

Multi-agent learning programming assignment, study document

Declaration of authenticity

1. I declare that I have knowledge of the Education and Exam regulations of the Utrecht Graduate School of Natural Sciences, and the relevant master-specific annex.
2. I declare that this assignment is my own work and demonstrates my own abilities and knowledge and does not involve plagiarism or teamwork other than that authorised within the assessment task for this unit.
3. I have taken proper and reasonable care to prevent this work from being copied by another student.
4. I declare that I have not contracted another person to do the work for me or allowed another person to edit and substantially change my work.
5. So that the assessor can properly assess my work, I give this person permission to act according to the Utrecht OER policy and practice to reproduce this work and provide a copy to another member of staff for the purpose of cross checking and moderation and to take steps to authenticate its originality.
6. I declare that the signature below is my own, and is the signature as it appears on my passport and/or driving license and/or identity card.

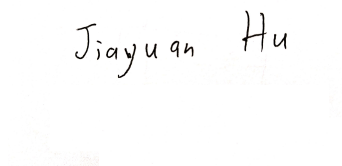
Name: Jiayuan Hu

Student number: 6876587

Signature:

Place: Utrecht, Netherlands

Date: June 26, 2020

A handwritten signature in black ink that reads "Jiayuan Hu". The signature is written in a cursive style with some capital letters.

Name: Jacqueline Isabel Wagner

Student number: 6670555

Signature:

Place: Utrecht, Netherlands

Date: June 26, 2020

A handwritten signature in blue ink that reads "J. Wagner". The signature is written in a cursive style with a large initial 'J'.

Overview

This study document supplements the programming assignment of the multi-agent learning course 2020. It is organized as follows. Section 1 introduces the occasion of this assignment, its context and its purpose. It also briefly mentions our specific take on this assignment, i.e., where we think that our elaboration of the assignment is special and may deviate favorably from other elaborations. Section 2 introduces concepts, notation and terminology that is necessary to understand this document. Section 3 introduces the learning algorithms that are going to be compared, and defines them as precisely as possible. We also motivate in this section why it is important that especially these 8 algorithms are compared. Section 4 introduces our algorithms, named *SmartBully*, *SmartGreedy* and *SmartHybrid*. Section 5 explains how the algorithms selected are going to be compared, and how results are going to be processed and interpreted. Section 5.1 explains the concept of the grand table. Section 5.2 explains comparison by evolutionary means, in particular the replicator dynamic [Hofbauer et al., 1998, Weibull, 1997]. Section 5.3 explains how learning algorithms may be compared by computing Nash equilibria of the grand table. Section 6 discusses our coding approach and specific design decisions. Section 7 summarizes our results. Section 8 discusses our results. We discuss the algorithm we proposed. In particular, we discuss how it compares to other algorithms and why it performs the way it does. We also briefly compare our results with [Airiau et al., 2007, Bouzy and Métivier, 2010, Zawadzki et al., 2014]. Section 9 summarizes our work and highlights its features. Section A is an appendix with a reference to our code with instructions on how run it.

Contents

1	Introduction	3
2	Concepts and terminology	3
3	Algorithms	4
4	Our Algorithms	5
5	Methods	6
5.1	Introducing the Grand Table	6
5.2	Introducing the Replicator Dynamic	7
5.3	Introducing the Nash Equilibria of the Grand Table	7
6	Code	7
7	Results	9
7.1	Results Obtained Using the Grand Table	9
7.2	The replicator dynamic	10
7.3	Nash Equilibria in the Grand Table	11
8	Discussion	12
9	Conclusion	14
A	Code	16
B	Additional Results	17
B.1	Grand Tables	17
B.2	Replicator Dynamics	17
B.3	Nash Equilibria	18

1 Introduction

With the number of applications requiring *multiple agents* to communicate and cooperate to serve a common goal rapidly growing, the field of *multi-agent learning* has established its position as “the most fertile grounds for interaction between game theory and artificial intelligence” [Shoham et al., 2007]. While traditional single-agent AI considers only one agent in each environment, *multi-agent systems* consist of autonomous agents interacting with each other in shared environments. Hereby, an agent’s actions influence the actions of all other agents in the same environment, which, in turn, influence the agent’s actions. While hand-crafting agents’ behaviours in advance may work for certain simple applications, in general, this approach is too brittle and simply not sufficient, especially as the number of agents in an environment and the space of possible actions increases. Instead, agents must continuously reevaluate their behaviour and learn from their environment, enabling the multi-agent system as a whole to gradually improve.

Action selection, the task of an agent deciding which action to take next in a multi-agent system, lies at the heart of multi-agent learning. Although selecting an action from a pre-defined, finite set of actions may seem trivial on the surface, dynamic and unpredictable environments, as well as real-time demands make this task a highly complex one. Faced with computational limitations, agents must constrain the process of action selection as they can not consider all possible outcomes before selecting an action. Here, multi-agent learning has brought forward state-of-the-art algorithms facilitating adaptive and efficient learning and action selection in multi-agent systems by combining approaches from both machine learning and traditional game theory.

As applications have significantly increased in complexity, multi-agent learning approaches are progressively proving themselves indispensable for various problems, from robotics and data mining to decision support systems and resource management. With topics such as autonomous driving at the forefront of the recent *AI craze*, the need for adaptive learning-based solutions is greater than ever. Amidst this growing interest in multi-agent learning, the amount of available literature is exploding. From the 1951 Fictitious Play algorithm [Brown, 1951] to novel Deep Reinforcement Learning algorithms [Foerster et al., 2016] [Sukhbaatar et al., 2016], the choices to consider when designing multi-agent systems are ample. However, while Deep Reinforcement Learning algorithms are high in demand, understanding, implementing and running these algorithms can be challenging and costly. Moreover, the effectiveness of simpler algorithms is often overlooked. Thus, this paper provides an extensive overview and comparison of common, easy-to-use baseline multi-agent learning algorithms. Additionally, we will propose three novel adaptive learning-based algorithms, coined *SmartBully*, *SmartGreedy* and *SmartHybrid*, and will subsequently compare their performance to the baseline algorithms by letting all algorithms compete against each other. *SmartBully*, *SmartGreedy* and *SmartHybrid* form extensions to the baseline algorithms by uncovering and bridging shortcomings. Unlike other approaches, *SmartBully*, *SmartGreedy* and *SmartHybrid* use only existing concepts and, thus, maintain simplicity and low complexity while, nonetheless, continuously outperforming their baselines. Moreover, we evaluate our algorithms using statistical functions beyond assessing only *mean performance*. As a result, we provide a more comprehensive analysis and are able to give more robust recommendations for future multi-agent systems.

2 Concepts and terminology

To evaluate each strategy $s_i \in \{s_1, \dots, s_z\}$, we will consider 2-agent matrix games. A matrix-game is a game in which each agent, the *row-player* and the *column-player*, has a finite number of actions to choose from. In the following, we will refer to the row-player as the *player*, and the column-player as the *opponent*. Each matrix-game will be played for $n \geq$ rounds. The set of actions available to the player are given by

$$A = \{a_1, \dots, a_p\} \quad (1)$$

whereby the opponent’s actions are denoted by

$$A^{(opp)} = \{a_1^{(opp)}, \dots, a_q^{(opp)}\}. \quad (2)$$

The entry $\pi_{ik}^{(j)}$ in the resulting $(p \times q)$ *payoff-matrix* Π details the player’s payoff for selecting action $a_i \in A$ if the opponent selected action $a_k^{(opp)} \in A^{(opp)}$ in round $j \leq n$. Moreover, we will evaluate all strategies using multiple payoff-matrices $\Pi_l \in \Pi = \{\Pi_1, \dots, \Pi_r\}$ whereby Π is commonly called a *matrix suite*.

3 Algorithms

Amidst thousands of articles and multiple books published on the topic of multi-agent learning, selecting the optimal learning algorithm for an application is challenging. While learning algorithms can differ on many features, the degree to which an algorithm *exploits* and *explores* is commonly used as a comparison. In the following, we will introduce 8 multi-agent learning algorithms, each employing a different strategy for balancing the trade-off between *exploitation* and *exploration*, which we will then quantitatively compare in [section 7](#).

A-Select The perhaps most simple of all learning-based algorithms is termed A-Select. This *exploration-only* algorithm entirely forgoes *exploitation* by randomly selecting an action $a_i \in A$ at each round $j \leq n$.

Epsilon-Greedy A simple learning approach considering both *exploration* and *exploitation* is the Epsilon-Greedy algorithm. By selecting a random action $a_i \in A$ ϵ -percent of the time, epsilon-greedy behaves similarly to A-Select. However, $(100 - \epsilon)$ -percent of the time, Epsilon-Greedy *exploits* whichever action $a_j \in A$ has been the most successful to date. Hereby, Epsilon-Greedy keeps track of the average payoffs earned by each action $a_i \in A$ and simply *exploits* the action yielding the highest average payoff. For our implementation, we selected the commonly-used value $\epsilon = 10$.

Satisficing Play A further algorithm weighing both *exploration* and *exploitation* is the Satisficing Play algorithm. This algorithm employs *aspirations*, which are updated in each round $j \leq n$ after playing an action $a_i \in A$ using

$$\alpha^{j+1} = \lambda \cdot \alpha^j + (1 - \lambda) \cdot \pi_i^j. \quad (3)$$

If the payoff achieved by playing action $a_i \in A$ at $j \leq n$ surpasses the aspiration α^j , a_i is played again in round $j + 1$, otherwise the algorithm selects a random action from the space of possible actions. In our implementation, α^1 is initialized as the average payoff for a randomly selected action and λ is set to 0.1.

Softmax While the ϵ -Greedy and Satisficing Play algorithms consider both *exploitation* and *exploration*, one major downside remains. As the algorithm selects all strategies with equal probability during *exploration*, it is equally as likely to select the worst-performing action as it is to select the best-performing action. An algorithm that considers past payoffs during *exploration* is the Softmax algorithm. Although the Softmax algorithm assigns the highest probability to the best-performing action as is the case in the ϵ -Greedy algorithm, other actions are also selected with a probability according to their past payoffs. The algorithm employs so-called *qualities* Q_i for each action $a_i \in A$ to keep track of each actions past performance. Q_i is initialized as the average payoff of action $a_i \in A$ and is henceforth updated in each round $j \leq n$ using

$$Q_i^{(j)} = Q_i^{(j-1)} + \lambda \cdot (\pi_i^{(j)} - Q_i^{(j-1)}) \quad (4)$$

with the learning rate λ . Finally, the softmax selection rule gives each action's probability of being selected as

$$p_i = \frac{e^{Q_i/\tau}}{\sum_{j=1}^n e^{Q_j/\tau}}. \quad (5)$$

Hereby, the *temperature* τ indicates the extent to which past payoffs influence an actions selection probability. For $\tau \rightarrow 0$, the Softmax algorithm approaches the ϵ -Greedy algorithm with $\epsilon = 0$. For our implementation, we selected $\lambda = 0.1$ and $\tau = 1.0$.

Proportional Regret Matching Similar to the Softmax algorithm, the Proportional Regret Matching, or short PRM, algorithm uses the history of play to influence an action's selection probability. However, while the Softmax algorithm lets each action's past payoff determine its selection probability, PRM uses each action's *regret*. Hereby, the regret of an action $a_k \in A$ is defined as the payoff a player could have earned by selecting a_k given the opponent's response when they selected a different action $a_i \in A, i \neq k$ instead. In each round $j \leq n$, the PRM algorithm then chooses an action $a_i \in A$ proportional to its regret $r_i^{(j-1)}$ and then updates the regret $r_k^{(j-1)}$ of all other actions $a_k \in A, k \neq i$ using

$$r_k^{(j)} = \begin{cases} r_k^{(j)} & \text{if } \pi_k^{(j)} < \pi_i^{(j)} \\ \pi_k^{(j)} - \pi_i^{(j)} & \text{else} \end{cases} \quad (6)$$

whereby $\pi_i^{(j)}$ denotes the payoff received in round j after playing action a_i and $\pi_k^{(j)}$ denotes the payoff the player could have received in round j if they had played a_k instead. In our implementation, the regret of each action $a_i \in A$ is initialized using the average payoff of a_i .

Upper confidence bound Often described as “optimism in the face of uncertainty”, the Upper Confidence Bound, or short UCB, algorithm uses optimistic guesses regarding each action’s payoffs to select the next action. The algorithm keeps track of the average payoff of each action $a_i \in A$ in round $j \leq n$ and *exploits* the action which yields the highest expected payoff p_i according to

$$p_i = \mu_i + \sqrt{\frac{2 \cdot \log(j)}{n_i}} \quad (7)$$

whereby μ_i refers to the action a_i ’s average past payoff and n_i denotes the number of rounds action a_i was selected in. In our implementation, μ_i is initialized using

$$\mu_i^{(0)} = \frac{\max(\pi(a_i)) + \min(\pi(a_i))}{2} \quad (8)$$

As is evident by Equation 7, if an action yields lower payoffs than expected, the algorithm’s optimistic guess will decrease, causing the likelihood of this action being selected to decrease and the algorithm to *explore* other actions. However, if an action performs well over time, the algorithm will *exploit* it, incurring a symbiosis between *exploitation* and *exploration*.

Fictitious Play Fictitious Play “combines two elements that are common to many models of economic decision making: a prediction of the future based on past observations, and an optimal response given the prediction” [Young, 2004]. At the heart of Fictitious Play lie so-called *beliefs*, which document for each action $a_i \in A$ the cumulative payoffs a player could have received given the opponent’s past behaviour. In each round $j \leq n$, the algorithm *exploits* the action $a_i \in A$ with the highest belief $b_i^{(j)}$, the action which would have led to the highest cumulative payoff over the history of play. Therefore, the algorithm’s predictions of the future rest on the assumption that opponents are playing stationary strategies that are independent and consistent with the history of play, allowing the algorithm to select the optimal action in response. Once again, our implementation uses the average payoff of an action $a_i \in A$ to initialize $b_i^{(0)}$.

Bully Lastly, the Bully algorithm forms the counterpart to the *non-exploitative* A-Select algorithm introduced above. This *non-exploratory* algorithm is stateless and selects the same action in each round of play. Hereby, the algorithm determines for each possible action $a_i \in A$ which action $a_k^{(opp)} \in A^{(opp)}$ the opponent would play in response to maximize their expected payoff. The algorithm then decides which action $a_i \in A$ yields the highest expected payoff in return to the opponent’s hypothesized behaviour, leaving only a single best-response action to be *exploited* in each round.

4 Our Algorithms

This paper proposes *three extensions* to existing strategies, as well as a novel *hybrid* approach. The proposed strategies coined *SmartBully*, *SmartGreedy* and *SmartHybrid* systematically uncover and bridge shortcomings in their respective baseline strategies.

SmartGreedy While ϵ -Greedy is a simple, yet effective action selection algorithm balancing both *exploration* and *exploitation*, one key vulnerability remains and is easily capitalized by an opponent following a different strategy. During *exploitation*, ϵ -Greedy selects the action $a_i \in A$ which has resulted in the highest average payoff to date given by

$$\text{average_payoff}(a_i) = \frac{\sum_{k=1}^{n_k} \pi_i^{(k)}}{n_i} \quad (9)$$

whereby n_i denotes the number of rounds action a_i was selected in. From Equation 9 it follows that ϵ -Greedy is slow to react to sudden change in an opponent’s strategy as it utilizes the entire history of play to obtain the average payoff of each action. As a result, if an action has been optimal thus far, ϵ -Greedy will continue to select it even after a shift in an

opponent's strategy has rendered it sub-optimal, causing the player to lose out. This effect is especially pronounced if the gap between the former best action's average payoff and the new best action's average payoff is significant. An opponent who is aware of this flaw in the ϵ -Greedy strategy can thus *exploit* strategic, frequent and sudden strategy changes.

Our *SmartGreedy* strategy proposes a fast and cost-effective solution to this problem. Instead of using the highest average payoff to date, SmartGreedy considers only the past η rounds. Thus, SmartGreedy selects the action $a_i \in A$ which maximizes

$$\text{average_payoff}(a_i) = \frac{\sum_{k=(n_k-\eta)}^{n_k} \pi_i^{(k)}}{n_i}. \quad (10)$$

In our implementation, we set the hyper-parameter η to 20 while maintaining $\epsilon = 10$ for comparison.

SmartBully Similar to ϵ -Greedy, Bully's key weakness is easily exploitable by opponents playing a different strategy. Bully begins by evaluating for each action $a \in A$ which action $a^{(opp)} \in A^{(opp)}$ the opponent would play as a best-response. Using the set of best-response actions $\{a_h^{(opp)}, \dots, a_k^{(opp)}\} \in A^{(opp)}$, Bully selects the action $a_i \in A$ which maximizes

$$\frac{\sum_{l=h}^k \pi_{il}}{k}. \quad (11)$$

However, as Bully *always* selects the best overall action in response to the opponent's hypothesized actions, the opponent can easily deviate by selecting actions outside of the anticipated behaviour. Instead of selecting one action that maximizes the expected payoff over a set of best-response actions $\{a_h^{(opp)}, \dots, a_k^{(opp)}\} \in A^{(opp)}$, SmartBully determines for each action $a^{(opp)} \in A^{(opp)}$ which action $a_i \in A$ maximizes the expected payoff π_{il} resulting in a set of optimal actions $A^{(SmartBully)}$. Moreover, SmartBully records the average payoff received by playing each action in $A^{(SmartBully)}$. In each round, SmartBully then selects an action from $A^{(SmartBully)}$ which has been optimal thus far using the average payoff.

SmartHybrid Although the two previously introduced strategies differ in many ways, their commonality is their weakness: Each strategy may perform well against different opponents, results, however, are not necessarily consistent. While one strategy may yield far above average payoffs playing against an opponent using a particular strategy, the same strategy might perform slightly above average against other opponents. Therefore, while some strategies result in a high *mean* performance, they also display a high *standard deviation*, making the results unreliable and highly dependent on the opponent's strategy. To bridge this gap, our algorithm *SmartHybrid* combines both the *SmartGreedy* and *SmartBully* strategies to a more robust alternative. For each of the two strategies, SmartHybrid thus keeps track of the average past payoff received. In each round, SmartHybrid then lets the strategy, which has resulted in the highest average payoff, decide which action to take.

5 Methods

We will compare all prescribed and strategies using three different approaches, namely the *grand table*, *replicator dynamic*, and *Nash equilibria*. The grand table mainly focuses on comparing the average payoffs for each strategy, among other statistical metrics. The replicator dynamic, on the other hand, takes an evolutionary approach that gives proportions to strategies and tracks the changes of these proportions. The Nash equilibria approach considers the grand table as a game between a row and a column player with the goal of finding the optimal action, in this case, the optimal strategy.

5.1 Introducing the Grand Table

The grand table's idea is to compare the strategies by making them play against each other and tracking the average payoff for each strategy. Assuming there are z different strategies to be compared. Each cell $g_{i,j}$ in the underlying grand table, a $n \times n$ matrix, contains the payoff that the *row player* received by playing strategy s_i against the column player playing strategy s_j averaged over n rounds. A further column represents the mean payoffs μ achieved by the row player playing strategy s_i against all possible strategies s_j . Additionally, we added two extra columns to calculate the

standard deviation σ and the *median* performance $\bar{\mu}$ of a strategy's average payoff. Using the standard deviation, we can observe whether a strategy displays consistent high results against all other strategies, or whether a strategy performs well only against a select few strategies while performing sub-par against others. Moreover, as the average payoffs received by a strategy against all other strategies may not stem from a normal distribution, using the median payoff, which is less affected by outliers and skewed data, may yield a more accurate comparison across strategies.

As the resulting mean, standard deviation, and median values are subject to randomness, we will compare the strategies in different environments, in this case, by *restarting* the grand table for each payoff matrix in the matrix suite. For r payoff matrices, the grand table will thus need $r - 1$ restarts. After each restart, each strategy pair will play n rounds. Moreover, we will use three unique matrix suites Π , specifically the *fixed* Π_{fixed} , *random int* Π_{int} and *random float* Π_{float} matrix suites. While Π_{fixed} contains only square matrices, the number of rows and columns in Π_{int} and Π_{float} are from the interval $(1, 5]$ and not necessarily equal. Moreover, the Π_{fixed} and Π_{int} matrices contain integer payoff values from $(0, 3]$, whereas the Π_{float} matrices contain float values from $(0.0, 3.0]$.

We provide a list of all parameters used in our implementation in [Table 1](#).

Matrix Suite	Strategies	Restarts ($r - 1$)	Rounds per Restart (n)
Fixed	All	9	1000
Random Int	All	19	1000
Random Float	All	19	1000

Tab. 1: grand table parameter settings

5.2 Introducing the Replicator Dynamic

The replicator dynamic describes an evolutionary approach to measure the strength of each strategy. Hereby, each strategy s_i in a population of strategies $S = \{s_1, \dots, s_z\}$ is assigned a proportion which evolves over time with regards to the strength of s_i . Hereby, a weaker strategy, meaning a strategy with weaker performance, will gradually lose its share while stronger ones will gain share. These changes are observable by visualizing the *proportions history*.

Initially, the strategies are given some proportions $P = (p_1, \dots, p_z)$, $0 \leq p_i \leq 1$ and $\sum_i p_i = 1$, equally or not equally. Then using row i of the grand table M , the expected score of a strategy score(s_i) is given by

$$\text{score}(s_i) = M_i p_i. \quad (12)$$

The proportion p_i of strategy s_i is updated using the Hadamard product of s_i and p_i to yield

$$q_i = p_i \circ s_i \quad (13)$$

which is subsequently normalized using

$$p_i^{(new)} = \frac{q_i}{\sum_{j=1, \dots, z} q_j}. \quad (14)$$

The proportions P are updated iteratively until they are stable. In our implementation, we perform the replicator dynamic on all three matrix suites, with and without our own strategy, using uniform and non-uniform starting proportions. We detail these combinations in [Table 2](#).

5.3 Introducing the Nash Equilibria of the Grand Table

Lastly, finding the Nash equilibria of a grand table is also a useful method for comparing strategies. The idea behind this is that the grand table can be viewed as a payoff matrix whereby the space of possible actions for each player corresponds to the strategies in the grand table. Therefore, the optimal action profile for a player represents the best strategies that the player can select. In our implementation, we calculate the Nash Equilibria using the grand tables for each of the three matrix suites, with and without our strategies.

6 Code

As the program is designed using several classes, we will provide an overview of how we implemented each class and explain important decisions made during the coding process. For our project, we followed the *ground-up approach* by

Matrix Suite	Own Strategies?	Uniform?	starting properties
Fixed	Yes	Yes	$\frac{1}{9}$ for every strategy
Fixed	Yes	No	[0.12, 0.08, 0.06, 0.15, 0.05, 0.21, 0.06, 0.09, 0.18]
Fixed	No	Yes	$\frac{1}{8}$ for every strategy
Fixed	No	No	[0.22, 0.19, 0.04, 0.06, 0.13, 0.10, 0.05, 0.21]
Random Int	Yes	Yes	$\frac{1}{9}$ for every strategy
Random Int	Yes	No	[0.12, 0.08, 0.06, 0.15, 0.05, 0.21, 0.06, 0.09, 0.18]
Random Int	No	Yes	$\frac{1}{8}$ for every strategy
Random Int	No	No	[0.22, 0.19, 0.04, 0.06, 0.13, 0.10, 0.05, 0.21]
Random Float	Yes	Yes	$\frac{1}{9}$ for every strategy
Random Float	Yes	No	[0.12, 0.08, 0.06, 0.15, 0.05, 0.21, 0.06, 0.09, 0.18]
Random Float	No	Yes	$\frac{1}{8}$ for every strategy
Random Float	No	No	[0.22, 0.19, 0.04, 0.06, 0.13, 0.10, 0.05, 0.21]

Tab. 2: combinations for the replicator dynamic

first implementing the strategies and matrix suites, before designing the game and grand table and, lastly, implementing the replicator dynamic and Nash equilibria.

MatrixSuite We adopted a straightforward approach for implementing the matrix suites. When a new payoff matrix is needed, the instance updates its payoff matrix by generating random integer values for `RandomIntMatrixSuite` or random float values for `RandomFloatMatrixSuite` within a pre-defined range using the random Python-package.

Strategies As suggested, we implemented each Strategy as the sub-class of the meta-class Strategy, each using the methods `initialize`, `get_action` and `update`. For strategies which require access to the payoff matrices, as well as the payoff histories, we converted both to `numpy` arrays, to enable the use of simple built-in functions, such as `numpy.argmax`, `numpy.sum` or `numpy.reshape`.

Game To play a game using two competing strategies for n rounds, the method `play` of the class `Game`, calls the method `play_single_round` n -times. Hereby, `play_single_round` simply calls the `get_action` and `update` methods of both strategies before returning the row player's payoff which is then averaged over all rounds in play.

GrandTable For the `GrandTable`, we added the method `execute` to play all games, meaning letting all strategies compete against each other, for n rounds each, using all payoff matrices in a single matrix suite. For each payoff matrix in the matrix suite, the method `execute` calls the method `execute_one_matrix` which lets all strategies compete against each other on a single payoff matrix by calling the method `play` of the `Game` class. Hereby, `execute_one_matrix` receives the average payoff for the row player playing against a single strategy from `play` and combines these into a `numpy` array which it returns to `execute`. The method `execute` gathers these average payoffs received by the row player for each strategy and payoff matrix in a `numpy` array, which is finally used to construct the grand table.

ReplicatorDynamic In our implementation, the replicator dynamic automatically executes the grand table before calculating and updating the proportions until the change in the last 20 rounds is less than 0.01. In then end, `matplotlib` visualizes the proportions history using random colours for each strategy.

Nash `Nash_equilibria` simply calls `run_gambit` using a set of strategies and the corresponding grand table without the statistical metrics.

7 Results

7.1 Results Obtained Using the Grand Table

Using the grand table approach introduced in [subsection 5.1](#), we compared the performance of all strategies discussed in [section 3](#) and [section 4](#). The mean and median performance of each strategy, as well as the standard deviation, are listed in [Table 3](#) and [Table 4](#) for the fixed and RandomInt matrix suites respectively. The results obtained for the RandomFloat matrix suite, which are very similar to those of the RandomInt matrix suite, are provided in [subsection B.1](#).

Strategy	Mean	Standard Deviation	Median
SmartHybrid	4.99	0.26	4.91
SmartBully	4.70	0.45	4.60
Bully	4.89	0.37	4.98
SmartGreedy	4.86	0.26	4.81
ϵ -Greedy	4.64	0.37	4.61
ASelect	3.39	0.37	3.25
UCB	4.90	0.31	4.78
Satisficing	4.39	0.36	4.50
Softmax	4.97	0.30	4.91
Fictitious	4.65	0.37	4.51
PRM	4.93	0.26	5.00

Tab. 3: results obtained using all strategies and the fixed matrix suite

Strategy	Mean	Standard Deviation	Median
SmartHybrid	2.58	0.14	2.64
SmartBully	2.56	0.16	2.62
Bully	2.56	0.13	2.61
SmartGreedy	2.58	0.08	2.60
ϵ -Greedy	2.55	0.11	2.59
ASelect	2.02	0.05	2.03
UCB	2.64	0.11	2.68
Satisficing	2.42	0.15	2.44
Softmax	2.33	0.10	2.35
Fictitious	2.48	0.06	2.49
PRM	2.62	0.11	2.66

Tab. 4: results obtained using all strategies and the RandomInt matrix suite

As is evident from [Table 3](#), our algorithm *SmartHybrid* considerably outperforms all other strategies in the mean performance on the fixed matrix suite with the exception of the *Softmax* strategy which is only slightly inferior. Moreover, our *SmartGreedy* strategy trumps the performance of ϵ -Greedy by a large margin. While ϵ -Greedy is one of the worst-performing strategies, *SmartGreedy* makes it into the top 5. Additionally, both algorithms display significantly lower standard deviations than their baseline counterparts, while *SmartHybrid* achieves the lowest standard deviation overall. Further, well-performing algorithms are *PRM* and *UCB*, which also display a low standard deviation. While *Bully* also achieves high mean scores, it has the second-highest overall standard deviation, on par with *ASelect*, which performs the worst out of all strategies on mean performance. Moreover, our algorithm *SmartBully* performs significantly worse than *Bully*. Notably, most strategies' median performance is significantly lower than the mean performance, with the exception of *PRM*, which improves slightly on median performance.

From [Table 4](#), it follows that *UCB* is the best performing algorithm on the RandomInt matrix suites in both mean and median performance, tightly followed by *PRM* and *SmartBully*, which all three maintain very low standard deviations. Interestingly, on this matrix suite, the mean performance of *Bully* is on par with that of *SmartBully*, with a slightly higher standard deviation for the latter. Once again, *SmartGreedy* outperforms ϵ -Greedy on both mean performance

and standard deviation. Although *ASelect* exhibited a much smaller standard deviation, the mean performance was, once again, the lowest overall. Notably, on the `randomInt` matrix suite, all algorithms displayed a higher median than mean performance.

7.2 The replicator dynamic

As the starting proportions of replicator dynamic are pre-defined, we could only choose one of our strategies to include in the comparison. Given the reliably high mean and median performance of *SmartHybrid*, as well as the generally very low standard deviation, we selected *SmartHybrid* to use in the experiments involving the replicator dynamic.

Figure 1 and Figure 2 detail the evolution of the proportions of each strategy using all strategies on the `fixed` matrix suite with non uniform and uniform starting properties, respectively. It is clear from both Figure 1 and Figure 2 that *PRM* outperforms all other strategies. However, using the non uniform starting properties requires more iterations, whereby the proportions of *Softmax* and *UCB* are initially higher.

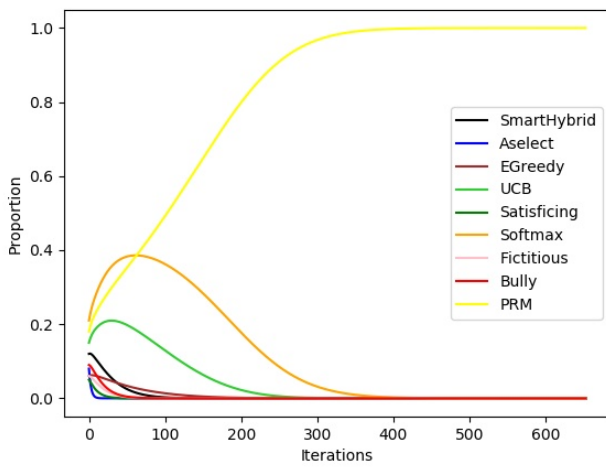


Fig. 1: replicator dynamic using non uniform starting values and the `fixed` matrix suite with our own strategies

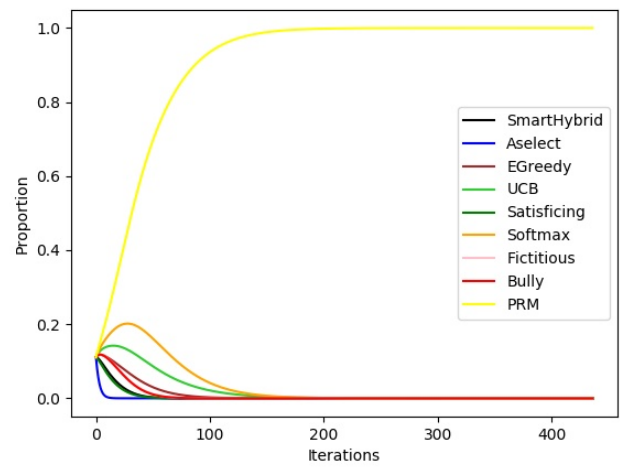


Fig. 2: replicator dynamic using uniform starting values and the `fixed` matrix suite with our own strategies

Figure 3 and Figure 4 visualize the evolution of the proportions of each strategy using all strategies on the `randomInt` matrix suite with non uniform and uniform starting properties, respectively. Figure 3 and Figure 4 show that *UCB* outperforms all other strategies. Once again, however, using the non uniform starting properties requires more iterations, whereby the proportions of *SmartHybrid* are initially higher.

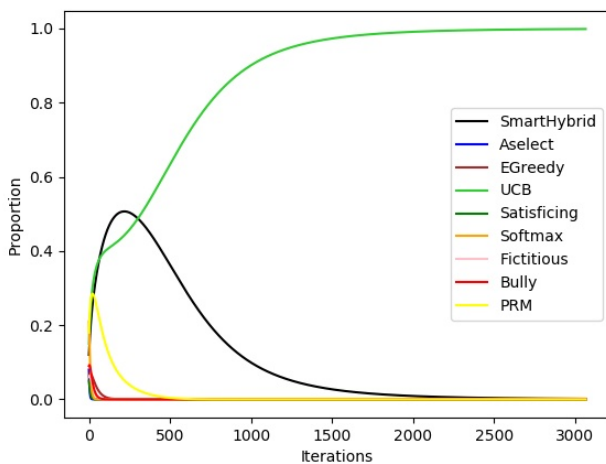


Fig. 3: replicator dynamic using non uniform starting values and the `randomInt` matrix suite with our own strategies

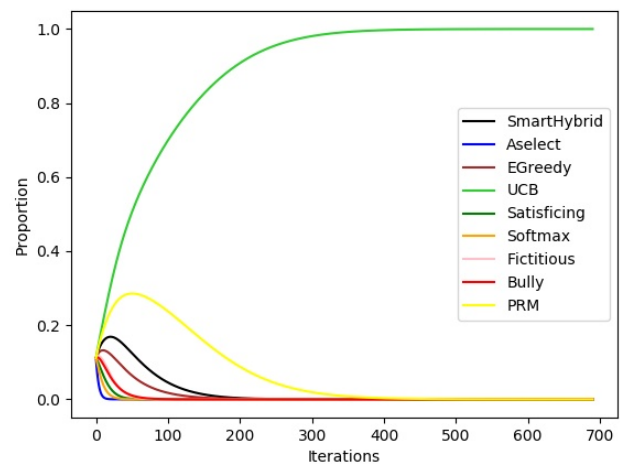


Fig. 4: replicator dynamic using uniform starting values and the `randomInt` matrix suite with our own strategies

Lastly, [Figure 5](#) and [Figure 6](#) detail the evolution of the proportions of each strategy using all strategies except our own on the randomInt matrix suite with non uniform and uniform starting properties, respectively. While *PRM* takes the lead in [Figure 3](#), when using the non-uniform starting properties, the proportions of *UCB* and *PRM* oscillate with *UCB* taking the overall lead.

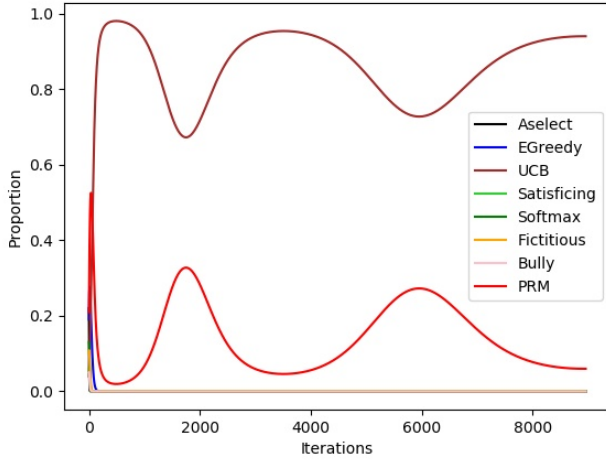


Fig. 5: replicator dynamic using non uniform starting values and the randomInt matrix suite without our own strategy

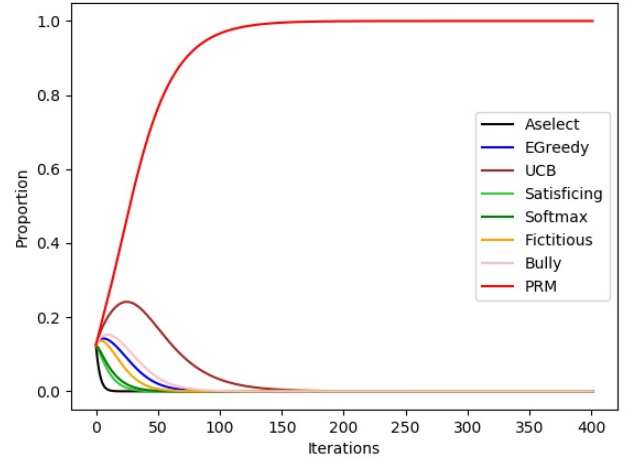


Fig. 6: replicator dynamic using uniform starting values and the randomInt matrix suite without our own strategy

We detailed all results for the 12 different settings outlined in [subsection 5.2](#) in [subsection B.1](#).

7.3 Nash Equilibria in the Grand Table

Using the grand table for the fixed matrix suite on all strategies described in [Table 3](#), we obtained *four* Nash Equilibria, as is evident from [Table 5](#).

Nash Equilibria	proportions row player	proportions col player
1	Smart Hybrid 0.46 Bully 0.19 PRM 0.35	Smart Hybrid 0.46 Bully 0.19 PRM 0.35
2	Bully 0.14 UCB 0.74 PRM 0.12	Bully 0.14 UCB 0.74 PRM 0.12
3	Smart Hybrid 0.36 Bully 0.22 UCB 0.24 PRM 0.18	Smart Hybrid 0.36 Bully 0.22 UCB 0.24 PRM 0.18
4	UCB 1.00	UCB 1.00

Tab. 5: Nash Equilibria obtained using all strategies and the fixed matrix suite

Moreover, for the RandomInt matrix described in [Table 4](#), we obtained *one* Nash Equilibria when using all strategies, namely (*UCB*, *UCB*). This result is highlighted in [Table 6](#).

Nash Equilibria	proportions row player	proportions col player
1	UCB 1.00	UCB 1.00

Tab. 6: Nash Equilibria obtained using all strategies and the randomInt matrix suite

We included all remaining results in [subsection B.3](#).

8 Discussion

In the preceding section, we compared the results of all three of our algorithms *SmartHybrid*, *SmartGreedy* and *SmartBully* with eight popular multi-agent learning algorithms. We compared these algorithms using both the mean and median performance, as well as the standard deviation from the mean. While an overall high mean performance is indicative of a well-performing algorithm, we motivate the necessity of other statistical methods, namely the standard deviation and median performance. Strategies might perform extremely well against select strategies while performing significantly worse against others. However, an algorithm that only performs well against a few other algorithms is not the overall best algorithm. Instead, an algorithm with a slightly lower mean performance but a significantly lower standard deviation would be a better choice if the opponent's strategy profile is unclear, as it's high performance is consistent. Further, an algorithm that displays a significantly lower median performance than mean performance has major outliers in the low end, which is also not desirable. Thus, an algorithm with a similar or higher median than mean performance is more desirable.

From the results obtained using the grand table approach, it is clear that the consistently best-performing algorithms are *SmartHybrid*, *PRM* and *UCB*, with the latter taking the lead. All three algorithms consistently outperform the other algorithms on both mean performance, as well as the standard deviation. However, we note that *PRM* performed slightly better than the other two algorithms in most experiments, and, importantly, *always* had a higher median performance than mean performance, while maintaining a very low standard deviation. Next, our *SmartGreedy* approach trumped the traditional ϵ -Greedy strategy on all matrix suites, while simultaneously reducing the standard deviation. Thus, it appears as though our hypothesis that reducing the history of play available to the ϵ -Greedy strategy significantly increases reactivity, is valid. Unfortunately, our *SmartBully* algorithm did not fare so well, as it was not able to improve the traditional *Bully* strategy. Overall, our grand table experiments showed that *Satisficing*, *ASelect*, the traditional ϵ -Greedy and *Fictitious Play* are consistently outperformed. Moreover, during our experiments, we observed high variability among the top three best-performing algorithms, which motivates the need for the replicator dynamic approach. Additionally, some algorithms performed noticeably worse on the randomInt and randomFloat than on the fixed matrix suites, highlighting the need for diverse payoff matrices in comparing multi-agent learning algorithms.

The results from the replicator dynamic underline our grand table results, once again putting forward *SmartHybrid*, *UCB* and *PRM* as top-performing strategies, with *PRM* winning on the fixed matrix suite and *UCB* winning on the randomInt matrix suite for both starting properties. Similarly, the results of the Nash equilibria include *SmartHybrid*, *UCB* and *PRM*, with *UCB* slightly in the lead. Although the replicator dynamic mostly results in one *best-choice* strategy, uncovering the Nash equilibria in the grand table yields multiple strategies. However, one important distinction must be made: the Nash equilibria are obtained using the grand table, which corresponds to one iteration for the replicator dynamic. While a strategy may appear like a good choice using the grand table of Nash equilibria approach, the replicator dynamic can be used to make a definitive decision. Thus, to truly decide if an algorithm is better than others, employing multiple iterations as is the case in the replicator dynamic appears superior.

Comparison with Work of Airiau et al.

In their 2007 paper *Evolutionary Tournament-based Comparison of Learning and Non-learning Strategies for Iterated Games* [Airiau et al., 2007], Airiau et al. describe a tournament-based approach for comparing multi-agent learning strategies. They compare 8 strategies by letting them compete against each other in 57 2×2 matrix games over 1000 rounds. However, similar to our paper, they conclude that “a one-shot tournament may not be representative of the strength of a strategy” [Airiau et al., 2007]. As a result, they propose a head-to-head, as well as an evolutionary approach.

In their head-to-head approach, Airiau et al. compare the average payoff of each strategy over the course of 100 tournaments. Hereby, they assign a fixed strategy to each player while assigning each strategy to 100 players. In their evolutionary approach, on the other hand, Airiau et al. study the evolution of a large population, which initially contains all strategies with an equal proportion, specifically 100 players per strategy. At the end of each iteration, they select the proportion with which each strategy will contribute to the next iteration. For this, Airiau et al. use three methods: the pure and modified tournament selection mechanism and the fitness proportionate selection mechanism. While the pure tournament selection mechanism selects strategies from a uniform distribution, the modified

tournament selection mechanism selects strategies proportionate to their score. The more intricate fitness proportionate selection mechanism selects strategies according to

$$p_i = \frac{\frac{1-\delta}{\delta_{max}}}{2}. \quad (15)$$

While the head-to-head approach proposed by Airiau et al. is very similar to our grand table approach, some differences must be noted. Although, Airiau et al. use significantly more payoff matrices, namely 57, our paper includes the randomInt and randomFloat payoff matrices *in addition* to 2×2 payoff matrices. In their paper, Airiau et al. acknowledge that “it would be interesting to see if these results generalize to large samples of randomly generated $n \times n$ games with cardinal payoff”. From our results, it is clear that certain algorithms may perform well on square payoff matrices, while performing noticeably worse on the randomInt and randomFloat matrices. As a result, the results detailed by Airiau et al. may not generalize and are thus not suitable for drawing conclusions.

Further, the evolutionary approach using the modified tournament selection mechanism discussed in [Airiau et al., 2007] is very similar to our replicator dynamic. However, Airiau et al.’s approach selects strategies with a probability proportionate to the cumulative results, while we select strategies with a probability proportionate to the mean result.

Although the more intricate fitness proportionate selection mechanism requires far more iterations to converge, it is clear from the paper that the oscillations observed on the pure and modified tournament selection mechanism, as well as our replicator dynamic, are absent. As a result, the proportions may not converge when using larger numbers of evolutions and, thus, the proposed fitness proportionate selection mechanism may be a better choice.

Lastly, as Airiau et al. selected a different group of multi-agent learning algorithms, our results are not directly comparable. However, their paper includes numbers on *ASelect* and *Fictitious Play*, which are not high-performing in both Airiau et al.’s and our experiments. Moreover, similar to our paper, their statistical evaluation also included the standard deviation from the mean performance.

Comparison with work of Bouzy et al.

In their 2010 paper *Multi-agent Learning Experiments on Repeated Matrix Games* [Bouzy and Métivier, 2010] Bouzy et al. empirically compared several multi-agent learning algorithms. Some of the learning algorithms they included are also evaluated in this paper, namely *UCB* and *Bully*. They used a two-player game in which each player aims to maximize their cumulative payoffs whereby each player has three actions yielding payoffs within $[-9, 9]$. Bouzy et al. randomly draw 100 matrix games whereby each pair of players plays 3,000,000 rounds for each game.

Using their tournament framework, the first results they discovered are the performance ranking evolution according to number of rounds played. The *FP*, *JR*, and *HMC* take first three ranks at the beginning, i.e. at 100 rounds. And *UCB*, *M3*, and *Sat* are ranked first at the end. Then they took an novel approach called *eliminate by rank*. They first studied whether an algorithm’s performance is influenced by the presence or absence of other given algorithm. The corresponding ranking is more stable than the normal ranking mechanism as it remains approximately unchanged after 30,000 rounds. Based on these result, they introduce an *elimination mechanism* which removes the worst player either if they are significantly worse than the second-worst player or if the ranking has been stable over a prolonged period of time. The authors also conducted experiments in cooperative and competitive games to see whether the different classes of games would effect the players’ performance. For instance, the players are compared in only cooperative games, only competitive games, games with different number of actions, and specific matrix games. Similar to our paper, they found out that *UCB*, among others, was the best-performing. Moreover, they conclude that adding fixed-width windows and state features to a multi-agent learning algorithm will significantly improve its rank.

Compared with our three approaches, the *grand table*, *replicator dynamic*, and *Nash equilibria*, the first evaluation method of this paper can be considered a *dynamic grand table*. It traces the ranking of each algorithm in each round of the game. This idea is also practical for building an algorithm which selects the best strategy at each stage of a game, e.g. choosing *JR* at the beginning and *UCB* at the end, similar to our *SmartHybrid* algorithm. Overall, a more powerful algorithm can be built by aggregating existing strategies.

Comparison with work of Zawadzki *et al.*

In their 2008 paper, Zawadzki *et al.* introduced the novel MultiAgent Learning Testbed, coined *MALT*, for comparing multi-agent learning algorithms which they subsequently used to run an experiment of “unprecedented in size” [Zawadzki *et al.*, 2014]. Similar to our paper, Zawadzki *et al.* conclude that considering only the mean payoff for each strategy is not sufficient. Instead, they devised a criterion titled *probabilistic domination*, which is stronger, as a strategy that probabilistically dominates another strategy also has a higher mean payoff. Hereby, a strategy probabilistically dominates another strategy if its *cumulative distribution function* (CDF) is below the other strategy’s CDF along the entire curve.

By comparing the average payoffs of each strategy before determining which strategy is probabilistically dominated by other strategies, Zawadzki *et al.* showed that the mean payoff is simply a piece of a larger puzzle. Finally, their paper determines *Q*-Learning to be the only algorithm which is not probabilistically dominated by other strategies, despite other strategies initially yielding promising results when comparing their mean performance. Unfortunately, their paper includes only two algorithms, which we compared in this paper, namely *ASelect* and *Fictitious Play*, which both did not yield high payoffs in our results. Nonetheless, this paper motivates the need for more extensive statistical methods in comparing multi-agent algorithms.

9 Conclusion

In this paper, we proposed three novel multi-agent learning algorithms, coined *SmartHybrid*, *SmartGreedy* and *SmartBully* and subsequently conducted experiments to compare the performances between these proposed algorithms and eight other popular multi-agent learning algorithms. Our experiments uncovered that *UCB* and *PRM* as the best-performing algorithms, with our *SmartHybrid* just slightly behind. Moreover, our paper motivates the necessity of considering various statistical methods beyond just the mean performance, as well as multiple matrix suites, including non-quadratic matrix games. Lastly, our paper shows that to truly compare a set of algorithms, one must consider a multitude of iterations.

In the future, further experiments could include using different strategies in the *SmartHybrid* algorithm to achieve an overall higher result. As *SmartHybrid* alternates between *SmartGreedy* and *SmartBully*, which both performed significantly worse than *UCB* and *PRM*, and yet managed to fall just short of the latter two. It is, therefore, conceivable that other strategies may lead to better mean performance. Considering that our *SmartHybrid* algorithm is implemented to consider an arbitrary number of strategies, we view our *SmartHybrid* algorithm as an important contribution.

References

- [Airiau *et al.*, 2007] Airiau, S., Saha, S., and Sen, S. (2007). Evolutionary tournament-based comparison of learning and non-learning algorithms for iterated games. *Journal of Artificial Societies and Social Simulation*, 10(3):7.
- [Bouzy and Métivier, 2010] Bouzy, B. and Métivier, M. (2010). Multi-agent learning experiments on repeated matrix games. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 119–126.
- [Brown, 1951] Brown, G. W. (1951). Iterative solution of games by fictitious play. *Activity analysis of production and allocation*, 13(1):374–376.
- [Foerster *et al.*, 2016] Foerster, J., Assael, I. A., De Freitas, N., and Whiteson, S. (2016). Learning to communicate with deep multi-agent reinforcement learning. In *Advances in neural information processing systems*, pages 2137–2145.
- [Hofbauer *et al.*, 1998] Hofbauer, J., Sigmund, K., *et al.* (1998). *Evolutionary games and population dynamics*. Cambridge university press.
- [Shoham *et al.*, 2007] Shoham, Y., Powers, R., and Grenager, T. (2007). If multi-agent learning is the answer, what is the question? *Artificial intelligence*, 171(7):365–377.
- [Sukhbaatar *et al.*, 2016] Sukhbaatar, S., Fergus, R., *et al.* (2016). Learning multiagent communication with backpropagation. In *Advances in neural information processing systems*, pages 2244–2252.

[Weibull, 1997] Weibull, J. W. (1997). *Evolutionary game theory*. MIT press.

[Young, 2004] Young, H. P. (2004). *Strategic learning and its limits*. OUP Oxford.

[Zawadzki et al., 2014] Zawadzki, E., Lipson, A., and Leyton-Brown, K. (2014). Empirically evaluating multiagent learning algorithms. *arXiv preprint arXiv:1401.8074*.

A Code

Note: please use `python -m pip install -U matplotlib` and `python -m pip install -U numpy` if you have not installed matplotlib and numpy yet.

How to use our code The example code are all included in `Main.py`. To check all the results of this paper, simply run `Main.py`. We will also give examples separately below.

To obtain a grand table described in [subsection 5.1](#):

```
grand_table = GrandTable(matrix_suite, strategies, k - 1, N)
grand_table.execute()
print(grand_table)
```

The grand table will be displayed on the terminal. `matrix_suite` is any instance of `FixedMatrixSuite`, `RandomIntMatrixSuite`, or `RandomFloatMatrixSuite` class. `strategies` is a list containing strategies. `k - 1` is the restart times. `N` is the number of rounds for each game.

To use replicator dynamic and draw a graph of proportions:

```
replicator_dynamic = ReplicatorDynamic(proportions, grand_table)
replicator_dynamic.run()
```

This will save the figure of the replicator dynamic result in `img/` folder. `proportions` is a list containing the probability of each strategy, which has to be the same length with the strategies of `grand_table`. We have included 4 proportions as examples in `Main.py`. `grand_table` is any instance of `GrandTable`.

To check the Nash equilibria of a grand table:

```
Nash.nash_equilibria(grand_table)
```

The Nash equilibria results will be displayed on the terminal. `grand_table` is any instance of `GrandTable`.

B Additional Results

B.1 Grand Tables

Strategy	Mean	Standard Deviation	Median
SmartHybrid	1.96	0.08	1.96
SmartBully	1.98	0.17	2.02
Bully	2.01	0.11	2.02
SmartGreedy	2.05	0.12	2.11
ϵ -Greedy	2.00	0.09	2.01
ASelect	1.42	0.06	1.43
UCB	2.09	0.13	2.09
Satisficing	1.78	0.15	1.78
Softmax	1.78	0.12	1.83
Fictitious	1.91	0.06	1.93
PRM	2.07	0.13	2.07

Tab. 7: results obtained using all strategies and the RandomFloat matrix suite

B.2 Replicator Dynamics

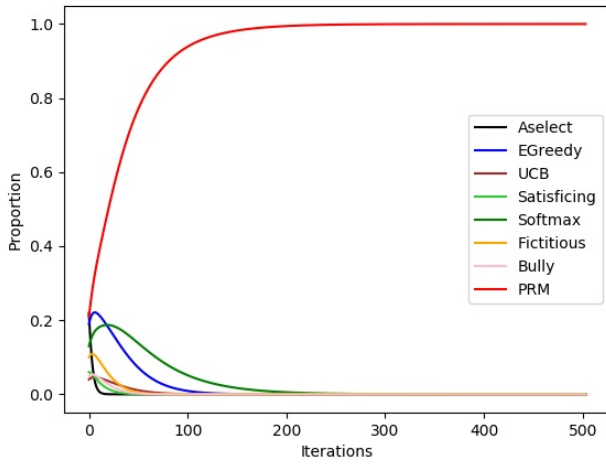


Fig. 7: replicator dynamic using non uniform starting values and the fixed matrix suite without our own strategy

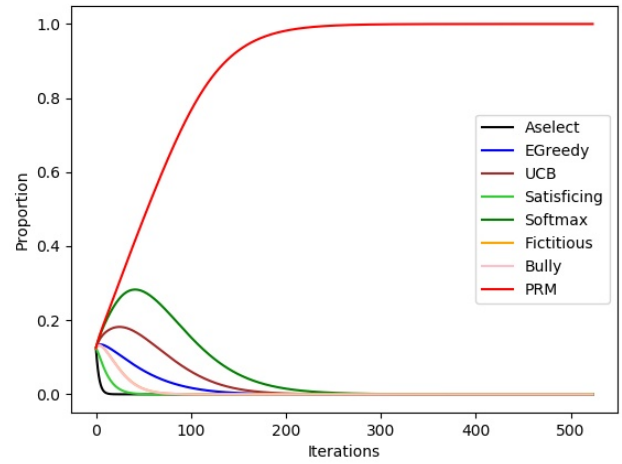


Fig. 8: replicator dynamic using uniform starting values and the fixed matrix suite without our own strategy

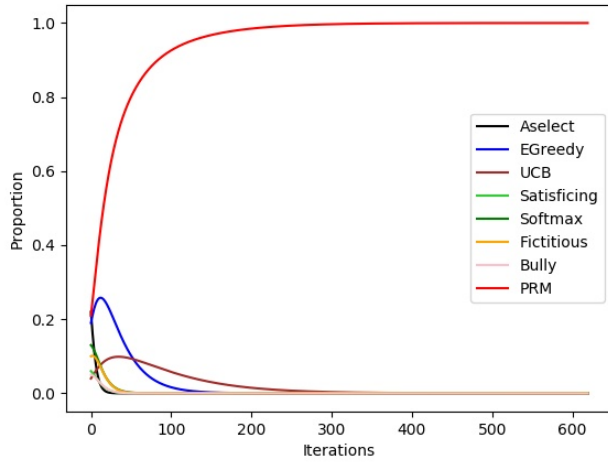


Fig. 9: replicator dynamic using non uniform starting values and the randomFloat matrix suite without our own strategy

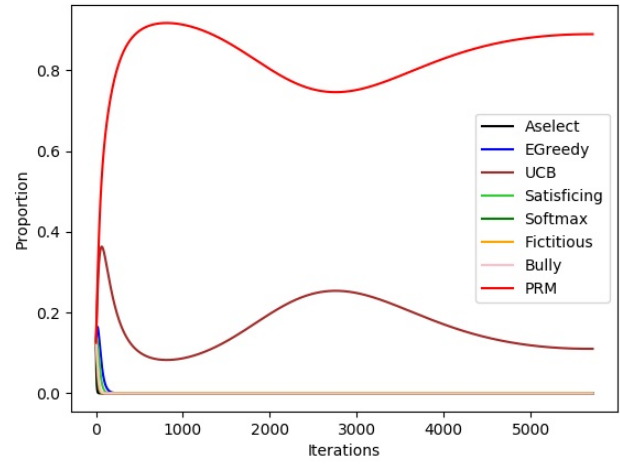


Fig. 10: replicator dynamic using uniform starting values and the randomFloat matrix suite with our own strategy

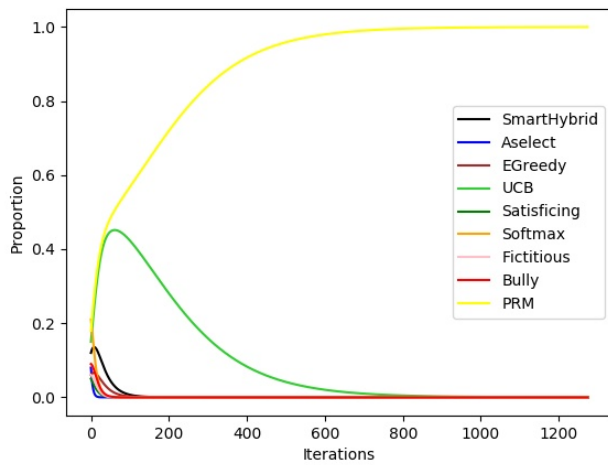


Fig. 11: replicator dynamic using non uniform starting values and the randomFloat matrix suite with our own strategy

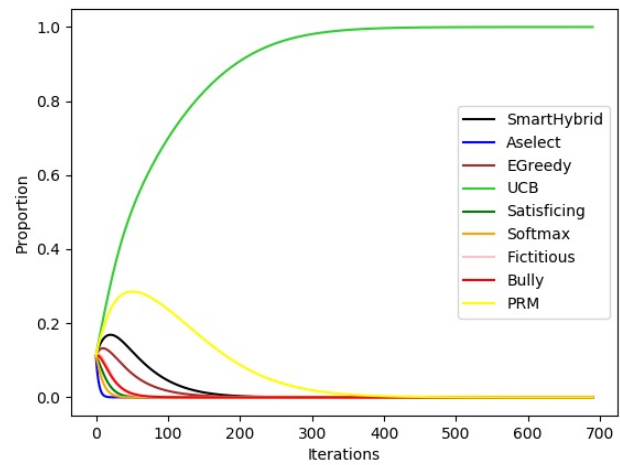


Fig. 12: replicator dynamic using uniform starting values and the randomFloat matrix suite without our own strategy

B.3 Nash Equilibria

Nash Equilibria	proportions row player	proportions col player
1	SmartGreedy 0.48 UCB 0.06 PRM 0.45	SmartGreedy 0.48 UCB 0.06 PRM 0.45
2	UCB 0.06 Satisficing 0.94	UCB 0.06 Satisficing 0.94
3	UCB 0.18 PRM 0.82	UCB 0.18 PRM 0.82

Tab. 8: Nash Equilibria obtained using all strategies and the randomFloat matrix suite