = True
    return Trainer(game, agents).train(iterations, False)


def evaluate(agent:Agent, index=0, round=10000, mat=False):
    game = agent.game
    n = game.action_counts[1 - index]
    agents = [0, 0]
    agents[index] = agent
    m = 0

    for i in range(n):
        prior = np.zeros(n)
        prior[i] = 1
        agents[1 - index] = Agent(game, 1- index, prior)

        if not mat:

            outcome = Trainer(game, agents).train(round, out=False)
            pure_regret = outcome[1 - index].avg_utility
        else:
            cur_mat = game.payoff_matrix[1 - index]
            pure_regret =  agents[index].strategy @ cur_mat @ np.transpose(agents[1 -
                                                         index].strategy)
        m = max(m, pure_regret)
    return m

def faceoff(a1, a2, round=10000, mat=True):
    game = a1.game
    agents = [a1, a2]

    # outcome = Trainer(game, agents).train(round, out=False)
    cur_mat = game.payoff_matrix[1]
    if mat:
        return agents[0].strategy @ cur_mat @ np.transpose(agents[1].strategy)
    else:
        outcome = Trainer(game, agents).train(round, out=False)
        return outcome[1].avg_utility

if __name__ == "__main__":
    pass
```

C Infrastructure for Blotto games

```
from math import comb
from abc import ABC, abstractmethod
import numpy as np
import random
from numpy import cumsum
from numpy.random import rand
from collections.abc import Iterable
from infrastructure import *

def partition_matrix(k, s):
    """
    K: Maxmimal number of partitions
    S: Number to be partitioned

    Output:
        Returns a (K+1) by (S+1) matrix whose i,j th entry is the number of ways to
                                                    partition the number j
        into the sum of i non-negative numbers where the order does not matter
    """
    out = np.array([np.array([1] + [0] * s)], dtype=int)
    for i in range(1, k + 1):
        mul = np.zeros(s + 1, dtype=int)
        for j in range(len(mul)):
            if (j - len(mul) + 1) % i == 0:
                mul[j] = 1
        prod_out = np.polymul(mul, np.flip(out[-1, :]))
        row = np.flip(prod_out)[:s + 1]
        out = np.r_[out, [row]]
    return out

def soldier_dist(i, N, S, out):
    if S == 0:
        out += [0] * N
        return np.array(out, dtype=int)
    if N == 1:
        out.append(S)
        return np.array(out, dtype=int)
    temp = 1
    for s in range(S + 1):
        if s >= 1:
            temp = (temp * (N + s - 2)) // s
        if temp > i:
            out.append(S - s)
            return soldier_dist(i, N - 1, s, out)
        else:
            i -= temp

def soldier_dist_collapsed(i, N, S, lookup_mat, out, base=0):
    if S == 0:
        out += [base] * N
        return np.array(out)

    if N == 1:
        out.append(base + S)
        return np.array(out)

    for s in range(S + 1):
        temp = lookup_mat[N - 1, S - s * N]
        if temp > i:
            base += s
            out.append(base)
            return soldier_dist_collapsed(i, N - 1, S - s * N, lookup_mat, out, base)
        else:
            i -= temp
    raise Exception("You are bad at coding >:(")

class Blotto(Game):
    def __init__(self, N, S_arr):
        self.N = N
        self.S_arr = S_arr
        self.player_count = 2
        self.action_counts = [comb(N + self.S_arr[i] -1, N-1) for i in range(self.
                                                              player_count)]
        self.soldier_dists = [[soldier_dist(i, self.N, self.S_arr[j], []) for i in range(
                                                              self.action_counts[j])]
                              for j in range(self.player_count)]
        self.strategy_maps = [{i:f"{self.soldier_dists[j][i]}" for i in
                                range(self.action_counts[j])} for j in range(self.
                                                                                  player_count)
                                                                                  ]
        self._setup()

    def _utility(self, index: int, *actions: Iterable) -> float:
        dists = [self.soldier_dists[i][actions[i]] for i in range(self.player_count)]
        if index == 0:
            return sum(dists[0] > dists[1]) - sum(dists[0] < dists[1])
        else:
            return -self._utility(0, *actions)

def average_payoff(*arrs):
    s = 0
    A, B = arrs
    for a in A:
        diff = a - B
        s += sum(diff > 0) - sum(diff < 0)

    return s / len(A)

class Collapsed_Blotto(Game):
    def __init__(self, N, S_arr):
        self.N = N
        self.S_arr = S_arr
        self.player_count = 2
        if len(set(S_arr)) == 1:
            self.lookup_mats = [partition_matrix(self.N, self.S_arr[0])] * 2
        else:
            self.lookup_mats = [partition_matrix(self.N, self.S_arr[i]) for i in range(
                                                            self.player_count)]

        self.action_counts = [self.lookup_mats[i][-1, -1] for i in range(self.player_count
                                                        )]

        self.soldier_dists = [[soldier_dist_collapsed(i, self.N, self.S_arr[j], self.
                                                        lookup_mats[j], [])
                            for i in range(self.action_counts[j])] for j in range(self.
                                                                  player_count)]

        self.strategy_maps = [{i:f"{self.soldier_dists[j][i]}" for i in
                              range(self.action_counts[j])} for j in range(self.
                                                                    player_count)
                                                                    ]

        self._setup()

    def _utility(self, index: int, *actions: Iterable) -> float:
        dists = [self.soldier_dists[i][actions[i]] for i in range(self.player_count)]
        util = average_payoff(*dists)
        if index == 0:
            return util
        else:
            return -util

class Surrogate_Blotto_Agent(Agent):
    def __init__(self, game: Collapsed_Blotto, b_game:Blotto, index: int, prior=None,
                                            save_strat_hist=False):
        self.game: Collapsed_Blotto
        self.b_game = b_game
        super().__init__(game, index, prior, save_strat_hist)

    def update(self, actions: Iterable):
        # return super().update(actions)
        pass

    def action(self) -> int:
        collapsed_action = super().action()
        dist = soldier_dist_collapsed(collapsed_action, self.game.N, self.game.S_arr[self.
                                                    index], self.game.lookup_mats[self.
                                                    index], [])
        dist = np.random.permutation(dist)
        for i in range(self.b_game.action_counts[self.index]):
            if all(dist == soldier_dist(i, self.game.N, self.game.S_arr[self.index], [])):
                return i
        print(dist)
        print(soldier_dist(i, self.game.N, self.game.S_arr[self.index], []))
        raise Exception(">:(")
```

D Actual code

This section makes use of code defined above heavily.

D.1 Finding strategies for Blotto games

The following code finds the strategies with probability above some threshold.

```
from infrastructure import find_CE
from Blotto_infra import *
import numpy as np
```

```
g = Collapsed_Blotto(3, (5, 5))
SE = Single_Evaluation(g, 25000, agent=Regret_Minimisation_Agent)
SE.viable_strategies()

strat = SE.get_strategies()[0]
evaluate(Agent(g, 0, strat),round=5000, mat=True)
```

The function evaluate tests the strategy obtained against the class of pure strategies and outputs the test statistic M_x

D.2 Timing training iterations

```
from infrastructure import find_CE
from Blotto_infra import *
import numpy as np
import time

t_arr = []
for s in range(5,11):
    start = time.time()
    g = Blotto(5, (s, s))
    SE = Single_Evaluation(g, 5000, agent=Regret_Minimisation_Agent)
    t_arr.append(time.time() - start)
```

References

[1] Robert J. Aumann. Subjectivity and correlation in randomized strategies. *Journal of Mathematical Economics*, 1(1):67–96, 1974.

[2] Robert J. Aumann. Correlated equilibrium as an expression of bayesian rationality. *Econometrica*, 55(1):1–18, 1987.

[3] Martin Cripps. Correlated equilibria and evolutionary stability. *Journal of Economic Theory*, 55(2):428–434, 1991.

[4] Dean P. Foster and Rakesh V. Vohra. Calibrated learning and correlated equilibrium. *Games and Economic Behavior*, 21(1):40–55, 1997.

[5] Sergiu Hart. Adaptive heuristics. *Econometrica*, 73(5):1401–1430, 2005.

[6] Sergiu Hart. Discrete colonel blotto and general lotto games. 2007.

[7] Sergiu Hart. Dynamics and equilibria. Presidential Address, GAMES 2008 (July 2008), 2008.

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

[9] Lagerbaer (https://math.stackexchange.com/users/4171/lagerbaer). Integer partition with fixed number of summands but without order. Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/54635 (version: 2011-07-30).

[10] John Nash. Non-cooperative games. *Annals of Mathematics*, 54(2):286–295, 1951.

[11] Neller and Lanclot. An introduction to counterfactual regret minimization. 2013.

[12] Evanston R. W. ROSENTHAL. Correlated equilibria in some classes of two-person games. 1974.

[13] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007.