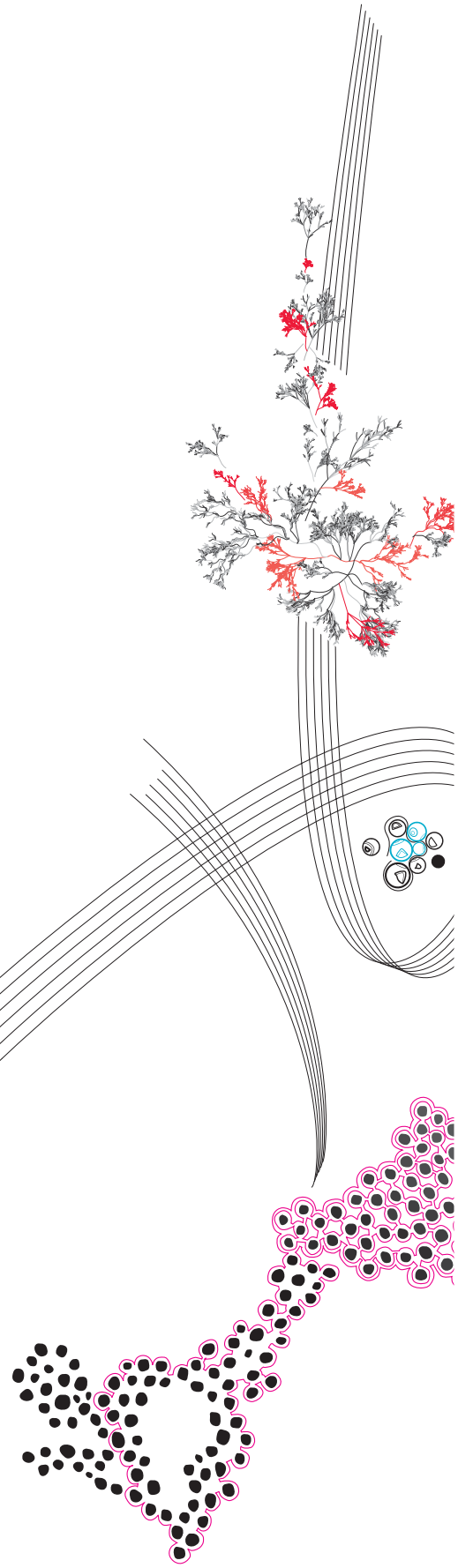

BSc Thesis Applied Mathematics

Weak acyclicity of the iterated prisoner's dilemma with a memory of one period.

Daan Koorn

Supervisor: J.M. Meylahn

July, 2023

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Preface

I want to thank Janusz M. Meylahn from the bottom of my heart for supervising my thesis. He gave me honest critical feedback with useful suggestions on how to improve. I learned much, not only mathematically but also on how to improve my writing. He also deserves credit for giving up some of his time to give feedback and answer my questions outside the predetermined appointments.
Next I want to thank Fanaat and Bellettrie for letting me use the Drakenkelder to study and its denizens for their help and support. I specifically want to thank Lu, Elise, and Marleen for the weekly dinners and for their motivation. Lastly I want to thank: Ruben, Nico, Remco, Justen, Emiel, Peter, and Nander for their help and suggestions and for listening to me and help point out some mistakes.

Weak acyclicity of the iterated prisoner's dilemma with a memory of one period.

Daan Koorn[a]

July, 2023

Abstract

I develop a Python algorithm that visualizes the combined state values of the ϵ-greedy policies of an iterated prisoner's dilemma with a memory of one period. From this one can read the best response to a given policy of the opposing player. I also attempt to show that the iterated prisoner's dilemma is weakly acyclic. This can be done by constructing a potential function for the game. The two candidate functions explored turn out not to be potential functions. The approach developed here does however lend itself to extensions to games with more states and actions.

Keywords: Best response, Exploration, Prisoner's dilemma, Value state, Weakly acyclic.

1 Introduction

In a world where people are trying to automate and optimize we often train artificial intelligence (AI) to do as much as possible for us. We are now automating the training and therefore learning of AIs. One way to learn is using reinforcement learning, here the AI tries to maximise a reward signal [10]. A form of reinforcement learning is Q-learning [12] which is based on temporal differences [10].
The concept of reinforcement learning has been expended into multi-agent reinforcement learning [6] where multiple agents[1] try to learn in the same environment. Learning in dynamic situations is said to be one of the grand challenges in multi-agent reinforcement learning[13]. In [3] the authors present a multi-agent Q-learning algorithm that converges. This algorithm is significant in the sense that agents learn in a semi-dynamic environment where each agent has its own goal. One of the conditions for this algorithm to converge is that the game is weakly acyclic.

Another article [7] present the mutual pure strategy[2] best-response dynamics for a some games including the prisoner's dilemma. Of note are their graphs with conditions on the rewards for each best response to exist.
A different article [5] looks at what makes temporal-difference reinforcement learning with ϵ-greedy strategies cooperative, they found that the win-stay, lose-switch (Pavlov) strategy leads to stable cooperation in an iterated prisoner's dilemma.

In this paper, I aim to contribute to this literature by looking at the best response dynamics for the iterated prisoner's dilemma with memory one with the inclusion of exploration. This is unique in the sense that I use exploration and all pure policies with a

[a]Email: d.koorn@student.utwente.nl
[1]I use the terms agent and player interchangeably
[2]I use the terms strategy and policy interchangeably

memory of one period contrary to [7]. I will not go into the detail of cooperation as done in [5].
The best response to a policy played by an opponent, links the game to being weakly acyclic. A game is said to be weakly acyclic if every improvement path is finite. This means that for every combination of starting policies the players can only improve their own rewards by changing policies a finite amount of times.
My second goal is to show that the iterated prisoner's dilemma with a memory of one period is weakly acyclic. One could calculate by hand all improvement paths and see if they are all finite. My aim is to show that the prisoner's dilemma is weakly acyclic by constructing a potential instead.

Structure

In section 2, I introduce the specific setting I am working in and the corresponding notation. Next, in section 3, I explain which methods are used to obtain the results, discussed afterwards in section 4. Here I show the best responses and afterwards I discuss the weak acyclicity. Lastly in section 5 I conclude the results and discuss how to move forward.

2 Setting

In the standard prisoner's dilemma [2], also used in [7], there are two criminals who work together. They both get caught for committing a crime and are interrogated separately by the police. Here they both have two options. The first one is to stay silent and don't tell anything (cooperating with their partner) and the second is to betray their partner (defecting) and work together with the police. These actions are denoted by: C and D respectively.
One can refer to the outcome of the actions by listing the actions played by each player. If both player defected, the outcome would be DD, If player one cooperates and player two defects the outcome can be called CD. I also refer to the outcomes as states with state 1 being DD, state 2 DC, state 3 CD and state 4 CC

Player 1	Player 2	
	D	C
D	P, P	T, S
C	S, T	R, R

Table 1: Prisoner's dilemma

The parameters P, T, S, R denote the rewards (where a higher reward coincides with a lower punishment). For the prisoner's dilemma, we assume that: $S < P < R < T$. This implies that it is in the advantage of an individual to betray their companion while it is preferable to work together over both betraying each other. This is exactly why this is a dilemma, betraying the other is more beneficial, but the moment both players do this they both receive a lesser reward than if they both worked together.
When one plays the same game against the same opponent repeatedly it is called an iterated prisoner's dilemma. This gives the game a new aspect, namely reacting to what your opponent does. If a person betrays the other, the other can do the same back, the same holds for cooperating. This policy is called TIT FOR TAT (TFT) and is one of the stronger policies for an iterated prisoner's dilemma [4]. In an environment with other

policies, some random, others intricate, TFT got the highest average score. After the results were distributed, the competition for the best policy was repeated and TFT was again the best of 62 entries. The reason for this was that TFT was never the first to defect, but it does reciprocate. On the other hand, it is also forgiving and willing to cooperate afterwards again. It was also shown that in a random environment TFT can learn to dominate and is resilient to invasions by other policies.
TFT relies only on a memory of one, namely the action the opponent played last round. An interesting question is whether more intricate policies that have a longer memory perform better than TFT. Strangely, that is not the case if the same game is repeated indefinitely. The agent with the shortest memory determines the effective memory of the opponent [9]. This implies that if we include TFT as a policy, including policies that require a longer memory it is only beneficial against some or none of the other policies. I will therefore look only at the policies that require a memory of one. I encode this memory by the states discussed earlier.

After having looked at the game from a micro level, let us now determine the policies on a macro level. I will use π_i to denote the strategy used by player i with $i \in \{1, 2\}$ as in [3, 7, 10]. Since both players have a memory of one period they remember the results of the last game played, i.e. one of the states $\sigma = \{DD, DC, CD, CC\}$. Based upon this, they can choose their next move. The policies they can choose can therefore depend upon the results of the previous round. Given the limitation of pure policies with an exploration rate, this can result in 16 different policies for each player. The combination of actions from the player can result from going from each state to any state in the next iteration.

Exploration is used in reinforcement learning, like Q-learning, to get an estimate of the value of taking an action. This way an agent can take an action and learn from the consequences of taking such an action to get a better picture of the whole system and the long/short term benefits of taking each action [10]. Using exploration does however influence the policies and how they interact with each other. An example of this is the Grimm trigger policy. This policy co-operates, but if the opponent plays defect once it will always play defect in the future. In a game with exploration, the always cooperate policy will try defecting at some point. This triggers the Grimm trigger to play all defect until it itself explores.
TFT is another policy that feels the consequences of exploration. If an opponent explores the defect action, TFT reacts to that by retaliating. This will cause two opposing TFT policies to continue to retaliate until one of them explores at the right time. This can again result in the overall performance of these policies being worse than without exploration.

The exploration ensures that every policy pair will at some point in an iterated prisoner's dilemma go to every state, so every part of a policy becomes important at some point. Adding exploration to the system means that the transitions are no longer deterministic, but probabilistic.
The probability to explore is given by ϵ. When an agent decides to explore they choose uniformly between one of their actions as defined in [10]. This results in the following transition probabilities $\rho(\sigma_1 \to \sigma | \pi_1, \pi_2)$. Given a state, the policies of player one and player two without the exploration would determine the next state. With the exploration this gives a transition probability of:

$$\left(1 - \epsilon + \frac{\epsilon}{2}\right)^2 = 1 - \epsilon + \frac{\epsilon^2}{4}. \tag{1}$$

The state opposite (see table 1) to the one determined by the pure policies would require

both players to explore, resulting in a transition probability of:

$$(\frac{\epsilon}{2})^2 = \frac{\epsilon^2}{4}. \tag{2}$$

The two remaining states require one of the players to explore but not the other:

$$\frac{\epsilon}{2} \cdot (1 - \frac{\epsilon}{2}) = \frac{\epsilon}{2} - \frac{\epsilon^2}{4}. \tag{3}$$

The probabilities themselves do not change, the variables in the probability transitions are which one to use when going from one state to another dependent on the policies of the players.

Q-learning is a model of reinforcement learning introduced in [12], where this Q-value indicates the expected discounted reward from taking a specific policy from a specific state. I do not have a specific starting state. Adding one would require giving both players an additional choice that only matters in the first state. Due to the exploration this only has a limited impact on the combined value state. When looking at the infinite discounted expected rewards the contribution from the starting state goes to zero when the discount factor is high. Therefore, I assume that we start in a random state with a uniform distribution. This means that the value state from time t can be expressed as:

$$V_t(\sigma_1, [\pi_1, \pi_2]) = r(\sigma_1) + \delta \cdot \sum_{\sigma} \rho(\sigma_1 \to \sigma | \pi_1, \pi_2) V_{t+1}(\sigma, [\pi_1, \pi_2]). \tag{4}$$

Here $r(\sigma)$ represents the immediate rewards from state σ and δ represents the discount factor. From this one can make a 16×16 matrix consisting of all the possible pure policies with ϵ-greedy exploration for both the players. Then each entry of the matrix will consist of the Q-values resulting from the policies the two players can use. With these values, I attempt to show that the set of strategies is weakly a-cyclic using a potential function.

3 Method

In order to determine if the prisoner's dilemma is weakly acyclic I made a python code that can be separated into 3 parts. The first part is making a 16×16 matrix with all the possible pure policies for each player. The second part consists of calculating all the value functions given the policies of the players. The third part is substituting the variables and making the potential function. The full code can be found in the appendix.

3.1 Policy matrix

With the prisoner's dilemma, a 1 corresponds with taking the defect action and a 2 with the co-operate action. I purposefully am not using binary to code as in [7]. The reason for this is twofold. Firstly, I intended this code to be applicable to a wider scope of problems including 3 or more actions. For this I could still have used 0 and 1, but since the audience is predominantly mathematicians, I start counting at one instead of zero.
Each player has 16 different pure policies they can choose from. In order to see how each policy fares against the policies from the other player, I construct a 16×16 matrix with all the possible combinations of policies. In 'makinglistformatrix' as defined in the appendix, I start by creating a list containing ones with a length equal to the amount of states. After this, I construct a list containing this list with the ones, in order to get a long list equal to the amount of pure policies. I end up with a list containing; [1,1,1,1], sixteen times.

This list is one of the four inputs for the function that constructs all possible policies. For this recursive algorithm, called 'recursivematrixmaker', I use four inputs; the list I want to change the policies of, which of the entries of the policy I need to change, the number of states and lastly the amount of different actions each player can take in a single prisoner's dilemma game.
The first thing that happens in the recursive part of the algorithm is dividing the length of the list by the amount of actions. With this, I divide the inputted list in equal parts. Then I change the entries of the policies depending on in which part they are. So after one iteration we go from a list containing: [1,1,1,1] sixteen times to two list. The first list contains [1,1,1,1] eight times and the second consist of [2,1,1,1] eight times. I then check if the state changed was the last state in the sequence, if that is not the case, I recursively call the algorithm twice once for each list. The only other input I change is the entry of the policy that needs to be changed. Afterwards the algorithm returns the new list. This list contains all the possible pure strategies.
After having determined all the different policies, I loop over all the different policies twice to create a 16×16 matrix of all the different combination of policies each player can use. Each entry of the matrix looks as follows: [[2,1,1,2], [1,1,1,2]]. The first entries correspond to the policy player one uses, in this case WSLS, and the second list corresponds to the policy of player two, Grimm Trigger.

Lastly, I change the order of some policies of player two. When the last actions played by player 1 is defect and the last action from player 2 is to co-operate we start at the state DC. This is an advantageous state for player 1, hence if player 1 is playing TFT [1,2,1,2] their policy says to play co-operate in the next state, since player 2 co-operated last round. If player 2 is also playing TFT, he would play defect next round. However, looking at the policy [1,2,1,2] it tells us to co-operate. The reasoning behind this inconsistency is that the states DC and CD are looked at from player 1's perspective. If we were to look at it from player 2's perspective, it would be the other way around. To solve this, we change the order of policies of player two. This way the policies of the players match instead of the actions they play in each state.

3.2 Value functions

The calculations of the value functions 'Calculating_expected_long_term_value' are automated using the 16×16 policy matrix. I cycle over the matrix and calculate the combined value from the policy of both players.
I start off with defining all the possible transition probabilities as in equations: 1, 2 and 3. Next I have the symbolic expressions of the equations for all four states e.g. for state 1, (DD):

$$\begin{aligned} V_t(DD, [\pi_1, \pi_2]) = r(DD) + \\ \delta \cdot (\rho(DD \to DD|\pi_1, \pi_2) \cdot V_t(DD, [\pi_1, \pi_2]) + \\ \rho(DD \to DC|\pi_1, \pi_2) \cdot V_t(DC, [\pi_1, \pi_2]) + \\ \rho(DD \to CD|\pi_1, \pi_2) \cdot V_t(CD, [\pi_1, \pi_2]) + \\ \rho(DD \to CC|\pi_1, \pi_2) \cdot V_t(CC, [\pi_1, \pi_2])). \end{aligned}$$

Moving all the terms of $V_t(DD, [\pi_1, \pi_2])$ to the same side and dividing both sides of the equation yields the formula for the state value.

$$\begin{aligned}(1 - \delta \cdot (\rho(DD \to DD|\pi_1, \pi_2)) \cdot V_t(DD, [\pi_1, \pi_2]) = \\ r(DD) + \rho(DD \to DC|\pi_1, \pi_2) \cdot V_t(DC, [\pi_1, \pi_2]) \\ + \rho(DD \to CD|\pi_1, \pi_2) \cdot V_t(CD, [\pi_1, \pi_2]) \\ + \rho(DD \to CC|\pi_1, \pi_2) \cdot V_t(CC, [\pi_1, \pi_2])).\end{aligned}$$

$$\begin{aligned}V_t(DD, [\pi_1, \pi_2]) = &\frac{r(DD) + \rho(DD \to DC|\pi_1, \pi_2) \cdot V_t(DC, [\pi_1, \pi_2])}{(1 - \delta \cdot (\rho(DD \to DD|\pi_1, \pi_2))} \\ &+ \frac{\rho(DD \to CD|\pi_1, \pi_2) \cdot V_t(CD, [\pi_1, \pi_2])}{(1 - \delta \cdot (\rho(DD \to DD|\pi_1, \pi_2))} \\ &+ \frac{\rho(DD \to CC|\pi_1, \pi_2) \cdot V_t(CC, [\pi_1, \pi_2]))}{(1 - \delta \cdot (\rho(DD \to DD|\pi_1, \pi_2))}.\end{aligned}$$

After I have collected the formulas for each state, I will look at the policies of the players. I start by looking at the first entry of both the policies. This action decides what the players do when they are in the first state (DD). With this information, I can substitute the correct transition probabilities. These transition probabilities only depend on the action taken in that state, I can therefore create the formulas relatively easy. I do this for all the states and end up with four state value equations.
When the formulas are known for all the states one can substitute them and solve the system of equations to get the state value expressed as a formula of the rewards S, P, R, T, the discount δ and the exploration ϵ.
If we assume the starting state is uniformly chosen at random, we can define the combined state value as follows:

$$V_t(\sigma, [\pi_1, \pi_2]) = \frac{V_t(DD, [\pi_1, \pi_2]) + V_t(DC, [\pi_1, \pi_2]) + V_t(CD, [\pi_1, \pi_2]) + V_t(CC, [\pi_1, \pi_2])}{4}.$$

Each combination of two policies in the input results in a single symbolic value as output. This results in a 16×16 matrix with the value player one can expect from playing a policy against a specific policy.

3.3 Showing weakly acyclic

The original goal of the thesis was to compare the combined values for the different policies and solve these inequalities for 2 to 4 variables to calculate the best response. Python, specifically SymPy, is not able to solve multivariate equations [11].

I will circumvent this by substituting the variables with specific values and constructing a potential function.

'A function $W : y \to \Re$ is an *ordinal potential* for Γ, if for every $i \in N$ and for every $y^{-i} \in Y^{-i}$

$$u^i(y^{-i}, x) - u^i(y^{-i}, z) > 0 \text{ iff } W(y^{-i}, x) - W(y^{-i}, z) > 0 \text{ for every } x, y \in Y^i \text{ '[8]. (5)}$$

Here N is the number of agents and Γ represents a strategic game. By lemma 3.2 in [8] and from [1] 'we call a strategic game weakly acyclic (respectively, BR-weakly acyclic) if

for any joint strategy there exist a finite improvement path (respectively, BR-improvement path) that start at it.' An improvement path is a sequence of policies where with every policy switch that agent improves his own reward. This implies the iterated game is weakly acyclic. Since we only allow pure strategies with exploration, we have a finite game. For creating a potential, I need the rewards from both players. These rewards can be obtained by taking the transpose of the rewards player one can expect.
The first potential I try is the sum of the combined value of both players. The second potential I try is the sum of the normalized combined value of both players. For this normalization, I only normalize each column. This way, the best response against a given policy gets the value of one. Lastly, I plot the matrices in the form of a heatmap to show the results.

4 Results

Using the code described in section 3, we can print the matrix with the symbolic expressions. These 256 expressions together are practically unworkable. By substituting the rewards together with the discount and exploration we get a heatmap which can show the best responses for a given policy of the opponent. I show the heatmap where the columns are normalized in figure 1. For the heatmap of the combined state values see figure 7 in the appendix.

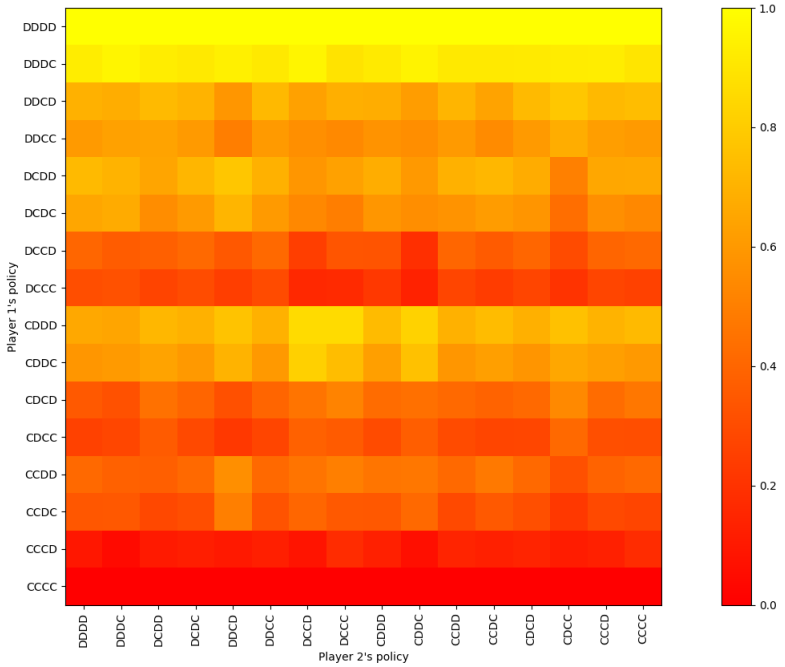

Figure 1: Best response given player 2's policy.
Here I set $\epsilon = 0.4$, $\delta = 0.2$

The heatmaps can be read by first selecting the policy player 2 is using[3], and secondly one can look at the column for the best response where one represents the best response.

[3] A reminder that the policies are written from player 1's perspective. This means that the actions used in state 2 and 3 are swapped for player 2.

In order to visualize the results I do need to set the rewards in the prisoner's dilemma. I received some unpublished work from my supervisor to compare my results with. These assume the following rewards : $T = 1.2$, $R = 1$, $P = 0$ and $S = -0.2$, I will from here on out be using these values as well.

The following figure from my supervisor shows what the best response against the Grimm trigger policy (DDDC) is, depending on the discount factor and the exploration rate. The 1 represents the area when all defect (DDDD) is the best response, 2 is Grimm trigger and 16 stands for always cooperating (CCCC).

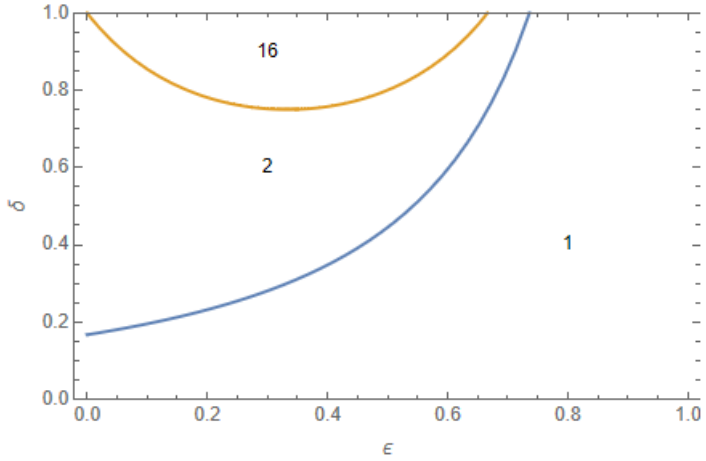

Figure 2: Best response to Grimm trigger.

In figure 1, I plotted the best response for $\epsilon = 0.4$ and $\delta = 0.2$. In the following figure, one can see the best responses for $\epsilon = 0.4$ and $\delta = 0.6$

As can be seen, the best response to the Grimm trigger policy is to play Grimm trigger. This corresponds with what figure 2 shows.

On the line in figure 2 where the best response switches from Grimm trigger to always cooperating, the code still shows Grimm trigger as the best response. When increasing the discount from 0.75758 to 0.83322 the heatmap does show that both Grimm trigger and the always cooperate policy are the best responses.

It makes sense that both codes do not agree on the value of δ as the value states defined in this thesis are slightly different from the standard. The value state is 5.396 when using an δ of 0.83322 the value state of Grimm trigger is 3.832 and that of CCCC is 3.67 when using the δ from figure 2. This difference is greater than a single factor, δ or any other. I was not able to find the cause of this deviation in time.

Something to note in figure 4 is that there are 8 different best responses to the Grimm trigger policy. This makes sense since Both DDDC and CCCC are best responses it is implied that it does not matter if one chooses to play D or C in the first 3 states and $2^3 = 8$. Any combination of these actions is therefore a best response as long as the player cooperates in state 4.

The benefit from the heatmaps is that they show the best response for every policy played by player 2. One can also find the equilibria from this. For example, If player 2 plays Grimm trigger, the best response is always cooperating. We can also determine the best response to this by looking at what player one would do if player 2 would always

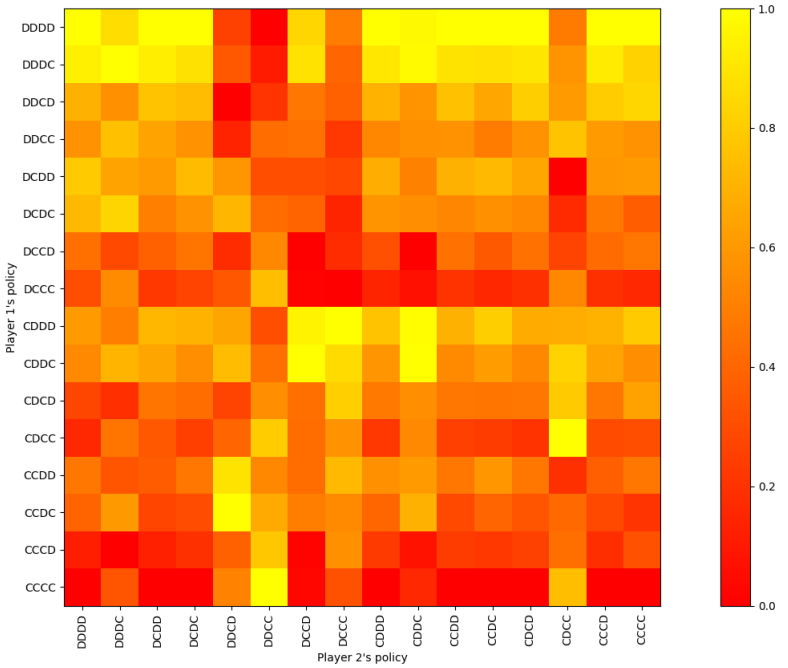

Figure 3: Best response given player 2's policy.
Here I set $\epsilon = 0.4$, $\delta = 0.6$

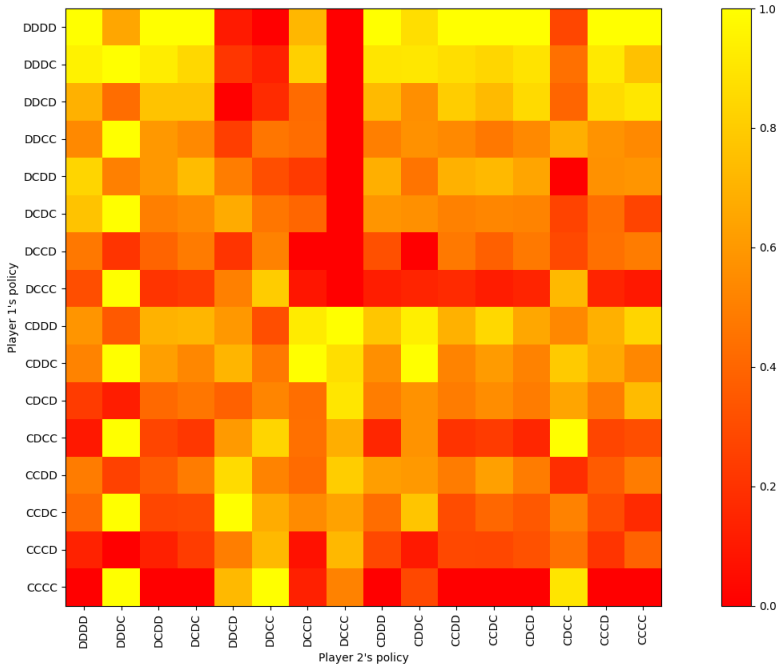

Figure 4: Best response given player 2's policy.
Here I set $\epsilon = 0.4$, $\delta = 0.83322$

cooperate, namely all defect. The best response to all defect is to play in kind and also always defect. When the best response to a policy is playing the same policy as the round before, the game has reached an equilibrium.
Since the prisoner's is a symmetric game the transpose of the matrix represents the rewards

obtained by the other player.

4.1 Potential function

For the potential function, I got two heatmaps, one for the sum of the value states and one from the sum of the normalized state values.

The first one is obtained by summing the combined value state functions for both players, the combined state value heatmap of player one can be found in the appendix 7.

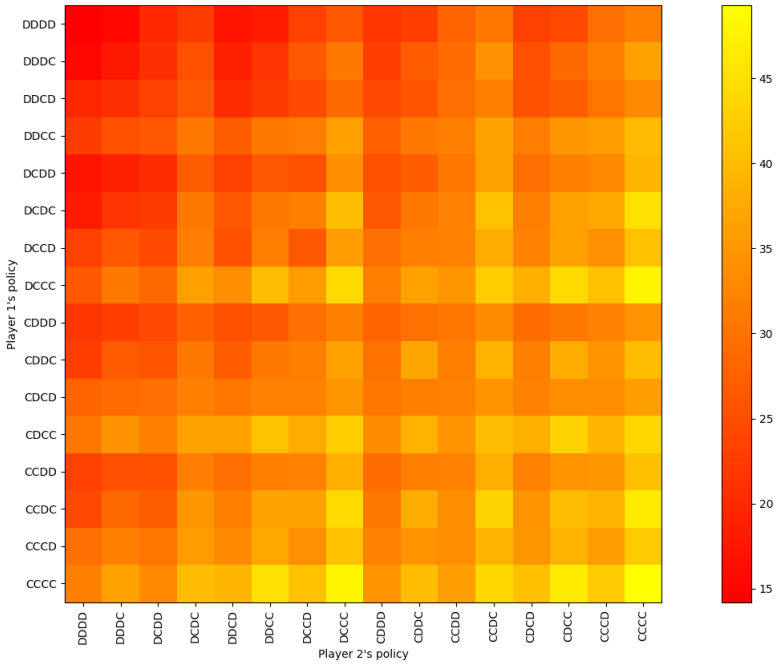

Figure 5: Potential from the sum of state values
Here I set $\epsilon = 0.4$, $\delta = 0.9$

It is immediately visible from this graph that the mutual cooperation has the highest potential. But we see in figure 8 that the best response against always cooperating is to always defect. Hence, the state value from playing DDDD is larger than the state value of policy CCCC. But for a potential this is the other way around, therefore this cannot be a potential by the definition as seen in equation 5. The reason that this doesn't work is that mutual cooperation results in the highest total rewards, but it does not take into consideration that for each single person this is not the optimal policy.

For the second attempt at the potential I added up the normalized state values. These normalized values are of each column, so the best response has a value of one. This implies that a value of two means a mutual best response and therefore an equilibrium.

While in some cases the potential makes sense, with the all defect policy of the WSLS policies which are both known best responses in some scenarios without exploration [7], this potential shows that the policy pair: CDCC and Grimm trigger has a higher potential than playing the always cooperating policy against the Grimm trigger policy. From figure 8 in the appendix, we can see that this is not the case and that by equation 5 this also cannot be a potential.

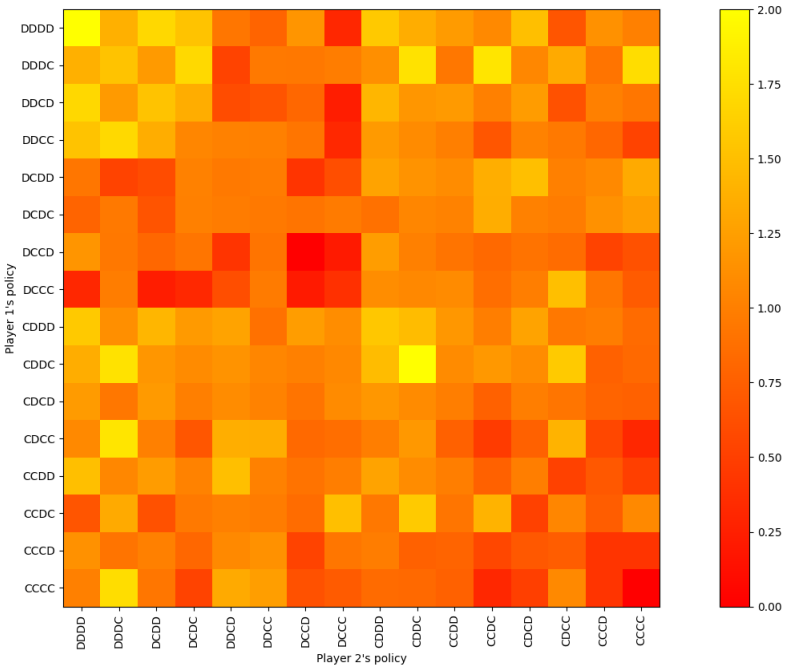

Figure 6: Potential from the sum of the normalized state values $\epsilon = 0.4$, $\delta = 0.9$

5 Conclusions

Contributions

The code developed in this thesis can be used for most rewards in a two-player, two-action game. It has specifically been developed for the prisoner's dilemma, but as the rewards can be substituted late in the process it can be used for other two-player, two action games. One does need to be careful in some scenarios where state values can become zero as no safeguard for this has been implemented. The code is also built with a discount factor in mind and is not usable without it.

The heatmaps are an alternative representation of best response networks. It not only shows the best response but also shows the other possible responses. This thesis includes exploration into the iterated prisoner's dilemma and can show the impact of different values.

There are some inconsistencies with the work from my supervisor. This could possibly be explained by a difference in how the state values are defined. This remains a guess as no connection between the results have been shown yet.
This thesis shows that one can theoretically use a potential to show a game is weakly acyclic though it does not manage to use this to show the prisoner's dilemma is weakly acyclic. It does explore two methods of constructing a potential and show that they do not work.

Extensions

There are a couple of possible extensions to this thesis. Presumably the most important is to redefine the value states to give a reward based upon the next state entered instead of the state the game is at. This was an unorthodox move based upon an incorrect understanding

of the material at hand and not having the necessary time to change multiple equations and checking if they work correctly.

Another option is to delve deeper into the potentials and trying to find one or proving it doesn't exist for certain rewards, discount factors, and exploration rate. Options for this are to look into squaring before adding the normalized values or taking a combination of the normalized state values or state-action values.

One could try to model in a different environment that can solve multivariate inequalities. Then one can take the symbolic programming output from the value states and solve them. I do have to warn that the symbolic output is extremely long and practically unusable. Together with the amount of equations that need to be solved it would be wise to take the speed at which the environment can solve the equations into account.

One can also speed up the calculation of the state values. At the moment more than 1024 symbolic equations are created, many of which are the same. With some smart programming this could presumably be improved. The same holds for solving the system of equations. While there are "only" 256 of these, this still requires some computation. In python there is an inbuilt function for this, which could presumably also be sped up.

Once the shortcomings of my approach have been addressed, there would be some other options to extend, namely adding more actions, players, or memory to the games or focussing on a different set of games like the stag hunt or hawk-dove (also referred to by "chicken").

Possibilities to extend the code

Some part of the code have been designed with expansion to more states or more agents in mind. The function 'recursivematrixmaker' in the appendix 6.2 has been designed to work for games with more than 2 actions. In the second part of the code in appendix 6.3 the idea is that by substituting the probabilities, a third action or one more player can be added without much complication. It would however cost some time to write out all the equations and possibilities.

References

[1] Krzysztof R. Apt and Sunil Simon. "A classification of weakly acyclic games". In: *Theory and Decision : An International Journal for Multidisciplinary Advances in Decision Science* 78.4 (Apr. 2015), pp. 501–524. issn: 0040-5833. doi: 10.1007/s11238-014-9436-1.

[2] Aykut Argun, Agnese Callegari, and Giovanni Volpe. "Simulation of Complex Systems". In: *Simulation of Complex Systems* (Dec. 2021). doi: 10.1088/978-0-7503-3843-1.

[3] Gürdal Arslan and Serdar Yüksel. "Decentralized Q-Learning for Stochastic Teams and Games". In: *IEEE Transactions on Automatic Control* 62.4 (Apr. 2017), pp. 1545–1558. issn: 0018-9286. doi: 10.1109/TAC.2016.2598476.

[4] Robert Axelrod and William D Hamilton. "The Evolution of Cooperation". In: *New Series* 211.4489 (1981), pp. 1390–1396.

[5] Wolfram Barfuss and Janusz M. Meylahn. "Intrinsic fluctuations of reinforcement learning promote cooperation". In: *Scientific Reports* 13.1 (Jan. 2023). issn: 20452322. doi: 10.1038/s41598-023-27672-7.

[6] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. "A comprehensive survey of multiagent reinforcement learning". In: *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews* 38.2 (Mar. 2008), pp. 156–172. issn: 10946977. doi: 10.1109/TSMCC.2007.913919.

[7] J. M. Meylahn and L. Janssen. "Limiting Dynamics for Q-Learning with Memory One in Symmetric Two-Player, Two-Action Games". In: *v2022 (20220101)* 2022 (2022). issn: 1099-0526. doi: 10.1155/2022/4830491.

[8] Dov Monderer and Lloyd S. Shapley. "Potential Games". In: *Games and Economic Behavior* 14.1 (May 1996), pp. 124–143. issn: 0899-8256. doi: 10.1006/GAME.1996.0044.

[9] William H. Press and Freeman J. Dyson. "Iterated Prisoner's Dilemma contains strategies that dominate any evolutionary opponent". In: *Proceedings of the National Academy of Sciences of the United States of America* 109.26 (June 2012), pp. 10409–10413. issn: 00278424. doi: 10.1073/PNAS.1206569109/ASSET/8958A58B-36A3-4D38-8CC5-AD36C70FE6D8/ASSETS/GRAPHIC/PNAS.1206569109I77.GIF. url: https://www.pnas.org/doi/abs/10.1073/pnas.1206569109.

[10] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. Ed. by A Bradford Book. 2nd edition (W.I.P.) Cambridge, Massachusetts and London, England: The MIT Press, 2014.

[11] *SymPy 1.12 documentation*. url: https://docs.sympy.org/latest/index.html.

[12] Christopher J.C.H. Watkins and Peter Dayan. "Technical Note: Q-Learning". In: *Machine Learning* 8.3-4 (1992), pp. 279–292. issn: 0885-6125. doi: 10.1023/A:1022676722315.

[13] Yaodong Yang and Jun Wang. "An Overview of Multi-Agent Reinforcement Learning from Game Theoretical Perspective". In: (Nov. 2020). url: https://arxiv.org/abs/2011.00583v3.

6 Appendix

6.1 Heatmaps

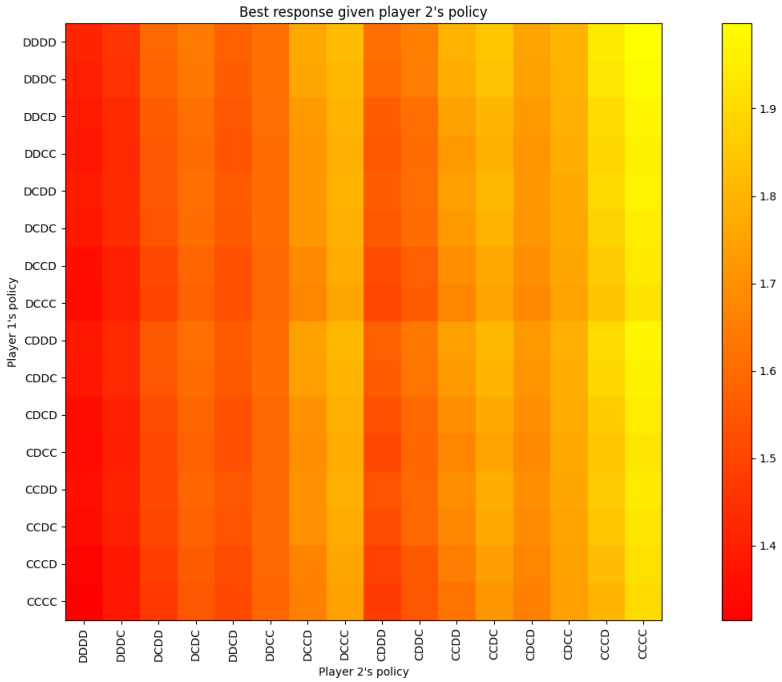

Figure 7: Combined state Values. $\delta = 0.2$, $\epsilon = 0.4$

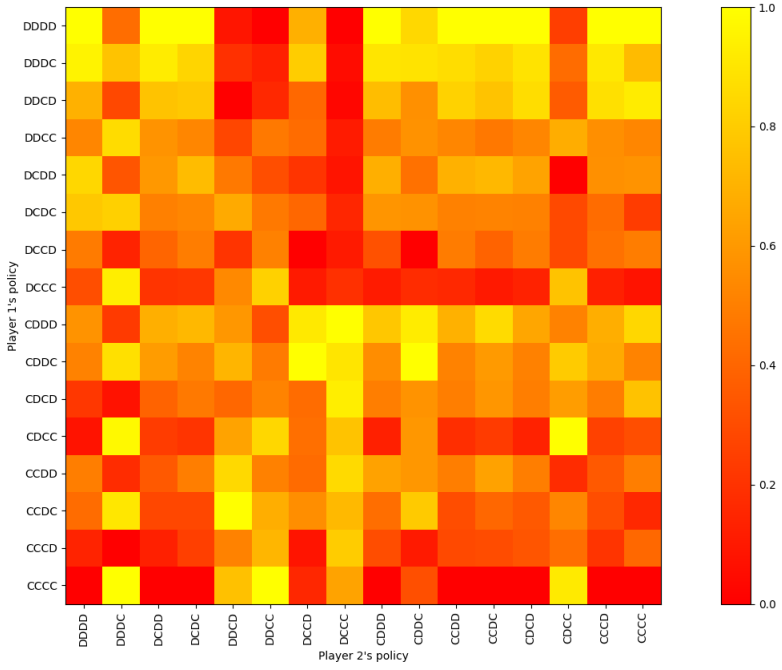

Figure 8: Combined state Values. $\delta = 0.9$, $\epsilon = 0.4$

6.2 Part one of the python code

```
from sympy import *

def makinglistformatrix(number_of_states: int, amount_of_choices:
                                                    int) -> list:
```

```
    size_of_matrix = number_of_states ** amount_of_choices
    row = []
    matrix = []
    for i in range(number_of_states):
        row.append(1)
    for i in range(size_of_matrix):
        matrix.append(row.copy())
    # works only for 2 choices at the moment.
    returnt = recursivematrixmaker(matrix, 0, number_of_states,
                                               amount_of_choices)

    returnt2 = returnt.copy()

    the_matrix = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    for i in range(len(returnt)):
        first_entry_of_the_whole_row = returnt[i]
        the_row = []
        for j in range(len(returnt2)):
            the_row.append([first_entry_of_the_whole_row, returnt2[j]])
        the_matrix[i] = the_row
    #i am manually exchanging the strategies as in the states CD and DC
                        the strategies slighly differ otherwise (so wsls
                        is not defined correctly when just switching
    for i in range(len(the_matrix)):
        the_matrix[i][4],the_matrix[i][2]=the_matrix[i][2],the_matrix[i][4]
        the_matrix[i][3],the_matrix[i][5]=the_matrix[i][5],the_matrix[i][3]
        the_matrix[i][10],the_matrix[i][12]=the_matrix[i][12],
                                            the_matrix[i][10]
        the_matrix[i][11],the_matrix[i][13]=the_matrix[i][13],
                                            the_matrix[i][11]

    return the_matrix


def recursivematrixmaker(list_to_change: list, state_to_change: int,
                         number_of_states: int, amount_of_choices: int) -> list:

    length_of_the_list = len(list_to_change)
    which_lists_to_change = Rational(length_of_the_list, amount_of_choices)
    for i in range(amount_of_choices):
        startlist = i * which_lists_to_change
        endlist = which_lists_to_change * (i + 1)
        for j in list(range(startlist, endlist)):
            list_to_change[j][state_to_change] = i + 1

        if state_to_change != number_of_states - 1:
            recursivematrixmaker(list_to_change[startlist:endlist],
            state_to_change + 1, number_of_states, amount_of_choices)
```

```
    return list_to_change
```

6.3 Part two of the python code

```
import sympy.categories
from sympy import *


def Calculating_expected_long_term_value(matrix: list) -> object:
    '''
    This function takes a set of integers (1 or 2) as the actions of each
                        player in in each state. (basically it is the basis
                                            for all the transitions)
      #it takes integers instead of booleans so it is easier/possible to
                                            extend this code later on.
    :param matrix: is a m by n matrix where m is the number of people
                   and n the number of states. The values of the matrix
                   correspond with the actions. In this case 1 is
                   defect and 2 is coopereate.
    :type list
    :return: object
    '''

    # Here i define the variables (symbols for sympy) so python is able
                                                    to give symbolic returns
    # discount is a value such that 1 means no discount and 0 means you
                            don't care about the future. is alpha is the
                            discount (e.g. 5% then discount is 0.95)
    discount = Symbol('discount', real=True)
    suckers_reward = Symbol('suckers_reward', real=True)
    temptation_reward = Symbol('temptation_reward', real=True)
    cooperative_reward = Symbol('cooperative_reward',real=True)
    uncooperative_reward = Symbol('uncooperative_reward', real=True)
    exploration_rate = Symbol('exploration_rate', nonnegative=True,real=True)

    # All 4 different states. So i can calculate the value of each of them.
    DD = Symbol('DD')
    CD = Symbol('DC')
    DC = Symbol('CD')
    CC = Symbol('CC')

    # State 1 is DD
    # State 2 is DC
    # State 3 is CD
    # State 4 is CC
    # the rewards are always from player ones perspective.
    # every reward is a combination of the other states so we only need to
                            assign what the probability is to go to each
                            state for each of the possible states
```

```
transaction_one_minus = 1 - exploration_rate + (exploration_rate**2)/4
transaction_exploration = exploration_rate/2 - (exploration_rate ** 2)/4
transaction_less_likely_exploration = (exploration_rate ** 2)/4

# parameter to assign the probabilities to
prob_parameter_one = Symbol('prob_parameter_one')
prob_parameter_two = Symbol('prob_parameter_two')
prob_parameter_three = Symbol('prob_parameter_three')
prob_parameter_four = Symbol('prob_parameter_four')
prob_parameter_five = Symbol('prob_parameter_five')
prob_parameter_six = Symbol('prob_parameter_six')
prob_parameter_seven = Symbol('prob_parameter_seven')
prob_parameter_eight = Symbol('prob_parameter_eight')
prob_parameter_nine = Symbol('prob_parameter_nine')
prob_parameter_ten = Symbol('prob_parameter_ten')
prob_parameter_eleven = Symbol('prob_parameter_eleven')
prob_parameter_twelve = Symbol('prob_parameter_twelve')
prob_parameter_thirteen = Symbol('prob_parameter_thirteen')
prob_parameter_fourteen = Symbol('prob_parameter_fourteen')
prob_parameter_fifteen = Symbol('prob_parameter_fifteen')
prob_parameter_sixteen = Symbol('prob_parameter_sixteen')

# DD= reward+ ...DC +...CD+...CC/...

eqnDD = Eq(DD, (uncooperative_reward + discount * (prob_parameter_one *
        DC + prob_parameter_two * CD + prob_parameter_three * CC)) /
                (1 - discount * prob_parameter_four))
eqnDC = Eq(DC, (temptation_reward + discount * (
             prob_parameter_five * DD + prob_parameter_six *
                         CD + prob_parameter_seven * CC)) /
                         (1 - discount * prob_parameter_eight))
eqnCD = Eq(CD, (suckers_reward + discount * (prob_parameter_nine *
        DD + prob_parameter_ten * DC + prob_parameter_eleven * CC)) /
                    (1 - discount * prob_parameter_twelve))
eqnCC = Eq(CC, (cooperative_reward + discount * (
             prob_parameter_thirteen * DD + prob_parameter_fourteen *
                             DC + prob_parameter_fifteen * CD)) /
                             (1 - discount * prob_parameter_sixteen))

if matrix[0][0] == 1:
    if matrix[1][0] == 1:
        eqnDDresult = eqnDD.subs(
            [(prob_parameter_one, transaction_exploration),
            (prob_parameter_two, transaction_exploration),
             (prob_parameter_three, transaction_less_likely_exploration),
             (prob_parameter_four, transaction_one_minus)])
    else:
        eqnDDresult = eqnDD.subs(
            [(prob_parameter_one, transaction_one_minus),
```

```
            (prob_parameter_two, transaction_less_likely_exploration),
             (prob_parameter_three, transaction_exploration), (
             prob_parameter_four, transaction_exploration)])
else:
    if matrix[1][0] == 1:
        eqnDDresult = eqnDD.subs(
            [(prob_parameter_one, transaction_less_likely_exploration),
            (prob_parameter_two, transaction_one_minus),
             (prob_parameter_three, transaction_exploration),
             (prob_parameter_four, transaction_exploration)])
    else:
        eqnDDresult = eqnDD.subs(
            [(prob_parameter_one, transaction_exploration),
            (prob_parameter_two, transaction_exploration),
             (prob_parameter_three, transaction_one_minus),
             (prob_parameter_four, transaction_less_likely_exploration)])
if matrix[0][1] == 1:
    if matrix[1][1] == 1:
        eqnDCresult = eqnDC.subs(
            [(prob_parameter_five, transaction_one_minus),
            (prob_parameter_six, transaction_exploration),
             (prob_parameter_seven, transaction_less_likely_exploration),
             (prob_parameter_eight, transaction_exploration)])
    else:
        eqnDCresult = eqnDC.subs(
        [(prob_parameter_five, transaction_exploration),
        (prob_parameter_six, transaction_less_likely_exploration),
        (prob_parameter_seven, transaction_exploration),
        (prob_parameter_eight, transaction_one_minus)])
else:
    if matrix[1][1] == 1:
        eqnDCresult = eqnDC.subs(
            [(prob_parameter_five, transaction_exploration),
            (prob_parameter_six, transaction_one_minus),
             (prob_parameter_seven, transaction_exploration),
             (prob_parameter_eight, transaction_less_likely_exploration)])
    else:
        eqnDCresult = eqnDC.subs([(
            prob_parameter_five, transaction_less_likely_exploration),
            (prob_parameter_six, transaction_exploration),
            (prob_parameter_seven, transaction_one_minus),
            (prob_parameter_eight, transaction_exploration)])
if matrix[0][2] == 1:
    if matrix[1][2] == 1:
        eqnCDresult = eqnCD.subs(
            [(prob_parameter_nine, transaction_one_minus),
            (prob_parameter_ten, transaction_exploration),
             (prob_parameter_eleven, transaction_less_likely_exploration),
             (prob_parameter_twelve, transaction_exploration)])
```

```
    else:
        eqnCDresult = eqnCD.subs(
            [(prob_parameter_nine, transaction_exploration),
            (prob_parameter_ten, transaction_one_minus),
             (prob_parameter_eleven, transaction_exploration),
             (prob_parameter_twelve, transaction_less_likely_exploration)])
else:
    if matrix[1][2] == 1:
        eqnCDresult = eqnCD.subs(
        [(prob_parameter_nine, transaction_exploration),
        (prob_parameter_ten, transaction_less_likely_exploration),
        (prob_parameter_eleven, transaction_exploration),
        (prob_parameter_twelve, transaction_one_minus)])
    else:
        eqnCDresult = eqnCD.subs(
        [(prob_parameter_nine, transaction_less_likely_exploration),
        (prob_parameter_ten, transaction_exploration),
        (prob_parameter_eleven, transaction_one_minus),
        (prob_parameter_twelve, transaction_exploration)])
if matrix[0][3] == 1:
    if matrix[1][3] == 1:
        eqnCCresult = eqnCC.subs(
            [(prob_parameter_thirteen, transaction_one_minus),
            (prob_parameter_fourteen, transaction_exploration),
             (prob_parameter_fifteen, transaction_exploration),
             (prob_parameter_sixteen, transaction_less_likely_exploration)])
    else:
        eqnCCresult = eqnCC.subs(
            [(prob_parameter_thirteen, transaction_exploration),
            (prob_parameter_fourteen, transaction_one_minus),
             (prob_parameter_fifteen, transaction_less_likely_exploration),
             (prob_parameter_sixteen, transaction_exploration)])
else:
    if matrix[1][3] == 1:
        eqnCCresult = eqnCC.subs(
        [(prob_parameter_thirteen, transaction_exploration),
        (prob_parameter_fourteen, transaction_less_likely_exploration),
        (prob_parameter_fifteen, transaction_one_minus),
        (prob_parameter_sixteen, transaction_exploration)])
    else:
        eqnCCresult = eqnCC.subs(
        [(prob_parameter_thirteen, transaction_less_likely_exploration),
        (prob_parameter_fourteen, transaction_exploration),
        (prob_parameter_fifteen, transaction_exploration),
        (prob_parameter_sixteen, transaction_one_minus)])

# solving the system of equations
Answer_from_equations = solve([eqnDDresult, eqnCDresult,
            eqnDCresult, eqnCCresult], DD, CD, DC, CC)
```

```
    answerDD = Answer_from_equations[DD]
    answerDC = Answer_from_equations[DC]
    answerCD = Answer_from_equations[CD]
    answerCC = Answer_from_equations[CC]

    answer = answerDD + answerDC + answerCD + answerCC / 4
    return (answer)
```

6.4 Part three of the python code

```
import copy

from sympy import *
from All_possibilities_in_one import *
from start_working_on_matrix import *
import csv
import matplotlib.pyplot as plt
from numpy import *
from copy import deepcopy
#needed for substitution here
discount = Symbol('discount', real=True)
suckers_reward = Symbol('suckers_reward', real=True)
temptation_reward = Symbol('temptation_reward', real=True)
cooperative_reward = Symbol('cooperative_reward', real=True)
uncooperative_reward = Symbol('uncooperative_reward', real=True)
exploration_rate = Symbol('exploration_rate', nonnegative=True, real=True)


input_matrix_list= makinglistformatrix(4, 2)

for i in range(len(input_matrix_list)):
    for j in range(len(input_matrix_list[i])):
        input_matrix_list[i][j]=Calculating_expected_long_term_value
                                    (input_matrix_list[i][j])

for i in range(len(input_matrix_list)):
    for j in range(len(input_matrix_list[i])):
        input_matrix_list[i][j]=input_matrix_list[i][j].subs(
                    [(uncooperative_reward,0),(cooperative_reward,1),
                    (temptation_reward, 1.2),(suckers_reward,-0.2),
                    (discount, 0.9),(exploration_rate, 0.4)])

data=input_matrix_list.copy()
for i in range(len(input_matrix_list)):
    for j in range(len(input_matrix_list[i])):
            data[i][j]=float(input_matrix_list[i][j])

                    # data2=copy(deepcopy(data))
                    #this gives an obious wrong best response
```

```
                        wich CCCC being the equilibrium
                    # for i in range(16):
                    #     for j in range(i,16):
                    #         data2[i][j],data2[j][i]=
                    #               data2[j][i],data2[i][j]
                    #
                    # potential=copy(deepcopy(data))
                    # for i in range(16):
                    #     for j in range(16):
                    #         potential[i][j]=
                    #           potential[i][j]+data2[i][j]
for i in range(16):
    for j in range(i,16):
        data[i][j],data[j][i]=data[j][i],data[i][j]
for i in range(16):
    minvalue=min(data[i])
    for j in range(16):
        data[i][j]=data[i][j]-minvalue

for i in range(16):
    maxvalue=max(data[i])
    for j in range(16):
        data[i][j]=data[i][j]/maxvalue

for i in range(16):
    for j in range(i,16):
        data[i][j],data[j][i]=data[j][i],data[i][j]

data2=copy(deepcopy(data))
for i in range(16):
    for j in range(i,16):
        data2[i][j],data2[j][i]=data2[j][i],data2[i][j]

potential=copy(deepcopy(data))
for i in range(16):
    for j in range(16):
        potential[i][j]=potential[i][j]+data2[i][j]


plt.imshow(potential,cmap='autumn')
# plt.title('Best response given player 2\'s policy' )
plt.xlabel('Player 2\'s policy')
plt.ylabel('Player 1\'s policy')
plt.colorbar()
visiblenumbers=copy(deepcopy(potential))
for i in range(16):
    for j in range(16):
        visiblenumbers[i][j]=round(potential[i][j],3)
        # print('checking')
```

```
        # print(visiblenumbers[i][j])
        # print(data[i][j])



for i in range(16):
    for j in range(16):
        plt.annotate(str(visiblenumbers[i][j]), xy=(j, i),
                     ha='center', va='center', color='black')
label_x_ax=['DDDD','DDDC', 'DCDD', 'DCDC', 'DDCD', 'DDCC', 'DCCD',
        'DCCC','CDDD', 'CDDC', 'CCDD','CCDC','CDCD', 'CDCC', 'CCCD','CCCC']
label_y_ax=['DDDD','DDDC', 'DDCD', 'DDCC', 'DCDD', 'DCDC', 'DCCD',
        'DCCC','CDDD', 'CDDC', 'CDCD','CDCC','CCDD', 'CCDC', 'CCCD','CCCC']


plt.xticks(range(16),\
           label_x_ax,rotation=90)
plt.yticks(range(16),\
           label_y_ax)
plt.show()
```