School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

Understanding and Implementing Counterfactual Regret Minimisation.

Mohammed Salbih Ashraf

Submitted in accordance with the requirements for the degree of
BSc Computer Science

2023/2024

COMP3931 Individual Project

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Final Report	*PDF file*	*Uploaded to Minerva (01/05/24)*
Link to online GitHub code repository	*URL*	*Sent to supervisor and assessor (30/04/24)*

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student) ___ Mohammed Salbih Ashraf___

Summary

This report concerns the application and implementation of the Counterfactual Regret Minimisation algorithm. This algorithm is typically used to solve 2 player zero-sum games such as Rock Paper Scissors and Kuhn Poker. The algorithm calculates the ‘regret’ of an action based on its performance in a turn of a game. It then accumulates this regret from every previous turn to inform a new strategy for the next turn. After a certain number of games, this accumulated regret determines our optimal strategy.

My main achievements regarding this report are understanding the process of the algorithm and how it incorporates game theory concepts into it. We will also explore its use in zero-sum games such as Rock Paper Scissors and Kuhn Poker. We will be implementing our version of the regret minimisation algorithm on Rock Paper Scissors and examining an existing solution for Kuhn Poker. To end we will run tests using both solutions to determine the accuracy of these algorithms and understand any other metrics.

Acknowledgements

[1] Mr. Mark Walkley – Supervisor for the project

Table of Contents

Introduction and Background Research

1.1 Introduction to Solving Simple Games

A solved game means that there is a strategy in place that a player could follow at each decision point that cannot be improved upon [10]. Any deviation from the strategy will be worse for the player value-wise. Strategies in general are commonplace when playing any game but, strategies that guarantee a ‘winning situation’ are fewer and more complex to determine. Most players decide their strategy, learning from past games and their opponent's past moves. This is where computing can do most of the bulk for us, being able to simulate games much faster than any human could. However, solving a game computationally means considering every single decision a player can make and what value they gain from that decision to make a good judgement on a strategy. This is where the application of game theory is required.

1.1.1 Game Theory

The Stanford Encyclopedia of Philosophy [5] describes Game theory as “The study of how interacting choices of economic agents produce outcomes for the preferences (utilities) of those agents, where the outcomes in question might have been intended by none of the agents”. In more simple terms, we can mathematically represent what are the value propositions available when a player reaches a decision point. What is the best action a player can take?

One great example of this concept is the ‘prisoner’s dilemma’, a thought experiment regarding game theory put forward by Merrill Flood and Melvin Dresher [20].

Traditional game theory can be easily applied to zero-sum games.

1.1.2 Zero-sum Games

These are games where ‘the gain of one (or some) player(s) must be balanced by the loss of the other(s), and the game is referred to as “zero-sum”’ [14]. This concept can be represented mathematically if we think of a player value gained as an integer number thus meaning the opponent value gained would be negative of that integer.

Red \ Blue	A	B	C
1	30 / −30	−10 / 10	20 / −20
2	−10 / 10	20 / −20	−20 / 20

Figure 1.1.2.1 shows a generic payoff table for the red player for some zero-sum game.

The table represents each value proposition for each combination of player and opponent decisions. These values can be any number for each entry if they are opposing. Of course, the higher the value, the more to lose/gain. This table can be modelled as a matrix, thus enabling the possibility of solving it entirely using techniques such as linear programming [17].

1.1.3 Nash Equilibrium

The ‘Nash Equilibrium’ was introduced as a concept by John Forbes Nash Jr [16]. The idea is that an optimal strategy has been found in a game. If either player were to deviate from their ‘optimal’ strategy, then they would only lose more, and the other player would gain more. This is the ideal scenario we want to end up in when solving games.

David M. Kreps provides a more mathematical definition in his book about game theory [15].

1.1.4 Existing Solutions

As discussed before, we can use linear programming in situations where we can model the game as a matrix [17]. We formulate each line in the matrix as a constraint for player 1 and we define a maximisation function as the sum of player 1's actions multiplied by the value of each action. However, as a game becomes more complex, linear programming may not be sufficient. This could be due to more complicated games requiring a lot more constraints or actions or players thus creating large systems of equations that become exponentially more computationally expensive to solve.

There is also the idea of solving games algebraically using a similar matrix formulation of a game. We again use the payoffs for each player multiplied by the probability of creating that scenario (probability of player 1 action * probability of player 2 action). We end up with 2 equations that we need to pick the probabilities for players 1 and 2 that maximise both. Like linear programming, this approach can become computationally expensive with more complex games that introduce more variables and players. In some cases, it could be impossible to solve it algebraically. Both methods also assume a linear relationship between a player’s strategy and payoffs. Many real-world games are non-linear and don’t have a one-to-one relationship between strategy and payoff.

A better method for solving games is to instead approximate a solution and make a judgment based on the results. We can do this with Regret Minimisation.

1.1.5 Regret Minimisation Algorithm

The Regret Minimisation algorithm takes an iterative approach to solving games. Instead of calculating the exact solution considering every single variable, we play many games and learn from each game using a regret system [1]. For example, in RPS, if I pick rock and lose to paper, then I regret not picking paper (for the draw), but I regret not picking scissors (for the win) even more. The algorithm plays many games and uses the total regret accumulated from every turn to inform the strategy for the next turn. After a certain number of games we can take the total regret to approximate a solution. In this project, I want to explore the use of this algorithm and how it can be used to solve single-stage single-action games such as Rock Paper Scissors, and multi-stage games such as Kuhn Poker.

1.2 Rock Paper Scissors (RPS)

Rock Paper Scissors is a single-action game where 2 people pick between actions rock paper and scissors usually pre-empting by a countdown. Rock beats Scissors, Scissors beats Paper and Paper beats Rock. Choosing the same action results in a draw.

Single-action games are typically binary. There is either a winner/loser or a draw. In RPS there are either differing actions meaning someone has won, or the same action which creates a draw. This section is dedicated to fitting the regret minimisation model onto Rock Paper Scissors.

1.2.1 Utility/Value

First, we must define what utility value we get from each game of RPS. Utility is a way of numerically measuring how good or bad an outcome of a game is. In RPS it's as simple as Win, Lose, and Draw which we can represent as 1, -1, and 0 respectively. If player 1 picks rock and player 2 picks scissors, that is a +1 utility for player 1 and a -1 utility for player 2. A table can be derived showing the utility of every possible combination of rock paper scissors. Each entry shows the utility for player 1 and player 2 respectively.

	R	***P***	***S***
R	0, 0	-1, 1	1, -1
P	1, -1	0, 0	-1, 1
S	-1, 1	1, -1	0, 0

1.2.2 Calculating Regret

Using these utilities, we can determine what result a certain action got. From that action, we can calculate regret for each action in the game. Regret is also a numerical value measuring how much better or worse playing an action is compared to the one just played. If I played Rock and lost to Paper, then we would regret not playing Paper to get a draw, but we would regret even more not playing Scissors for the win. Alternatively, if I had played Rock and beat Scissors, then I would be glad that I didn't play Scissors, but even more so not playing Paper.

To calculate regret, we take the utility of an action in a turn and minus the utility of playing the other action in that turn. In the example of playing rock and losing to paper.

Rock Regret = (-1) – (-1) = 0

Paper Regret = (0) – (-1) = 1

Scissors Regret = (1) – (-1) = 2

For every turn, we should calculate regret for each action and add on the regrets of those actions in the previous turn. For example, if for the next turn, we got regrets 2, 0, and 1 for RPS respectively, then our total regrets for RPS would be 2, 1, and 3 respectively.

1.2.3 Regret Matching

Using definitive choices in a game like RPS isn't a good idea. If a player knew we always picked against what we lost to, they could exploit that to win more games. Instead of deciding that scissors are the only choice we can make next turn, we should increase the likelihood that we pick scissors and paper proportional to the regrets we calculated in the previous turn. To calculate these probabilities, we take the cumulative regrets of an action and divide by the sum of regrets of every action. Following from the previous regret sum of 2, 1, and 3 of RPS respectively.

Rock Probability = 2/6

Paper Probability = 1/6

Scissors Probability = 3/6

These probabilities represent our strategy for the next turn. We will randomly pick an action following the distribution of those probabilities.

1.2.4 Regret Minimisation

Eventually, by playing many iterations of RPS we would end up with the last strategy used. However, this ending strategy will be biased towards how the last round played out. This means we also need an average strategy based on the strategy used in every single turn.

This can just be calculated by adding the strategy of a turn to a total ‘strategy sum’. Then we can normalise that to receive the average strategy that the computer has suggested we play.

1.2.5 Nash Equilibrium

Obviously how the opponent plays will determine what the average strategy will look like, and the number of iterations would likely give a more accurate result. However, if we also changed the strategy of the opponent based on the games as well as the player, we should reach a point where both players have the same strategy which would be playing each action randomly with a 1/3 probability for each.

1.3 5 Action Game

If we wanted to increase the number of actions, in this case, 5, it just requires that we calculate 5 different action utilities and regrets. For this project we’ve decided to use Rock, Paper, Scissors, Lizard, Spock as my 5-action game which plays like Rock, Paper, Scissors but with 5 actions instead. The utility table of this game is shown below.

	R	***P***	***S***	***L***	***SP***
R	0, 0	-1, 1	1, -1	1, -1	-1, 1
P	1, -1	0, 0	-1, 1	-1, 1	1, -1
S	-1, 1	1, -1	0, 0	1, -1	-1, 1
L	-1, 1	1, -1	-1, 1	0, 0	1, -1
SP	1, -1	-1, 1	1, -1	-1, 1	0, 0

1.4 Kuhn Poker

Kuhn poker is a 2 player-simplified version of regular poker defined by Harold E. Kuhn [8]. The game starts with both players betting a chip into a pot. Then both players are handed 1 of three cards marked as Jack, Queen, King (1, 2, 3). The card given is only known to the player holding it. We alternate the play starting with player 1. On any given turn a player can either bet 1 chip or pass their turn. This is a zero-sum game meaning that the player that wins a round, gains an amount of chips that the other player loses. Many different scenarios can occur which will be shown in the table below.

Player 1	***Player 2***	***Player 1***	***Result***
pass	pass		*+1 to player with the higher card*
pass	bet	pass	*+1 to player 2*
pass	bet	bet	*+2 to player with the higher card*
bet	pass		*+1 to player 1*
bet	bet		*+2 to player with the higher card*

1.4.1 Sequential Games

Kuhn poker is different from RPS as there are multiple stages of play rather than just 1 action determining the game. The player may end up in a certain stage of the game where their strategy would be different from another stage. We could reformulate this as a single-action game if a player were to decide to bet or pass on both turns depending on their card. But that doesn't consider adapting to the other players' turn and using our past games to determine what that means for our strategy.

1.4.2 Tree Formulation

Instead, we will represent this game as a tree, with states and edges. Each state represents either a chance node which is the card distribution stage, or a decision node which is the point at which the player needs to decide between pass or bet. For chance nodes, the edges coming out represent an outcome of that chance event along with the probability of it occurring. For decision nodes, the edges represent the actions and the player's probability of picking that action.

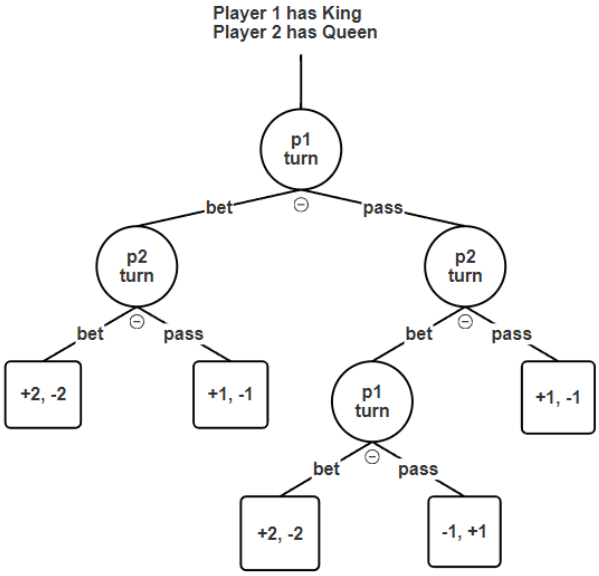


Figure 1.4.2.1 shows 1 possible game state after a King and Queen has been dealt to players 1 and 2 respectively.

This tree shows what the payoffs are for player 1 and player 2 respectively for each combination of actions that could occur. Each circle represents a decision node and each square a payoff. Of course, a different tree would exist for all other combinations of player 1's and player 2's cards. The probability of ending up on a tree would be 1/6 as it would be the chance of getting any of the 3 cards multiplied by the chance of the opponent getting 1 of the 2 other cards left.

1.4.3 Information Sets

An Information set is a game state that a current player can be in. These are typically formatted as the card that the current player has followed by the actions (if any) from the previous plays. If player 1 has an Ace to begin with, the Information Set is [A]. For player 2 holding a King and going second against player 1's bet, then the information set is [Kb]. For player 1 holding a Queen after passing the first turn, and player 2 betting on the second turn, the information set would be [Qpb].

For player 1 there are 6 information sets, 3 at the start for their first given card, and 3 for when they pass and player 2 bets with their assigned card. We can denote these as:

[K] [Q] [J] [Kpb] [Qpb] [Jpb]

For player 2 there are also 6 information sets all based on the card they get and what action player 1 chooses on their turn. We denote these as:

[Kb] [Kp] [Qb] [Qp] [Jb] [Jp]

If we think of the whole game tree, there are 2 different nodes within each information set. For example, for [Kb], we know that player 2 has a King and player 1 just bet but player 1 could have either a Queen or a Jack. So those are 2 different scenarios for that information set which is the same for all information sets. The graph below shows how certain nodes have the same information set (same colour) at the same stage solely based on the knowledge of the player at that stage.

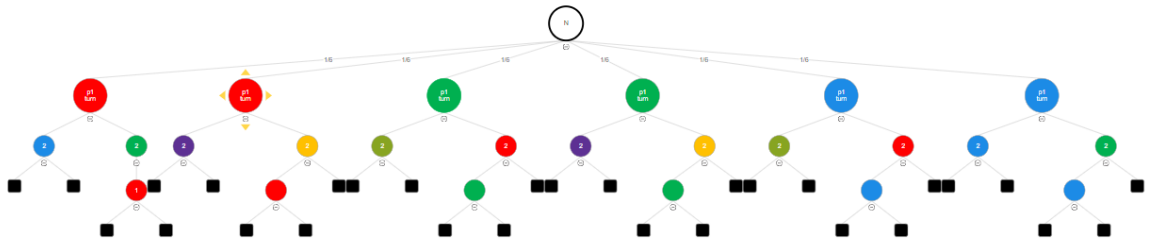


Figure 1.4.3.1 shows the full game tree with every combination of cards dealt to players 1 and 2.

1.4.4 Counterfactual Regret Minimsation (CFR)

Counterfactual Regret Minimisation or CFR uses the same regret-matching process we used for Rock Paper Scissors, however with 2 more factors we have to consider. (1) We must consider the probability of reaching each information set depending on the player's strategy, and (2) an idea of passing forward the game state information along with the probabilities of getting to certain points and passing backward the utility information from the information sets.

We want to calculate the probabilities (beliefs) of getting to each information set from the top of the tree to the bottom. Then from the bottom of the tree, we can calculate the action utilities going up the tree to which we can perform our regret matching.

1.4.5 Beliefs

Beliefs are the player's beliefs of what the other player's card is. Lets assume each players strategy, no matter what information set they are in, is always 2/3 probability of betting and 1/3 probability of passing.

Scenario 1: First action player 1

If we are at information set [K] but we want to know exactly which node we are at as it could be that player 2 has a Q or a J. We can separate the other player's cards as either higher (H) or lower (L). In these information sets at the start of a game, the belief will always be ½ as they could either get H or L at random.

Scenario 2: Player 2 response

When we get to player 2 information sets such as [Qb], we know that there is a 50/50 chance that player 1 has either a King or a Jack. But, since player 1 is randomising their actions based on the card they were dealt (in our case 2/3 for bet and 1/3 for pass)the probability of seeing a player 1 bet might be different depending on what they were dealt. So to find out the probability of whether player 1 has a King or Jack based on player 2's information set of [Qb], we need to calculate the conditional probability that player 1 has a King or Jack based on the observation that they played bet. Here we can use Bayes' Law.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

For the King scenario we have:

$$Pr(K|Qb) = \frac{P(Qb|K) \times P(K)}{P(Qb)} = \frac{\frac{1}{2} \times \frac{2}{3}}{\frac{1}{2} \times \frac{2}{3} + \frac{1}{2} \times \frac{1}{3}} = \frac{2}{3}$$

Which makes Jack 1 – 2/3 = 1/3.

So our belief at [Qb] for player 1 having a higher card (King) is 2/3 and a lower card (Jack) is 1/3.

Scenario 3: Player 1 Response after Pass Bet.

Now we perform the same action but for all nodes at information sets [Kpb] [Qpb] [Jpb]. This is when player 1 has passed and player 2 has bet. For example for information set [Kpb], we need to find the probabilities that player 2 was dealt the Queen and the Jack given that player 1 was dealt a King and the actions followed were pass from player 1 and bet from player 2. Using Bayes' Law we can.

$$Pr(Q|Kpb) = \frac{P(Kpb|Q) \times P(Q)}{P(Kpb)} = \frac{\frac{1}{2} \times \frac{1}{3} \times \frac{2}{3}}{\frac{1}{2} \times \frac{1}{3} \times \frac{2}{3} + \frac{1}{2} \times \frac{1}{3} \times \frac{2}{3}} = \frac{1}{2}$$

This makes the probability of player 2 having a Jack also 1/2. When the beliefs are calculated for each node in every information set, we get a table that could look like the one below. Calculations for these specific tables can be found in Professor Bryce's video [4]

Player 1:	strategy		Beliefs	
Information Set	bet	pass	H	L
K	2/3	1/3	1/2	1/2
Q	1/2	1/2	1/2	1/2
J	1/3	2/3	1/2	1/3
Kpb	1	0	2/3	1/3
Qpb	1/2	1/2	3/4	1/4
Jpb	0	1	3/5	2/5

Table 1.4.5.1 Strategies and beliefs for each information set for player 1.

Player 2:	strategy		Beliefs	
Information Set	bet	pass	H	L
Kb	1	0	3/5	2/5
Kp	1	0	3/7	4/7
Qb	1/2	1/2	2/3	1/3
Qp	2/3	1/3	1/3	2/3
Jb	0	1	4/7	3/7
Jp	1/3	2/3	2/5	3/5

Table 1.4.5.2 Strategies and beliefs for each information set for player 2.

1.4.6 Action Utilities

Now that we know the player's belief at every information set, we can travel back up the tree to calculate the expected action utilities of each decision at each information set.

Starting at the bottom, information sets [Kpb] [Qpb] [Jpb] have only 2 decisions, either bet or pass. Using the payoffs from the game tree [5], for the information set [Jpb], betting always gives a -2 payoff and pass gives -1. These are our action utilities for betting and passing for information set [Jpb]. However, For information set [Qpb] betting can give +2 if player 2 has Jack, and -2 if player 2 has King. If we pass our action utility is always -1. We need to use the beliefs of the information set [Qpb].

Using the example beliefs from the table above, we know there a 3/4 chance of having King and a 1/4 chance of having Jack. So our action utility for betting at [Qpb] is (-2 x 3/4) + (2 x 1/4) = -1. We perform these action utility calculations at every information set going up the tree. Using the table example above we get the following action utilities.

Player 1	strategy		Beliefs		Action utilities	
Information Set	bet	pass	H	L	bet	pass
K	2/3	1/3	1/2	1/2	5/4	3/2
Q	1/2	1/2	1/2	1/2	-1/2	-1/3
J	1/3	2/3	1/2	1/3	-5/4	-1
Kpb	1	0	2/3	1/3	2	-1
Qpb	1/2	1/2	3/4	1/4	-1	-1
Jpb	0	1	3/5	2/5	-2	-1

Table 1.4.6.1 Updated table 1.4.5.1 with action utilities for each action in each information set for player 1.

Player 2	strategy		Beliefs		Action utilities	
Information Set	bet	pass	H	L	bet	pass
Kb	1	0	3/5	2/5	2	-1
Kp	1	0	3/7	4/7	17/14	1
Qb	1/2	1/2	2/3	1/3	-2/3	-1
Qp	2/3	1/3	1/3	2/3	0	1/3
Jb	0	1	4/7	3/7	-2	-1
Jp	1/3	2/3	2/5	3/5	-11/10	-1

Table 1.4.6.2 Updated table 1.4.5.2 with action utilities for each action in each information set for player 2.

1.4.7 Expected Utilities

Since we have a strategy for the player when betting or passing, we can calculate expected utilities for each information set. Expected utilities are the value we would gain from each information set if we pick randomly betting or passing based on the strategy of the player. For player 2 at information set [Qb], the strategy is to bet with a 1/2 chance and pass with a 1/2 chance and we get action utilities of -2/3 and -1 respectively. Our expected utility at information set [Qb] is (1/2 x -2/3) + (1/2 x -1) = -5/6.

1.4.8 Regret Matching

Now with the expected utility at each information set as well as the utility for each action in that information set, we can start regret matching. For example, using table 1.4.6.1, at information set [K] we have an expected utility of 4/3 and action utility of 5/4 for bet and 3/2 for pass.

At information set [Kb]

Regret of betting: 5/4 – 4/3 = -1/12 -> 0

Regret of passing: 3/2 – 4/3 = 1/6

So here we regret not passing as much so we need to increase the probability of passing and decrease the probability of betting. We also need to weigh these regrets by the probability of getting to certain information sets. For [K] the probability of getting to this set is 1/3 (dealt a King). So, we multiply each regret by 1/3 to give 0 for a bet and 1/18 for a pass.

Now we can add these regrets to our cumulative regrets for each action at each information set, like Chapter 1.2.3 for RPS.

1.4.9 Nash Equilibrium

In this game, the Nash equilibrium is not the same strategy for both players as for RPS. Since player 1 and player 2 are dealing with different amounts of information at different points in the game, their strategies will be different. We also know that there is a slight bias in this game as player 2 will always be reacting to a player 1 move.

1.4.10 Recursion

Calculating utilities from any point in the tree requires us to first calculate the utilities of the nodes below until we hit the payoff node. This implies the use of recursion of this algorithm which will be further explained in the implementation in chapter 2. This could also suggest a longer time complexity as we would need 1 to 3 times more calls to the algorithm for every iteration.

1.5 Aims and Objectives

The aim of this project is to explore the implementation and performance of Counterfactual Regret Minimisation in conceptually different games. The games in question being Rock Paper Scissors, a 5-action variant of Rock Paper Scissors, and ending with Kuhn poker (a multi-stage game).

Objectives of this report are to implement the regret minimisation algorithm on Rock Paper Scissors and its 5-action variant. Examine an existing solution for Kuhn Poker. Perform multiple tests on these solutions measuring accuracy and time and note how these change depending on different variables such as the number of iterations or an opponent-set strategy.

Methods

2.1 Software

Using the coding language Python, we can create a program that takes the strategy of the opponent and uses regret minimisation to obtain an ‘optimal’ strategy. We can iterate this process many times to obtain a more accurate average strategy.

2.1.1 Python

Python is a very versatile and accessible coding language with a lot of libraries available to streamline the coding process. While there are alternatives such as Java and C which may even be more efficient for larger games, python is a language I am comfortable writing in and is more readable than others. It should also be noted that most solutions that exist for this algorithm are also written in Python. The general structure of the code will be based on the Java code from Nellar and Lanctot [1].

There will also be some methods based on code written by Pranav Ahluwalia [2]

2.1.2 GitHub

I have used GitHub to version control my code. During the coding process, I frequently committed files when I had implemented a new function or streamlined other functions. It's also used to store my report and for general testing. The repository should only contain 3 Python files: regret minimisation for RPS, regret minimisation for RPSLSP (5 action RPS), and CFR minimisation for Kuhn Poker. The link to the repository is given below.

https://github.com/uol-feps-soc-comp3931-2324-classroom/final-year-project-Salbih100.git

2.2 Solution: Rock Paper Scissors

2.2.1 Algorithm Structure

Referring to the abstract view of the algorithm in Chapter 1.2, we want to have a training function that repeats a given number of times performing the steps to accumulate regret of each action. This will inform the strategy for the next loop We also want to keep track of our sum strategy (the total of every strategy used) to calculate an average strategy. The algorithm can be structured as follows.

1. Obtain a strategy based on our accumulated regrets.
2. Select an action based on the probabilities of our strategy.
3. Select an action based on the probabilities of the opponent's strategy.
4. Calculate action utilities for the player against the opponent's action.
5. Calculate regrets based on the utility of each action.

6. Update our accumulated regrets for each action and update our strategy sum.
7. Goto 1.
8. Exit loop only if a certain number of iterations has occurred.
9. Calculate the average strategy based on the total of every single strategy that was used.

To obtain the Nash equilibrium, we would need to also accumulate regrets for the opponent and select their strategy to which they can pick an action. Then obtain the average strategy for the opponent as well as the player. This will be described in more detail in Chapter 2.2.9

2.2.2 Solution Structure

Like the Java code by Nellar and Lanctot [1] we will use a class to represent our 'trainer'. This allows us to keep our variables, arrays, and methods strictly to the instance of the class. This also allows us to share the contents of the variables and arrays across all the methods within the 'trainer' class. There will be 5 main methods which will be described below. There will also be other methods used for testing and displaying information regarding the algorithm. There will be a code segment at the bottom outside of the class to initialise the class and run various 'test' methods.

2.2.3 Information Storage

We Initialise our class with '`__init__(self)`' to perform any starting calculations and initialise any needed variables. For this algorithm, we need a way of storing information about each of the actions of rock paper scissors. This can be done easily through arrays of length 3. For example, the value at index 0 represents the value for rock in any given use. 3 arrays are initialised, one for the strategy, the sum of all strategies, and the sum of all regrets, all of these for each rock, paper, and scissors. We can also initialise these same arrays but for the opponent for when we need to change the opponent's strategy as well. For future proofing, we need a constant variable to store the number of actions that are present in this game. This will help during our 5-action game.

2.2.4 Obtaining the Strategy

The '*get_strategy(self)*' method will be used in every iteration of the algorithm to obtain a new strategy that the player will use for their next turn. As discussed in Chapter 1, we will need the regret sum of the actions. We will also need a normalising sum variable as a means of storing the sum of the regret sums to distribute the probabilities proportionally.

To start we clear our class strategy array. Then we iterate through it to assign each action of the strategy array to the corresponding regret sum of that action. We also add each value of the regret sum to the normalising sum variable.

Next, we iterate through the newly assigned strategy array but divide each value by the normalising sum to create our finalised strategy array with probabilities of playing each

action equal to 1. If the normalising sum is 0, then we assign a value of (1 / num_of_actions) to each action of the strategy array. In the case of RPS, that is 1/3 for each action.

2.2.5 Choosing Our Action

The '*get_action(self, strategy)*' function randomly chooses an action based on the probabilities of the current strategy array. We assign a variable with a random float between 0 and 1 and then use the cumulative probability of choosing our actions from our strategy array. For example, let's say our strategy array is [0.3, 0.4, 0.3] and we pick a random float of 0.75. Then we can determine that the value is higher than 0.3, higher than 0.3 + 0.4 but lower than 0.3 + 0.4 + 0.3 thus we pick the last value (scissors) in the array as our action.

2.2.6 Accumulating Regrets

The '*train(self, iterations)*' method is where our main algorithm will take place using all the previous functions to calculate the regret of a set number of games. We are passed through a variable called 'iterations' to determine how many times our for loop should repeat (how many games should be played. In this loop, we obtain a strategy for this term and determine our action played based on our strategy. Also, we take the opponent's set strategy and use the same get action method to determine their action in this iteration of an RPS game. Then we use if-else statements to set the action utility array (for the player) for that turn. This idea was explained in chapter 1.2.1. At the end, we can calculate the regret of each action based on that iteration and update our regret sum using the calculations from Chapter 1.2.2. We also want to add the strategy that was just used in that iteration to our sum strategies array. This process will repeat for the set amount of iterations that was provided.

2.2.7 Calculating average strategy

The 'get_avg_strategy(self)' method gives us the average strategy played when considering every single iteration of the game. We follow the same structure as the get strategy method, this time using the strategy sum array. This means initializing a normalising sum variable to distribute the probabilities of our average strategy evenly just as done in the get strategy method.

2.2.8 Nash Equilibrium

The 'nash_equilibrium(self, iterations)' method follows the same structure as the training method, but now we have an opponent regret that we also update based on the turn played. We perform the same calculation but multiply each action by minus 1 (the opposite of the player's regret). This method should be used when we want the opponent's strategy to also change along with the player effectively creating the optimal strategy for playing RPS. When initialising the class, we can choose between a set opponent strategy

using the ‘train’ method or an evolving opponent strategy using the ‘nash_equilibrium’ method.

2.3 Solution: Five action game

Changing the algorithm to work for a five-action game is very simple. We just need to change the constant that is the NUM_ACTIONS to 5 instead of 3. This also means updating all arrays to be of length 5 instead of 3. This should keep our methods working the same way. The only methods we need to change are our train and nash equilibrium methods. We update the action utilities via the table provided in Chapter 1.2.5 instead of the one based on regular RPS. This also requires updating the regrets of all 5 actions. Coding-wise, this mostly consists of creating more if-else statements for the other combination of actions.

2.3.1 Evaluation

For Rock Paper Scissors and its 5-action variant, we already know the optimal strategy when playing against another player. For RPS it is played randomly with a 1/3 probability of picking each action. For the 5-action game, it is played randomly with a 1/5 probability of picking each action. This doesn’t consider a set strategy from the opponent, but we can assume a set strategy that favours Rock would warrant a counter strategy that heavily pushes Paper and avoids Scissors.

We will perform tests that change an opponent's strategy to measure the accuracy of our algorithm. We will also measure the error from our optimal strategy when an opponent is allowed to change their strategy. Additionally, we will measure the time taken to solve by setting a timer before we start the algorithm to when we finish all iterations. I will also change the number of iterations and note how that affects accuracy and time.

2.4 Solution: Kuhn Poker

For Kuhn Poker, I will be examining a solution from Varuna Jayasiri [7]. Explanations and graphical tests can be found on his Google Colab page [8].

2.4.1 Information Storage

Before implementing the algorithm itself, Varuna uses the libraries: ‘`NewType`’, ‘`List`’, and ‘`cast`’ to create objects that hold information such as players, actions, and chances (J, Q, K). We also create a history string that stores the card dealt to players 1 and 2 as well as actions played in the previous turns if any. Also, dictionaries are used to hold the regret and strategy arrays as there are separate strategies that need to be updated for all 12 information sets. Finally, there is the method ‘`info_set(h)`’ that returns an information set for a given history.

2.4.2 Utility Methods

Throughout this implementation, there are many methods solely used for utility purposes. There are 'getter' methods for strategy, regret, and actions. Some methods returns true or false based on whether we are at a terminal node. It does this by checking if the length of our history is smaller than 2.

2.4.3 Terminal Utility

The '`is_terminal(h)`' method returns true if we are at the end node of our game (the payoff). We do this by first checking if our history is longer than 2. Then we check the history string. If the last character is a 'p' for pass OR if the last 2 characters are 'bb' for 2 bets. If any of the conditions are false then we return false.

The '`terminal_utility(h, i)`' method calculates the payoff if we are at a terminal node. We calculate a winner based on the first 2 characters of our history which are our cards for p1 and p2 respectively. If p1 is stronger than p2 then the winner is p1 given a +1 value. Then we return payoffs for all the possible end situations which are 'bp', 'bb', and 'p'. The payoffs can be found at [4]. This utility value return is flipped if we are player 2.

2.4.4 Sample Chance

The '`is_chance(h)`' method checks if we are at a chance node (when cards are dealt). If the history string is smaller than 2, we return true.

The '`sample_chance(h)`' method picks what card is given to each player. We pick a random integer between 0,2 (for the 3 cards) and set that as the card. However we also check whether the card has already been dealt by seeing if it is in our history string If so, we choose another random card until we get a unique one. We return the card chosen as a character ('K', 'Q, 'J').

2.4.5 Accumulating Regrets

We use '`update_regrets(I, v, va, i)`' to accumulate regret for both actions ***a*** in an information set ***I***. We add on (va[a] – v) where va[a] is the utility if we always take action a at an information set and v is the utility from this point of the history playing our current strategy.

2.4.6 Strategy and Average Strategy

We '`calculate_new_strategy(I)`' like RPS where we require a normalising sum which is '`r_plus_sum`' in this method. For every action possible at the current information set, we create a strategy of that action in that set to be the cumulative regret of that action in that set divided by the sum of the cumulative regrets of both actions. If we have a below 0 normalising sum, we set both strategies to be 1 / '`n_actions`' which in this case would be 1/2.

‘`update_cumulative_strategy(I, pi_i)`’ adds to our new strategy calculated for each action in an information set after multiplying it by the sum of probabilities needed to get to that information set.

‘`calculate_average_strategy(I)`’ is similar to RPS where we set the average strategy of an action in an information set by dividing the cumulative strategy of that action by the cumulative strategy of both actions for that information set. Again, if the normalising sum here is less than 0, we assign 1/2 to both actions in the information set.

2.4.7 Main Algorithm

The pseudo-code for the main Counterfactual Regret Algorithm (CFR) can be found on page 12 of Neller and Lanctot’s report [1]. To clarify the `cfr(h, I, pi)` method returns a utility ***u***. First, we check for if we are at the start or end of a game. If we are at the start, we pick cards for the players. If we are at the end we return the utility of that end node. We then set variables for the current info set at history ***h,*** our current player, our utility ***ua*** if we always take action ***a*** for a history ***h,*** and an overall utility ***u*** for this information set. We will also set variables ***va*** for the counterfactual value of reaching ***h*** if we always took action ***a,*** and ***v*** as the counterfactual value at a non-terminal history ***h.***

For both actions ***a*** in the info set, we copy the probability of reaching ***h*** if all players except I took actions that lead to ***h*** with probability of 1 into a variable ***pi_next***. We then multiply that variable by the strategy for player ***I*** of picking action ***a*** (***prob_action***).

We then perform the CFR algorithm again (recursive) on the new history with the added action, and the new probability variable. We would keep calling CFR until we reach a terminal node for this game.

After, we update our utility variable ***u*** with the returned utility value from our CFR method multiplied by ***prob_action.*** We set ***pi_neg_i*** (which would be the probability of reaching ***h*** if player ***i*** took all of its actions that lead to ***h*** with probability of 1) to 1. We update ***pi_neg_i*** by multiplying it by the ***pi*** of the opposite player, for each probability in ***pi.***

Lastly, we set the value of our actions in ***va*** to ***pi_neg_i*** * ***ua[a]*** for all actions. Then we set our counterfactual value ***v*** as ***pi_neg_i*** * ***u.*** We lastly call our update regrets and update cumulative strategy methods with our new variables.

2.4.8 Evaluation

Since we are using an external solution, our options for testing are limited. However, like RPS, we can measure the time of the algorithm by setting a timer before we run the main method. We would expect to see an exponential growth of time compared to the number of iterations due to the recursive nature of CFR.

We can also measure accuracy by comparing the results for each action for each information set compared to existing solved solutions to Kuhn poker. We can then measure the percentage error for each action in each information set to determine where errors are the largest. In the end, we can calculate an average error to compare how different numbers of iterations affect overall accuracy. We expect a lower percentage error for a larger number of iterations used. We can use print statements to print out the probabilities for each action and the time taken.

Results

3.1 Rock Paper Scissors

For Rock Paper Scissors there are 3 tests we will perform on the algorithm. (1) Opponent uses mathematically solved strategy (1/3, 1/3, 1/3) (2) Opponent uses set biased strategy (0.4, 0.3, 0.3) (3) Opponent changes strategy using Nash equilibrium. We will test all with both 10,000 and 1,000,000 iterations. We will also measure the time taken and accuracy compared to the expected results.

3.1.1 Opponent Strategy (1/3, 1/3, 1/3)

10,000 iterations – Time taken (s): 0.0335.

Action	Action Probability	Expected Result	Error
Rock	0.4019014	0.3333333	20.58%
Paper	0.269793	0.3333333	19.05%
Scissors	0.3283056	0.3333333	1.5%

1,000,000 iterations – Time taken (s): 3.1971.

Action	Action Probability	Expected Value	Error
Rock	0.3389359	0.3333333	1.68%
Paper	0.3224602	0.3333333	3.26%
Scissors	0.3386039	0.3333333	1.58%

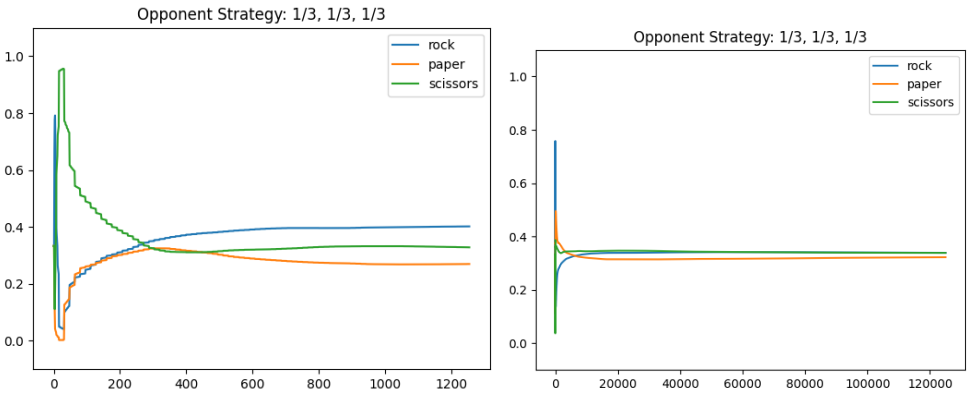


Figure 3.1.1.1 and Figure 3.1.1.2 Show the change in action probabilities for each action: Rock Paper Scissors over the number of iterations (10,000 and 1,000,000 respectively)

3.1.2 Opponent Strategy (0.4, 0.3, 0.3)

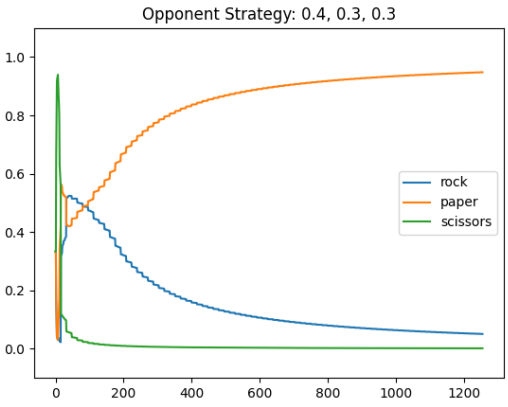
Note: *For Actions with an expected value of 0, we will calculate error by (1 - action probability) against an expected value of 1.*

10,000 iterations – Time Taken (s): 0.0320.

Action	Action Probability	Expected Value	Error
Rock	0.0506881	0	5.07%
Paper	0.9477926	1	5.22%
Scissors	0.0015193	0	0.15%

1,000,000 iterations – Time taken (s): 3.1591.

Action	Action Probability	Expected Value	Error
Rock	0.0009948	0	0.099%
Paper	0.9990046	1	0.101%
Scissors	7e-07	0	6.99e-05%



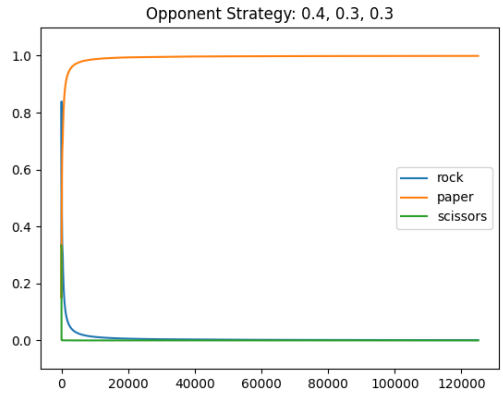


Figure 3.1.2.1 and **Figure 3.1.2.2** show the change of action probability for each rock paper and scissors with opponent strategy (0.4, 0.3, 0.3) for 10,000 iterations and 1,000,000 iterations respectively.

3.1.3 Nash Equilibrium

10,000 Iterations – Time taken (s): 0.0351.

Action	Player Action Probabilities	Opponent Action Probabilities	Expected Value (Both)	Player Error	Opponent Error
Rock	0.3251962	0.3462859	0.3333333	2.44%	3.88%
Paper	0.3090767	0.3475631	0.3333333	7.27%	4.26%
Scissors	0.3657271	0.3061508	0.3333333	9.71%	8.15%

1,000,000 iterations – Time taken (s): 3.6111.

Action	Player Action Probabilities	Opponent Action Probabilities	Expected Value (Both)	Player Error	Opponent Error
Rock	0.3275012	0.3381471	0.3333333	1.74%	1.44%
Paper	0.3373142	0.3313360	0.3333333	1.19%	0.59%
Scissors	0.3351847	0.3305167	0.3333333	0.55%	0.84%

3.1.4 Results Analysis

The results of Rock Paper Scissors seem to be standard for our expectations. We see that the more iterations used in the algorithm, the closer the accuracy is to the optimal strategy. We see an average error of 13.71% for 10,000 iterations compared to the average error of 2.17% for 1,000,000 iterations. For the set opponent strategy, we observe the algorithm by far favouring Paper as that is the action that counters the opponent's favour of Rock. For an opponent-evolving strategy, we see that we are given close to 1/3 probabilities for all actions for both players showing that playing randomly with a 1/3 chance is the optimal strategy for Rock Paper Scissors.

For the set opponent strategy and Nash equilibrium, we observe much more accurate results compared to the opponent optimal strategy. I will assume that the poor results for the 1/3 split strategy could have been caused by the random number generation. For example, there could’ve been long stretches of the algorithm picking the same actions randomly which may skew the regrets of certain actions.

We also notice a linear correlation between the number of iterations and the time taken to compute. From this, we can confirm that the algorithm is O(n) for time complexity.

3.2 Rock Paper Scissors Lizard Spock

For this game we will follow the same test basis of using the mathematically solved opponent strategy, a set opponent strategy, and a changing opponent strategy (Nash Equilibrium). Our mathematically solved opponent strategy will be an even split of 0.2 for all actions. Our set opponent strategy will be (0.4, 0,2, 0,2, 0.1, 0.1) for actions Rock Paper Scissors Lizard Spock respectively.

3.2.1 Opponent Strategy (0.2, 0.2, 0.2, 0.2, 0.2)

10,000 Iterations – Time taken (s): 0.0430.

Action	Action Probability	Expected Result	Error
Rock	0.17948	0.2	10.26%
Paper	0.24457	0.2	22.285%
Scissors	0.12533	0.2	37.335%
Lizard	0.2126	0.2	6.3%
Spock	0.23802	0.2	19.01%

1,000,000 iterations – Time taken (s): 4.2607.

Action	Action Probability	Expected Result	Error
Rock	0.19607	0.2	1.965%
Paper	0.20586	0.2	2.93%
Scissors	0.19349	0.2	3.255%
Lizard	0.20687	0.2	3.435%
Spock	0.1977	0.2	1.15%

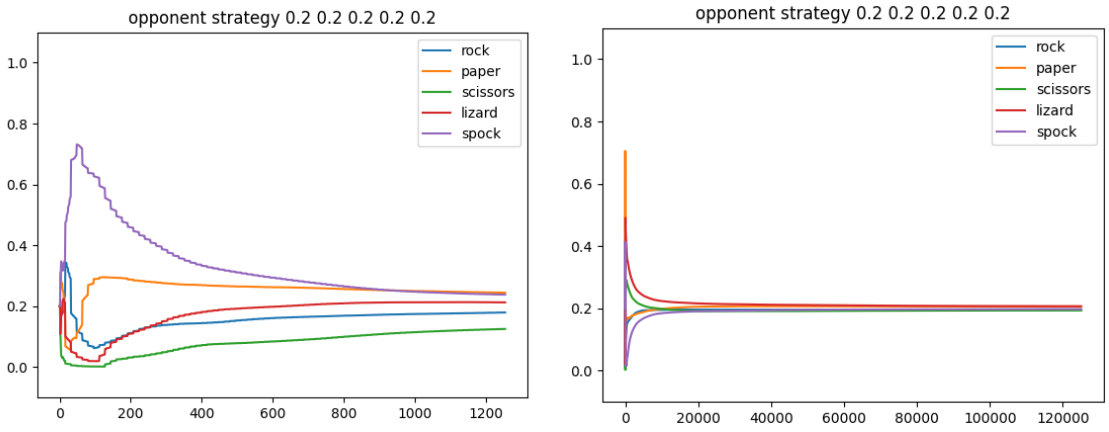


Figure 3.2.1.1 and Figure 3.2.1.2 Show the change in action probabilities for each action: Rock Paper Scissors Lizard Spock over the number of iterations (10,000 and 1,000,000 respectively)

3.2.2 Opponent Strategy (0.4, 0.2, 0.2, 0.1, 0.1)

10,000 iterations – Time Taken (s): 0.0320.

Action	Action Probability	Expected Value	Error
Rock	0.00038	0	0.039%
Paper	0.26443	1	73.55%
Scissors	0.00058	0	0.058%
Lizard	0.00039	0	0.039%
Spock	0.73422	1	26.58%

1,000,000 iterations – Time taken (s): 3.1591.

Action	Action Probability	Expected Value	Error
Rock	0.0	0	0%
Paper	0.23069	1	76.93%
Scissors	0.0	0	0%
Lizard	0.0	0	0%
Spock	0.7693	1	23.07%

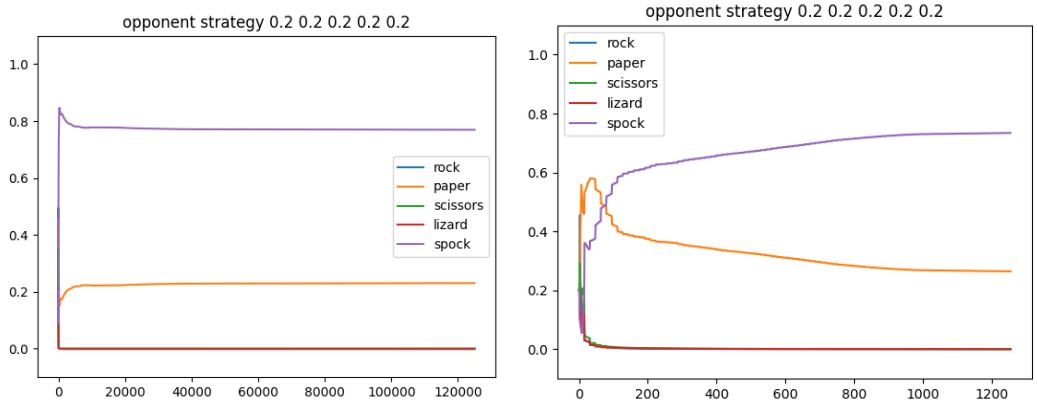


Figure 3.2.2.1 and Figure 3.2.2.2 Show the change in action probabilities for each action: Rock Paper Scissors Lizard Spock over the number of iterations (10,000 and 1,000,000 respectively)

3.2.3 Nash Equilibrium

10,000 Iterations – Time taken (s): 0.0505.

Action	Player Action Probabilities	Opponent Action Probabilities	Expected Value (Both)	Player Error	Opponent Error
Rock	0.18398	0.21410	0.2	8.01%	7.05%
Paper	0.16511	0.22464	0.2	17.445%	12.32%
Scissors	0.20064	0.20363	0.2	0.32%	1.815%
Lizard	0.23490	0.17698	0.2	17.45%	11.51%
Spock	0.21534	0.18063	0.2	7.67%	9.685%

1,000,000 iterations – Time taken (s): 4.7908.

Action	Player Action Probabilities	Opponent Action Probabilities	Expected Value (Both)	Player Error	Opponent Error
Rock	0.20203	0.20083	0.2	1.015%	0.415%
Paper	0.19823	0.19956	0.2	0.88%	0.22%
Scissors	0.20157	0.19895	0.2	0.785%	0.525%
Lizard	0.19655	0.20071	0.2	1.725%	0.355%
Spock	0.20160	0.19992	0.2	0.8%	0.04%

3.2.4 Results Analysis

We again observe with the opponent playing the optimal strategy, poor results with an average percentage error of 19.04%. This could be the fault of the algorithm falling into biases when observing the same results many times in a row.

For the set strategy, we see very interesting results that favour 2 actions (Paper and Spock). This is due to these 2 actions being the actions that counter the opponent's favoured move of rock. This, in turn, means the algorithm can choose any combination of probabilities for these 2 actions since they both work against the opponent's favoured move. It should be noted however that Spock was favoured more than Paper in this specific scenario due to the opponent having scissors as a higher probability action. This would explain the lesser use of Paper than Spock. For the opponent's evolving strategy, we again see expected results converging to 0.2 which is mathematically the optimal solution for this 5-action game.

It seems that this game takes slightly longer to solve averaging 0.04 and 4.4 seconds for 10,000 and 1,000,000 iterations respectively compared to RPS having 0.03 and 3.2 seconds. This is likely due to the 2 extra actions and the more action utilities and regrets that need to be calculated.

Judging from the results the best number of iterations for both versions of RPS is 1,000,000 giving accurate results without resulting in long computation times.

3.3 Kuhn Poker

For these tests, I will be utilising a Kuhn poker solution by Jonathan Gardner M.D.[9]. We will only be changing the number of iterations (3,000 and 30,000) as there are many information sets to cover. This solution will show the probabilities of all 12 information sets. We will be measuring time performance as well as the percentage error of our results compared to expected results. The information sets are highlighted in red and blue for players 1 and 2 respectively.

3,000 Iterations – Time taken (s): 0.8401.

Information Set	Strategy to Bet	Expected Result	Error
K	0.8451	1	15.49%
Q	0.0659	0	6.59%
J	0.2619	0.33333	21.42%
Kb	0.9796	1	2.04%
Qb	0.3562	0.33333	6.86%
Jb	0.0230	0	2.3%
Kp	0.9701	1	2.99%
Qp	0.0643	0	6.43%
Jp	0.3358	0.33333	0.74%
Kpb	0.9807	1	1.93%
Qpb	0.6098	0.66666	8.53%
Jpb	0.0216	0	2.16%

30,000 Iterations – Time taken (s): 28.4594.

Information Set	Strategy to Bet	Expected Result	Error
K	0.8835	1	11.65%
Q	0.0187	0	1.87%
J	0.2918	0.33333	12.45%
Kb	0.9928	1	0.72%
Qb	0.3489	0.33333	4.67%
Jb	0.0074	0	0.74%
Kp	0.9909	1	0.91%
Qp	0.0225	0	2.25%
Jp	0.3298	0.33333	1.05%
Kpb	0.9933	1	0.67%
Qpb	0.6324	0.66666	5.14%
Jpb	0.0070	0	0.7%

3.3.1 Results Analysis

We observe similar ideas to the previous 2 games showing with more iterations we get more accurate results. The average percentage error for 3,000 iterations and 30,000 iterations is 6.46% and 3.59% respectively.

The only outlier is the first action when given a King which seems to stay at that 0.8 probability. This could be because with both players playing optimally, player 1 will have a lower overall utility compared to player 2 (this is due to player 2 being able to react to a player 1 move). This value according to Harold E. Kuhn [3] himself is -1/18 and +1/18 for players 1 and 2 respectively. The game may be trying to change to the obvious idea of betting with King to gain a higher utility for that turn. This explains the overall higher error for player 1 moves compared to player 2 moves.

We observe that the times taken for both algorithms seem to have a large difference depending on iterations. This makes sense as the algorithm is recursive calling itself at least 3 times to get to the bottom of the game tree. With this information, we can deduce that this algorithm has a time complexity of O(3^n). Ideally, we would not want to run this algorithm for more than 300,000 iterations without incurring very long computation times.

Discussion

4.1 Conclusions

Concluding this report, regret minimisation proves to be an efficient algorithm for solving zero-sum games. We discovered from the results that straightforward games such as RPS could converge to a solution quickly with minimal performance issues or accuracy issues. On the other hand, Kuhn poker suffered more with accuracy and time complexity. We observe more complicated games with multiple stages require recursive use of the algorithm which severely affects time complexity. Also having a range of payouts for the algorithm to consider (instead of just winning or losing), causes more inaccuracies as we must decide whether it's more important to maximise gain or minimise loss.

However, despite weaker performance for more complex games, regret minimisation does approximate a solution much faster than other methods. For example, solving Kuhn poker using linear programming requires solving a 64x64 matrix [11] which would be extremely computationally and time-consuming. Some games even become impossible to find exact solutions for using mathematical approaches which highlights the benefits of approximating a solution using algorithms such as regret minimisation.

4.2 Future Use cases for CFR

We could try to implement this algorithm for different types of games if we can represent them mathematically. Games such as checkers and chess use the regret minimisation idea but for each instance a game could be in. For chess that would be 10^123 unique positions. Given that we converged to the Kuhn poker solution with few iterations, we could investigate regret minimisation for standard poker [18]. Solving this would require vast amounts of computation and time as there are many more variables to account for such as more than 2 players, more than 3 turns, more cards to be dealt, and learning the value of each hand.

Outside of games, we can apply similar ‘regret’ based ideas to the world of robotics. Deep reinforcement learning follows the same principle of repeating a task over and over to accumulate ‘regrets’ to perform them more efficiently. For example, we can set a robot to travel to certain places. Through deep reinforcement learning it may develop faster ways of traversing its area. More can be found in this book by Lapan, M [19].

List of References

[1] Neller, T. and Lanctot, M. (2013). An Introduction to Counterfactual Regret Minimization 1 Motivation. [online] Available at: https://www.ma.imperial.ac.uk/~dturaev/neller-lanctot.pdf.

[2] Ahluwalia, P. (2020). Basic Counterfactual Regret Minimization (Rock Paper Scissors). [online] Pranav on Data Science. Available at: https://www.pranav.ai/CFRM-RPS.

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

[4] www.youtube.com. (n.d.). Counterfactual Regret Minimization (AGT 26). [online] Available at: https://www.youtube.com/watch?v=ygDt_AumPr0

[5] Wikipedia. (2023). Kuhn poker. [online] Available at: https://en.wikipedia.org/wiki/Kuhn_poker.

[6] Ross, D. (2014). Game Theory (Stanford Encyclopedia of Philosophy). [online] Stanford.edu. Available at: https://plato.stanford.edu/entries/game-theory/.

[7] Jayasiri, V. (2024). vpj/poker. [online] GitHub. Available at: https://github.com/vpj/poker

[8] colab.research.google.com. (n.d.). Google Colaboratory. [online] Available at: https://colab.research.google.com/github/vpj/poker/blob/master/kuhn_cfr/kuhn_cfr.ipynb#scrollTo=YIoSr3fm302A

[9] Gardner, J. (2024). jonathangardnermd/counterfactualRegretMin. [online] GitHub. Available at: https://github.com/jonathangardnermd/counterfactualRegretMin?tab=readme-ov-file

[10] Allis, Louis Victor (1994-09-23). Searching for Solutions in Games and Artificial Intelligence (PhD thesis) [online] Available at: http://fragrieu.free.fr/SearchingForSolutions.pdf

[11] AI Poker Tutorial. (2021). AIPT Section 3.2: Solving Poker – Toy Poker Games. [online] Available at: https://aipokertutorial.com/toy-poker-games/.

[12] Li, Y. (n.d.). DEEP REINFORCEMENT LEARNING: AN OVERVIEW. [online] Available at: https://arxiv.org/pdf/1701.07274

[13] Koller, D. and Megiddo, N. (1992). The complexity of two-person zero-sum games in extensive form. Games and Economic Behavior, 4(4), pp.528–552.

[14] Salah E. Elmaghraby (2003). Encyclopedia of Physical Science and Technology (Third Edition), 2003. Chapter 3.

[15] Kreps, D.M. (1989). Nash Equilibrium. Game Theory, pp.167–177.

[16] www.scientificlib.com. (n.d.). John Forbes Nash, Jr. [online] Available at: https://www.scientificlib.com/en/Mathematics/Biographies/JohnForbesNashJr.html

[17] cbom.atozmath.com. (n.d.). Game Theory problem using Linear programming method Method & Example-1. [online] Available at: https://cbom.atozmath.com/example/CBOM/GameTheory.aspx?q=lpp&q1=E1#:~:text=The%20linear%20programming%20technique%20is

[18] Carter, B. (2023). News: GTO Poker Theories - The Regret Minimization Framework. [online] Pokerstrategy.com. Available at: https://www.pokerstrategy.com/news/world-of-poker/GTO-Poker-Theories-The-Regret-Minimization-Framework_128267/

[19] Lapan, M. (2020). Deep reinforcement learning hands-on : apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more. Birmingham, UK: Packt Publishing Ltd.

[20] Axelrod, R. (1980). Effective Choice in the Prisoner's Dilemma. Journal of Conflict Resolution, 24(1), pp.3–25. doi:https://doi.org/10.1177/002200278002400101.

Other Sources Used:

[1] Chen, J. (2021). Nash Equilibrium. [online] Investopedia. Available at: https://www.investopedia.com/terms/n/nash-equilibrium.asp.

[2] Wikipedia. (2023). Kuhn poker. [online] Available at: https://en.wikipedia.org/wiki/Kuhn_poker.

[3] Zinkevich, M., Johanson, M., Bowling, M. and Piccione, C. (n.d.) (2007). Regret Minimization in Games with Incomplete Information. [online] Available at: https://proceedings.neurips.cc/paper/2007/file/08d98638c6fcd194a4b1e6992063e944-Paper.pdf.

[4] Lanctot, M., Waugh, K., Zinkevich, M. and Bowling, M. (n.d.) (2007). Monte Carlo Sampling for Regret Minimization in Extensive Games. [online] Available at: https://proceedings.neurips.cc/paper/2009/file/00411460f7c92d2124a67ea0f4cb5f85-Paper.pdf.

[5] Counterfactual Regret Minimization. (n.d.). Available at: https://web.mit.edu/~gfarina/www/6S890/L14_cfr.pdf

[6] Nikhil R. (2023). Counterfactual Regret Minimization or How I won any money in Poker? [online] Available at: https://rnikhil.com/2023/12/31/ai-cfr-solver-poker.html#:~:text=Counterfactual%20regret%20is%20how%20much

Appendix A
Self-appraisal

A.1 Critical self-evaluation

At the start of this project, I had set a meeting with my supervisor to talk about the project at hand. We were able to reduce the task of exploring regret minimisation down to the implementation of the algorithm on Rock Paper Scissors and Kuhn Poker. I started coding my solution for Rock Paper Scissors having frequent meetings with my supervisor to discuss issues or any misunderstandings on my behalf. I had completed a fair amount of work and decided to put aside the project to work on my other modules for that semester. After that, I took a break from work entirely to focus on Ramadan and Eid. As a Muslim, this is extremely important to take time off to focus on my religion during this month. However, coming back to the project, I realised I did not have as much time as anticipated. This also came with the realisation that implementing Kuhn poker was a much larger task than I had initially thought. After discussing with my supervisor, we decided the best move forward was for me to analyse a Kuhn poker solution instead which I could break down and explain in my report. The rest of the weeks leading up to the deadline were of examining the Kuhn poker solution as well as writing up my report.

A.2 Personal reflection and lessons learned

I think a big problem that I suffered from was a lack of schedule and more importantly, researching the time needed for certain goals for my project. While I had a vague schedule in my outline, this turned out to not be very realistic in terms of the effort required for those tasks. I also should’ve factored in the month of Ramadan in the outline as I had assumed I would be able to work efficiently in that month.

Generally, I have learned that big projects like these require a lot of planning as well as a deep understanding of the subject matter beforehand to properly allocate tasks. I've also learned to set aside periods where I know I will not be working as well as periods to focus on my other university work.

A.3 Legal, social, ethical, and professional issues

One key aspect of this algorithm is the fact that it approximates the best strategy in each game. The issue comes with games such as poker which fit perfectly in use with an algorithm like this. Poker is such a game where real money is involved and introducing ‘unofficial’ ways to increase your chances of winning can be problematic. Most online gambling websites have rules that using external algorithms to ‘cheat’ is against their policy and they can ban you from their casinos.

A.3.1 Legal issues

Legally speaking, cheating in any in-person gambling game can warrant up to a $10,000 fine or even prison time. This can also lead to issues with the developer of such algorithms used by the person, and they may be brought in for questioning as well.

Other than that, we know that implementations of these algorithms are publicly available online, and legal for anyone to use or change.

A.3.2 Social issues

Gambling games in general are disliked by the public due to their exploitative and addictive nature. One would be concerned that someone could see how regret minimisation is used to solve games such as the ones present in casinos. This, in turn, misleads them to believe they could make money from using the algorithm when the house is always favoured in most of these games.

A.3.3 Ethical issues

Since these algorithms are mathematically created and do not collect or use any external data, there are next to no ethical concerns regarding regret minimisation. Furthermore, there is no real threat from the use of this algorithm to any external user. The only concern would be its use by casinos to calculate odds in their games.

A.3.4 Professional issues

As I worked on this project and report with only help from my supervisor, I cannot say I have encountered any professional issues worth bringing up. All issues that could have come up would’ve been the fault of me alone.

Appendix B
External Materials

Coded solutions for RPS and 5-action RPS will be shown below. A GitHub link will also be made available below. My GitHub repository has been made available to both my supervisor and assessor.

https://github.com/uol-feps-soc-comp3931-2324-classroom/final-year-project-Salbih100.git

rps_trainer.py

```
import random
import matplotlib.pyplot as plt
import time

'''
This program allows for Basic CounterFactual Regret Minimisation
for ROCK, PAPER, Scissors, Lizard, Spock. This Program is mostly based
on the Paper by Todd W.Neller and MarcLanctot. There are also functions
based on code by Pranav Ahluwalia. Links will be below.

https://www.ma.imperial.ac.uk/~dturaev/neller-lanctot.pdf
https://www.pranav.ai/CFRM-RPS
'''

# Rock Beats  Scissors and loses to Paper
# Paper Beats Rock and loses to Scissors
# Scissors Beats Paper and loses to Rock

# This is the main class for the trainer.  RPS = Rock, Paper, Scissors
class rpsTrainer:
    def __init__(self, opp_strategy):
        self.ROCK = 0
        self.PAPER = 1
        self.SCISSORS = 2
        self.NUM_ACTIONS = 3

        # Initialising player arrays
        self.regret_sum = [0,0,0]
        self.strategy = [0,0,0]
        self.strategy_sum = [0,0,0]

        # Arrays to keep track of strategies at certain iterations
        self.rockstrats = []
        self.paperstrats = []
        self.scissorsstrats = []
```

```
        # Initialising opposition arrays
        self.opp_strategy = opp_strategy
        self.opp_regret_sum = [0,0,0]
        self.opp_strategy_sum = [0,0,0]

    # Gets the current strategy for the player.
    # Returns an array for the strategy and the sum of strategies
    # Strategy array follows structure like this [0.4, 0.3, 0.1] as the
probabilities of playing Rock, Paper, Scissors respectively
    def get_strategy(self):
        normalising_sum = 0
        self.strategy = [0,0,0]

        # If regret less than or equal to 0, that option is not factored into
the strategy
        for x in range(self.NUM_ACTIONS):
            if self.regret_sum[x] > 0:
                self.strategy[x] = self.regret_sum[x]
            else :
                self.strategy[x] = 0
            normalising_sum += self.strategy[x]

        # Normalises strategy probabilities so they add to 1
        # Initial strategy is split evenly between RPS
        for x in range(self.NUM_ACTIONS):
            if normalising_sum > 0:
                self.strategy[x] = self.strategy[x] / normalising_sum
            else :
                self.strategy[x] = 1.0 / self.NUM_ACTIONS
            self.strategy_sum[x] += self.strategy[x]


        return self.strategy, self.strategy_sum

    # Same as player get_strategy() but for the opponent
    def get_strategy_opp(self):
        normalising_sum = 0
        self.opp_strategy = [0,0,0]

        for x in range(self.NUM_ACTIONS):
            if self.opp_regret_sum[x] > 0:
                self.opp_strategy[x] = self.opp_regret_sum[x]
            else :
                self.opp_strategy[x] = 0
            normalising_sum += self.opp_strategy[x]

        for x in range(self.NUM_ACTIONS):
```

```
            if normalising_sum > 0:
                self.opp_strategy[x] = self.opp_strategy[x] / normalising_sum
            else :
                self.opp_strategy[x] = 1.0 / self.NUM_ACTIONS
            self.opp_strategy_sum[x] += self.opp_strategy[x]


        return self.opp_strategy, self.opp_strategy_sum

    # Gets an action based on the probabilities of the player strategy
    # def get_action(self, strategy):
    #     r = random.uniform(0,1)
    #     if r >= 0 and r < strategy[0]:
    #         return 0
    #     elif r >= strategy[0] and r < strategy[0] + strategy[1]:
    #         return 1
    #     elif r >= strategy[0] + strategy[1] and r < sum(strategy):
    #         return 2
    #     else:
    #         return 0

    def get_action(self, strategy):
        r = random.uniform(0,1)
        a = 0
        cumulative_probability = 0

        while ( a < self.NUM_ACTIONS -1):
            cumulative_probability += strategy[a]
            if r < cumulative_probability:
                break
            a += 1
        return a

    # Training algorithm based on https://www.pranav.ai/CFRM-RPS
    def train(self, iterations):
        iteration = 0
        action_utility = [0,0,0]
        for i in range(0,iterations):
            avg = self.get_avg_strategy()

            # Keep track of the strategies every 100 iterations (for graph
production)
            if iteration & 100 == 0:
                self.rockstrats.append(avg[0])
                self.paperstrats.append(avg[1])
                self.scissorsstrats.append(avg[2])

            # Retrieve Actions
```

```
            t = self.get_strategy()
            strategy = t[0]
            strategySum = t[1]

            my_action = self.get_action(strategy)
            # Define an arbitrary opponent strategy from which to adjust
            other_action = self.get_action(self.opp_strategy)
            # Opponent Chooses scissors
            if other_action == 2:
                # Utility(Rock) = 1
                action_utility[0] = 1
                # Utility(Paper) = -1
                action_utility[1] = -1
            # Opponent Chooses Rock
            elif other_action == 0:
                # Utility(Scissors) = -1
                action_utility[2] = -1
                # Utility(Paper) = 1
                action_utility[1] = 1
            # Opopnent Chooses Paper
            else:
                # Utility(Rock) = -1
                action_utility[0] = -1
                # Utility(Scissors) = 1
                action_utility[2] = 1

            # Add the regrets from this decision
            for i in range(self.NUM_ACTIONS):
                self.regret_sum[i] += action_utility[i] -
action_utility[my_action] + 0.1
            iteration +=1

    # Nash equilibrium based on https://www.pranav.ai/CFRM-RPS
    def nash_equilibrium(self, iterations):
        iteration = 0
        action_utility = [0,0,0]
        strategy_sum1 = [0,0,0]
        strategy_sum2 = [0,0,0]

        regret_sum1 = [0,0,0]
        regret_sum2 = [0,0,0]

        for x in range(iterations):

            t1 = self.get_strategy()
            strategy1 = t1[0]
            strategy_sum1 = t1[1]
            my_action = self.get_action(strategy1)
```

```
            t2 = self.get_strategy_opp()
            strategy2 = t2[0]
            strategy_sum2 = t2[1]
            opp_action = self.get_action(strategy2)

            # Opponent Chooses scissors
            if opp_action == self.NUM_ACTIONS - 1:
                # Utility(Rock) = 1
                action_utility[0] = 1
                # Utility(Paper) = -1
                action_utility[1] = -1
            # Opponent Chooses Rock
            elif opp_action == 0:
                # Utility(Scissors) = -1
                action_utility[self.NUM_ACTIONS - 1] = -1
                # Utility(Paper) = 1
                action_utility[1] = 1
            # Opopnent Chooses Paper
            else:
                # Utility(Rock) = -1
                action_utility[0] = -1
                # Utility(Scissors) = 1
                action_utility[2] = 1


            # Add the regrets from this decision
            for i in range(0,self.NUM_ACTIONS):
                self.regret_sum[i] += action_utility[i] -
action_utility[my_action]
                self.opp_regret_sum[i] += -(action_utility[i] -
action_utility[my_action])
            # print("self", self.strategy_sum)
            # print("opp", self.opp_strategy_sum)
        return strategy_sum1, strategy_sum2

    # Compares players actions to opponents and change BOTH strategies
accordingly
    # Returns player strategy and opponent strategy
    def rps_to_nash(self, iterations):
        strats = self.nash_equilibrium(iterations)
        s1 = sum(strats[0])
        s2 = sum(strats[1])
        for i in range(3):
            if s1 > 0:
                strats[0][i] = strats[0][i]/s1
            if s2 > 0:
                strats[1][i] = strats[1][i]/s2
```

```
        return strats[0], strats[1]

    # Get the average strategy using the sum of every strategy
    def get_avg_strategy(self):
        avg_strategy = []
        for x in range(self.NUM_ACTIONS):
            avg_strategy.append(0)

        normalising_sum = 0
        for x in range(self.NUM_ACTIONS):
            normalising_sum += self.strategy_sum[x]

        for x in range(self.NUM_ACTIONS):
            if (normalising_sum > 0):
                avg_strategy[x] = self.strategy_sum[x] / normalising_sum
            else:
                avg_strategy[x] = 1.0 / self.NUM_ACTIONS

        return avg_strategy
        #return (self.print_avg_strategy(avg_strategy))

    # Creates a graph to show how the avg strategy changes with iterations
    def show_graph(self, graph_title):
        #plt.title("Opponent Strategy: 1/3 1/3 1/3")
        plt.title(graph_title)
        plt.axis([None, None, -0.1, 1.1])
        plt.plot(self.rockstrats, label="rock")
        plt.plot(self.paperstrats, label="paper")
        plt.plot(self.scissorsstrats, label="scissors")
        plt.legend(loc='best')
        plt.show()
        pass

    # Prints out avg strategy in a readable way
    def print_avg_strategy(self):
        avg_strategy = self.get_avg_strategy()

        round_value = 7

        string = "Trainer Strategy \n"
        string += "Rock: "
        string += str(round(avg_strategy[0],round_value))
        # string += "\n"
        string += " Paper: "
        string += str(round(avg_strategy[1],round_value))
        # string += "\n"
        string += " Scissors: "
```

```
        string += str(round(avg_strategy[2],round_value)) + "\n"

        return string

    # Prints out opponent strategy in readable way
    def print_opp_strategy(self):
        opps = self.opp_strategy

        round_value = 7

        string = "\nOpponent Strategy \n"
        string += "Rock: "
        string += str(round(opps[0], round_value)) + " "
        string += "Paper: "
        string += str(round(opps[1], round_value)) + " "
        string += "Scissors: "
        string += str(round(opps[2], round_value)) + "\n"

        return string

# ------------------------------------------------------------------------
------- #

# Main method to run trainer
# use train() method to train player against a static opponent strategy
# use rps_to_nash() to train both player and opponent to obtain optimal RPS
strategy
# use show_graph() when using train() only
# print_opp_strategy() and print_avg_strategy() to print out strategies

def main_method():
    # Input opponent strategy
    #opp_strategy = [1/3, 1/3, 1/3]
    opp_strategy = [0.4, 0.3, 0.3]
    #opp_strategy = [0.7, 0.15, 0.15]
    graph_title = "Opponent Strategy: 0.4, 0.3, 0.3"
    trainer = rpsTrainer(opp_strategy)
    start = time.time()
    iterations = 1000000
    # trainer.train(iterations)

    m = trainer.rps_to_nash(iterations)
    print("Player Strategy:", m[0])
    print("Opponent Strategy:", m[1])

    print(trainer.print_opp_strategy())
    print("Number of Iterations:", iterations, "\n")
    print(trainer.print_avg_strategy())
```

```
    print("Time taken (s):", time.time() - start)
    trainer.show_graph(graph_title)
    pass

main_method()
```

rpslsp_trainer.py

```
import random
import matplotlib.pyplot as plt
import time

'''
This program allows for Basic CounterFactual Regret Minimisation
for ROCK, PAPER, Scissors, Lizard, Spock. This Program is mostly based
on the Paper by Todd W.Neller and MarcLanctot. There are also functions
based on code by Pranav Ahluwalia. Links will be below.

https://www.ma.imperial.ac.uk/~dturaev/neller-lanctot.pdf
https://www.pranav.ai/CFRM-RPS
'''

# Rock Beats Lizard and Scissors and loses to Paper and Spock
# Paper Beats Rock and Spock and loses to Scissors and Lizard
# Scissors Beats Paper and Lizard and loses to Rock and Spock
# Lizard Beats Paper and Spock and loses to Rock and Scissors
# Spock Beats Rock and Scissors and loses to Paper and Lizard

# This is the main class for the trainer.  RPSLSP = Rock, Paper, Scissors,
Lizard, Spock
class rpslspTrainer:
    def __init__(self, opp_strategy):
        self.ROCK = 0
        self.PAPER = 1
        self.SCISSORS = 2
        self.LIZARD = 3
        self.SPOCK = 4
        self.NUM_ACTIONS = 5

        # Initialising player arrays
        self.regret_sum = [0,0,0,0,0]
        self.strategy = [0,0,0,0,0]
        self.strategy_sum = [0,0,0,0,0]

        # Arrays to keep track of strategies at certain iterations
        self.rockstrats = []
        self.paperstrats = []
        self.scissorsstrats = []
```

```
        self.lizardstrats = []
        self.spockstrats = []

        # Initialising opposition arrays
        self.opp_strategy = opp_strategy
        self.opp_regret_sum = [0,0,0,0,0]
        self.opp_strategy_sum = [0,0,0,0,0]

    # Gets the current strategy for the player.
    # Returns an array for the strategy and the sum of strategies
    # Strategy array follows structure like this [0.4, 0.3, 0.1] as the
probabilities of playing Rock, Paper, Scissors respectively
    def get_strategy(self):
        normalising_sum = 0
        self.strategy = [0,0,0,0,0]

        # If regret less than or equal to 0, that option is not factored into
the strategy
        for x in range(self.NUM_ACTIONS):
            if self.regret_sum[x] > 0:
                self.strategy[x] = self.regret_sum[x]
            else :
                self.strategy[x] = 0
            normalising_sum += self.strategy[x]

        # Normalises strategy probabilities so they add to 1
        # Initial strategy is split evenly between RPS
        for x in range(self.NUM_ACTIONS):
            if normalising_sum > 0:
                self.strategy[x] = self.strategy[x] / normalising_sum
            else :
                self.strategy[x] = 1.0 / self.NUM_ACTIONS
            self.strategy_sum[x] += self.strategy[x]


        return self.strategy, self.strategy_sum

    # Same as player get_strategy() but for the opponent
    def get_strategy_opp(self):
        normalising_sum = 0
        self.opp_strategy = [0,0,0,0,0]

        for x in range(self.NUM_ACTIONS):
            if self.opp_regret_sum[x] > 0:
                self.opp_strategy[x] = self.opp_regret_sum[x]
            else :
                self.opp_strategy[x] = 0
            normalising_sum += self.opp_strategy[x]
```

```
        for x in range(self.NUM_ACTIONS):
            if normalising_sum > 0:
                self.opp_strategy[x] = self.opp_strategy[x] / normalising_sum
            else :
                self.opp_strategy[x] = 1.0 / self.NUM_ACTIONS
            self.opp_strategy_sum[x] += self.opp_strategy[x]


        return self.opp_strategy, self.opp_strategy_sum

    # Gets an action based on the probabilities of the player strategy
    def get_action(self, strategy):
        r = random.uniform(0,1)
        a = 0
        cumulative_probability = 0

        while ( a < self.NUM_ACTIONS -1):
            cumulative_probability += strategy[a]
            if r < cumulative_probability:
                break
            a += 1
        return a

    # Training algorithm based on https://www.pranav.ai/CFRM-RPS
    def train(self, iterations):
        iteration = 0
        action_utility = [0,0,0,0,0]
        actions = 5
        for i in range(0,iterations):
            avg = self.get_avg_strategy()

            # Keep track of the strategies every 100 iterations (for graph
production)
            if iteration & 100 == 0:
                self.rockstrats.append(avg[0])
                self.paperstrats.append(avg[1])
                self.scissorsstrats.append(avg[2])
                self.lizardstrats.append(avg[3])
                self.spockstrats.append(avg[4])

            # Retrieve Actions
            t = self.get_strategy()
            strategy = t[0]
            strategySum = t[1]

            my_action = self.get_action(strategy)
            # Define an arbitrary opponent strategy from which to adjust
```

```
other_action = self.get_action(self.opp_strategy)

# Opponent Chooses Rock
if other_action == 0:
    # Utility(Paper) = 1
    action_utility[1] = 1
    # Utility(Scissors) = -1
    action_utility[2] = -1

    # Utility(Lizard) = -1
    action_utility[3] = -1
    # Utility(Spock) = 1
    action_utility[4] = 1

# Opopnent Chooses Paper
elif other_action == 1:
    # Utility(Rock) = -1
    action_utility[0] = -1
    # Utility(Scissors) = 1
    action_utility[2] = 1

    # Utility(Lizard) = 1
    action_utility[3] = 1
    # Utility(Spock) = -1
    action_utility[4] = -1

# Opponent Chooses scissors
elif other_action == 2:
    # Utility(Rock) = 1
    action_utility[0] = 1
    # Utility(Paper) = -1
    action_utility[1] = -1

    # Utility(Lizard) = -1
    action_utility[3] = -1
    # Utility(Spock) = 1
    action_utility[4] = 1


# Opopnent Chooses Lizard
elif other_action == 3:
    # Utility(Rock) = 1
    action_utility[0] = 1
    # Utility(Paper) = -1
    action_utility[1] = -1
    # Utility(Scissors) = 1
    action_utility[2] = 1
```

```
                # Utility(Spock) = -1
                action_utility[4] = -1

            # Opopnent Chooses Spock
            elif other_action == 4:
                # Utility(Rock) = 1
                action_utility[0] = -1
                # Utility(Scissors) = 1
                action_utility[2] = -1
                # Utility(Paper) = 1
                action_utility[1] = 1

                # Utility(Lizard) = 1
                action_utility[3] = 1



            # Add the regrets from this decision
            for i in range(self.NUM_ACTIONS):
                self.regret_sum[i] += action_utility[i] -
action_utility[my_action] + 0.1
            iteration +=1

    # Nash equilibrium based on https://www.pranav.ai/CFRM-RPS
    def nash_equilibrium(self, iterations):
        iteration = 0
        action_utility = [0,0,0,0,0]
        strategy_sum1 = [0,0,0,0,0]
        strategy_sum2 = [0,0,0,0,0]

        regret_sum1 = [0,0,0,0,0]
        regret_sum2 = [0,0,0,0,0]

        for x in range(iterations):

            t1 = self.get_strategy()
            strategy1 = t1[0]
            strategy_sum1 = t1[1]
            my_action = self.get_action(strategy1)

            t2 = self.get_strategy_opp()
            strategy2 = t2[0]
            strategy_sum2 = t2[1]
            other_action = self.get_action(strategy2)

            # Opponent Chooses Rock
            if other_action == 0:
                # Utility(Paper) = 1
                action_utility[1] = 1
```

```
        # Utility(Scissors) = -1
        action_utility[2] = -1

        # Utility(Lizard) = -1
        action_utility[3] = -1
        # Utility(Spock) = 1
        action_utility[4] = 1

    # Opopnent Chooses Paper
    elif other_action == 1:
        # Utility(Rock) = -1
        action_utility[0] = -1
        # Utility(Scissors) = 1
        action_utility[2] = 1

        # Utility(Lizard) = 1
        action_utility[3] = 1
        # Utility(Spock) = -1
        action_utility[4] = -1

    # Opponent Chooses scissors
    elif other_action == 2:
        # Utility(Rock) = 1
        action_utility[0] = 1
        # Utility(Paper) = -1
        action_utility[1] = -1

        # Utility(Lizard) = -1
        action_utility[3] = -1
        # Utility(Spock) = 1
        action_utility[4] = 1


    # Opopnent Chooses Lizard
    elif other_action == 3:
        # Utility(Rock) = 1
        action_utility[0] = 1
        # Utility(Paper) = -1
        action_utility[1] = -1
        # Utility(Scissors) = 1
        action_utility[2] = 1

        # Utility(Spock) = -1
        action_utility[4] = -1

    # Opopnent Chooses Spock
    elif other_action == 4:
        # Utility(Rock) = 1
```

```
                action_utility[0] = -1
                # Utility(Scissors) = 1
                action_utility[2] = -1
                # Utility(Paper) = 1
                action_utility[1] = 1

                # Utility(Lizard) = 1
                action_utility[3] = 1


            # Add the regrets from this decision
            for i in range(0,self.NUM_ACTIONS):
                self.regret_sum[i] += action_utility[i] -
action_utility[my_action]
                self.opp_regret_sum[i] += -(action_utility[i] -
action_utility[my_action])
            # print("self", self.strategy_sum)
            # print("opp", self.opp_strategy_sum)
        return strategy_sum1, strategy_sum2

    # Compares players actions to opponents and change BOTH strategies
accordingly
    # Returns player strategy and opponent strategy
    def rps_to_nash(self, iterations):
        strats = self.nash_equilibrium(iterations)
        s1 = sum(strats[0])
        s2 = sum(strats[1])
        for i in range(5):
            if s1 > 0:
                strats[0][i] = strats[0][i]/s1
            if s2 > 0:
                strats[1][i] = strats[1][i]/s2

        return strats[0], strats[1]

    # Get the average strategy using the sum of every strategy
    def get_avg_strategy(self):
        avg_strategy = []
        for x in range(self.NUM_ACTIONS):
            avg_strategy.append(0)

        normalising_sum = 0
        for x in range(self.NUM_ACTIONS):
            normalising_sum += self.strategy_sum[x]

        for x in range(self.NUM_ACTIONS):
            if (normalising_sum > 0):
                avg_strategy[x] = self.strategy_sum[x] / normalising_sum
```

```
            else:
                avg_strategy[x] = 1.0 / self.NUM_ACTIONS

        return avg_strategy
        #return (self.print_avg_strategy(avg_strategy))

    # Creates a graph to show how the avg strategy changes with iterations
    def show_graph(self):
        plt.title("opponent strategy 0.2 0.2 0.2 0.2 0.2")
        plt.axis([None, None, -0.1, 1.1])
        plt.plot(self.rockstrats, label="rock")
        plt.plot(self.paperstrats, label="paper")
        plt.plot(self.scissorsstrats, label="scissors")
        plt.plot(self.lizardstrats, label="lizard")
        plt.plot(self.spockstrats, label="spock")
        plt.legend(loc='best')
        plt.show()
        pass

    # Prints out avg strategy in a readable way
    def print_avg_strategy(self):
        avg_strategy = self.get_avg_strategy()
        round_value = 5

        string = "Trainer Strategy \n"
        string += "Rock: "
        string += str(round(avg_strategy[0],round_value))
        # string += "\n"
        string += " Paper: "
        string += str(round(avg_strategy[1],round_value))
        # string += "\n"
        string += " Scissors: "
        string += str(round(avg_strategy[2],round_value))
        string += " Lizard: "
        string += str(round(avg_strategy[3],round_value))
        string += " Spock: "
        string += str(round(avg_strategy[4],round_value)) + "\n"

        return string

    # Prints out opponent strategy in readable way
    def print_opp_strategy(self):
        opps = self.opp_strategy
        string = "\nOpponent Strategy \n"
        string += "Rock: "
        string += str(opps[0]) + " "
        string += "Paper: "
        string += str(opps[1]) + " "
```

```
        string += "Scissors: "
        string += str(opps[2]) + " "
        string += "Lizard: "
        string += str(opps[3]) + " "
        string += "Spock: "
        string += str(opps[4]) + "\n"

        return string

# ------------------------------------------------------------------------
------- #

# Main method to run trainer
# use train() method to train player against a static opponent strategy
# use rps_to_nash() to train both player and opponent to obtain optimal RPS
strategy
# use show_graph() when using train() only
# print_opp_strategy() and print_avg_strategy() to print out strategies

def main_method():
    # Input opponent strategy
    # opp_strategy = [0.4, 0.2, 0.2, 0.1, 0.1]
    opp_strategy = [0.2, 0.2, 0.2, 0.2, 0.2]
    trainer = rpslspTrainer(opp_strategy)
    start = time.time()
    #trainer.train(1000000)

    m = trainer.rps_to_nash(1000000)
    print("Player Strategy:", m[0])
    print("Opponent Strategy:", m[1])

    print(trainer.print_opp_strategy())
    print(trainer.print_avg_strategy())
    print("Time taken (s):", time.time() - start)
    trainer.show_graph()
    pass

main_method()
```