Dyanmics of Learning and Iterated Games

Project 3

Regret Minimisation Learning - A Computer Implementation

---

cid : 01496079
email : sc9018@ic.ac.uk
MSc Applied

6^{th} January 2023

Table of Contents

0.1 Introduction

The following project investigates the mathematically rich Colonel Blotto strategic game. In this project, after introducing the basic mechanics of the Blotto game we consider the existence of Nash equilibria and the nature of these Nash equilibria for both integer value allocation games and non-integer value allocation games. We then pivot to consider the wider field of regret learning and how an agent can be trained to use regret quantified and treated in a mathematical way to inform actions in the future such that they outperform other agents in direct competition. Particular consideration is given to the regret minimisation process and we apply it to an instance of the Blotto game to see how Nash Equilibria may be obtained using such an approach. In the final part of the project, we consider how theoretically with the help of Blackwell's approachability theorem we can assure regret minimisation. Finally, we look at the connections between regret learning and reinforcement learning by implementing a Q-learning algorithm to train an agent to play colonel Blotto against another agent which plays only random actions with an even probability distribution (which is, in theory, a strong strategy for colonel Blotto). We see that after sufficient training iterations, the Q-learning agents significantly and reliably outperform the random agent. This is also the part of the project the author is most proud and fond of.

0.2 Blotto

Colonel Blotto is a game that hinges on strategic mismatch between competing agents. The Colonel Blotto game takes inspiration from wartime allocation of soldiers across a battlefront. With colonel Blotto himself being a fictitious leader of a fictitious army in deadly conflict with an adversary. The basic idea, in a warfare setting, is to allocate your troops in such a way as to gain an advantage over your adversary's troops (how this exactly works will be addressed shortly). If this is achieved then the overall conflict is won in your favor. Blotto is a very powerful mathematical tool in modeling situations of direct competition since it very neatly generalises to more general resource allocation situations in instances of competition. Countless such applications exist however political campaigning, counter-terrorism planning, research and development spending allocations as well as instances of sporting competition can all be effectively modeled using the colonel Blotto framework [1].

In more concrete terms a standard edition of Colonel Blotto consists of two players in direct competition with one another in a warfare-like setting. Each side has a respective army with S soldiers. Hence the opposing armies are of equal size. The conflict takes place across a large battlefront (e.g. WWI style battle fronts) and so we can think of the entire front as consisting of individual, smaller battlefields. For the Blotto game, the number of battlefields is held to be less than the total number of troops available for each army. Hence N is the number of battlefields we have

$$N < S \tag{1}$$

The actions each player (in this case Blotto and the opposing general) can take is deciding how many troops to allocate to each individual battlefield. This is done such as to obtain an overall victory across the entire front and to win the game. Victory on any given battlefield very simply is obtained by the side which allocates a majority of troops to that battlefield. It follows that the payoff function for the first player in a two-player Blotto game can be formalised as

$$p(x_1, x_2) = \sum_{i=1}^{n=2} f_1(x_1^i, x_2^i) \tag{2}$$

where x_1 and x_2 are the respective strategy profiles for each player and f_1 is evaluated as

$$f_1(x_1, x_2) = \begin{cases} 1 & x_1^i > x_2^i \\ -1 & x_2^i < x_1^i \\ 0 & x_1^i = x_2^i \end{cases} \tag{3}$$

Hence in game theoretical terms the pay off can simply be thought to be the number of battle fields "won" across the entire front.
Upon some deliberation it becomes clear that the values in each utility vector sum to zero and thus Blotto is a zero sum game. This makes sense in a warfare context, where someones gain is someone else's direct loss however the game's payoff could be reformulated into a constant sum variant.
To illustrate an example of the zero sum blotto game we can pick a simple example where $S, N = (20, 5)$. We randomise the allocation of Blotto and the Enemies troops over the five battlefields.

Blotto-Enemy [S,N = (20,5)]					
Battlefield	N_1	N_2	N_3	N_4	N_5
Blotto	4	4	6	1	5
Enemy	2	13	1	2	2

(4)

From this we can see that Blotto has won this game since we has won three out of the five possible battlefields (N_1, N_3, N_5).
Thinking about the random case above it is evident that there exists no unbeatable strategy when both sides have the same amount of soldiers. As a result if this game were to be repeated many times the strategy giving a victory is continually changing, provided the losing player makes adjustments to change his losing situation [1].
Indeed knowing any information about the oppositions strategy is extremely powerful as we can see that if this is the case not even all the troops at ones disposal are needed to result in a win. In the example above if the Enemy had known about Blotto's troop allocation he would only have needed to use 12 soldiers across N_1, N_2, and N_4 to win the game.

0.3 Representing Blotto

While the premise of the colonel Blotto game is reasonably straightforward it becomes clear that the number of available strategies each player can play rapidly becomes very large as the number of soldiers assigned and number of battlefields increase. At its very core, the number of actions available to a player is a question of how to distribute S entities between N different locations. This is a well-understood problem in combinatorics and a mathematically identical question would be something like how many ways a certain number of sweets could be distributed between a certain number of people. In combinatorics such a problem is generalised to a "stars and bars" context. We use the symbol $*$ (star) to represent a soldier and a | (bar) to represent divisions between battlefields. It is important to remember that in this formulation of the game the soldiers are considered to be all of the same time and so indistinguishable while the battlefields are distinguishable. Suppose we had three battlefields (N = 3) and fifteen soldiers (S = 15).

In the notation of stars and bars we can represent sending four troops to N_1 (battlefield 1), 8 to N_2(battlefield 2) and 1 soldier to N_3(battlefield 3) as

$$\star\star\star\star\,\big|\,\star\star\star\star\star\star\star\star\big|\star \tag{5}$$

Hence the number of possible actions a player has (the number of ways he can assign his troops). The number of ways a player could assign his troops for this particular case is

$$\frac{17!}{15!2!} = \binom{17}{2} \tag{6}$$

where the latter is simply 17 choose 2. The was obtained since we have 15 stars and 2 bars for which we want to find out the number of possible arrangments while discounting repeats. Hence we divide by the number of repeated entities, in this case, 15 stars and 2 bars.[2] In a general setting with S soldiers and N battlefields the number of actions each player has is

$$\binom{S+N-1}{N-1} \tag{7}$$

From this, we can see that the number of possible actions each player can have becomes enormously large for all but the smallest values for S and N. As a result, formulating this game in matrix form while possible is mostly not very practical since in the majority of cases the matrix would be too large to fit on a piece of paper.

In principle however, there is nothing to stop us from formulating this game in matrix form. Where every row and column would be a specific action a player could take and the corresponding positions in the matrix being action profiles (i.e. one of the possible combinations the two players could play ahavenst one another). The entries of the matrix would house a vector containing the respective payoff for both players as obtained from the payoff function (

0.4 Existence of Nash Equilibria of Blotto

To show the existence of a Nash equilibrium in the case of integer allocations to each battlefield we begin by formalising the Blotto game (as it is described by Neller and Lanctot).

We describe the game in normal form with the tuple (N,A,u) where N is the number of players, A is the set of all the possible combinations of actions from all the players involved and u is a function which gives the utility for the situation and player in question. For the Blotto game we have only two players and so $N = 2$. The set of all possible combinations of actions of the two players is then given by $A = S_1$ x S_2 where S_i represents the set of actions player i can take. Finally in the context of the Blotto game the utility function u maps from the set of A to a set of vectors which give the respective payoffs for each player for that specific action profile.

To illustrate this we consider an instance of the Blotto game where we have two battlefields and each player has three soliders ($N = 2, S = 3$). Let us consider the action profile where player one allocates soldiers $[1, 2, 1]$ meaning one soldier on the first battlefield, two on the second and one on the final battlefield and player two has allocated soldiers [2,0,2]. To find the utility vector we use our payoff function (2) for both players as follows.

$$\text{The action profile is : } (\underbrace{[1,2,1]}_{x_1}, \underbrace{[2,0,2]}_{x_2}) \tag{8}$$

The payoff for player one is

$$\begin{aligned} P_1(x_1, x_2) &= f_1(x_1^1, x_2^1) + f_1(x_1^2, x_2^2) + f_1(x_1^3, x_2^3) \\ &= f_1(1,2) + f_1(2,0) + f_1(1,2) \\ &= -1 + 1 - 1 \\ &= -1 \end{aligned} \tag{9}$$

and thus the first entry in our utility vector is -1. The payoff for player two is then

$$\begin{aligned} P_1(x_1, x_2) &= f_2(x_1^1, x_2^1) + f_2(x_1^2, x_2^2) + f_2(x_1^3, x_2^3) \\ &= f_2(1,2) + f_2(2,0) + f_2(1,2) \\ &= 1 - 1 + 1 \\ &= 1 \end{aligned} \tag{10}$$

Where here the function f_2 is defined in the same way as f_1 in equation
To show that in the form described, the Blotto game has a Nash equilibrium provided the allocation of soldiers to a battlefield is an integer we begin by introducing the concept of a best response map which is crucial in explaining the existence of a Nash Equilibrium.

Definition 0.4.1 (Best Response Map)**.** To define the best response map we consider a game in normal form described by the tuple (N, S, U). N is the set of players taking part in the game. The set S is the set of all combinations of actions of all the players (i.e. $S = S_1 \times S_2 \times ... \times S_n$) and the set U is a tuple containing a payoff function u_i for each $i \in N$. We additionally define S_i to be the set of action player i can take. The best response map is a map for each player $i \in N$ and is defined as

$$\mathbf{BR}_i : S_{-i} \to \mathcal{P}(S_i) \tag{11}$$

where for each set of strategy combinations excluding the strategy of player i we map to a subset of the set of the strategies player i can play. The image under this map is defined as

$$s_{-i} \mapsto \{s_i^* : \forall s_i \in S_i, u_i(s_*^i, s_{-i}) \geq u_i(s_i, s_{-i})\} \tag{12}$$

where s_i^* is the best possible reply to a set of strategy combinations which are played by all the other players. This means that the set s_i^* of strategies for player i outperforms or performs just as well as all the possible strategies any other player could play.

A key theorem we then need to consider, having introduced the best response map is one relating a Nash Equilibrium to the best response map.

Theorem 0.4.1 (Nash Equilibrium)**.** *For a game defined in normal form (N, S, U) a strategy combination of all the players s^* is a Nash Equilibrium if and only if for all players i in the set N we have that $s_i^* \in \boldsymbol{BR}_i(s_{-i}^*)$*

This theorem codifies mathematically the more informal idea that in a Nash Equilibrium state, no player wishes to deviate from their strategy since that strategy is the best response to the strategy combination of all the other players and so deviation cannot result in an increased payoff.
We can now introduce the famous existence theorem of a Nash Equilibrium given by John Nash

Theorem 0.4.2 (Existence of Nash Equilibrium)**.** *For a game in normal from (N,S,U) with a finite number of players and associated action profiles there exists at least one Nash Equilibrium for the normal form game*

To show that this is indeed the case we want to show that the best response map has a fixed point. To show this we can do it by using Kakutani's fixed point theorem.

Theorem 0.4.3 (Kakutani's Fixed Point Theorem)**.** *For a set K which is closed, bounded, convex and non-empty where*

$$K \subset \mathbb{R}^n \tag{13}$$

and for a correspondence

$$f : K \to \mathcal{P}(K) \tag{14}$$

where the image of the correspondence, $f(x)$ for all $x \in K$ is compact and convex it follows that f has a fixed point,

$$x \in K \text{ where } x \in f(x) \tag{15}$$

One looks to now use Kakutani's fixed point theorem to show that there exists a Nash equilibrium for the colonel Blotto game in the case that the allocation to each battlefield is an integer [3].

For a game in normal form (N,S,U) we can apply Kakutani's fixed point theorem by letting the set K be the set of all combinations of actions of every player, S and by letting the best reply map BR_i be equivalent to the correspondence f. Having made these assignments it follows that if we can show that [4]

- S is compact, convex and non-empty
- $\mathbf{BR}_i(s_i)$ is non-empty for any s_i
- $\mathbf{BR}_i(s_i)$ is convex for any s_i

In the case of the Blotto game with integer battlefield allocation we have the situation where there exist only a finite set of actions a player may take. The set S is closed since it contains it boundary points and if a player wished to play a mixed strategy the probability of playing any specific action could be one and so the set is closed. The set S is bounded since we have a finite number of actions and if a mixed strategy is chosen the probability of playing any strategy is bounded by the laws of probability since the chance of playing an action must lie in the interval $[0, 1]$. Finally, the set S is also non-empty. This follows immediately provided that the setup of the Blotto game is not the case where the number of battlefields is zero and the number of soldiers is zero. Indeed if this were the case there is not much of a game to discuss. To show that the set S is convex we consider the set of all convex combinations of all points within the set. Since we have an integer allocation of soldiers we have the set S being finite and hence the set S contains all convex combinations of its points and thus it follows that S is a convex set. Hence we conclude that the set S of all possible combinations of actions of every player is non-empty, closed, bounded and convex.

The best response for the set of all player's mixed strategies is non-empty since from the definition of best response there exists a response to whatever the other players play which is the best possible response in that case. To show that the set of best responses is convex we can use a similar argument as we did for the set S in such as that since there is an integer allocation of soldiers per battlefield we have a finite set of points which all have corresponding best response depending on the players considered. These points of best responses can be taken in convex combination and since the set is finite the best response set contains all these points and thus it is convex.

Provided that these conditions hold, by Kakutani's fixed point theorem the best response correspondence has a fixed point and thus any game which is finite (the number of possible actions are finite) has a Nash Equilibrium as defined in the definition (0.4.1).

If we briefly consider the case where the allocation to a battlefield is not an integer we immediately see that our set of all combinations of all plays S is no longer finite. The approach followed above no longer works since we cannot verify convexity and closed and boundedness. While we cannot prove that a Nash Equilibrium must exist there exist many examples of games where in this situation a Nash Equilibrium nonetheless is present.

0.5 Nash Equilibria of Blotto

In this section, we consider a specific form of the more general Colonel Blotto game with the following conditions. There are only two battlefields in the game and so $N = 2$ and the armies allocated to each side are not equal in size. Additionally, the deployment of soldiers to a battlefield need not be

an integer number. Since the scenario of soldiers not being integers is not very physically realisable it makes sense to simply consider this as a resource allocation endeavor where the resource in question can quite reasonably not take integer values. As a result, in this section, the "soldiers" will simply be referred to as "resources". Altering the parameters in such a way means we can make the colonel Blotto game even more generally applicable to a wider host of strategic competitions we may hope to model in the real world. An example of this could be election campaigning across two different electoral regions since it is rare for the resources of both sides to be equal and generally the nature of electoral campaigning resources (e.g. money) can comfortably take non-integer values. Our aim is to characterise the set of all possible Nash Equilibria allowing for a comprehensive understanding of the game under consideration.

Before beginning our exposition on the set of all Nash equilibria we recall that a Nash Equilibrium is intuitively an action profile where neither party can improve their payoff by unilaterally deviating from their current strategy. We immediately note that since we have noninteger allocations to our battlefields, the approach of showing the existence of Nash Equilbria as discussed in section (0.4) is no longer applicable. Nevertheless, there is still rich mathematics behind all the possible Nash Equilibria which arise in this version of the Blotto game. We begin characterising the set of all NE by first considering some key theorems and ideas which let us identify all possible Nash equilibrium in all possible scenarios. A final note before beginning is that we make a slight alteration to the Blotto game (as done in the paper by Macdonell et al.) where by the resource superior player (i.e Blotto) is awarded a win on a battlefield where the two sides allocate an equal amount of resources. In this exposition, we follow the treatment of Macdonell et al. [5]

In this implementation of the Blotto game one side always has a greater amount of resources (larger army) than the opposing force and to mathematically codify this we first consider the constraints each player has when allocating troops to each battlefield. In general B signifies the resources of Blotto and E are the resources of the enemy. We let the troops Colonel Blotto allocates to the i^{th} battlefield be denoted by b_i. From reasonable physical constraints it follows that

$$b_1 \geq 0, \quad b_2 \geq 0 \quad \text{and} \quad b_2 \leq f(b_1) \tag{16}$$

This enforces the fairly obvious constraint that we cannot have negative resources on a battlefield and also that the function f signifies some kind of relationship between the allocations between the two battlefields. We shall for the time being consider only a linear resource constraint.. If we let the oppositions resource allocation to the i^{th} battlefield be denoted by e_i then we see that the same relations hold as in (16) except we then have a function g to indicate the resource constraint relation between battlefields. Having defined these constraints we can further impose the condition that the resources of Colonel Blotto are always greater than the opposition. Mathematically this is described by the equation given by Macdonell et. al

To further aid the analysis we stipulate that there exist values B_1 and B_2 such that if we allocate to a battlefield the value B_1 the corresponding battlefield will have a zero allocation and similarly for a zero allocation on a battlefield the corresponding battlefield has an allocation of B_2. In other words, there is a largest allocation to each respective battlefield such that the corresponding battlefield has a troop allocation of 0. A consequence of this is that we only have two points of intersection of our resource constraint function with the axes. The corresponding values of E_1 and E_2 are defined in a similar way.

Finally, we impose the condition that our constraint functions are both continuous and strictly decreasing. Discontinuity in the constraint function is an unrealistic scenario and a strictly decreasing function also makes physical sense since if we had portions of the constraint function which were

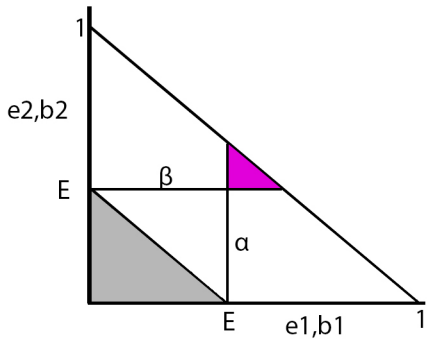

Figure 1: the purple region shows the region of allocation for Blotto such that we have a Nash Equilibrium given the resource constraint $E \in [0, \frac{1}{2}]$. Lines α and β bound both regions.

horizontal then this would suggest that for two different troop allocations to one battlefield we could have the same number of troops allocated to the other battlefield. If we are to use up all our troops then this cannot be the case. While we do not necessarily need to use all the troops for this analysis we restrict ourselves to this scenario. We also remark that due to the constraints imposed by (**??**) our analysis takes place in the positive quadrant however the conditions on the constraint functions being continuous and strictly decreasing are universally true beyond just the first quadrant.

Before coming to formalise a theoretical framework to obtain the set of Nash Equilibria for a given instance of the game within the outlined form we consider the first two special cases where we can arrive at the set of Nash Equilibria in an intuitive way. This then allows us to then see more general constructions which then allow us to consider more general cases. Specifically, we consider the normalised case of this game where normalisation in this context simply means that the resources allocated to Colonel Blotto are 1 and that the enemy allocation is a fraction of Blotto's allocation. The two cases we first consider are when $E \in [0, \frac{1}{2}]$ and the case where $E \in [\frac{1}{2}, \frac{2}{3}]$.

Case: $E \in [0, \frac{1}{2}]$ In this scenario, we can see just by inspection that Blotto has the resources to win overall game regardless of what the enemy does. If Blotto simply sends at least E resources to each battlefield he is guaranteed to win. Regardless of what strategy the Enemy plays, provided Blotto is a rational actor and sends at least E resources to battlefields one and two his expected payoff cannot be less than 2 since he always wins both battlefields and correspondingly the Enemy has an expected payoff of 0. Graphically this can be expressed in figure (1). It is worth remarking that on figure (1) we have a single region where Blotto can allocate his resources such that the resource constraints are obeyed and that the system is in a position of Nash Equilibrium for any potential choice of allocation E makes. The lines α and β provide a useful way of demarcating this region and the usefulness of lines of such a nature will be focused on as we continue in the exposition.

Case: $E \in [\frac{1}{2}, \frac{2}{3}]$ To see if we can bring more meaning to the lines demarcating the region of play where Blotto can allocate his troops such that we have a Nash Equilibrium we consider the further case where the Enemy has more resources such that a win every time is no longer guaranteed for Blotto. Strategies such as splitting his forces in half and sending each to a respective battlefield would be a very silly idea and in general, no rational agent would pick such a play since it is very easily countered by the Enemy simply overwhelming Blotto on a single battlefield thus ensuring that

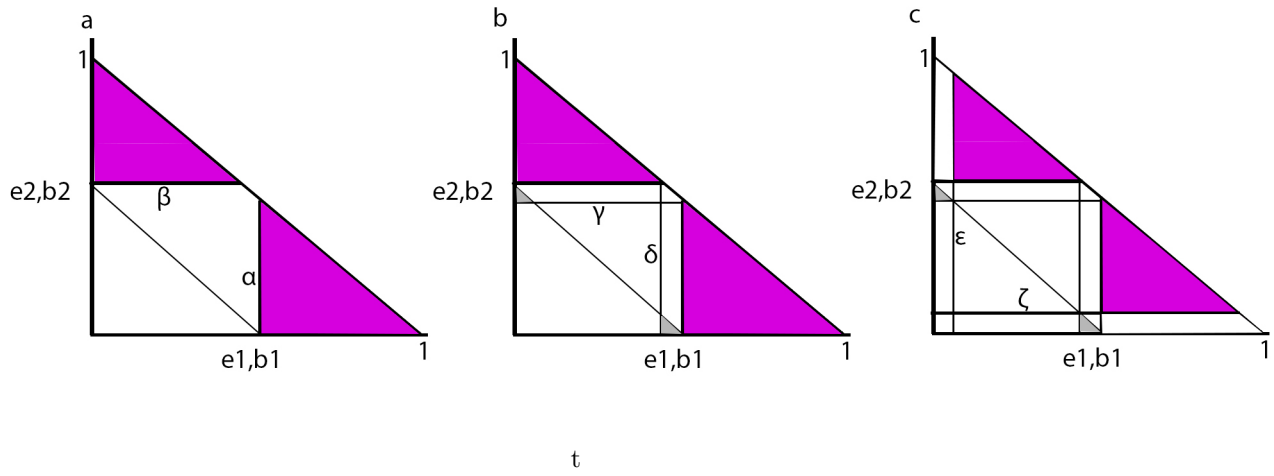

Figure 2: We consider the construction of the regions of Nash Equilibria give the resource constraint $E \in [\frac{1}{2}, \frac{2}{3}]$. Panel a. shows the lines α and β along with troop allocations. Panel b. shows the lines δ and γ along with the minimum amount of troops the enemy must deploy in our scenario to win on the opposing battlefield. Panel c. shows lines ϵ and ζ and the final constraint on blotto's allocation.

he does not lose. While Blotto cannot ensure victory every game, to maximise his expected payoff Blotto can send at least E troops to a battlefield. The enemy will most likely send as many troops as possible to a single battlefield since there is little reason to split forces and lose on both battlefields and so Blotto hopes for the case where both himself and the Enemy allocate the majority of their troops to the same battlefield. In this way, Blotto is guaranteed to win since Blotto's residual force will be larger than what the enemy allocated since the enemy decided to concentrate its forces. Of course, it is possible that in the case where the allocation of the larger forces of both players is not to the same battlefield, we have the case where Blotto loses a battlefield and the overall game is a tie however with this approach one battlefield victory is guaranteed every time. Panel a. in figure (2) shows the two regions in which he should allocate at least E troops such as to win at least one battlefield each game. The line α and the line β demarcate the minimum amount of troops to allocate to ensure a victory on that battlefield. As we established previously, the Enemy hopes for a mismatch between the battlefields where both sides commit the majority of their forces. In the case of a mismatch, the Enemy must still commit more than what is remaining of Blotto's force once he has committed at least E troops to the other battlefield. Panel b. in figure (2) contains lines γ and δ which demarcate the region where the Enemy must play in order to ensure that in the case of a mismatch, he does indeed win at least one battlefield and is not overwhelmed on both. Finally if Blotto attacks one battlefield with a minimum of E soldiers he must ensure that if the enemy targets that same battlefield with the majority of its troops Blotto can take advantage of the situation. Hence he must send a minimum amount of troops to the other battlefield to counter the smaller force the Enemy could have sent there while prioritising the battlefield where both players commit the majority of their troops. The lines on panel 3 in figure (2) show and ζ demarcating these regions.

From figure (2) we now consider with particular attention the lines and what they demarcate. We immediately see how these lines, if continued to their respective axes pick out the regions shaded either grey or purple. This is of significance since these regions contain strategies where neither player can unilaterally deviate from and immediately receive a better payoff from what they already received. Since these lines appear to demarcate the regions of Nash equilibria we look to characterise

them further. Considering the pair of lines ζ and b we see how if started from the x-axis they both have the same value the first time they meet the first resource constraint in their path (traveling left to right). As such we can think of the lines ζ and β as being lines corresponding to the situation where we assign a certain amount of resources to both players to a battlefield such that on the other battlefield the assignment of resources for both players is exactly the same.

If in figure (2), if Blotto assigned E_1 troops to battlefield one we can define a function $h(E_1)$ which gives the amount of resources committed to battlefield one by the Enemy such that on battlefield two the two sides allocate the same amount of troops. The lines which illustrate this are α and η. This is crucial since it allows us to find a line that represents the minimum amount of soldiers Blotto must send to battlefield one in the region of a Nash Equilibria by following the reasoning outlined in the case (provided some further conditions hold which we shall discuss later on) $E \in [\frac{1}{2}, \frac{2}{3}]$. The same reasoning applies to battlefield two where we see that the lines β and ζ represent regions where if Blotto sends E_2 troops to battlefield two then if the Enemy sends a certain amount of troops to battlefield two given by a function $p(E_2)$ then both sides allocate the same amount of troops to battlefield one. Again this then demarcates the minimum amount of troops Blotto must send to battlefield two in a region of Nash Equilibria. The corresponding regions demarcating the regions where the Enemy allocates troops in the case of a Nash Equilibrium can then be expressed using the resource constraint function f where the minimum amount of soliders the Enemy must allocate to battlefield two depending is given by applying f^{-1} the appropriate amount of times to the function p.

We can formalise these functions h and p as being ones assign equal troop allocations to the other battlefield for both players we can define them as Macdonell et al. did

$$\begin{aligned} h(x) &\equiv g^{-1}(f(x)) \quad \text{for battlefield 1} \\ p(y) &\equiv g(f^{-1}(y)) \quad \text{for battlefield 2} \end{aligned} \tag{17}$$

To elucidate these functions and explain how they can be useful it is worth considering the simple case where both f and g are our linear resource constraints as defined in (16). Considering the function h, the idea is that if x is Colonel Blotto's allocation to battlefield 1 then $h(x)$ is the opposition's allocation to battlefield 1. To see what the consequence of $h()$ is we consider the concrete example where $B = 10$ and $O = 8$. It follows that our linear constraints are then

$$\begin{aligned} f(x) &= 10 - x, \qquad g(x) = 8 - x \\ f^{-1}(x) &= 10 - x, \quad g^{-1}(x) = 8 - x \end{aligned} \tag{18}$$

If colonel Blotto assigns $x = 5$ troops to the first battlefield then we have the following arrangement.

$$\begin{aligned} &\text{Battlefield 1 [Blotto]:} \quad 5 \text{ troops}, && \text{Battlefield 1 [Opposition]:} \quad g^{-1}(f(5)) = \quad 3 \text{ troops} \\ &\text{Battlefield 2 [Blotto]:} \quad f(5) = 10 - 5 = 5 \text{ troops}, && \text{Battlefield 2 [Opposition]:} \quad g^{-1}(h(5) = 8 - 3 = 5 \text{ troops} \end{aligned} \tag{19}$$

and so we see that in this construction $h(x)$ is defined such that for a given troop allocation for both Blotto and the Opposition on battlefield 1, the function $h(x)$ ensures that the troop allocation on the other battlefield is the same for both players. The function $p(y)$ works in exactly the same way ensuring that given allocations to battlefield 2 we have the exact same allocation of troops to battlefield 1.

When comparing the case of $E \in [0, \frac{1}{2}]$ and the case of $E \in [\frac{1}{2}, \frac{2}{3}]$ we note that the independent regions in which Blotto and the enemy can play in such as to have a Nash Equilibrium vary. In the case of $E \in [0, \frac{1}{2}]$ we see that there is only one region for both players where they allocate resources.

In the case of $E \in [\frac{1}{2}, \frac{2}{3}]$ we see that there are now two regions for both Blotto and the Enemy to allocate their troops. From the regions mapped out from values of the functions h and g as defined in equations (17), we can see that we can consider the system to be in a partition depending on how many regions there are for each player to play in. We could consider the first case of being a partition of $n = 1$ and the second case of being the situation of partition $n = 2$. To identify what system a partition is in we can consider the interval a value such as E_1 (or E_2) falls in with relation to its corresponding value E_2 (or E_1). Before generalising we consider the specific case of the two partition case and note that E_2 must lie in the interval

$$E_2 \in (f(E_1), f(h(E_1))] \tag{20}$$

when comparing the $n = 1$ case to the $n = 2$ case we see how the number of lines given by $h^{n-2}(E_1)$ i.e the lines which demarcate battlefield one allocation such that we have even battlefield two allocation when acted upon by the resource constraint of player one give the start of the interval in which E_2 must be placed within. The other end of the interval can be arived at by a similar comparion between $n = 1$ and $n = 2$ case and so for a general blotto competition in a partition n we have

$$E_2 \in (f(h^{n-2}(E_1)), f(h^{n-1}(E_1))] \tag{21}$$

We see that as n increases we get to the stage where the interval E_2 grows narrower and practically we get closer to the scenario where both sides have equal resources.

Having thought about the partition a scenario of the Blotto game may lie in it is also worth trying to define the sets which contain the acceptable allocations for a Nash Equilibrium for both players. Again we shall stick to considering the $n = 2$ partition case (i.e. where $E \in [\frac{1}{2}, \frac{2}{3}]$) and then look to generalise the way we construct the set further on in the exposition. From the lines drawn in FIG PANEL A and B we can see that the regions are bounded by the lines given by our functions h and p. Looking at these lines, the regions they map out and the overall resource constraint for each player we can define for partition $n = 2$ in the case where $E \in [\frac{1}{2}, \frac{2}{3}]$, the regions for Blotto's allocation such that we have a Nash Equilibrium as being the set of points given by

$$T_1^b \equiv \{(b_1, b_2) : (b_1 \geq h(E_1), \quad b_2 \geq E_2, \quad b_2 \leq f(b_1)\} \tag{22}$$

$$T_2^b \equiv \{(b_1, b_2) : (b_1 \geq E_1, \quad b_2 \geq p(E_2), \quad b_2 \leq f(b_1)\} \tag{23}$$

The regions for the Enemy's troop allocations can be correspondingly described by the sets

$$T_1^e \equiv \{(e_1, e_2) : (e_1 > f^{-1}(p^{-1}h(E_2)), \quad e_2 \geq f(E_1), \quad e_2 \leq g(e_1)\} \tag{24}$$

$$T_2^e \equiv \{(e_1, e_2) : (e_1 > f^{-1}(E_2), \quad e_2 > f(h^{-1}(E_1)), \quad e_2 \leq g(e_1)\} \tag{25}$$

The lines given by all the key functions used in the sets defined in equations (22), (27) and (24), (29) are drawn in FIGURE 5. This further illustrates the significance of the functions h and p in finding the regions of allocations that constitute a Nash Equilibria. In practice, this region can be thought of as the region of surplus resources Blotto has such that if a point is chosen in this region Blotto can outperform the Enemy. Graphically one can see this in figure (2) and the functions h and p make greater intuitive sense as they demarcate the regions of superior resources as well as illustrating their original definition of enforcing an equal number of resources on the opposite battlefield. To explain why we enforce conditions of $b_1 \geq h^{2-1}(E_1)$ we can think back to the definition of h and remark that if the enemy assigns E_1 to battlefield 1 we assign $h(E_1)$ such that there is an equal allocation to battlefield 2. An equal allocation means we have a tie on this battlefield and so if we use our greater resources to ensure $b_1 \geq h^{2-1}(E_1)$ we map out part of a region where Blotto beats the Enemy (the full region can subsequently be mapped out by considering the condition on b_2.

While we have been considering the regions T_1^b, T_2^b, T_1^e and T_2^e as demarcating the allocations of resources for both players which results in a Nash Equilibrium so far we have been making an implicit assumption about the probabilities with which strategies from specific regions are played. For those regions to indeed be points that yield Nash Equilibria we have to ensure that any deviation from a player does not improve the payoff. If we just momentarily consider the regions for Blotto, T_1^b and T_2^b in figure 2 we see that for an arbitrary enemy allocation $x, g(x)$ the probability of Blotto playing in the part of set T_1^b where Blotto loses on battlefield 1 must be less than or equal to the probability of Blotto playing in the region of T_2^b where Blotto would win on battlefield two multiplied by the associated payoff for all acceptable values x of the Enemy allocation. This ensures that no deviation is worthwhile for Blotto and hence the regions T_1^b, T_2^b, T_1^e and T_2^e are indeed Nash Equilibria. To formalise this we can define the sets (as done by Macdonell et. al)

$$j_b^{x,i} \equiv \{(b_1, b_2) : ((b_1, b_2) \in T_i^b, b_1 < x)\} \tag{26}$$

$$k_b^{x,i} \equiv \{(b_1, b_2) : ((b_1, b_2) \in T_{i+1}^b, b_2 \geq g(x))\} \tag{27}$$

$$j_e^{x,i} \equiv \{(e_1, e_2) : ((e_1, e_2) \in T_{i+1}^e, b_1 \leq x)\} \tag{28}$$

$$k_e^{x,i} \equiv \{(e_1, e_2) : ((e_1, e_2) \in T_i^e, e_2 > f(x))\} \tag{29}$$

where for the case of Blotto $j_b^{x,i}$ may be thought of as the set of actions within T_i^b where Blotto loses on battlefield one for an enemy allocation of $x, g(x)$. Similarly $k_b^{x,i}$ may be thought of as the set of actions in T_{i+1}^b where Blotto wins on battlefield two. If we let S_i be the set of actions a player can take then in our case an element of S_i would be an ordered pair of numbers signifying the number of soliders allocated to each battlefield. Every associated action, in general, is chosen with a certain probability and so since we assign probabilities to a set of actions we can define the probability measures μ_b and μ_e respectively for Blotto and the enemy. A pure strategy is simply a mixed strategy where the probability of playing a certain action has an associated probability of 1. We then place the probability measures in the context of our regions T_i^b and T_i^e such that we can obtain a set of probabilities measures Ω^B for each allocation region. It follows then we can enforce the following properties on the probability measures such that our designated regions are Nash Equilibria.

The first property we enforce for the case of $n = 2$ the probability of playing an action in the region T_i^b can be expressed in terms of the payoffs as follows

$$\mu_b(T_i^b) = \frac{w^{2-i}}{\sum_{j=0}^{1} w^j} \quad \text{for all i = 1, 2} \tag{30}$$

we quickly see that if all payoff values w are the same we have that the probability of an allocation in T_i^b is $\frac{1}{2}$. And then to enforce that no deviation is payoff increasing in the regions of Nash Equilibira we require that for the probability measures μ_b and μ_e

$$\forall x \in [h^{2-i}(E_1), f^{-1}(p^{i-1}(E_2))] \quad \text{we have } \mu_b(j_b^{x,i}) \leq \mu_b(k_b^{x,i})w \tag{31}$$

$$\text{for all } i < 1, 2 \tag{32}$$

We immediately see that two similar conditions can also be enforced on μ_e the probability measure of the enemy playing an allocation.

$$\mu_b(T_i^e) = \frac{w^{i-1}}{\sum_{j=0}^{2-1} w^j} \quad \text{for all i = 1, 2} \tag{33}$$

and as a similar condition to enforce the deviation to another strategy while is a region of Nash Equilibria does not increase the payoff we have

$$\forall x \in [f^{-1}(p^{i-1}((E_2)), h^{2-i-1}((E_1)] \quad \text{we have } \mu_b(j_e^{x,i}) \leq \mu_e(k_e^{x,i})w \tag{34}$$
$$\text{for all } i < 1 \tag{35}$$

Thus we have characterised the set of Nash Equilibira for the two partition case where $E \in [\frac{1}{2}, \frac{2}{3}]$. We have seen the use of the functions h and p in identifying the regions of Nash Equilibria and the associated properties we must enforce on our probability measures such that they are in fact Nash Equilibria. The aim now is to extend the anaylsis to the general n partition case and we proceed in doing this by again considering differences between the partition $n = 2$ and $n = 1$ and seeing how we can obtain general relations to find the regions T_i^b and T_i^e for n partitions. Since for every increase in the number of partitions of the system we see that for that associated n partition we apply the h function $n - i$ times in the case of the lower bound for the blotto battlefield allocation on battlefield one and for a similar case with the p function which is used to demarcate the lower bound of the battlefield two allocation for blotto regions we arrive at the following set of expression giving all the possible allocations in the partition case n provided $n \geq 2$. These sets are defined by Macdonell et al. as,

$$\begin{aligned} T_i^b &\equiv \{(b_1, b_2) : (b_1 \geq h^{n-i}(E_1), b_2 \geq p^{i-1}(E_2), b_2 \leq f(b_1)\} \quad \text{for all i = 1,...,n} \\ T_i^e &\equiv \{(e_1, e_2) : (e_2 > f^{-1}(p^{i-2}(E_2)), e_2 > f(h^{n-i-1}(E_1)), e_2 \leq g(e_1)\} \quad \text{for all i = 2,...,n - 1} \end{aligned} \tag{36}$$

The generalisation for the allocation of the probability mass follows the argument that in the n partition case we must play in the regions T_i^b with a probability of $1/n$ and so in the more general n partition case the probability mass allocation properties become

$$\mu_b(T_i^b) = \frac{w^{n-i}}{\sum_{j=0}^{n-1} w^j} \quad \text{for all i = 1, ...,n} \tag{37}$$

and

$$\mu_b(T_i^e) = \frac{w^{i-1}}{\sum_{j=0}^{n-1} w^j} \quad \text{for all i = 1,...,n-1} \tag{38}$$

The idea that the deviation with full expenditure does not increase the payoff if just a single player chooses to do so also can be generalise when considering that for the μ_e case, the interval under consideration is given on one side by how many times our h "equal allocation" function splits the space in which we can distribute resources. As a result, the mass deviation restriction properties in the general case become

$$\forall x \in [h^{n-i}(E_1), f^{-1}(p^{i-1}(E_2))] \quad \text{we have } \mu_b(j_b^{x,i}) \leq \mu_b(k_b^{x,i})w \tag{39}$$
$$\text{for all } i < 1, 2, ..., n-1 \tag{40}$$

and

$$\forall x \in [f^{-1}(p^{i-1}((E_2)), h^{n-i-1}((E_1)] \quad \text{we have } \mu_e(j_e^{x,i}) \leq \mu_e(k_e^{x,i})w \tag{41}$$
$$\text{for all } i < 1, ..., n-1 \tag{42}$$

If we enforce these properties on the sets defined for our system it follows on from the properties that the following three theorems hold. We discuss why this is the case in section (0.8) and simply now present them and then use them to find a general why to characterise the expected payoff depending

on the proportion of resources the Enemy receives compared to Blotto. Having established the key properties and framework for the analysis of the possible Nash equilibria we quote without proof three key theorems from Macdonell et al. (a discussion of the proofs may be found in section 0.8)

Theorem 0.5.1 (Payoffs). *If Blotto plays a strategy $\mu_B \in \Omega^B$ and the Enemy plays a strategy $\mu_E \in \Omega^E$ the payoff for Blotto and the enemy are*

$$\textit{Blotto payoff:} \quad \frac{\sum_{j=0}^{n} w^j}{\sum_{j=0}^{n-1} w^j} \quad \textit{, Enemy payoff:} \quad \frac{\sum_{j=1}^{n-1} w^j}{\sum_{j=0}^{n-1} w^j}$$

Theorem 0.5.2 (Nash Equilibrium). *For $\mu_B \in \Omega^B$ and $\mu_E \in \Omega^E$ a pair of strategies $\{\mu_B, \mu_E\}$ are a Nash Equilibrium*

Theorem 0.5.3 (Set of Nash Equilibrium). *The complete set of Nash Equilibira of the two battlefield variant of Colonel Blotto is the set $\{\mu_B, \mu_E\}$ where $\mu_B \in \Omega^B$ and $\mu_E \in \Omega^E$.*

From the expected payoff theorem and the enforced probaility mass distribution we see that increasing the parition to the $n = 3$ case would yeild an expected payoff for Blotto of $\frac{4}{3}$ and an Enemy pay off of $\frac{2}{3}$. We note thus that as we get a more partioned system the payoffs become more and more equal verifying the idea that as payoffs increase we reduce the asymmetry in the resource allocation. We conclude by remarking that for the general Blotto to Enenmy resource allocation expression

$$E \in (\frac{n-1}{n}, \frac{n}{n+1}) \tag{43}$$

the expected payoff for Blotto is $\frac{n+1}{n}$ and the corresponding payoff for the enemy is $\frac{n-1}{n}$. The notion of Nash Equilibria in the case of an infinite number of strategies translates to the case of there being infinite corresponding Nash Equilibria. In practice a region of infinite Nash equilibria may be treated by just considering the support of the sets containing the points which constitute a Nash Equilibria and computationally a sensible discretisation of the number of feasible points should be considered such that the situation may be computationally tractable.

0.6 Regret Learning

Regret Matching

Regret learning is an approach to decision-making where by we consider what alternative strategies would have been superior to the strategy that was indeed chosen. The amount by which alternative strategies would have been better can be quantified and is considered the regret. We wish to then proceed in such a way that we choose strategies that minimise this regret hence leading to improved performance and payoff in the game.

The concept of regret matching is fundamental to regret-learning algorithms and often forms the core of more advanced regret-based learning algorithms such as counterfactual regret minimisation [6]. The idea of regret matching is that for a game that is iteratively played a player making use of regret minimisation will at each iteration of the game compare the strategy it played to all the possible strategies. A measure termed the regret is calculated by comparing the actual obtained payoff the player received to the payoffs the player could have received had they implemented a different strategy. In practice the regret may be obtained by subtracting the possible payoff from the alternative strategy from the actual payoff received. We see then that in cases where the player

played a strategy with the highest possible payoff the regret the player incurs will be negative and thus we can think of the player as experiencing "positive regret". Alternatively, if the player played a strategy that obtained a much lower payoff than was possible from a superior strategy the player will incur regret. This can then be used to inform the future strategies chosen when the game is played the next time around. In practice, there will be a wide range of alternative strategies a player could have played. Not all will induce the same amount of regret (i.e. some strategies could be better but not that much better when compared to the best alternative strategy) and so to inform future play a player implementing a regret matching scheme will choose its strategy the next time around with a probability proportional to positive regrets values. The benefit to choosing future strategies in this probabilistic way is that we still learn via our regret which strategies performed better but rather than playing the best possible strategy (i.e. the one which minimised the regret the most) we play better strategies in proportion to how much better they our. This stops the player's moves from being entirely predictable and thus easily played against by an opposing player.

To formalise this we consider a normal form game where the number of players is two, S_i is the set of actions a player i can take each time the game is played and u is a payoff function mapping from every action profile to the utility vector. It is considered that S_i is finite. One way to formalise the regret is a to consider three possible actions from S_i. Here we consider a_0, a_1 and a_2. We then define a function as follows

$$\text{switch}_i(a_0, a_1, a_2) = \begin{cases} a_2 & \text{if} \quad a_0 = a_1 \\ a_0 & otherwise \end{cases}$$

the utility of this function is that we can use it to quantify the regret as follows

$$\text{regret}_i((x_i, x_{-i}), switch_i(x_i, a_1, a_2)) = u_i(x_i, x_{-i}) - u_i(\text{switch}_i(x_i, a_1, a_2), x_{-i}) \tag{45}$$

The regret function $\text{regret}_i(x, \text{switch}_i(x_i, a_1, a_2))$ evaluates the regret a player has when playing a_1 instead of a_2 for a situation where the opponent in both cases in considered to have been playing x_{-i}. This lets us set up a notion of regret however in our analysis we would like to analyse the regret as the game progresses in time and more and more iterations of the game are played. To obtain an expression for the regret upto a point in time we can first consider the difference between what the player played and what they could have played on average up to iteration t of successively playing that game [7].

$$\text{Difference}_1^t(a_1, a_2) = \frac{1}{t}(\sum_{\gamma=1}^{t} u_1(\text{switch}_1^\gamma(x_i, a_1, a_2)) - u_i(x_i^\gamma, x_{-i}^\gamma)) \tag{46}$$

The regret then upto iteration t can be expressed with the following relation

$$\text{Regret}_1^t(a_1, a_2) = \max(\text{Difference}_A^t(a_1, a_2), 0) \tag{47}$$

The use of the maximum function ensures that we only consider the actions which could have benefited us and improved our situation had we played them. In theory if we have always taken the best action at each iteration of the game then the difference function would be negative and so we would have no regrets and hence we should set our regret function to give zero.

The concept of regret matching was first proposed by Hart and Mas-Colell and an important theorem of their work stated that in a normal form game situation with two players, where both employed the use of regret matching, that the "joint distribution of play" converges to the set of correlated equilibria [8].

The joint distribution of the actions of the two players in question is simply the frequencies at which different combinations of actions have occurred up to that point in the process. This can be expressed by an m x n matrix if player 1 has m actions and player 2 has n actions (provided that the frequencies are nonnegative and sum to 1) [8]. A correlated equilibria is an equilibria of a game that is obtained when some kind of trusted process randomly assigns recommendations for the action to be taken by an agent in a way that yields outcomes from which it is not beneficial for an agent to deviate from. Thus as we iterate the process with both agents carrying out regret matching the relative frequencies at which actions are carried out converge to the set of correlated equilibria. This is a crucial result since it suggests that an agent using regret matching eventually converges to the set of correlated equilibria just like a rational agent playing would. This is also useful in a way. Convergence to Nash Equilibria often can lead to situations where, although neither player benefits from unilateral deviation, we are not in a place where we can get the best "overall" outcome for the system when considered holistically. In the correlated equilibrium situation, we include all Nash Equilibria as they are a subset of the correlated equilibria but we also achieve situations where the sum of the overall payoff exceeds what we could achieve in a Nash Equilibrium without the players wishing to deviate from their strategies.

0.7 Colonel Blotto and Regret Learning

We now consider applying the process of regret matching to an instance of the colonel Blotto game as outlined in section 0.1. We focus on the specific case where the number of battlefields is $N = 3$ and the soldiers available to each side are $S = 5$. We restrict the game only to integer allocations of soldiers. To formalise this we express the considered normal form game as with the tuple (p, A, u).

- $p = 1, 2$, where p is the set of all players which in our case is two
- S_i denotes the set of all actions a player can choose
- $A = S_1 \times S_2$ is the set of all possible combinations of actions that can be played
- $u = u_1, u_2$ where u_i is the payoff function for a given player as outlined in section 0.1

In our specfic scenario where $N = 3$ and $S = 5$ we can use our work from section 0.3 to quickly calculate that each player has a possible

$$\frac{(S + N - 1)!}{S!(N - 1)!} = \binom{7}{2} = 21 \tag{48}$$

actions to choose from. This further means that there are a possible 441 action profiles available in the game. The set of all actions a player could take is listed below

$$\{[401], [122], [230], [104], [041], [113], [050], [212], [221], [500], [311], \tag{49}$$

$$[032], [320], [023], [302], [203], [014], [131], [140], [005], [410]\} \tag{50}$$

where the order of the three numbers in each triplet denotes the number of troops sent to the battlefield corresponding to the position of the number in the triplet (e.g [401] is an allocation of four soldiers to battlefield one, zero soldiers to battlefield two and a single solider to battlefield one).

In this context the regret matching algorithm proceeds as follows. If we imagine ourselves to be Blotto, on the first iteration we choose a strategy randomly from the set of all possible actions. Since there are 21 possible strategies we choose each strategy with an equal probability of $\frac{1}{21}$. For illustration purposes, we suppose Blotto chose the action corresponding to the troop allocation [401] and the Enemy chose the troop allocation corresponding to [122]. In this case, we see that the Enemy

wins battlefield two and battlefield three and so Blotto loses this game and receives a payoff of -1. At this point we consider all the possible alternatives that could have been played along with their respective payoffs had they been played againsts the Enemies [122]. These payoffs are

$$[-1, 0, 1, 0, -1, 0, -1, 0, 0, -1, -1, 0, 0, 0, 0, 1, -1, 0, 0, -1, -1] \tag{51}$$

Here the position of the number corresponds to the action he could have chosen and the value of the number is the payoff Blotto would have received had he chosen that action. From this we can then calculate the regret Blotto experiences by subtracting his action from each payoff in the above payoff vector (51) yielding.

$$[0, 1, 2, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 2, 0, 1, 1, 0, 0] \tag{52}$$

This regret then informs our future play. In the next iteration of this game Blotto plays his mixed strategy with the probability of choosing an action proportional to the amount of positive regret incurred. Since in the previous iteration Blotto lost, every other possible action was either just as bad, better by a payoff value of 1 or better by a payoff value of two (i.e. he could not have performed worse) and so there are no negative regrets in this case. The probabilities of the next action he chooses are thus

$$[0, \frac{1}{15}, \frac{2}{15}, \frac{1}{15}, 0, \frac{1}{15}, 0, \frac{1}{15}, \frac{1}{15}, 0, 0, \frac{1}{15}, \frac{1}{15}, \frac{1}{15}, \frac{1}{15}, \frac{2}{15}, 0, \frac{1}{15}, 1, 0, 0] \tag{53}$$

and we see that actions that had a higher regret are chosen more frequently than those with less regret.

In the computer implementation that follows we consider the case where both players use regret matching alternatively when playing against one another. This means that Blotto will update his regret-matched probability distribution every other instance the game is played and in a similar fashion the Enemy will do the same. We then analyse the strategies produced and see if they do indeed converge to strategies that are Nash Equilibria. Before discussing the results of the different scenarios considered we first look at the key functions which facilitated this implementation along with the overall structure of the program that ran the simulation.

An important part of the process is picking which strategy our player using regret matching will play. The idea of the below algorithm is to take only the regret value of each possible action we could have otherwise played where our regret is positive and assign it to our strategy vector. To turn this into a probability vector we normalise by the sum of the values which are inputted into the strategy vector. If we are in the case where all the regrets are negative (i.e. the case where we played one of the best possible strategies) then the value of the normalising sum is zero and hence we assign the strategy to be a probability vector where every possible action is played with equal probability. Finally, we add the strategy probabilities to a running sum that records the cumulative strategies over all training iterations. This entire process is given in algorithm(1)

Algorithm 1 getStrategy

```
number_of_players ← 2
number_of_actions ← 21
number_of_battlefields ← 3
normalising_sum ← 0
for i = 1 TO number_of_actions do
    if regret_sum[player][i] > 0 then
        strategy[player][i] ← regret_sum[player][i]                ▷ This is a comment
    else
        strategy[player][i] ← 0
        normalising_sum = normalising_sum + strategy[player][i]
    end if
end for
for j = 1 TO number_of_actions do
    if normalising_sum > 0 then
        strategy[player][j] ← strategy[player][j]/normalising_sum
    else
        strategy[player][j] = 1/num_actions
    end if
    strategy_sum[player][j] = strategy_sum[player][j] + strategy[player][j]
end for
```

The next core key algorithmic process is to determine the action given a strategy probability vector. We do this by generating a random number from a uniform distribution between $[0, 1]$ and then iterating over the possible number of actions checking if that number is less than the sums of the probabilities up to that point in the iteration. It allows us to assign an action to a player given the probabilities stipulated by the strategy vector. Algorithm (2) outlines this process formally.

Algorithm 2 getAction

```
random_val ← number chosen randomly from uniform distribution [0,1]
counter ← 0
cumulative_prob ← 0
while counter < (number_of_actions - 1) do
    cumulative_prob ← cumulative_prob + strategy[counter]
    if random_val < cumalative_prob then
        action ← counter
    end if
    counter = counter + 1
end while
```

The actions a player can take are central to the entire computational process and so the following algorithm calculates all possible solider allocations a player can make (i.e. the actions they can take) given a number of battlefield and a number of soldiers. Algorithmically this is done by obtaining a list of length $S \times N$ where the values are from 0 to S which can be thought of as being a reference two each solider, and we multiply by N. From this list we calculate all possible permutations of length N. From the list of all possible permutations, we then extract those where the total allocation of soldiers is 5. This yields the number of possible ways a player could distribute his S soldiers over N battlefields and thus is a list of all possible actions a player could take. This process is formalised by algorithm (3)

Algorithm 3 generate_IDs

```
S ← number_of_soliders
N ← number_of_battlefields
permutation_candidates ← list(range(S+1)) * N
permutations ← The list of all successive N length permutations of the list permutation_candidates
for i in permutations do
    if sum(i) == S then
        ID_array ← i
    end if
end for
```

An important part of calculating the regret, which we need to ultimately inform our next choice of action, is calculating the payoff between two different chosen actions. This is done by looping through every solider allocation for each battlefield and comparing the numbers assigned. As previously depending on the allocations we record the number of wins, loses and draws and return the appropriate payoff. In this process we obtain the payoff for player one against player two however to achieve the opposite (i.e. player two against an action player one played) we simply swap the inputs to the algorithm. The algorithm for this is given by algorithm (4).

Algorithm 4 utility

```
battlefields_won ← 0
battlefields_lost ← 0
battlefiethe lds_drawn ← 0
for i in range(number_of_battlefields) do
    if No. of soldiers of p1 on battlefield i > No. of soldiers of p2 on battlefield i then
        battlefields_won ← battlefields_won + 1
    else if No. of soldiers of p1 on battlefield i == No. of soldiers of p2 on battlefield i then
        battlefields_drawn ← battlefields_drawn + 1
    else if No. of soldiers of p1 on battlefield i < No. of soldiers of p2 on battlefield i then
        battlefields_lost = battlefields_lost + 1
    end if
end for
if battlefields_won > battlefields_lost then
    utility ← 1
else if battlefields_won == battlefields_lost or battlefields_drawn == battlefield_num then
    utility ← 0
else
    utility ← −1
end if
```

For the algorithm to obtain the next strategy from the regrets experienced we need a way of obtaining the payoff from every other possible action we could have played against where the opponent played. The process in algorithm 5 uses algorithm 4 and loops through all the possible actions which could have been played and calculates the corresponding utility.

Algorithm 5 get_all_payoffs

```
for i in range(number of possible actions) do
    utility_array[i] ← utility value of i played against given action
end for
```

Having put in to place all the key parts of the computational process it is left for us now to train the agent in question. To train the agent we simply use the algorithms previously outlined to obtain a strategy and then to obtain an action based on the probabilities given by that strategy. We then find the payoffs of all possible actions we could have taken and subtract the action we did take for the agent in question such that we obtain the regret. This regret is then alternately updated such that we carry out regret matching alternately for each player. The process for this is given in algorithm (6).

Algorithm 6 training

```
for i in range(training iterations) do
    blotto_strat ← getStrategy(blotto)
    blotto_action ← getAction(blotto_strat)
    enemy_strat ← getStrategy(enemy)
    enemy_action ← getAction(enemy_strat)
    utility_array_blotto ← get_all_payoffs(enemy_action)
    utility_array_enemy ← get_all_payoffs(blotto_action)
    regret_blotto ← utility_array_blotto - utility_array_blotto[blotto_action]
    regret_enemy ← utility_array_enemy - utility_array_enemy[enemy_action]
    if i % 2 is equal to 0 then
        regret_sum[blotto] ← regret_sum[blotto] + regret_blotto
    else if i % 2 is equal to 1 then
        regret_sum[enemy] ← regret_sum[enemy] + regret_enemy
    end if
end for
```

The final part of the computer implementation is to then compute the average strategy. Since the regret sums themselves can fluctuate dramatically from iteration to iteration we take the average strategy over all iterations as the final strategy the regret matching algorithm provides.

Blotto Regret Matching Results

Before numerically presenting the results we recall that for a strategy to be a Nash Equilibrium we require it to be in the set of best responses to that strategy. To think about what equations would need to be satisfied such that the strategy chosen is a Nash Equilibrium we would require the formulation of a matrix where the element a_{ij} is the payoff obtained of a pure strategy i being played against a pure strategy j. In our set up this is a rather large matrix since there exists for each player 21 pure strategies. For the colonel Blotto game, for the strategy to be an interior Nash Equilibrium, we require that

$$(Ax)_1 = (Ax)_2 = ... = (Ax)_{21} = c \tag{54}$$

Hence to find the exact interior Nash Equilibrium one would have to solve the system

$$Ax = c \tag{55}$$

where A is the payoff matrix and c is the vector of constant values which indicates that the vector x, the strategy lies in the set of best responses and further is indeed a Nash equilibrium. For

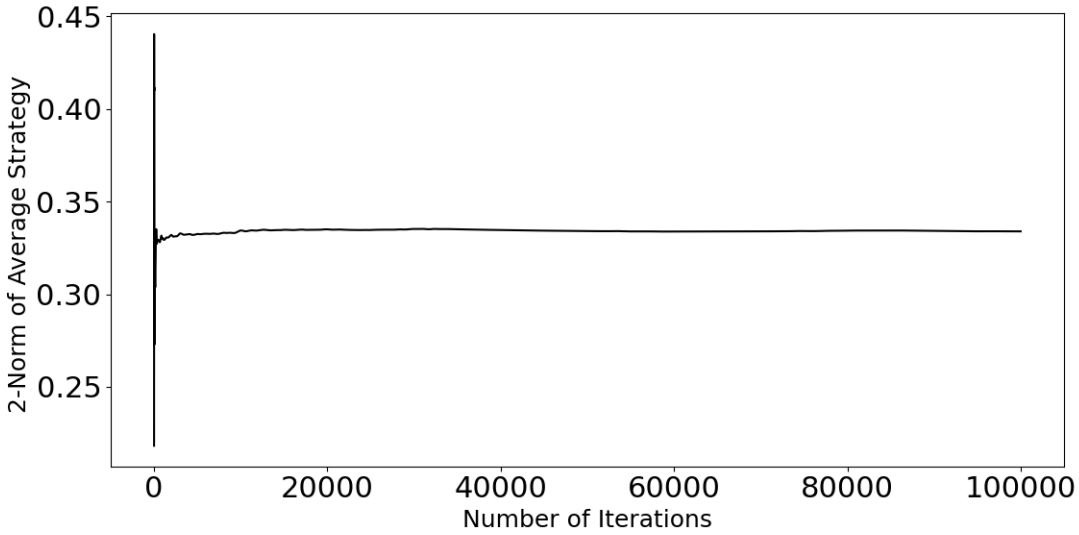

Figure 3: The number of iterations until values for player one converged

saddle point and board Nash equilibria we require that the lines of indifference intersect or lie on the lines composed from the convex combination of pure strategies. The strategies obtained by the regret matching algorithm all fall into the set of Nash Equilibria and into the wider set of correlated equilibria.

Since we converge to the set of correlated equilibria rather than a single point we get different strategies depending on the instance the program is run however for illustrative purposes we give the output of the program for both players for the case where the number of training iteration is 100,000. The outputted mixed strategies for blotto and the enemy were,

player 1 (blotto): [0.00005026, 0.00012307, 0.11221987, 0.00019274, 0.00005377, 0.10633304, 0.00000048, 0.00011795, 0.00024282, 0.00000048, 0.10575195, 0.11668019, 0.11351176, 0.11349998, 0.11304385, 0.11235979, 0.00011249, 0.10557318, 0.00008597, 0.00000048, 0.00004587]

player 2 (enemy): [0.00002419, 0.00002664, 0.10454329, 0.00001579, 0.000025, 0.11517811, 0.00000356, 0.00012176, 0.00003585, 0.00000095, 0.10950651, 0.1115391, 0.11575252, 0.10702005, 0.10862798, 0.11054649, 0.00001266, 0.1169863, 0.00002092, 0.00000095, 0.00001136]

where the position in each list is one of the 21 possible actions which may be played and the value of the list is the probability with which that action is played. The order of these actions is given in equation (49).

When considering convergence the process to assess the convergence was to find the 2-norm of the average strategy the algorithm calculated for each player up to that point in the training iteration. The use of the 2-norm prevented any issues with negative and positive values canceling out one another and as the solution given by the algorithm converged so did the value of the 2-norm of the average strategy for that iteration. By the theorem of Hart and Mas-Colell as long as the normalsing constant for the regret or transitioning between two actions is large we tend to the set of correlated equilibria as time tends to infinity. Looking at the figure (4) and at the figure (5) we see that after

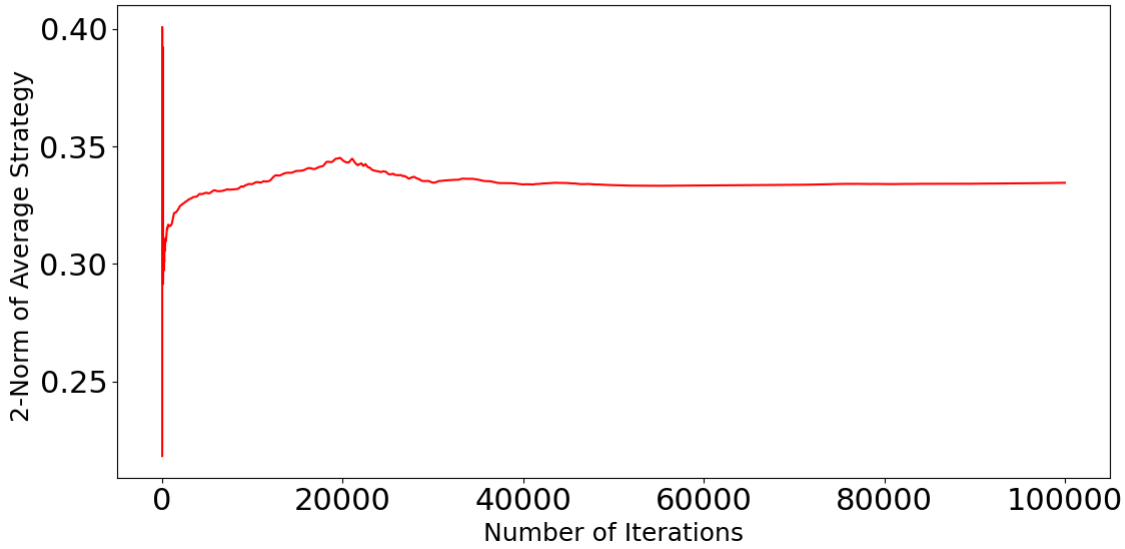

Figure 4: The number of iterations until values for player two converged

20,000 iterations we reach the point where numerically the tolerance between successive iterations lies below $1e-5$ and so we can claim convergence in accordance with the theory however in the case of player two we require far more iteraions (60,000) to reach the same level of convergence. While this is still in accordance with the theory the difference is striking. An investigation to see what the driving cause behind such behavoiur could be to check the 2-norm values of the regret sum to see if any potential erratic behaviour iteratin to iteration is having an impact on the overall convergence [6].

0.8 Proofs

In this section we return to the case of the Colonel Blotto game where the resource allocation is uneven and there are non-integer troop allocations to each battlefield. This is the problem we discussed in section (0.5) and here we aim to focus on the first two theorems outlined in that section and hope to give a fuller understanding of these theorems as well as why they are true. Again the exposition of the proofs is based the writtings by Macdonell et. al [5]

Before embarking on the analysis we first introduce two sets which allow us to pick out the first and second pair in an ordered pair containted within a set as done by Macondell et. al S

$$\Psi_1(S) \equiv \{x_1 \mid (\exists x_2 \in \mathbb{R} \quad \text{such that} \quad (x_1, x_2) \in S)\} \tag{56}$$
$$\Psi_2(S) \equiv \{x_2 \mid (\exists x_2 \in \mathbb{R} \quad \text{such that} \quad (x_1, x_2) \in S)\} \tag{57}$$

The usefulness of this operator Ψ is that if we have a set S, containing the allocations for Blotto and the allocations for the Enemy as an ordered pair where the first ordered pair is for Blotto and the second ordered pair is for the Enemy then we can use $\Psi_2(S)$ to find the number of soliders both Blotto and the Enemy sent to battlefield two. The same applies for battlefield one and $\Psi_1(S)$.

For convenience, we restate theorem 1 below

Theorem 0.8.1 (First Theorem: Payoffs)**.** *If Blotto plays a strategy $\mu_B \in \Omega^B$ and the Enemy plays a strategy $\mu_E \in \Omega^E$ the payoff for Blotto and the enemy are*

$$\textit{Blotto payoff:} \quad \frac{\sum_{j=0}^{n} w^j}{\sum_{j=0}^{n-1} w^j} \quad \textit{, Enemy payoff:} \quad \frac{\sum_{j=1}^{n-1} w^j}{\sum_{j=0}^{n-1} w^j}$$

To explain why this is the case we must begin by considering the problem from the perspective of the Enemy. If we are in the situation where the game is partitioned in an arbitrary state n we consider the situation where the Enemy plays a resource allocation from the set T_i^e (this set was introduced in section (0.5)). For the enemy to win on battlefield one we require that his resource allocation is greater than Blotto. For a strategy in T_i^e, we propose that any point for the enemies resource assignment to battlefield one denoted by e_1 is greater than Blotto's resource assignment to battlefield one (b_1) if that assignment came from a region $T_{i-1}^b, T_{i-2}^b, ..., T_1^b$. Using our operator Ψ we can express this as being the case where

$$b_1 \in \Psi_1(T_{i-1}^b) < e_1 \in \Psi_1(T_i^e) \tag{58}$$

and

$$e_1 \in \Psi_1(T_i^e) < b_1 \in \Psi_1(T_i^b) \tag{59}$$

to explain why this is the case we can refer to the two partition case section 0.5. We can see that if we choose the region T_2^e that for battlefield 1 its greatest lower bound (given by $f^{-1}(p^{-1}(E_2))$) is strictly less than the greatest lower bound of the corresponding region for Blotto T_2^e which is $h(E_1)$. Hence relation 59 is verified and we can quickly see that it holds for the general case for all $i \in 1, ..., n$. In the boundary case where $i = 0$ or $i = n$ we note that the strict inequalities become non-strict however the same reasoning holds. As a result, the chance that the enemy wins on battlefield 1 given that it plays a strategy from the set T_i^e is the probability that Blotto plays an allocation from the sets $T_1^b, T_2^b, ..., T_{i-1}^b$. We note that we only include the sets up to T_{i-1}^b due to the inequalities outlined in (62) and (63). Hence the chance of winning on battlefield 1 for the Enemy is

$$\mu_B(T_1^b \cup ... \cup T_{i-1}^b) \tag{60}$$

where μ_b is the probability measure of playing a certain strategy. This probability is equivalent to the sum of probabilities masses for the regions upto T_{n-1} starting with the probability mass of the region $T_{n-(i-1)}$ divided by the probability mass of the all the regions. The reason for this is that the Enemy only wins on battlefields with the index i or larger. This yields the chance of the Enemy winning on battlefield one as being

$$\frac{\sum_{j=n-i+1}^{n-1} w^j}{\sum_{j=0}^{n-1} w^j} \tag{61}$$

as given by Macdonell et al. which is the probability of the Enemy winning on battlefield 1. The exact line of reasoning can be applied to the probability of the enemy winning on battlefield two provided he plays in a region T_i^e. In the case of battlefield two allocations applying the same reasoning as for battlefield one we obtain the inequalities,

$$b_2 \in \Psi_2(T_{i+1}^b) < e_2 \in \Psi_2(T_i^e) \tag{62}$$

and

$$e_2 \in \Psi_2(T_i^e) < b_2 \in \Psi_2(T_i^b) \tag{63}$$

Hence the probability of the Enemy winning on battlefield two is equivalent to the probability Blotto plays an allocation from the sets $T_{i+1}^b, ..., T_n^n$. The chance of winning then can be expressed in terms of the probability masses with the same reasoning as applied in the case of battlefield one the only difference being our indexes from the sums are derived from relations (66) and (67),

$$\mu_B(T^b_{i+1} \cup ... \cup T^b_n) = \frac{\sum_{j=0}^{n-i-1} w^j}{\sum_{j=0}^{n-1} w^j} \tag{64}$$

in other words we find the proportion of probability masses upto one less than the region T_i that the enemy selects to choose their battlefield two resource allocation from. Following on from expectation values the total expectation value is the probability that the Enemy wins on battlefield 1 added to the probablity that the Enemy wins on battlefield two multiplied by the payoff of winning a battlefield. By laws of indecies this simplifies to

$$\frac{\sum_{j=1}^{n-1} w^j}{\sum_{j=0}^{n-1} w^j} \tag{65}$$

as given by the theorem.

We now look to show why the second claim of the theorem for the expected payoff of Blotto also is as is given by the theorem. If Blotto picks a resource allocation in the region T^b_i we can conclude (in a similar way as in the case of the Enemy) that in the case of battlefield one allocation the greatest lower bound of T^b_i is strictly greater than the greatest lower bound for the enemy in the corresponding region i. The greatest lower bound of region T^b_i for battlefield one is also greater than for any regions less than i. Thus by an extension of the relations (62),(63),(66),(67) we see that the probabiliy that Blotto wins on battlefield one having played an allocation from T^b_i is given by the probability that the enemy play an allocation from $T^e_1, T^e_2, ..., T^e_i$. This probabiliy is equal to the proportion of probability masses upto the point $i-1$ over all probability masses and so the chance Blotto wins on battlefield one having played T^b_i is given by Macdonell et al. is

$$\frac{\sum_{j=0}^{i-1} w^j}{\sum_{j=0}^{n-1} w^j} \tag{66}$$

in a similar way for battlefield two, if Blotto chooses a b_2 allocation from the set T^b_i we note that all enemy allocations from sets T^e_i where i is equal to or greater than the i of the set Blotto chose to play from (T^b_i) yields a win for Blotto. Hence winning for Blotto is given by the probabiliy that the enemy plays a battlefield two allocation from $T^e_i, T^e_{i+1}, ..., T^e_n$. Expressing this in terms of probability masses we get

$$\frac{\sum_{j=i-1}^{n-i} w^j}{\sum_{j=0}^{n-1} w^j} \tag{67}$$

To get the expected payoff we proceed as before multiplying the probability of winning on battlefield two by the payoff of winning a battlefield and then adding the probability of winning battlefield one given Blotto played a resource allocation from the set T^b_i. Upon some manipulation this yields the second result of the theorem

$$\frac{\sum_{j=1}^{n} w^j}{\sum_{j=0}^{n-1} w^j} \tag{68}$$

Theorem 0.8.2 (Nash Equilibrium)**.** *For $\mu_B \in \Omega^B$ and $\mu_E \in \Omega^E$ a pair of strategies $\{\mu_B, \mu_E\}$ are a Nash Equilibrium*

To explain why this property holds for all choices of appropriate probability measure for a strategy played in the non-integer allocation of the Blotto game we begin by noting that should these pairs of strategies $\mu_b \in \Omega^B$ and $\mu_e \in \Omega^E$ not constitute a Nash Equilibria. We briefly consider the case where a player plays a strategy that deviates from one of μ_B or μ_E such that the deviation is not

with all the entirety of the resource allocated. In other words, it is not a full expenditure deviation. If we are in the case of non-full expenditure deviations we immediately remark that they could not be payoff improving since the expected value of the payoff is either the same or increasing and so without a full expenditure commitment no improvement of the payoff is possible and so we need to consider only full expenditure payoffs.

For an allocation b_1, b_2 in the region T_i^b we consider a corresponding deviation where we choose a b_1^{dev}, b_2^{dev} such that they are not within any T_i^b. Regardless of the region T_i^b we deviate from provided we are not at $i = n$ or $i = 0$ we always have the case that there will lie a region of allocations b_1^{i+1}, b_2^{i+1} which is larger than your deviation and a region of allocation b_1^i, b_2^i that is smaller than your deviation. Specifically in the $i + 1$ region, we will have that $b_1^{dev} < b_1^{i+1}$ and that in the i region $b_2^{dev} < b_2^i$. For a selection b_1^{dev}, b_2^{dev} we then see that there is no payoff benefit to deviating with full expenditure from T_i^b. The only possible places where this could give an advantage on one battlefield are if the enemy strategies lie within T_i^e or T_{i+1}^e. It follows then that although the payoff may be higher on one battlefield the corresponding payoff on the other battlefield will be less and the probability of this occurring will be greater. As a result, the expected payoff is less and deviating from T_i^b yields no benefit in the payoff and so there is no benefit from unilaterally deviating. The same argument follows when we consider deviations from T_i^e and so we can conclude that the theorem holds.

0.9 Theoretical Arguments Behind Regret Learning

In this section we build on the ideas of regret learning and focus specifically on the use of Blackwell's approachability theorem to show that the regret may be minimised in a regret learning algorithm. For the purposes of this discussion, we shall only consider normal-form games in their zero-sum version. Our discussion is guided by [9][10]

To begin with we consider a central theorem from Von Neumann known as the Minimax theorem. If we considered mixed strategies for a game the theorem elegantly states that the greatest value of ones minimum gain is equal to the minimum value of the opponent maximum loss. Since we are working with mixed strategies the gain and loss are both the expected gain and loss. More formally if we define our game in normal form with the tuple (N, S, u) where N is the number of players, S is the set of all possible combinations of actions that may be taken and u is the set of payoff functions for each player. We let S_i be the set of all actions the player i can take and for a mixed strategy we can define $\sigma\_i(s)$ to be the probabilitiy that the player i chooses the action s in the set of actions S_i. We let the set of all mixed stratgies for player i be Δ_{σ_i} and the set of all mixed strategies for player $-i$ the opponent be $\Delta_{\sigma_{-i}}$ For a mixed strategy the payoff for player i where both player one and player two play a mixed strategy is

$$\sum_{\alpha \in S_i} \sum_{\beta \in S_{-i}} \sigma_i(\alpha)\sigma_{-i}(\beta)u_i(\alpha, \beta) \tag{69}$$

for some actions α and β. Note we let $-i$ denote the opposing player. The minmax theorem thus is expressed as

$$\max_{\sigma_i \in \Delta_{\sigma_i}} \min_{\sigma_{-i} \in \Delta_{\sigma_{-i}}} \sum_{\alpha \in S_i} \sum_{\beta \in S_{-i}} \sigma_i(\alpha)\sigma_{-i}(\beta)u_i(\alpha, \beta) = \min_{\sigma_{-i} \in \Delta_{\sigma_{-i}}} \max_{\sigma_i \in \Delta_{\sigma_i}} \sum_{\alpha \in S_i} \sum_{\beta \in S_{-i}} \sigma_i(\alpha)\sigma_{-i}(\beta)u_i(\alpha, \beta) \tag{70}$$

Having established the minimax theorem we look to consider the class of games where the payoffs are no longer scalars but rather they are vectors. We wish to obtain some kind of similar theorem that could then be used to exploit all the properties of systems to which the minimax theorem applies.

The minimax theorem works in the specific context where we have a normal form zero sum game with mixed strategies however the idea can be extended to more general mappings between sets. We quote without proof Sion's Minimax theorem which allows us to further our analysis

Theorem 0.9.1 (Sion Minimax)**.** *For the sets I and K which are convex and compact and a function $f : I \times K \to \mathbb{R}$ which is convex for $f(i, \cdot)$ and concave for $f(\cdot, k)$ we have the relation*

$$\inf_{i \in I} \sup_{k \in K} f(i, k) = \inf_{k \in K} \sup_{i \in I} f(i, k) \tag{71}$$

Thus if we propose to have a situation where the first player plays an action a from the set I and the second player plays an action b from the set K we can consider the case of games with a vector payoff. We can formalise a game with a vector payoff by again considering player one to a strategy s $\in$S$_i$ and player two playing a strategy $s_{-i} \in S_i$. For this action profile (s_i, s_{-i}) the payoff $p(s_i, s_{-i})$ is no longer a number but rather is a vector in $\mathbb{R}^n$ where n is finite. Hence we can consider the first player's payoff to be $p(a, b)$. From the minimax theorem we know that in the scalar case that the is a mixed action that player one chooses s_1 played against any pure strategy of player two yields the case that the payoff of that combination is greater or equal to the value of the zero-sum game for that payoff. In the vector setting, if there is a set Y and that if for all actions which could be chosen from the set I there exists an action from the set K such that the payoff $p(i, k) \in Y$ there is also an element in I where the converse is true and for all elements of K the payoff $p(i, k)$ is an element of the set Y. To show that something close to this notion is possible we consider the case of iterated play whereby the first player plays $i \in I$ repeatedly and the second player plays $k \in K$ repeatedly we can express the payoff loss in the vector-valued zero-sum game as being

$$\text{loss} = \frac{1}{n} \sum_{j=1}^{n} p(i_j, k_j) \tag{72}$$

where n is the number of times we repeated the game. We can then proceed to define a distance metric between the average loss after a certain amount of iterations and the arbitrary set Y which we used initially to wish properties upon the payoff function. The distance between the loss over n iterations and a point $y \in Y$ was (as supported by Sion Minimax)

$$d(\text{loss}, Y) = \inf_{y \in Y} \parallel \text{loss} - y \parallel \tag{73}$$

we can then define the term approachable where by the set Y is considered to be approachable if it is possible to find a repeated strategy where player one and player two play $i \in I$ and $k \in K$ repeatedly such that

$$lim_{n \to \infty} d(\text{loss}, Y) = 0 \tag{74}$$

We can now introduce Blackwell's Theorem which states

Theorem 0.9.2 (Blackwell's Theorem)**.** *For a set Y which in compact and convex in $\mathbb{R}^2$ with $\parallel x \parallel \leq Rforallx \in Y$. If for all $k \in K$ there exists $i \in I$ such that the payoff function $p(k, i) \in Y$ then Y is approachable and there exists a strategy where the distance between the average loss up to that point in the iteration and Y is*

$$d(loss_n, Y) \leq \frac{2R}{\sqrt{n}} \tag{75}$$

Using this key theorem it can be shown that for the process of regret minimisation on a standard domain, we can use Blackwells approachability theorem to guarantee that as the number of iterations tends to infinity we can minimise the regret of the system to zero.

0.10 Reinforcement Learning Algorithm

In this section we aim to look at training an agent to successfully play the colonel Blotto game as configured in section 0.7 (with $N = 3$ and $S = 5$) by using a form of reinforcement learning known as Q-Learning. From a high-level perspective, the overall concept behind any reinforcement learning algorithm is to expose an agent to an environment where in this environment it takes many different possible actions. Every time an action is taken the agent receives a reward, be it negative or positive, and the size and nature of the reward is then used to inform its future actions with the specific objective of maximising the reward over time. This is a very general idea and can be applied to a wide range of situations. Thinking about this in relation to the colonel Blotto game we can quickly infer that the environment is the action space and the payoffs can be used as the rewards received for an agent as it plays different strategies. Indeed the way normal-form games are formulated makes them very amenable to reinforcement learning without the problem of having to undergo prepossessing or reformulation as is the case with more general topics where reinforcement learning may be applied.

The idea behind Q-learning is that we choose what we think is the best possible choice of action out of all possible actions based on the present state of the system. We remark at this point that Q-learning is commonly referred to as model-free learning since there is no effort made to understand the underlying statistical distribution and mathematical equations which may govern the system. We simply choose actions based on the current state and the best possible reward we may achieve. At the heart of Q-learning lies the Q-table. The Q-table can be represented in matrix form where we assign a column to each possible action that could be taken and a row to the state. In our case for Colonel Blotto, the actions are simply how the five troops may be allocated to the three battlefields and the state we can consider to be the actions that have led up to that point. The hope is that for each action we assign a number that quantifies the quality of that action (hence q-value). Over time as the agent explores the set of actions, it can take and how the enemy plays we can train an agent to outperform the Enemy [11].

To formalise the ideas introduced we let A be the set of all possible actions our agent may take. In the colonel Blotto context, this is simply the different ways in which soldiers could be arranged across three battlefields. We then let S be the set of states for the model. In our context we let the state be the previous actions of both the agent we are training and the Enemy's state. The idea is that as we move to a new state we consider the associated reward (specifically the payoff) and then adjust the Q-values for all the actions accordingly. As is standard in reinforcement learning problems we let π be defined as the policy which is the agent's strategy for playing the game. We note here that we mean that this is the way in which it picks its actions as the game is iterated. We denote the associated probability of performing an action $a \in A$ while in a state $s \in S$ as $\pi(s, a)$.

To be able to make the problem mathematically tractable we assume that the problem which we wish to solve using the reinforcement learning approach is Markovian in nature. This means that all the information leading up to the time step before the present time step is grouped into one entity and the current time step depends exclusively on the previous time step. Actions, rewards and states which occurred several times steps back are all wrapped into just the previous time step. The Markov process we consider in the context of reinforcement learning is charactersied by the states and the probability of transitioning between the states. The probability with which we transition from one state to another is given by

$$\mathcal{P}_{s_{t+1},s} = \mathbb{P}(s_{t+1}, r_{t+1} \mid s_t, a_t) \tag{76}$$

Where $\mathbb{P}(\cdot)$ is the probability. From our discussion above where we can wrap the previous states, actions and rewards all into just the time step before and so we can express the probabiliy of transi-

tioning from one state to another as

$$\mathbb{P}(s_{t+1}, r_{t+1} \mid \underbrace{s_t, r_t, a_t, s_{t-1}, r_{t-1}, a_{t-1}, ..., s_0, r_t, a_0}_{\text{all described in just the previous time step}}) \tag{77}$$

We can formalise the reward a player expects to receive given a certain state by

$$R_s = \mathbb{E}[r_{t+1} \mid s_t] \tag{78}$$

ideally we would like the total reward across the progression of all the states played and this can be expressed as

$$T_t = \gamma^0 R_{t+1} + \gamma^1 R_{t+2} + ... \tag{79}$$

In the literature the γ term is introduced which is chosen to lie between $[0, 1]$. The purpose of this is to deal with the issue of the infinite horizon problem. Informally it is a useful way to allow us to treat an infinite sum as being finite. The total reward across the progression of all states is crucial since the agent uses it to inform the next action chosen in a given state. The policy gives us a mapping from a state S to the set of actions that could be performed and gives a probability for an action to be take. This can be expressed as

$$\pi(a \mid s) = \mathbb{P}(a_t \mid s_t) \tag{80}$$

We are now in a position to introduce the action value function which gives the expected reward from starting in a state s and a time t and taking an action a guided by a policy π. Using the expression in equation (79) as the total reward we can express the action value function as

$$Q_\pi(a, s) = \mathbb{E}_\pi[T_t \mid s_t, a_t] \tag{81}$$

This gives us our Q values and mathematically the larger the value of Q the better the action chosen with respect to maximising the expected reward. Our goal would be that the agent implements the optimal possible policy where the action chosen maximises the action value function for all possible states.

$$\pi^*(s) = \underbrace{\text{argmax}}_{a} Q(s, a) \tag{82}$$

The idea of the reinforcement learning process is to iteratively update the value of the action function since the reward an action yeilds at an earlier state might not be the same as the reward it yield in a future state. Using this difference allows us to obtain an expression for Q where we use the temporal difference in the nature of a current Q and previous Q to obtain the next Q. This process is called temporal difference learning in the literature[11][12]. Hence the value action function can be obtained by

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big(R_{t+1}(s, a) + \gamma \text{ argmax } Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big) \tag{83}$$

The idea then is to update this action value function and as the game progresses in the number of iterations our agent begins to pick better and better actions. The parameter α here is the learning rate and intuitively it can be thought of as the rate at which the Q-values are updated. Having a learning rate as very small means that in practice the Q-values are only altered by a very small amount and so not much learning really happens over a reasonable timescale. If however, the learning rate is large then learning occurs quickly and the Q-values are updated rapidly however at the expense of the updates potentially being sub-optimal. The learning rate is a number that lies in the interval $[0, 1]$.

Q-Learning Colonel Blotto - A Computer Implementation

To make the above theoretical description more concrete, the Q learning algorithm was used to program an agent to play colonel Blotto (N = 3, S = 5) against an enemy agent which always played a random allocation of soldiers to each of the three battlefields. We discuss with the use of pseudocode the most salient parts of the computer implementation and then consider the performance of the agent using the Q-learning approach.

The first important part of the overall algorithm is the initialisation of the q-table. We recall from our previous discussion that the q-table has entries for both the state of the system and the actions the agent can take. The actions the agent may take are simply how it can allocate soldiers (in our set up there are 21 different ways) and the state will be the previous actions of both the agent using Q-learning and the opponent agent. To effectively store this information we use a dictionary where the key is the state in question and the associated values are Q-values with which an action is taken. Hence each key will have a list of 21 values. The idea behind algorithm (7) is that if our Q-table is queried with a key that is not already an entry in the table rather than returning an error we create an entry with the key queried and the associated values are a list of the chance of doing each of the 21 possible actions an agent can take, each assigned an equal probability. The idea is that we initialise each action as being equally good and as we progress through the iterations we update the Q-value for a given action corresponding to a particular state. In practice we let the states be represented by strings as it allows for more interpretability.

Algorithm 7 Q_val_table_construction

S ← number_of_soliders
N ← number_of_battlefields
number_of_actions ← $\binom{N+S-1}{N-1}$
Q_values ← Empty Dictionary
if Q_values queried with key k not already in table **then**
    Q_values[k] ← $[\frac{1}{number\_of\_actions} * \text{number\_of\_actions}]$
end if

The states of the system we play are the pair of actions that contain the previous state of our agent and the previous state of our opponent. In practice however, when we initialise our algorithm we start from a point where there are no previous states. Thus to begin the iterative process we pick two states randomly, one for the agent and one for the enemy such that we synthetically generate two previous actions which can then be used to start the process. The process for doing this is illustrated in the algorithm (8). We remark that while the state can be just the previous actions of the agent and the enemy we can also have the state be the previous two pairs of actions of the state and the enemy. In practice, this can be altered by changing the range over which the loops in algorithm (8) loop over.

The next important part of the algorithm is to generate state values as strings such that they may be queried in our Q-table as keys. In practice, we will have obtained (either randomly generated or have been given to us by the algorithm) the state in question in the form of an action our agent played and an action our enemy played. An example of this could be the state $[[401], [221]]$ where here our agent played the allocation [401] and the enemy played the allocation [221]. The idea is to have assigned a list value to each possible action previously (using another dictionary) and so the state we are considering is given a key which can then be used to query the Q-table. The way we do this process algorithmically is outlined in algorithm (9).Recall from the construction of the Q-table that if that key is not already present then an entry will be created.

Algorithm 8 Generate_initial_state

```
prev_actions_list ← []                                   ▷ empty list
inner_list ← []                                          ▷ empty list
for i in range(2) do
    append to inner list a random action choice from all possible actions
end for
append to prev_actions_list the inner list
inner_list ← []                                          ▷ clear inner list
for j in range(2) do
    append to inner list a random action choice from all possible actions
end for
append to prev_actions_list the inner list
```

Algorithm 9 Row_to_string_state

```
state ← prev_actions_list
input_list ← []
for i in state do
    for j in i do
        append to input_list a string value to the chosen action
    end for
end for
```

We now come to the two crucial parts of the algorithm which implement the Q-value update rules as outlined in equation (83). The process of updating the Q-values works by inputting the actions of our agent and the enemy and the previous state of the system. We use the Q-value table and query it for the q-values of the previous state of the system. From the list of all possible actions and their associated q-values, we select the q-value associated with our agent's action. We then update this q-value using the update rule in equation (83). Finally we replace the old q-value in the q-value table with the new q-value.

Algorithm 10 q_val_update

```
action1 ← the action of our agent in the current time step
action2 ← the action of the enemy in the current time step
prev ← the previous state
state ← prev_actions_list
all_values ← query Q_value dictionary with key of previous state
value ← The q-value of the specific action1 from the set off all_values
new q-value ← value+(alpha)×(utility of action 1 (our agent) played against action 2 (the enemy))
× (reward + γ× (maximum value of the previous state) - value)
action1 Q value from Q_value table ← new q-value
```

Having updated the Q-values accordingly we then choose an action for our agent to take. Rather than always choosing the action with the largest Q-value we also make use of a parameter ϵ. This parameter ϵ allows the agent to explore random strategies rather than just focusing on the ones with the best Q-values all the time. The value of ϵ is multiplied by the discount rate γ at each iteration meaning that the chance of the agent picking a random action to play becomes less and less over time. This is beneficial since it means that at the start of the process when the Q-values are not very representative of the best actions we can explore all possible actions and as the Q-values improve we can focus on the actions which give the best payoff. Thus the idea is we do not neglect any strategies provdided that our discount rate is appropriate for the situation in question. Provided that we do not randomly sample the space we then look at the Q-values of the previous state and if there happen to be multiple actions that have equal Q-values then we randomly choose one of them. The process is given algorithm (12).

Algorithm 11 action_of_agent

```
random_number ← generate random number in interval [0,1]
if epsilon ≥ random_number then
    action_chosen ← randomly chosen action from the list of all possible actions
else
    state_row ← q values of state in question
    max q value ← find largest entry in state_row
    if there are two or more q values which are the largest then
        choose randomly between the actions assigned with the largest q-value
    end if
end if
```

Finally, we outline the algorithm for the training loop. The key processes taking place here are the discounting of the value ϵ meaning that over time we explore less and less of the space and focus on improving Q-values. We then update the Q-values for both the action taken and the actions not taken then update the state of the system. Finally, we track who wins or loses the current iteration as well as if the iteration happens to result in a draw.

Algorithm 12 training algorithm

```
action_one ← Action of Q-learning agent
action_two ← Random action of enemy agent
ϵ ← ϵγ
unchosen_actions ← list of all possible actions except for the chosen action by the Q-learning agent
Q-table ← Update Q-table for action chosen
for  i in range(lenght of unchosen_actions) do
    Q-table ← Update Q-table for action i of unchosen_chosen
end for
Update state
if Q-learning agent wins then
    Q-learning_agent_wins ← Q-learning_agent_wins + 1
else if Enemy agent wins then
    Random_agent_wins ← Random_agent_wins + 1
else
    draws ← draws + 1
```

Q-Learning Colonel Blotto - Results

The agent using a Q-learning algorithm was played against an agent randomly selecting its actions (each action played with equal probability). The performance of both agents were compared against one another and the success of an agent was the number of times it won compared to its opponent. The performance of the Q-learning algorithm was assessed with respect to both numbers of training iterations and hyperparameters (specifically the learning rate). Figure (5) shows the performance of the Q-learning agent against the random agent for only 200 training iterations. We see that the performance is very even with the random agent winning slightly more games out of the total 200 than the Q-learning agent. We then consider 25000 training iterations and see that the performance of the two agents is still very comparable with no noticeable difference. Starting at 5000 training iterations of the Q-learning agent, we start to see a noticeable improvement in the performance of our trained agent against the enemy and see that it comfortably wins a majority of games, see figure (6). Finally, we consider the case of 200,000 training iterations in figure (7) and see that the Q-learning agent

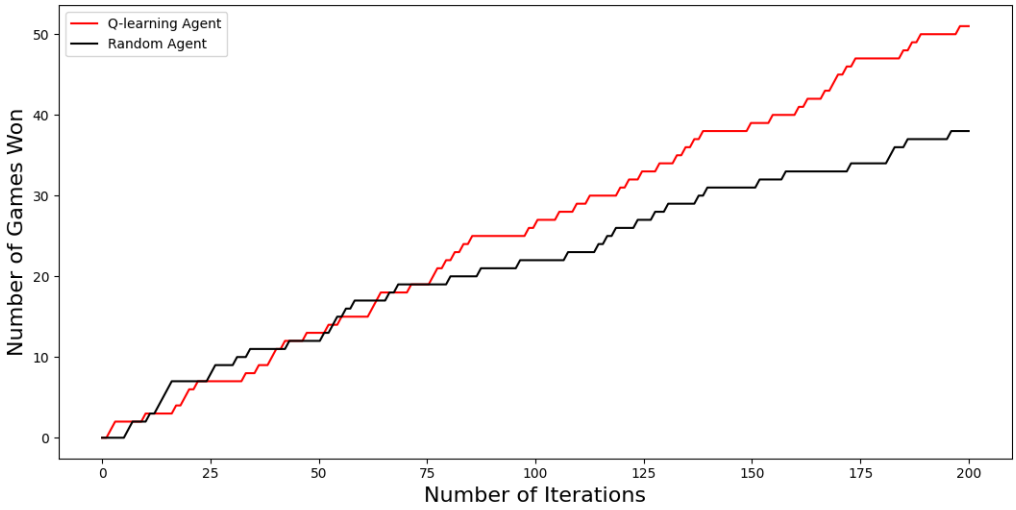

Figure 5: Q-Learning wins vs Random agent wins over 200 games

comprehensively outperforms the random agent. We conclude that given enough training iterations the Q-learning agent is superior to a randomly selected strategy (each action selected with equal probability). The need for so many training iterations can be explained by the need for the agent to explore to state space such that it can effectively update the Q-values. At lower iteration numbers it struggles to fully explore this space and so its performance is at best on par with the random agent.

In addition to model performance we also consider the impact of different values of the learning rate α on the performance of our agent. Setting very high learning rates caused the onset of the point at which the agent started to comprehensively outperform the random agent to be later than with more conservative values for the learning rate. While this did not seem to have a massive impact on the overall outcome of the agent it is worth noting that in other applications or instances of the Blotto game, having a high learning rate could lead to suboptimial performance of the agent since the Q-values with the highest values are often not always the best strategy to choose.

For our parting remarks, we consider the approach of Q-learning when applied to the case of non-integer battlefield allocation as well as comparing it to the performance of algorithms such as regret matching. Finally, we consider improvements and future work which could be made to the implementation of Q-learning for the Blotto game as shown in this section and its possible implications.

Q-learning and Non-Integer Battlefield Allocation In the integer allocation case covered the number of possible actions was finite and so applying a Q-learning scheme algorithmically was a natural extension of this. Although the state space could grow to be potentially very large it remained finite and so in theory, with enough computing power, such an approach could work in a variety of cases. In the case of non-integer allocations we see that the set of possible actions is no longer finite and so we must rethink our approach to how such a Q-learning algorithm could be implemented. While it is a complicated problem there is much literature in dealing with such continuous problems and one avenue of attack could be to discretise the troop allocation in a way such that we remain

Figure 6: Q-Learning wins vs Random agent wins over 6750 games

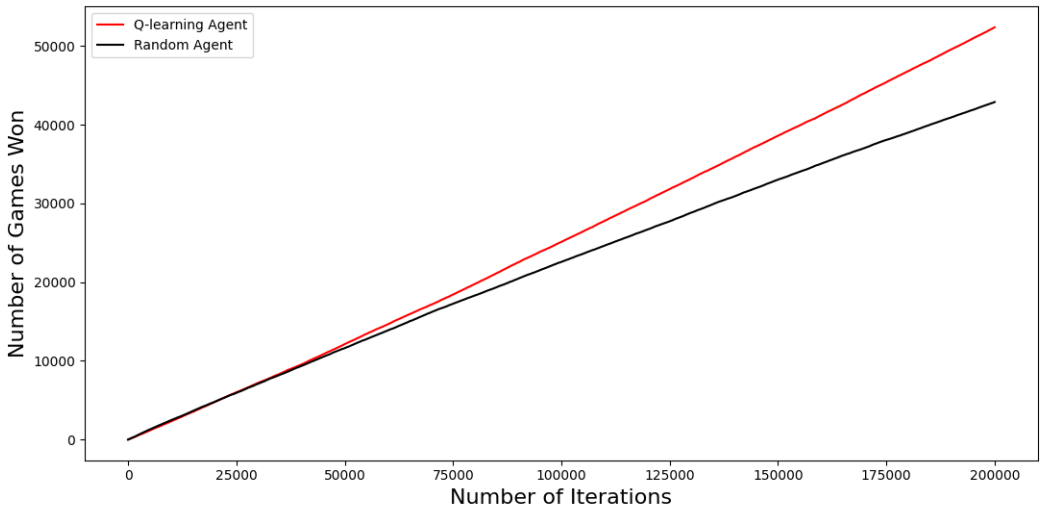

Figure 7: Q-Learning wins vs Random agent wins over 200,000 games

faithful to the probability mass distribution between different regions thus still being able to use a computational approach.

Q-learning and Regret Matching

The approach of obtaining a strategy using a regret-matching implementation is that we look to find a strategy that is a Nash Equilibrium and thus is a best response to what the other agent plays. This difference from reinforcement learning algorithms such as Q-learning which look to maximise the payoff against the opponent over time. This means they look to exploit the way the opponent pays such as to win as often as possible. This means that although it can lead to a superior performance than a random agent in a game like Blotto, a reinforcement agent could still be susceptible to being outperformed by another approach provided that it could be established that a specific type of reinforcement learning such as Q-learning was being used. In contrast, if we find a Nash Equilibrium of the system, while we may not outperform the opponent, the strategy cannot also be exploited in the same way a Q-learning algorithm could be.

Improvements to Q-learning and Blotto Finally, we consider potential improvements to the Q-learning scheme implemented to play Colonel Blotto. One immediate further modification of the implementation could be to make use of the rich literature body on deep neural networks and use a neural network to predict the Q-values rather than using an update rule. "Deep Q-learning" as it is known could allow for further improvement in both how quickly an agent learns and how good those Q-values are. A potential downside however is that we can start to lose interpretability as in the case of simple Q-learning since a deep neural network with hundreds of layers becomes very difficult to analyse[13].

0.11 CODE

Blotto Regret Matching Code for section 0.7

```
import numpy as np
import random
np.set_printoptions(suppress=True)
from itertools import combinations
import itertools
from matplotlib import pyplot as plt
from tqdm import tqdm

battlefield_num = 3
num_actions = 21
num_players = 2
S = 5

battlefields = np.zeros(battlefield_num)
regret_sum = np.zeros((num_players, num_actions))
strategy = np.zeros((num_players, num_actions))
strategy_sum = np.zeros((num_players, num_actions))
opp_strat = np.array([0.9,0.1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0])


def getStrategy(player):
    """
```

```
    obtain the strategy for a given player
    :param player: integer, player 1 [0] or player 2 [1]
    :return strategy: array of the given players strategy
    """
    normalising_sum = 0
    for i in range(num_actions):
        if regret_sum[player,i] > 0:
            strategy[player,i] = regret_sum[player,i]
        else:
            strategy[player,i] = 0
        normalising_sum = normalising_sum + strategy[player,i]
    for j in range(num_actions):                    #one could use numpy slices to get r
        if normalising_sum > 0:
            strategy[player,j] /= normalising_sum
        else:
            strategy[player,j] = 1/num_actions
        strategy_sum[player,j] += strategy[player,j]
    return strategy[player,:]


def getAction(strategy):
    rand_val = random.uniform(0,1)
    a = 0
    cumulative_prob = 0
    while(a < num_actions-1):
        cumulative_prob = cumulative_prob + strategy[a]
        if (rand_val < cumulative_prob):
            return a
        a = a + 1
    return a


def generate_ids(S,N):
    """
    generate list of all possible actions a commander can take given S soliders and N ba
    :param S: number of soliders
    :param N: number of battlefields
    :return IDarray: Numpy array of all possible actions with the index being equal to t
    """
    space = list(range(S+1)) * N
    ID_array = np.array(list(set(i for i in itertools.permutations(space,N) if sum(i) ==
    return ID_array




def utility(id_p1,id_p2,idlist):
    """
    :return utility: the utility of the first inputted player
    """
    battlefields_won = 0
```

```
    battlefields_lost = 0
    battlefields_drawn = 0
    for i in range(battlefield_num):
        if idlist[id_p1][i] > idlist[id_p2][i]:
            battlefields_won = battlefields_won + 1
        elif idlist[id_p1][i] == idlist[id_p2][i]:
            battlefields_drawn = battlefields_drawn +1
        elif idlist[id_p1][i] < idlist[id_p2][i]:
            battlefields_lost = battlefields_lost + 1
    if battlefields_won > battlefields_lost:
        return 1
    elif battlefields_won == battlefields_lost or battlefields_drawn == battlefield_num:
        return 0
    else:
        return -1

def getUtility(otherAction,id_list):
    """
    give table of utilities such that given otheraction we get all the payoffs of everyt
    """
    utilityarray = np.zeros(num_actions)
    for i in range(len(id_list)):
        utilityarray[i] = utility(i,otherAction,id_list)
    return utilityarray




def getaveragestrat(player):
    avgStrat = np.zeros(num_actions)
    normalising_sum = 0
    for i in range(num_actions):
        normalising_sum += strategy_sum[player][i]
    for j in range(num_actions):
        if normalising_sum > 0:
            avgStrat[j] = strategy_sum[player][j]/normalising_sum
        else:
            avgStrat[j] = 1.0/num_actions
    return avgStrat



def training(iterations):
    global regret_sum
    strategy_sum_blotto_list = []
    strategy_sum_boba_list = []
    norm_list_blotto = []
    norm_list_boba = []
    for i in tqdm(range(iterations)):
        blotto = 0
        boba = 1
```

```
        blotto_action = getAction(getStrategy(0))
        boba_fet_action = getAction(getStrategy(1))
        id_list = generate_ids(S, battlefield_num)
        util_val_blotto = utility(blotto_action, boba_fet_action, id_list)
        util_val_boba = utility(boba_fet_action, blotto_action, id_list)
        util_array_blotto = getUtility(boba_fet_action, generate_ids(S, battlefield_num))
        util_array_boba = getUtility(blotto_action, generate_ids(S, battlefield_num))
        regret_blotto = util_array_blotto - util_array_blotto[blotto_action]
        regret_boba = util_array_boba - util_array_boba[boba_fet_action]
        #alternate regret matching
        if i%2 == 0:
            regret_sum[blotto] = regret_sum[blotto] + (regret_blotto)
        elif i%2 == 1:
            regret_sum[boba] = regret_sum[boba] + (regret_boba)
        a = getaveragestrat(0)
        b = np.linalg.norm(a)
        norm_list_blotto.append(b)

        c = getaveragestrat(1)
        d = np.linalg.norm(c)
        norm_list_boba.append(d)
    return norm_list_blotto, norm_list_boba
iterations = 100000
norm_list, norm_list_boba = training(iterations)
norm_list_array = np.array(norm_list)
print("p1", getaveragestrat(0))
print("p2", getaveragestrat(1))
x_vals = np.linspace(0, iterations, iterations)
print(x_vals.shape, norm_list_array.shape)
plt.plot(x_vals, norm_list_boba, color="red")
plt.tick_params(axis='both', labelsize=22)
plt.xlabel("Number of Iterations", fontsize=18)
plt.ylabel("2-Norm of Average Strategy ", fontsize=18)
# plt.plot(x_vals, norm_list_boba)
plt.show()
```

Blotto Q-Learning Reinforcement Learning Algorithm

```
    from typing import DefaultDict
import numpy as np
import random
import matplotlib.pyplot as plt
from math import comb
np.set_printoptions(suppress=True)
from itertools import combinations
import itertools
from tqdm import tqdm


#Base code structure used update style of RPS implementation from eskehaack
#set parameters of Blotto game
battlefield_num = 3
```

```
S = 5 #no. of soliders
num_of_actions = comb(battlefield_num+S-1,battlefield_num-1)

#number of iterations
iterations = 10000

#set parameters of RL model
epsilon = 1.0 # random act chance
gamma = 0.9 #gamma value in q update rule
alpha = 0.1 #alpha in q update rule [learning rate]
lookback_amount = 2 #the state
reward = 0.1 #reward

#Init
AI_wins = 0 #player 2
Random_player = 0 #player 1
total_games = 0
draws = 0
game_outcome = 0

#init qtable
q_values = DefaultDict(lambda:[1/num_of_actions]*num_of_actions)    #works as desired

# Arrays to collect statistics for plots
y_vals_AI = np.array([])
y_vals_Random = np.array([])

#translate each action into an index for q vals dict
def generate_ids(S,N):

    """
    generate list of all possible actions a commander can take given S soliders and N ba
    :param S: number of soliders
    :param N: number of battlefields
    :return IDarray: Numpy array of all possible actions with the index being equal to t
    """
    space = list(range(S+1)) * N
    ID_array = list(set(i for i in itertools.permutations(space,N) if sum(i) == S))
    return ID_array

list_of_IDs = generate_ids(S, battlefield_num)
dict_of_actions = {i: list_of_IDs[i] for i in range(0,len(list_of_IDs))} #dict of action

#center val
center_val = 60

def get_key(action_alloc,input_dict):
    """
    take a battlefield soldier allocation profile
    :param action_alloc: the allocation across the battlefields as a tuple
    :param input_dict: the dictionary containing the corresponding key in our case input
```

```
    """
    key_list = list(input_dict.keys())
    val_list = list(input_dict.values())
    position = val_list.index(action_alloc)
    return position

#determine win or loss or draw
def utility(id_p1,id_p2,idlist):
    """
    :id_p1: action of p1
    :id_p2: action of p2
    :idlist: list of possible actions
    :return utility: the utility of the first inputted player
    """
    battlefields_won = 0
    battlefields_lost = 0
    battlefields_drawn = 0
    for i in range(battlefield_num):
        if idlist[id_p1][i] > idlist[id_p2][i]:
            battlefields_won = battlefields_won + 1
        elif idlist[id_p1][i] == idlist[id_p2][i]:
            battlefields_drawn = battlefields_drawn +1
        elif idlist[id_p1][i] < idlist[id_p2][i]:
            battlefields_lost = battlefields_lost + 1
    if battlefields_won > battlefields_lost:
        return 1
    elif battlefields_won == battlefields_lost
    or battlefields_drawn == battlefield_num:
        return 0
    else:
        return -1


#function to translate input numbers to string for q vals dict
def get_row(old_inputs):
    input_list = []
    for action_sets in old_inputs:
        for action in action_sets:
            input_list.append(get_key(action,dict_of_actions))
    input_list = str(input_list)
    return input_list




#function for updating the table
def update_table(action1, action2, old_actions):
    global q_values
    value = q_values[get_row(old_actions)][action2]
    new_value = value + alpha * utility(action2,action1,list_of_IDs) *
    ((reward) + gamma*np.max(state_row)-value)
```

```
    q_values[get_row(old_actions)][action2] = new_value



#old actions
#generate two random old actions to intialise the process
old_actions = []
inner_list = []
for j in range(2):

        inner_list.append(random.choice(list_of_IDs))
old_actions.append(inner_list)
inner_list = []
for i in range(2):
        inner_list.append(random.choice(list_of_IDs))
old_actions.append(inner_list)



#agent input
def agent_input():
    number = random.random()
    if epsilon >= number:
        action_alloc = random.choice(list_of_IDs)   #list of IDs really are a list of al
        action = get_key(action_alloc,dict_of_actions)
    else:
        state_row_saved = q_values[get_row(old_actions)]
        equal_max_val = [i for i, j in enumerate(state_row_saved) if j ==
        max(state_row_saved)]
        if len(equal_max_val) > 1:
            action = random.choice(equal_max_val)
        else:
            action = equal_max_val[0]
    return action

def enemy_input():
    action = random.randrange(21)
    return action


for i in tqdm(range(iterations)):

    action1 = enemy_input()
    action2 = agent_input()
    epsilon = epsilon*gamma
    game_outcome = utility(action2,action1,list_of_IDs)
    state_row = np.copy(q_values[get_row(old_actions)])
    rest_actions = list(range(0,21))
    rest_actions.pop(action2)
```

```
        update_table(action1,action2,old_actions)
        for j in range(len(rest_actions)):
            update_table(action1,rest_actions[j],old_actions)

        old_actions.append([dict_of_actions[action2],dict_of_actions[action1]])
        del old_actions[0]

        #plotting



        #collecting stats
        if game_outcome == 1:
            AI_wins += 1
        elif game_outcome == -1:
            Random_player += 1
        elif game_outcome == 0:
            draws += 1
        total_games = total_games + 1
        y_vals_AI = np.append(y_vals_AI, AI_wins)
        y_vals_Random = np.append(y_vals_Random, Random_player)



print("AI wins", AI_wins)
print("Random wins", Random_player)
print("draws", draws)
x_vals = np.linspace(0,iterations,iterations)
print(x_vals)
print(y_vals_AI)
print(y_vals_Random)
print(y_vals_AI.shape,y_vals_Random.shape)
plt.plot(x_vals,y_vals_AI, color="red", label="Q-learning Agent")
plt.plot(x_vals,y_vals_Random, color="black", label="Random Agent")
plt.ylabel('Number of Games Won', fontsize=16)
plt.xlabel('Number of Iterations', fontsize=16)
plt.legend(loc="upper left")
plt.show()
```

Bibliography

[1] Soheil Behnezhad et al. *Optimal Strategies of Blotto Games: Beyond Convexity*. 2019. DOI: 10.48550/ARXIV.1901.04153.

[2] Alan Tucker. *Applied Combinatorics*. wiley, 2012.

[3] Brown Jiang. *A tutorial on the Proof of the Existence of Nash Equilibria*. 2018.

[4] Mohammad T. Irfan. *Explanation of Nash's Theorem and Proof with Examples*. 2019.

[5] Mastronardi Macdonell. 2018.

[6] Nehler Lanctot. 2013.

[7] Nisan Noam et al. *Algorithmic Game Theory*. Cambridge university press, 2007.

[8] Mas-Colell Hart. 2000.

[9] Abernethy et al. 2013.

[10] P. Rigollet. *MIT: Mathemathics of Machine Learning*. Lecture Handouts, 2015.

[11] Joseph Christian G. Noel. *Reinforcement Learning Agents in Colonel Blotto*. 2022. DOI: 10.48550/ARXIV.2204.02785.

[12] Yufeng Zhang et al. *Can Temporal-Difference and Q-Learning Learn Representation? A Mean-Field Theory*. 2020. DOI: 10.48550/ARXIV.2006.04761.

[13] Yuxi Li. *Deep Reinforcement Learning: An Overview*. 2017. DOI: 10.48550/ARXIV.1701.07274.