ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

ΤΜΗΜΑ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΕΚΕΦΕ ΔΗΜΟΚΡΙΤΟΣ

ΙΝΣΤ. ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Δι-ιδρυματικό Πρόγραμμα Μεταπτυχιακών Σπουδών στην

«Τεχνητή Νοημοσύνη»

Course

Intelligent Agents and Multiagent Systems

AAMAS Assignment

Fictitious play and reinforcement learning for computing equilibria

Authors

Alexandros Filios - mtn2219

Nikolaos Chiotis - mtn2221

Athens, 2023

Table of Contents

1. Introduction

1.1 Background

Fictitious Play and Reinforcement Learning are two popular and well-studied approaches in game theory and computational intelligence. They are often used to model and understand the behavior of agents in strategic interactions, with the ultimate goal of finding a Nash equilibrium, the state where no player has an incentive to unilaterally change their strategy.

Fictitious Play was first introduced by John von Neumann and Oskar Morgenstern in their book "Theory of Games and Economic Behavior". In this approach, players make repeated play with their opponents, observing their strategies and updating their own strategies accordingly. The name "fictitious play" stems from the assumption that each player believes that the other players will continue to play their current strategies, even though the actual strategies of the opponents may change.

On the other hand, Reinforcement Learning is a subfield of Artificial Intelligence that focuses on the training of agents to optimize their behavior in sequential decision making tasks through the use of rewards. In the context of game theory, reinforcement learning algorithms can be used to model the behavior of agents playing a repeated game, by learning from their experiences with the environment and updating their policies accordingly.

For our implementation, we utilized the normal Fictitious Play algorithm, where each player updates their strategy based on their belief of the current strategies of the opponents, while assuming that the opponents will continue to play those strategies. We also utilized Q-Learning, a popular Reinforcement Learning algorithm, which uses the Q-values, or the expected discounted sum of rewards for each state-action pair, to update the policies of the agents.

In this report, we aim to present an overview of the normal fictitious play and Q-Learning algorithms and demonstrate their application in computing equilibria. Our implementation of these algorithms is not particularly novel, but serves as a basis for comparison between the two methods. Through this study, we aim to gain a deeper understanding of the strengths and limitations of these algorithms in computing equilibria and provide insights for future research in this area.

1.2 Overview

In this report, we aim to present an in-depth examination of the use of Fictitious Play and Q-Learning algorithms for computing equilibria in simple games. The purpose of this study is to compare the results obtained from the application of these algorithms in various simple games, such as Rock-Paper-Scissors, Matching Pennies, etc. Our focus is to examine the strengths and weaknesses of both algorithms in finding the Nash Equilibrium, and to discuss the differences in their convergence rates and accuracy of their results.

We will first provide a thorough explanation of the theoretical background of the algorithms and then present the implementation details and the results obtained. Finally, we will draw a conclusion on the

advantages and limitations of using these algorithms in the computation of equilibria in simple games and make recommendations for future research.

1.3 Motivation

The study of multi-agent systems has garnered significant attention in recent years due to its numerous real-life applications. The interactions between agents in these systems can lead to a variety of outcomes and it is of great interest to understand how equilibria can be computed in such settings. Equilibria are particularly important in the design of stable and efficient systems.

In the field of game theory, the concept of Nash Equilibrium has proven to be a powerful tool for understanding and predicting the behavior of multi-agent systems. Fictitious play and reinforcement learning are two well-known methods for computing Nash Equilibria. By analyzing the results of these algorithms in simple games, we aim to gain a deeper understanding of their properties and limitations.

Furthermore, the knowledge gained from this project can be applied to a range of multi-agent systems, such as airplane navigation systems and other applications in the fields of control, optimization, and coordination. In conclusion, the motivation behind this project is to contribute to the existing body of knowledge on the computation of equilibria in multi-agent systems, and to demonstrate the utility of the fictitious play and reinforcement learning algorithms in this context.

2. Theoretical Framework

2.1. Repeated & zero-sum stochastic games

Zero-sum repeated games refer to a type of repeated game where the total amount of reward or payoffs in each round is constant, and the sum of all rewards across all players is equal to zero. This means that for every win of one player, there must be an equal loss for the other player. Zero-sum repeated games are often used to model situations where there is a conflict of interests between two or more agents.

In such games, it is well known that the Nash Equilibrium strategy, which is a type of stable strategy where no player has an incentive to change their strategy given the strategies of the other players, is also the optimal strategy. This means that if both players play the Nash Equilibrium strategy, neither player can improve their payoff by unilaterally changing their strategy.

The Nash Equilibrium can be found by using different methods such as minimax algorithm, linear programming, and others. In the case of zero-sum repeated games, a specific algorithm called Minimax Q-Learning can be used to find the Nash Equilibrium. Minimax Q-Learning is a variant of the Q-Learning algorithm, which is a popular reinforcement learning algorithm, that takes into account the fact that the game is zero-sum.

2.2. Nash Equilibrium

Nash Equilibrium is a key concept in game theory that was first introduced by Nobel Prize winning economist John Nash. It is a state in which each player in a game has chosen a strategy that is optimal given the strategies of the other players. In other words, no player has an incentive to change their strategy given the strategies of the other players.

The concept of Nash Equilibrium has been widely studied and applied in various fields such as economics, political science, and computer science. In particular, it has proven to be a powerful tool for analyzing multi-agent systems and has been used to study a wide range of problems, including cooperative and non-cooperative games, market competition, and bargaining.

In a game with multiple players, a Nash Equilibrium is a set of strategies such that no player can improve their payoff by unilaterally changing their strategy, assuming that the other players' strategies are unchanged. In other words, a Nash Equilibrium represents a stable state in which no player has any incentive to change their behaviour.

2.3. Fictitious play (FP)

The algorithm starts with an initial strategy for each player and iteratively updates each player's strategy based on the observed strategies of the other players. The update rule for player i is as follows:

$$P(a) = \frac{w(a)}{\sum_{a' \in A} w(a')}$$

where

- $P(a)$ is the probability of an agent taking action a.
- $w(a)$ is the number of occasions action as appeared so far.
- $\sum_{a' \in A} w(a')$ is the sum of all actions, inside the available action vector A, that occurred so far.

The algorithm has been shown to converge to a Nash equilibrium, which is a type of equilibrium strategy where no player can improve their payoff by unilaterally changing their strategy, assuming that the other players strategies are stationary.

In this assignment, we will be using Fictitious play algorithm to train agents to play a repeated game and find equilibrium strategies. The implementation will involve creating simulations of the game and training the P(a) = w(a) ∑a'∈A w(a') P(a) w(a) ∑a'∈A w(a') agents using the Fictitious play algorithm. The agents will be trained to play against each other and learn from the strategies of the other players.

It's worth to mention that there are other variations of the fictitious play algorithm such as fictitious play with adaptive exploration, which uses an adaptive exploration to avoid the problem of getting stuck in a local equilibrium.

2.4. Reinforcement Learning (RL)

Reinforcement learning (RL) is a type of machine learning that focuses on training agents to make decisions by interacting with an environment and receiving rewards or penalties. In the context of games, RL can be used to find equilibrium strategies by training agents to play against each other and learn from the rewards or penalties of the game.

One popular algorithm for RL in games is Q-learning. Q-learning is a model-free algorithm that estimates the value of each action in a given state. The agent starts with an initial estimate of the value of each action and updates it as it experiences new states and rewards. The agent uses the following formula to update its Q-values:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

where:

- $Q(s,a)$ is the estimate of the value of taking action a in state s,
- α is the learning rate,
- r is the reward received after taking action a in state s,
- γ is the discount factor, and s' is the resulting state after taking action a in state s.

In this assignment, we will be using the Q-learning algorithm to train agents to play a repeated game and find equilibrium strategies. The implementation will involve creating a simulation of the game and training the agents using the Q-learning. The agents will be trained to play against each other and

learn from the rewards or penalties of the game. It's worth to mention that these are just a few examples of RL algorithms, there are many others algorithms that could be used such as SARSA, actor-critic, etc.

3. Implementation

In order to understand in depth the theoretical framework of how Fictitious Play and Reinforcement Learning can be used for computing Nash Equilibria in zero-sum stochastic games, we implemented the described algorithms in practice. In this section, it will be presented a description and explaining of the developments, which were done in Python.

We followed a generic pattern of implementations of the algorithms, so they can be used in various zero-sum games, without needing modification. This allowed us to perform multiple experiments that will be presented later in section 4.

For each algorithm, Fictitious Play & Reinforcement Learning, a Player class was defined, which implements the way that an agent should act and learn, based on these theories. Then, three functions that simulate the game process were created. Since the scope is to calculate the Nash Equilibrium of the game, these functions not only simulate the games, but also are checking if the Players-agents have reached a Nash Equilibrium, based on criteria that has been set by the user. Finally, they print the outcome of the game, if an Equilibrium was reached, as well as the number of time and rounds elapsed, together with two plots that display the policy of each user during the rounds.

3.1. Fictitious play (FP)

In order to implement Fictitious Play algorithm, we defined the class **FPZeroSumPlayer()**. Each instance of this class can play as an agent in zero-sum games, based on Fictitious Play algorithm.

Object creation takes as arguments the payoff matrix of the game, that the player is going to play and as an optional argument we can define the initial beliefs of the agent, otherwise the agent will use random beliefs:

```
player_1 = FPZeroSumPlayer(payoff_matrix, w_init=[1,1,3])
```

At the initialization of the object, if the *w_init* is not defined, the agent chooses a random value around 1, in order to define W, which is the count of the opponent's actions. Based on W, P is calculated, which corresponds to the probability of the opponent to play an action. Based on P, a Policy is initialized, by calculating the probabilities of each agent's action to be selected as best response.

After the initialization, the player is ready to play. Each round, we should call **take_action()** function. This function returns the action of the player for the round. This action is defined based on the belief of the agent on what the opponent will play (which is stored in P). The belief is the opponent's actions that has the highest probability to be played. Based on that, agent chooses the action that gives the best reward, if the opponent plays the belief:

```
action_1 = player_1.take_action()
```

When both players play simultaneously, the Player learns based on the opponent's action:

```
player_1.learn(opponent_action=action_2)
```

Function **learn()** take as argument the opponent's action, and uses it based on Fictitious Play algorithm. More specifically, it updates W, by increasing opponent's action count by 1. Then, it

updated P, by calculating the probability of the opponent to perform each action. Also, it calculates the reward based on the payoff matrix. Finally, it updates the policy of the agent.

3.2. Reinforcement Learning (RL)

For Reinforcement Learning implementation, we defined class **RLZeroSumPlayer()**. Each instance of this class can play as an agent in zero-sum games, based on minimax Q-learning algorithm, as described in the below figure:

// Initialize:
forall $s \in S, a \in A,$ *and* $o \in O$ **do**
  $Q(s, a, o) \leftarrow 1$
forall s *in* S **do**
  $V(s) \leftarrow 1$
forall $s \in S$ *and* $a \in A$ **do**
  $\Pi(s, a) \leftarrow 1/|A|$
$\alpha \leftarrow 1.0$
// Take an action:
when in state s, with probability $explor$ choose an action uniformly at random, and with probability $(1 - explor)$ choose action a with probability $\Pi(s, a)$
// Learn:
after receiving reward rew for moving from state s to s' via action a and opponent's action o
$Q(s, a, o) \leftarrow (1 - \alpha) * Q(s, a, o) + \alpha * (rew + \gamma * V(s'))$
$\Pi(s, \cdot) \leftarrow \arg\max_{\Pi'(s,\cdot)}(\min_{o'} \sum_{a'}(\Pi(s, a') * Q(s, a', o')))$
// The above can be done, for example, by linear programming
$V(s) \leftarrow \min_{o'}(\sum_{a'}(\Pi(s, a') * Q(s, a', o')))$
Update α

Figure 1: The minimax-Q algorithm[1]

Object creation takes as arguments the learning parameter ***alpha***, discount factor ***gamma***, parameter ***explor*** that identifies what percentage of round the agent will play randomly and what based on best policy, the set of ***actions*** that agent can play and the initial state ***state_init***. We included state option in the implementation, regardless that in the scope of the present document we will only examine single-state games, so the development can be futureproof in further experiments. RL Player, in contrast with FP Player does not know the pay-off matrix of the game, the rewards are becoming known to them at each round.

```
alpha = 1
gamma = 0.9
explor = 0.2
initial_state = 's'
actions = list(range(len(payoff_matrix)))

player_1 = RLZeroSumPlayer(alpha, gamma, explor, actions, initial_state)
```

At the initialization of the object, dictionaries **Q**, **V** and **Pi** are created. Q represents the Q value of each state - action - opponent action combination, the values are initiated to 1. V represent the minimax value for each state of the game for the agent and is also initiated to 1. Pi is the probability distribution of the actions, based on the type defined in Figure 1. Pi is initiated to 1/|actions|, for each agent's action.

In order to be consistent with the FP Player, RL Player is implemented using **take_action()** and **learn()** functions. Each round, we should call take_action(), that returns the action of the player for the round, this action is selected based on *explor* parameter that is set in the initialization of the Player object.

[1] The minimax-Q algorithm as described in: Shoham, Y., & Leyton-Brown, K. (2008). Multiagent systems: Algorithmic, game-theoretic, and logical foundations. Cambridge University Press.

This parameter corresponds to the percentage of actions that agent selects randomly in order to explore the actions and their rewards and the percentage of actions that agent selects based on the best policy that has formed until each round. So, this function returns a random action with probability *explor* and the action with the highest Pi with probability *1-explor*. To overpass this behavior, we can pass the optional argument ***learning*** in take_action(), that when is False, the agent always returns the best action.

```
action_1 = player_1.take_action()
```

When both players play simultaneously, the Player learns based on the opponent's action:

```
player_1.learn(reward=reward_1, opponent_action=action_2, state='s')
```

Function **learn()** take as argument the reward of the round, opponent's action, and the state of the game and use them based on minimax Q-learning algorithm. It updates the state of the game, and by using three helping functions, update_Q(), update_Pi() and update_V(), it updates the corresponding arrays. In more detail, **update_Q()** starts with calculating the learning parameter *alpha*, based on parameter *kappa* that indicated the times that the action – opponent action pair has been observed. Parameter *alpha* equals to 1/*kappa*. Based on the updated alpha and using the formula that is shown in Figure 1.

```
updated_Q = (1 - alpha)*current_Q + alpha*(reward + self.gamma*self.V[new_state])
```

Function **update_Pi()**, using linear programming finds the Pi values that maximizes the minimax value of the game. For solving the linear programming problem, pulp Python package was used. Finally, **update_V()** function updated the V value based on the formula that described in Figure 1.

3.3. Games Set-up

In order to make experiments more efficient, we defined 3 functions:

play_fp_game(), play_rl_game(), play_fp_rl_game()

Each function is used to simulate a repeated zero-sum stochastic game between two agents, with aim to reach a Nash Equilibrium. The functions are working in the same way, the only difference is that play_fp_game() simulates a game between 2 Fictitious Play Players, play_rl_game() that simulates a game between 2 Reinforcement Learning Players and play_fp_rl_game() that simulates a game between a FP Player and a RL Player.

Since the scope it to find a Nash Equilibrium, we have introduced 3 parameters, that helps to identify if a game has reached a Nash Equilibrium, in order to stop the game:

- **max_iterations**: The maximum numbers of rounds to be played if no Equilibrium is reached.
- **policy_delta_thres**: The threshold of the policy change for each round, in order to assume that the policy is stable for the two players, so we can conclude if we have a Nash Equilibrium.
- **stable_policy_delta_rounds_thres**: The number of rounds with stable policies that we believe is safe to conclude that we have a Nash Equilibrium.

For example, if we set:

max_iterations = 1000, policy_delta_thres = 0.001 and stable_policy_delta_rounds_thres=50

We are declaring that if the maximum change of the policy (for each action) is less than 0.1% and for the last 50 rounds, we can conclude the players have reached a Nash Equilibrium, if not the game will end after 1000 rounds.

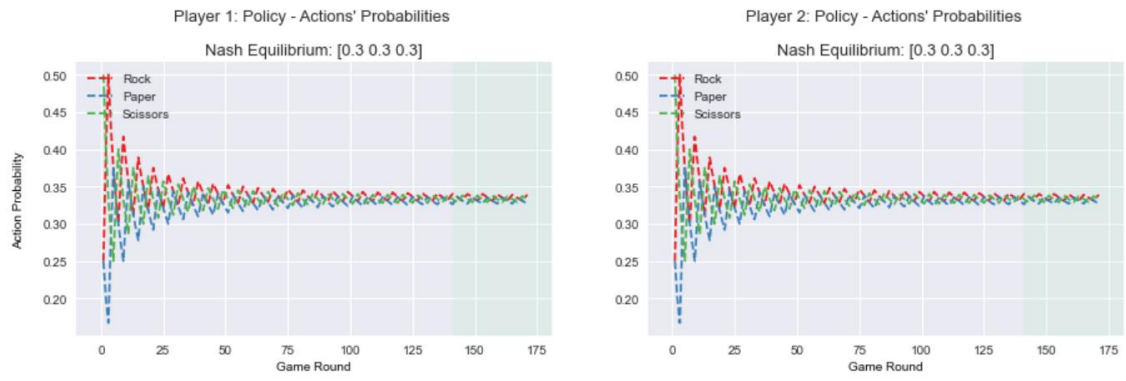
The process that is implemented in the functions starts with initiating a **game history**, which is a DataFrame, that stores for each round of the game, the actions of the players, the rewards, their policies and their Q and Pi values in the case of RL players. For example, for a Matching Pennies game, using a RL Player the game history looks like the following table:

	round	action_1	action_2	reward_1	reward_2	V1	V2	Q1(H,H)	Q1(H,T)	Q1(T,H)	Q1(T,T)	Q2(H,H)	Q2(H,T)	Q2(T,H)	Q2(T,T)	Pi1(H)	Pi1(T)	Pi2(H)	Pi2(T)
0	NaN	NaN	NaN	NaN	NaN	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	0.500000	0.500000	0.500000	0.500000
1	1.0	T	H	-1.0	1.0	1.000000	1.000000	1.000000	1.000000	-0.100000	1.000000	1.000000	1.900000	1.000000	1.000000	1.000000	0.000000	1.000000	0.000000
2	2.0	H	H	1.0	-1.0	1.000000	1.000000	1.900000	1.000000	-0.100000	1.000000	-0.100000	1.900000	1.000000	1.000000	1.000000	0.000000	0.000000	1.000000
3	3.0	H	T	-1.0	1.0	0.609677	1.279310	1.900000	-0.100000	-0.100000	1.000000	-0.100000	1.900000	1.900000	1.000000	0.354839	0.645161	0.310345	0.689655
4	4.0	H	H	1.0	-1.0	0.559793	1.291964	1.548710	-0.100000	-0.100000	1.000000	0.025690	1.900000	1.900000	1.000000	0.400188	0.599812	0.324405	0.675595

Then, it implements the round of the game based on the argument ***max_iterations***, that is given by the user. For each iteration, each player calls the take_action() function and then the learn() function. After this the game history is update. Finally, we check if the conditions for a Nash Equilibrium are met, so the game can be stopped, otherwise we move to the next round.

When the game is finished, a message is printed, informing the user if a Nash Equilibrium has reached:

```
game_log = play_fp_game(player_1, player_2, actions_map, max_iterations, policy_delta_thres, stable_policy_delta_rounds_thres)
Congrats! A Nash Equilibrium was reached!

Player 1 policy: [0.3 0.3 0.3]
Player 2 policy: [0.3 0.3 0.3]

Time elapsed: 9.46s
Number of rounds elapsed: 171
```



4. Experimental results

Based on the above developments we have performed experiments, so we can compare the two algorithms. All the experiments were based on single-state stochastic games, that include Matching Pennies, Rock-Paper-Scissors and another zero-sum game that is designed to have a Pure Strategy Nash Equilibrium. These games were played with 2 players each time, FP vs FP players, RL vs RL players and FP vs RL players. Each time the number of rounds and the time elapsed to reach an equilibrium were documented.

4.1. Matching Pennies

Matching Pennies was the first experiment with the agents that were developed. The game has 2 actions for each player, with the below pay-off matrix:

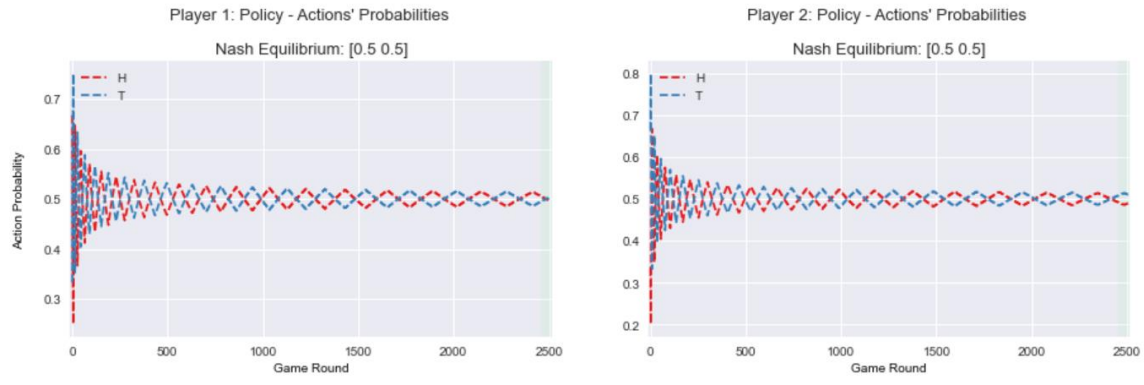
Matching Pennies	**Heads**	**Tails**
Heads	1,-1	-1,1
Tails	-1,1	1,-1

The criteria that we set in order to examine if we can reach a Nash Equilibrium, was that the agents should maintain a stable policy for 50 continues rounds, with a 1% max change of policy during these rounds.

4.1.1. Results with Fictitious Play

```
game_log = play_fp_game(player_1, player_2, actions_map, max_iterations, policy_delta_thres, stable_policy_delta_rounds_thres)
```

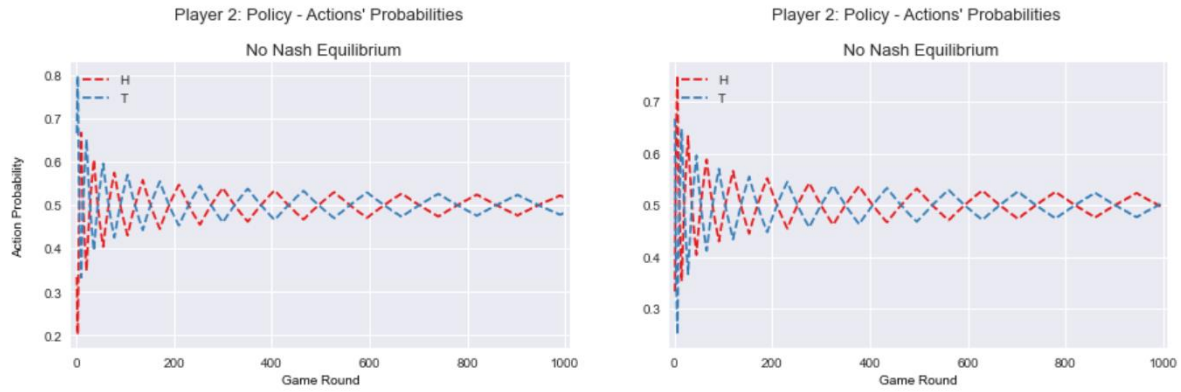
```
Congrats! A Nash Equilibrium was reached!

Player 1 policy: [0.5 0.5]
Player 2 policy: [0.5 0.5]

Time elapsed: 231.41s
Number of rounds elapsed: 2506
```



A Nash Equilibrium was reached after 2506 rounds and 3.85 minutes.

However, when still see that the probabilities of each action still have some difference, so for example if we try to make the criteria stricter, and have a maximum of policy change of 0.0001 for 50 rounds, we do not get an Equilibrium:

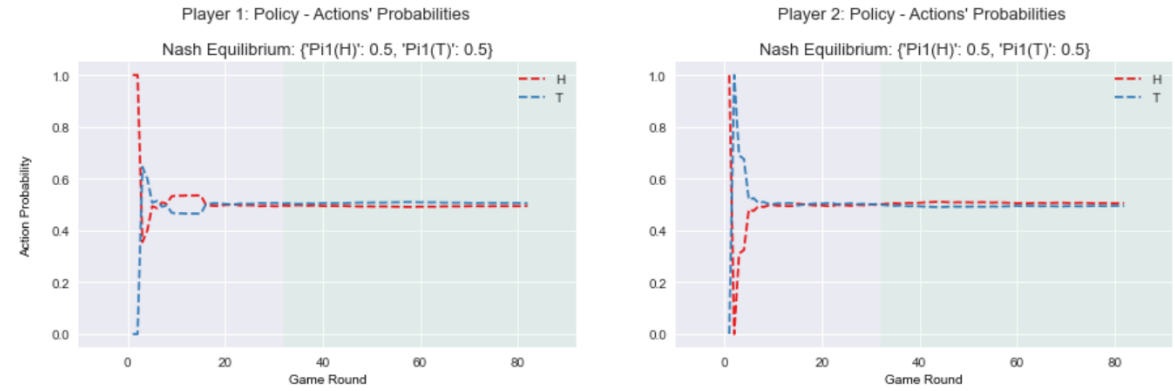
```
No Nash Equilibrium was reached based on the below defined criteria. 😢
Policy delta threshold: 0.0001
Number of rounds with stable policy needed: 50
Max rounds: 1000
```



4.1.2. Results with Reinforcement Learning

```
game_log = play_rl_game(player_1, player_2, payoff_matrix, actions_map, max_iterations, policy_delta_thres, stable_policy_delta_r
```

```
Congrats! A Nash Equilibrium was reached! 🎉

Player 1 policy: {'Pi1(H)': 0.5, 'Pi1(T)': 0.5}
Player 2 policy: {'Pi1(H)': 0.5, 'Pi1(T)': 0.5}

Time elapsed: 16.99s
Number of rounds elapsed: 82
```



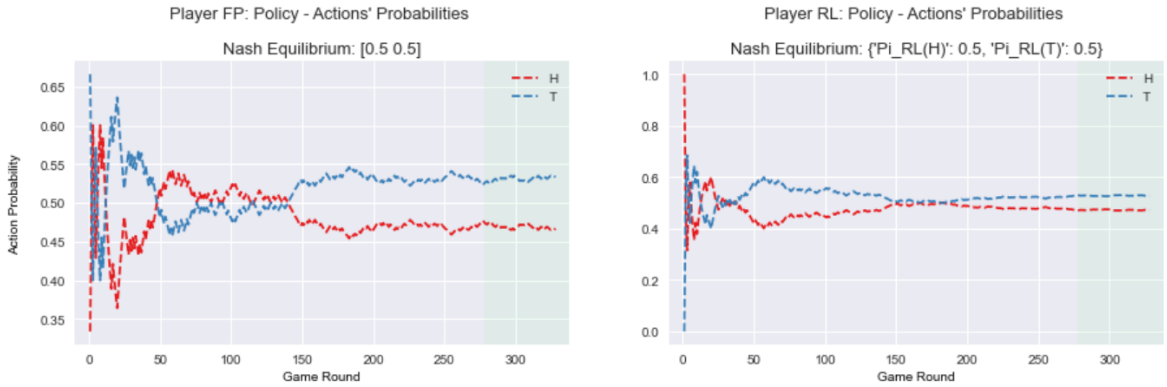
A Nash Equilibrium was reached after 82 rounds and 17 seconds.

Based on the above experiments' results, we can see that Reinforcement Learning can reach a Nash Equilibrium quicker, and the policies seem to merge to equilibrium much earlier than Fictitious Play.

4.1.3. Results with Fictitious Play vs Reinforcement Learning

```
game_log = play_fp_rl_game(player_FP, player_RL, payoff_matrix, actions_map, max_iterations, policy_delta_thres, stable_policy_de
```

```
Congrats! A Nash Equilibrium was reached!

Player FP policy: [0.5 0.5]
Player RL policy: {'Pi_RL(H)': 0.5, 'Pi_RL(T)': 0.5}

Time elapsed: 74.05s
Number of rounds elapsed: 328
```

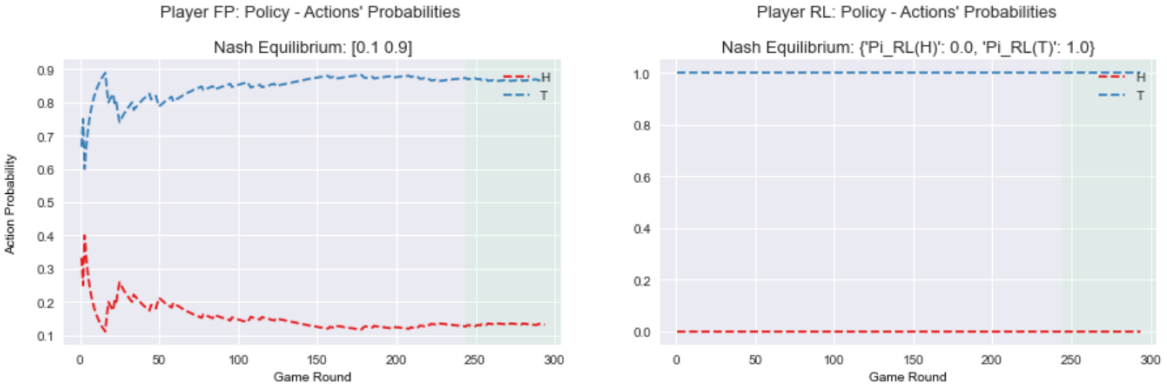


Based on the criteria we set, a stable policy was reached, but was not the standard Nash Equilibrium that we got as result in the other methods.

Also, the results seem sensitive to randomness, since when we tried to repeat the experiment, the results were different:

```
game_log = play_fp_rl_game(player_FP, player_RL, payoff_matrix, actions_map, max_iterations, policy_delta_thres, stable_policy_de
```

```
Congrats! A Nash Equilibrium was reached!

Player FP policy: [0.1 0.9]
Player RL policy: {'Pi_RL(H)': 0.0, 'Pi_RL(T)': 1.0}

Time elapsed: 64.23s
Number of rounds elapsed: 294
```



4.2. Rock-Paper-Scissors

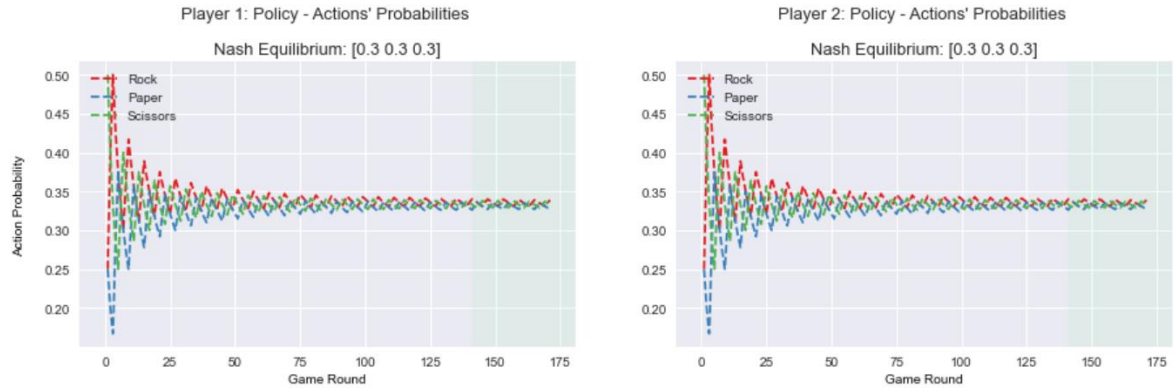
Rock-Paper-Scissors is a game with 3 actions for each player, with the below pay-off matrix:

Rock Paper Scissors	Rock	Paper	Scissors
Rock	0	-1,1	1,-1
Paper	1,-1	0	-1,1
Scissors	-1,1	1,-1	0

The criteria that we set in order to examine if we can reach a Nash Equilibrium, was that the agents should maintain a stable policy for 50 continues rounds, with a 1% max change of policy during these rounds.

4.2.1. Results with Fictitious Play

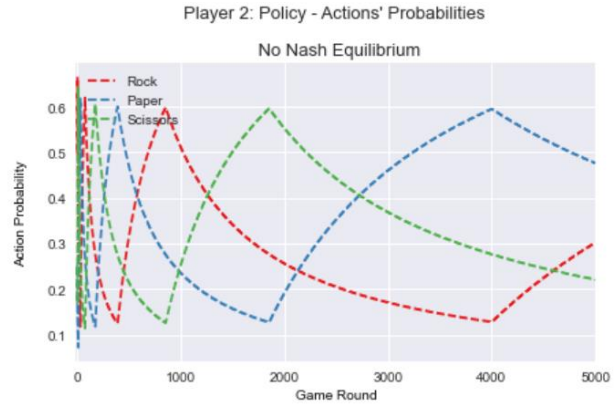
```
game_log = play_fp_game(player_1, player_2, actions_map, max_iterations, policy_delta_thres, stable_policy_delta_rounds_thres)
Congrats! A Nash Equilibrium was reached! 🎉

Player 1 policy: [0.3 0.3 0.3]
Player 2 policy: [0.3 0.3 0.3]

Time elapsed: 9.46s
Number of rounds elapsed: 171
```

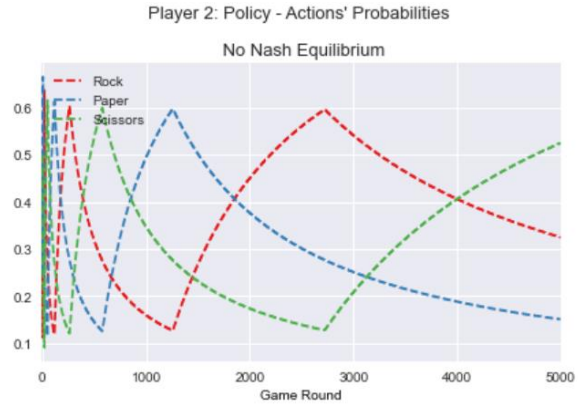


A Nash Equilibrium was reached after 171 rounds and 9.5 seconds.

One thing that we noticed is that Fictitious Play is sensitive to the initial beliefs that are set to the agents, for example, when we changed slightly the initial beliefs, the results were the following:

```
No Nash Equilibrium was reached based on the below defined criteria. 😢
Policy delta threshold: 0.01
Number of rounds with stable policy needed: 150
Max rounds: 5000
```

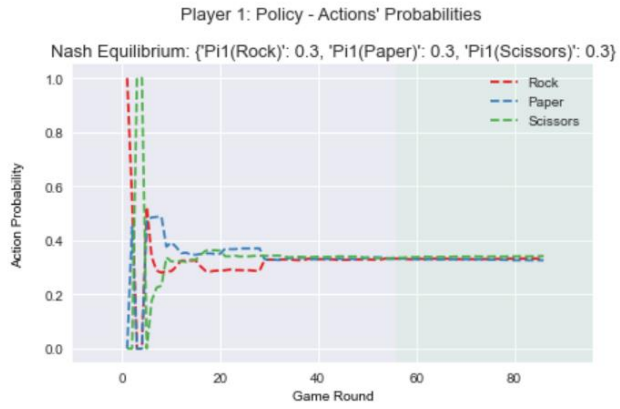


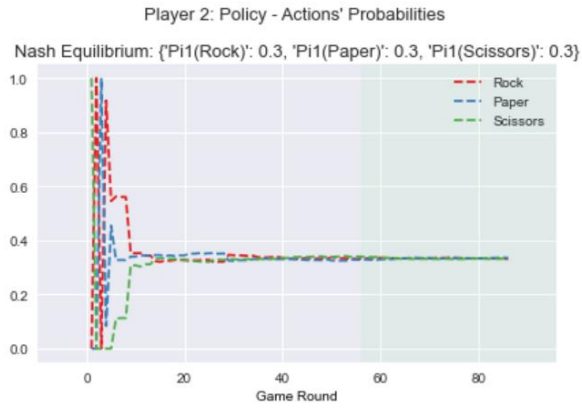


Although the probabilities after 5000 rounds are very slowly getting closer, the process seems to need a lot of time to converge to a Nash Equilibrium.

4.2.2. Results with Reinforcement Learning

```
Congrats! A Nash Equilibrium was reached! 🎉

Player 1 policy: {'Pi1(Rock)': 0.3, 'Pi1(Paper)': 0.3, 'Pi1(Scissors)': 0.3}
Player 2 policy: {'Pi1(Rock)': 0.3, 'Pi1(Paper)': 0.3, 'Pi1(Scissors)': 0.3}

Time elapsed: 17.07s
Number of rounds elapsed: 86
```





A Nash Equilibrium was reached after 86 rounds and 17 seconds.

Based on the above experiments' results, we can see that Reinforcement Learning can reach a Nash Equilibrium in less rounds, however each round seems to be more time consuming, on the other hand, Fictitious Play needs less time per round, due to its simple process. So, in cases that both methods can find the Nash Equilibrium in few rounds, Fictitious Play may be faster, like this example.

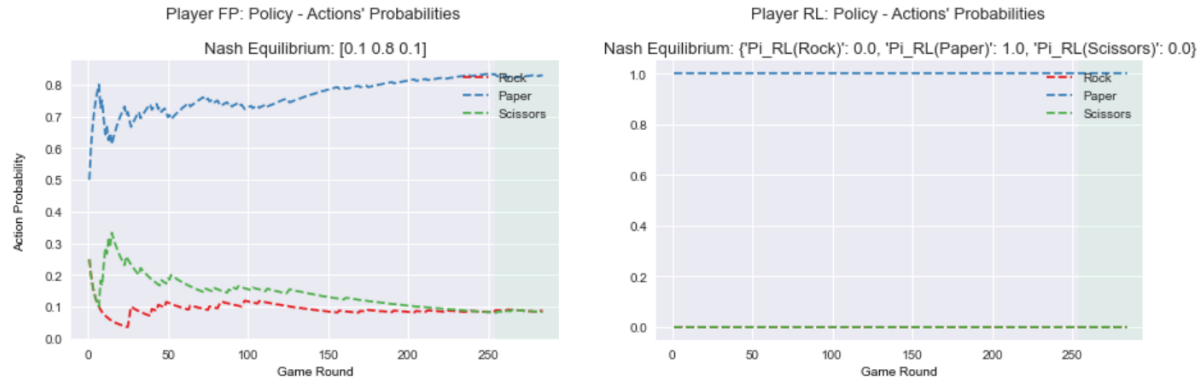
In any case, Reinforcement Learning seems again to have a really faster, in term of rounds, convergence to Nash Equilibrium.

Also, Reinforcement Learning is more robust to the initialization process.

4.2.3. Results with Fictitious Play vs Reinforcement Learning

```
game_log = play_fp_rl_game(player_FP, player_RL, payoff_matrix, actions_map, max_iterations, policy_delta_thres, stable_policy_de
```

```
Congrats! A Nash Equilibrium was reached!

Player FP policy: [0.1 0.1 0.8]
Player RL policy: {'Pi_RL(Rock)': 0.0, 'Pi_RL(Paper)': 1.0, 'Pi_RL(Scissors)': 0.0}

Time elapsed: 57.33s
Number of rounds elapsed: 284
```



Again, based on the criteria we set, a stable policy was reached, but was not the standard Nash Equilibrium that we got as result in the other methods.

4.3. Pure Equilibrium Game

Since the other games, concluded in a mixed strategy Nash Equilibrium, we wanted to test a game that was designed to have a pure Nash Equilibrium has 3 actions for each players, with the below pay-off matrix:

Pure Equilibrium Game	Action 1	Action 2	Action 3
Action 1	2,-2	0,0	1,-1
Action 2	-4,4	-3,3	2,-2
Action 3	1,-1	-2,2	-2,2

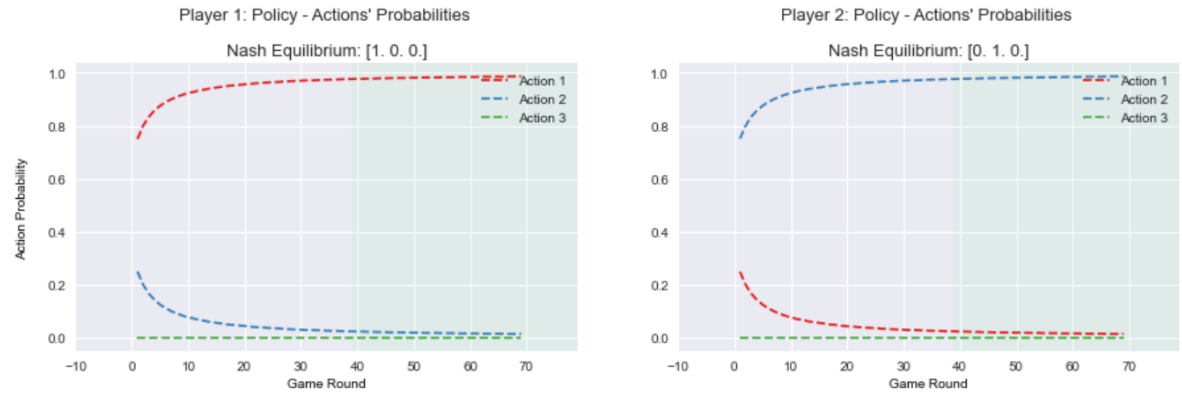
The pure equilibrium of the game is the set of actions Action 1 for row player and Action 2 for column player.

The criteria that we set in order to examine if we can reach a Nash Equilibrium, was that the agents should maintain a stable policy for 50 continues rounds, with a 1% max change of policy during these rounds.

4.3.1. Results with Fictitious Play

```
game_log = play_fp_game(player_1, player_2, actions_map, max_iterations, policy_delta_thres, stable_policy_delta_rounds_thres)
```

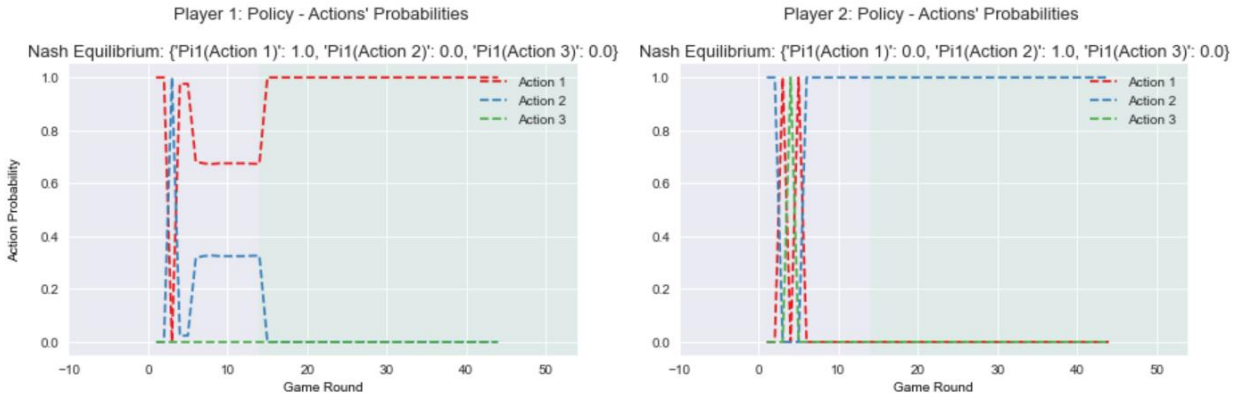
```
Congrats! A Nash Equilibrium was reached!

Player 1 policy: [1. 0. 0.]
Player 2 policy: [0. 1. 0.]

Time elapsed: 4.21s
Number of rounds elapsed: 69
```



A Nash Equilibrium was reached after 69 rounds and 4.2 seconds.

4.3.2. Results with Reinforcement Learning

```
Congrats! A Nash Equilibrium was reached!

Player 1 policy: {'Pi1(Action 1)': 1.0, 'Pi1(Action 2)': 0.0, 'Pi1(Action 3)': 0.0}
Player 2 policy: {'Pi1(Action 1)': 0.0, 'Pi1(Action 2)': 1.0, 'Pi1(Action 3)': 0.0}

Time elapsed: 9.66s
Number of rounds elapsed: 44
```

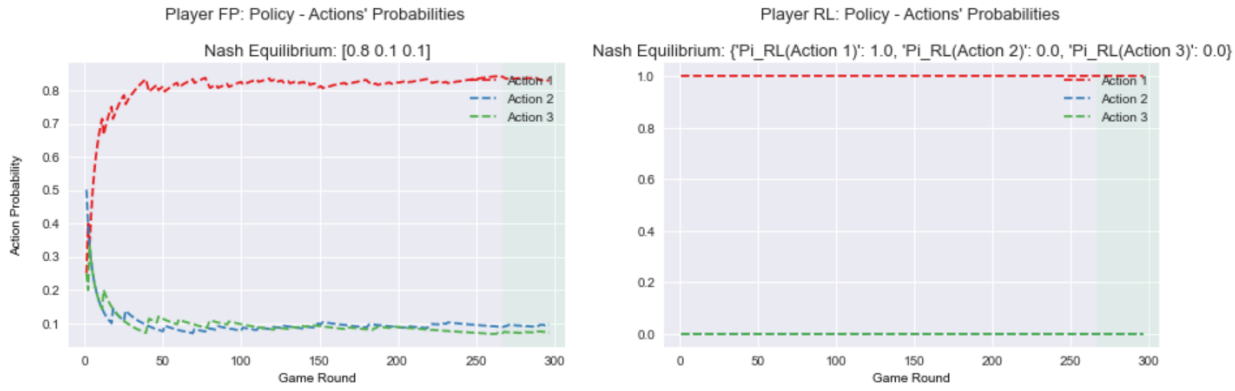


A Nash Equilibrium was reached after 44 rounds and 9.6 seconds.

Again, Reinforcement Learning reached a Nash equilibrium in less rounds and with moved earlier close to the equilibrium than Fictitious Play. However, due to the computation time caused by the solving of the linear programming at each round, Fictitious Play was faster in terms of time.

4.3.3. Results with Fictitious Play vs Reinforcement Learning

```
Player FP policy: [0.9 0.1 0. ]
Player RL policy: {'Pi_RL(Action 1)': 1.0, 'Pi_RL(Action 2)': 0.0, 'Pi_RL(Action 3)': 0.0}

Time elapsed: 54.10s
Number of rounds elapsed: 297
```



By using one FP and one RL player, no equilibria was calculated in this game. Based on the initial beliefs and initializations, the two algorithms may not converge to a Nash Equilibria when are combined.

5.Conclusion

In conclusion, the project aimed to explore the applications of Fictitious Play and Q-Learning algorithms in Multi-Agent Systems, specifically in repeated games. Through simulation and implementation, we were able to observe the performance of each algorithm and gain insight into their strengths and weaknesses. Our findings demonstrate the potential of using these algorithms to train agents to play against each other and find Nash Equilibria in repeated games. It is worth noting that our implementation was just one possible application of these algorithms and there is still much room for further exploration and improvement. The algorithms themselves are not without limitations and further research is needed to address these limitations and find new ways to apply them in real-world scenarios. In summary, this project provides a starting point for future studies on the applications of Fictitious Play and Q-Learning in Multi-Agent Systems. The knowledge gained through this project is useful in various fields, including gaming and the development of multi-agent systems for real-world applications such as air traffic control, network security, and economic modelling.