Utrecht University

MASTER THESIS

# Bandit algorithms applied to environments with multiple learners

*Author:*
Changlun Wang
*Student id:*
6212662

*Supervisor:*
Dr. G.A.W. (Gerard) VREESWIJK
*2nd examiner:*
Prof. dr. M.M. (Mehdi) DASTANI

*A thesis submitted in fulfillment of the requirements for the degree of Master of Artificial Intelligence*

*in the*

Intelligent Systems group
Department of Information and Computing Science

July 10, 2019

UTRECHT UNIVERSITY

# *Abstract*

Department of Information and Computing Science

Master of Artificial Intelligence

**Bandit algorithms applied to environments with multiple learners**

by Changlun Wang

Multi-agent learning algorithms are commonly used in a multi-agent environment. However, algorithms in such environments must learn the opponents' strategies in order to perform an agent's action, a process which increases the time and space complexity of an algorithm. In contrast, Bandit algorithms used in multi-armed bandit problems are agnostic to opponent behavior, which reduces algorithm complexity.

Given the different results of utilizing opponent behavior, this research investigates how bandit algorithms scale with an increasing number of players in a multi-agent environment in comparison to general-purpose multi-agent learning algorithms. The proposed hypothesis expects that the performance of bandit algorithms degrades slower than multi-agent learning algorithms when the number of players increases. To verify this hypothesis, a test bed has been developed in order to run a scalable tournament with configurable algorithm sets and game generators. More specifically, the Kolmogorov-Smirnov test and Spearman's rank correlation have been applied in order to inspect the distributions of test data and verify the hypothesis.

After analyzing the testing data, the performance of bandit algorithms has been concluded to degrade slower than multi-agent learning algorithms when the number of players increases under bounded conditions. Precisely, the difference of average rewards between two sets of algorithms continuously decreases when the number of player increases from 2 players to 7 players at 2-action games, even though multi-agent algorithms receive higher average rewards than bandit algorithms in these games. This result suggests the benefit of observing opponents' behavior for multi-agent algorithms is reduced as a number of players increases. However, the similar pattern was not found in 3-action and 4-action games, which sets a boundary to the proposed hypothesis.

Overall, this thesis built a testbed that analyzed the boundary of applying bandit algorithms to multi-agent environments by comparing the performance between two sets of algorithms in *m*-action *n*-player games. The experimental result showed that the proposed hypothesis partially holds on the condition from 2-action 2-player games to 2-action 7-player games. As such, the project paves the way to apply a bandit algorithm to a multi-agent environment, considering bandit algorithms have less time and space complexity and their performance degrade slower than multi-agent learning algorithms' performance as a number of player increases within the certain range.

**Keywords: multi-agent learning, multi-armed bandit, algorithm, test bed, multi-agent system**

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **BR** | **B**est **R**esponse |
| **FP** | **F**ictitious **P**lay |
| **MAL** | **M**ulti-**A**gent **L**earning |
| **MAB** | **M**ulti-**A**rmed **B**andit |
| **MAS** | **M**ulti-**A**gent **S**ystem |
| **QL** | **Q** **L**earning |
| **ECDF** | **E**mpirical **C**umulative **D**istribution **F**unction |
| **FMDP** | **F**inite **M**arkov **D**ecision **P**rocess |

# Chapter 1

# Introduction

## 1.1 Introduction

The field of *Multi-Agent Learning* (MAL) has attracted an increasing amount of research because MAL helps identify optimal solutions for various *Multi-Agent System* (MAS) problems, such as autonomous driving [20] and traffic light control [7], among others. The potential for wider applications has incentivized researchers to investigate not only the performance of MAL algorithms, but also the behavior that influences these algorithms. More specifically, researchers begun to consider how behavior influences an algorithm. By knowing the boundary of each algorithm in a given multi-agent environment, we can select a proper algorithm tackling various MAS problems in the real world.

In this project, we categorize all algorithms into two sets based on awareness of other agents in the environment. Awareness can be defined as whether the agent uses information from other agents in order to perform its action. For example, the agent may observe the last 50 actions of the opponent in order to perform its action. In this scenario, the agent is considered aware of other agents. On the other hand, the agent base an action on its own information, such as the average payoff of actions. Thus, the awareness of other agents' actions is used to divide algorithms into two sets — MAL algorithms and *Multi-Armed Bandit* (MAB) algorithms. Besides, a special case is an algorithm is not categorized if it uses neither opponents' information nor agent's information, such as Random algorithm.

The set of MAL algorithms, such as *Fictitious Play* (FP) [1] and *No-Regret* [6], allow an agent to observe opponents' actions in a multi-agent environment. The observation helps the agent learn from its opponents' strategies so that the agent can utilize the additional information to perform an action with higher rewards. For instance, the agent using FP stores the action history of its opponents. Based on the empirical frequency distribution of actions, the agent can best respond to its opponents during the subsequent iterations.

In contrast, the MAB algorithms are designed to tackle multi-armed bandit problems, such as *$\epsilon$-greedy* [4] [18], QL [5], *Softmax* [12], *UCB* [8], and *Exp3* [9]. The agent using MAB algorithms performs its own action based on its own information without knowing its opponents' actions. The agent decides by relying on the average payoffs of each action, rather than learning its opponents' strategies. For example, the agent adopting *$\epsilon$-greedy* spends $\epsilon$ % time exploring actions with the non-highest average payoffs among all actions while spending (1 - $\epsilon$)% time exploiting the action with the current highest average payoff. As such, the agent emphasizes the correlation between actions and corresponding average payoffs instead of considering the information associated with other agents.

### 1.1.1 Research questions

The MAL and MAB algorithms strive to maximize an agent's cumulative rewards. Whether using added opponents' information to perform an action differentiate them. However, this major distinction raises specific questions, which have formed the guiding research questions of this thesis. One question examines whether the observation of opponents' actions provides more information to an agent and asks if the additional information is always beneficial, leading to better performance, for example. If questions are only answered by specifying a given environment setting, then what conditions influence the answer? Do game selections, number of players, or number of player actions have an impact on the answer? In addition, what conditions help confirm the statement that MAL algorithms may outperform MAB algorithms, even though MAL algorithms take advantage of learning opponents' strategies?

By answering these questions, we are able to observe the performance of each algorithm and identify the correlation of performance between the two sets of algorithms.

## 1.2 Literature review

To answer the aforementioned questions, we have examined the existing literature. Although several experiments discussed the performance of MAL or MAB algorithms, we have identified three main limitations in the literature—algorithm selection, various testing environments, and different experiment goals. In this section, we explain these limitations and discuss why they are obstacles to answering the research questions.

### 1.2.1 Imperfect algorithm selection

Most of the literature focused on the performance of the same category, rather than choosing a similar amount of algorithms from each set. For example, [13] conducted a tournament, including 9 algorithms for comparison. However, 6 out of 9 algorithms, including generalized tit-for-tat (GTFT), best response to previous action (BR), FP, best response to FP, Bully, and Saby, are all MAL algorithms. The remaining three algorithms were Random, Maxmin, and Nash. Thus, a small amount of MAB algorithms were included. On the other hand, the tournament of another study, [19], focused on MAB algorithms. Six of the algorithms, including $\epsilon$-greedy, Boltzmann exploration (or Softmax), pursuit [3], reinforcement comparison, UCB1 and UCB1-Tuned, were MAB algorithms. However, a much larger tournament was conducted by Bouzy [17] and included 12 algorithms—JR [2], Minimax, FP, QL, S [10], M-Qubed, UCB, Exp3, HMC [6] Bully, Optimistic, and Random. Although the literature has included MAL, MAB and other non-learning algorithms in its evaluation framework, the tournament only ran 2-player matrix games, which limits the access of algorithm scalability.

The study [14] conducted a tournament which included 11 algorithms from both MAL and MAB algorithms, including FP, AWESOME, determined, meta, GIGA-WoLF, GSA, RVS, QL, Minimax-Q, Minimax-Q-IDR and Random, thereby constituting a well-balanced selection. However, several state-of-the-art algorithms, including UCB, Exp3, and $\epsilon$-greedy, were not chosen, a limitation which reduces the effectiveness of the experimental results. Indeed, the claim that one category of algorithms outperforms the other is not convincing if it relied on algorithms developed decades ago. While the experiment does not require all combinations of algorithms and games in order to be reliable, it should, as a minimum requirement, include representative algorithms based on MAL and MAB so that the two sets of algorithms can be objectively compared.

The literature discussed above indicated that the number of chosen MAL and MAB algorithms were neither the same nor similar in those experiments. We observed that most authors did not chosen algorithms based on whether they are MAL or MAB. Instead, they chose algorithms based on other research purposes. As a result, we have conducted a new experiment with two sets of algorithms with the capability of running in $m$-action $n$-player games in order to examine our research interest.

### 1.2.2 Result integration difficulties

Many experiments were conducted [13, 19, 17, 14] to provide insight into algorithm performance. If the results of these experiments can be directly merged, a ranking order of performance among algorithms is concluded for the algorithms in which we are interested. However, the various testing settings that significantly differ among these experiments limit our usage of them. Therefore, in the paragraphs below, we have examined the testing settings and decision choices of four papers and discussed how they differ.

To begin, Airiau, S. and Saha, S. and Sen, S. [13] adopted an evolutionary tournament to rank algorithms in order of performance. In this tournament, 57 2x2 conflicted games were played for 1,000 iterations. Of the 57 games, 51 had one Nash-equilibrium, while the remaining games had multiple equilibria. To remove bias for the agent only playing as a row player or column player, each agent played 1,000 iterations as a row player and then restarted for 1,000 iterations as a column player.

Kuleshov, Volodymyr and Doina Precup [19] conducted a tournament for MAB algorithm comparison. Since MAB algorithms only focus on the relation between actions and their corresponding payoffs, this tournament set the number of actions in a game by using the number of arms, K, from which the agent could choose. In this case, K was valued at 2, 5, 10, and 50. The expected payoff of each arm was given at random, on [0,1] with standard deviation at 1%, 10%, and 100%. The result of each game was given after 1,000 iterations.

Bouzy, Bruno and Marc Métivier [17] set payoffs in the range [-9, +9]. In contrast to the two previous experiments, these authors set the number of iterations up to 3,000,000 because the testing result indicated that the ranking of algorithms changes after 1,000 iterations. This observation increased its number of iterations to become larger than that of the aforementioned studies.

Zawadzki, Erik P. [14] conducted the largest of the mentioned experiments by including five action set sizes, 2x2, 4x4, 6x6, 8x8, and 10x10, and 11 algorithms. Moreover, the experiment included 600 game instances with 100,000 iterations, although 90,000 were not used for analysis due to the beginning settle. In total, 72,600 matches were evaluated for algorithm comparison.

Given the four experimental papers discussed above, we observed the various testing approaches and test settings used to conduct experiments. Some papers used the permutation of row and column players to eliminate bias. Testing settings varied in terms of maximum number of iterations, range of payoffs, size of action set, and number of games. However, we only examined variables associated with games, not parameters associated with algorithms. Certain experiments optimized the parameters of algorithms in the tournament to ensure that each algorithm performs its best during the experiment process. Yet most authors did not mentioned how they tuned these parameters. Therefore, these experiment distinctions in design purpose and detail led to difficulty not only in duplicating an existing experiment but also in integrating the testing results.

### 1.2.3 Experiment goals

At this point, we have discussed two main limitations. One limitation comes from the biased selection of algorithm combinations in existing experiments, which cannot perfectly mix two sets of algorithms, and the other comes from the difficulty of integrating experiment results. Next, we explain another critical limitation associated with our hypothesis. As we are interested in the question of whether MAB algorithms can be applied to the MAL environment, one purpose of conducting this-experiment is to determine the relation between the two groups of algorithms. Most literature [14] [13] [19] [17] focused on the relation between individual algorithms, as noted above. Previous studies often designed experiments to compare all algorithms and identify which algorithm is superior. This fundamental distinction creates variance in experiment design, especially in terms of how experiments choose algorithms in the beginning stage and how they analyze test data, either based on a single algorithm or group algorithms. Most importantly, this explains why the algorithm comparison plots presented in the literature did not discuss the relation between MAB and MAL algorithms.

In summary, we aim to categorize MAL and MAB algorithms based on whether the agent learns from the opponents' actions or simply relies on its own information. The distinction of both algorithms leads us to examine the conditions that affect their performance. Moreover, we question whether MAB algorithms can be applied in a multi-agent environment, regardless of their information use. With these questions in mind, the limitations present in the existing literature—such as biased algorithm selection, experiment settings, and different experiment purposes—have encouraged us to conduct this experiment in order to answer our research questions.

## 1.3 Hypothesis

We have discussed the key difference between MAB algorithms and MAL algorithms. Here, we provide a hypothesis, as noted in Table 1.1, which proposes how that difference influences algorithm performance. As such, we aim to predict the performance difference between MAB and MAL algorithms in different environment settings.

TABLE 1.1: Hypothesis setting and expected result

| No. of Players | Reward of MAB algorithms | Reward of MAL algorithms |
|---|---|---|
| 2 | X1 | Better than X1 |
| >2 | Similar to Y1 | Y1 |

The first row of the table presents a simple two-player game. The hypothesis expects that the MAL algorithm will outperform the MAB. In this case, the agent using the MAB algorithm would receive the reward X1, whereas the agent using the MAL algorithm would receive a reward larger than X1. This outcome results from the fact that the agent using MAL could observe the other agent in order to learn the strategies of its opponent. Learning opponents' strategies allows the MAL agent to achieve the best response, which receives maximum rewards. In contrast, the agent using MAB only plays the action based on the payoff of actions, without considering the opponent's action. While the agent may receive a better payoff than playing a random action, the action that the agent chooses cannot always be the optimized action because the payoff of the action is determined by the agent's action as well as the opponents' actions. If the opponents' actions change over time, the agent's payoff also changes, which makes the agent's action less optimal. Hence, we suggest that MAL outperforms MAB in a two-player game.

The second row of the table describes a game that involves more than two players. In this case, we expect an insignificant performance difference between MAL and MAB. As the table indicates, the agent using the MAL algorithm would receive the reward Y1, whereas the agent using the MAB algorithm would receive a reward similar to Y1. When the game involves more players, the complexity of opponent learning strategies increases. This increasing complexity offsets the benefit of MAL such that the agent can optimize his or her action based on knowledge of the opponents' strategy. Thus, the performance between the two sets of algorithms becomes subtle.

If this hypothesis holds at the end of the experiment, we could question why MAB algorithms are not used in a multi-agent environment when the number of players is larger than two, given that MAB algorithms achieve similar performance as MAL algorithms in an environment with more players. In addition, MAB algorithms use less information only associated with its own actions and payoffs. This feature allows them to be more easily implemented than MAL algorithms because they do not have to retrieve information from opponents. If the hypothesis only holds under specific conditions, we can confirm the application of MAB algorithms when such conditions are satisfied. On the other hand, if the hypothesis does not hold for any condition, questions should be raised for further investigation. For instance, if we agree that MAL algorithms are more applicable in a multi-agent environment, despite increases in a number of players, we must ask why the complexity of increasing players does not reduce the benefit of using opponents' information.

## 1.4 Thesis structure

The thesis begins with an overview of the existing literature and a presentation of our hypothesis. Chapters 2 and 3 introduce all of the algorithms and game types that can be used in the tournament. Chapter 4 discusses the about metrics used to evaluate algorithm performance. Next, Chapter 5 reviews the research methodology, which explains how we built the test framework to answer our research questions. After conducting the experiment, we will present the collected data and results in Chapter 6. Lastly, in Chapter 7 we offer a conclusion based on our experiment and discuss possibilities for future research.

# Chapter 2

# Algorithms

This chapter introduces two categories of algorithm, *multi-agent learning* algorithms (MAL) and *multi-armed bandit* algorithms (MAB) individually. To illustrate the core idea of these algorithms, 2-player games are used as examples. Some discussions are about the scalability of algorithms in terms of the number of players and the number of actions. In general, MAL algorithms that use more information in their calculation may suffer from increasing computation cost as the game size increases, while MAB algorithms using less information have less time and space complexity, e.g. linear or constant complexity. More detail of time and space complexity are discussed in each algorithm.

In addition, the term *agent* is used to indicate a player using an introduced algorithm, while the term *opponent* or *opponents* is used to indicate the remaining player(s) using other algorithms.

## 2.1 Multi-agent learning algorithms

MAL algorithms are based on the information from both an agent and its opponents. The information from its opponents, for example, could be opponents' action history that stores how many times each action has been played. Moreover, this action history could be completely used or partially used, e.g. only using the last 20 iterations for calculation, which is determined by algorithms. The opponent's information can also be combined with agent's information, e.g. based on opponents' action history, the agent chooses an action with the highest average rewards.

Yet, the cost of using more information might be the increasing space complexity or time complexity of algorithms. In addition, we cannot conclude this additional information benefits the performance for all kinds of game size until the analysis of the experiment results has been done. Both time and space complexity are listed in the end of discussion of each algorithm.

### 2.1.1 Fictitious play

*Fictitious Play* chooses its action based on opponent's action history. The algorithm calculates the frequency of each action that opponents have played in the past iterations so that the agent can take advantage of knowing which action is more likely to be played in the next iteration by his opponent. On top of this, the best response of the agent is to pick the action that gains the highest average rewards.

The pseudo code is given by:

**Algorithm 1** Fictitious play algorithm

```
function FP($opponentActionHistory$)
    $mostFrequentAction \leftarrow$ find maximum counts of actions in action history
    for $agentAction$ in $agentActions$ do
        $avgPayoff \leftarrow$ get average payoff by($agentAction, mostFrequentAction$)
        if $avgPayoff < maxAvgPayoff$ then
            $bestAction \leftarrow agentAction$
        end if
    end for
    return $bestAction$
end function
```

In practice code, if the frequency of opponent's actions are equally being played, the most frequent action is randomly chosen among actions with equal frequency.

Complexity analysis:

- Space complexity : $\mathcal{O}(n^2)$
- Time complexity : $\mathcal{O}(n^2)$

*Fictitious Play* uses not only agent's average rewards of actions, but also opponents' action history. This makes the space complexity increased quickly since the algorithm stores the counts of each action that have been played by each player, which is $\mathcal{O}(n^2)$. The time complexity is $\mathcal{O}(n^2)$ as well because the algorithm has to find which action is more likely being played by each player through looping opponent's action history of every player.

### 2.1.2 No-regret learning

*No-regret learning* chooses its action based on evaluating those agent's actions that would have been successful in the past. The term *regret* is explained as: an algorithm evaluates all hypothesized rewards of all actions that is played by the agent in the past. If the hypothesized rewards of an action was larger than the received rewards of a chosen action at that time, it produces *regret* on this non-chosen action by the value of difference between hypothesized rewards and received rewards.

After the evaluation of hypothesized rewards of each action, the agent then chooses its action that would have received the most hypothesized rewards in the past. It is also the same as choosing an action having the most regrets. The goal of the algorithm is to minimize the accumulated regrets among actions. The algorithm is separated into a few steps shown below.

- The agent maintains a hypothesized reward vector, that stores accumulated hypothesized rewards for each action if the agent might have played that action in the past.

- After each iteration, the agent updates hypothesized reward vector for each action.
- The agent chooses best action based on the action with the maximum accumulated hypothesized rewards in the hypothesized reward vector.

To illustrate how algorithm works, the pseudo code is given by:

**Algorithm 2** *NoRegretLearning* algorithm

```
1: function NoRegret(actionHistory)
2:     receivedReward ← rewards received by a chosen action in the last iteration
3:     for all actions do                ▷ update hypothesized rewards for each action
4:         hypoRewards ←  the rewards if action has been played
5:         hypoRewards[action] += (hypoReward - receivedReward)
6:     end for
7:     bestAction ← the action with maximum hypothesized rewards
8:     return bestAction
9: end function
```

Complexity analysis:

- Space complexity : $\mathcal{O}(n)$
- Time complexity : $\mathcal{O}(n)$

*No-regret learning* only needs a vector to store hypothesized rewards for each action so that the space complexity is $\mathcal{O}(n)$. Since the size of vector is the number of actions that the agent can choose. The time complexity is $\mathcal{O}(n)$ as well because the algorithm loops the hypothesized reward vector once to choose the best action with maximum value and update hypothesized reward vector once in each iteration.

## 2.2 Multi-armed bandit algorithms

MAB algorithms were invented to solve *Multi-Armed Bandit Problems* (MABP), which strives to gain rewards, as many as possible from a number of slot machines (one arm bandits). The total rewards are determined by these fixed limited set of resource as in competing choices. A MAB algorithm has to explore different slot machines over iterations to obtain reward information behind these machines, or exploit a particular slot machine that is considered to have higher rewards.

If MAB algorithms are applied to multi-agent environment, a number of slots are simply replaced with a number of opponents. The goal of MAB algorithms stays the same to figure out how to help an agent to maximize received rewards against its opponents. In addition, a MAB algorithm can adopt exploration and exploitation strategies to choose an action where it used in a multi-armed bandit problem.

In addition, the essential characteristic of MAB algorithms does not utilize any opponents' information. This is because MAB algorithms simply observe how many rewards that each slot generates in MABP rather than which action is performed by its opponents in multi-agent environment. Not using opponents' information makes most MAB algorithms to be designed more easily in terms of less space and time complexity. We will elaborate how each MAB algorithm works and what its space and time complexity are.

### 2.2.1 $\epsilon$-greedy and $N$-greedy

$\epsilon$-greedy chooses its action between exploration and exploitation, which is based on the value of $\epsilon$. In each iteration, the agent randomly chooses an action with the non-highest average rewards by the probability $\epsilon$, while the agent chooses the action with the highest rewards by the probability (1 - $\epsilon$). It means the agent spends $\epsilon$ time exploring a new action that would have given the agent more rewards while the rest of time to exploit the action with the highest average rewards in the past.

The probability of choosing action $i$ at the time step $t+1$ is represented by:

$$p_i(t+1) = \begin{cases} 1-\epsilon & \text{if } i = \underset{j=1,\dots,K}{\arg\max} \ \mu_j(t) \\ \epsilon & \text{otherwise} \end{cases} \tag{2.1}$$

$\mu_j(t)$ : empirical average rewards of action $j$ in time step $t$

The pseudo code of the algorithm is as:

**Algorithm 3** $\epsilon$-greedy algorithm

```
1: function ε-greedy(none)
2:     $rand \leftarrow$ generate random value between 0 to 1
3:     if $rand < \epsilon$ then
4:         $bestAction \leftarrow$ pick a random action
5:     else
6:         $bestAction \leftarrow$ action with the highest average rewards
7:     end if
8:     return $bestAction$
9: end function
```

We observe that the value of $\epsilon$ determines the proportion of exploration and exploitation. However, the value is fixed, which means that the agent always spends $\epsilon$ time on exploration no matter the number of iterations are. If a multi-agent environment is static and the agent has received stable average rewards for each action, any action with the non-highest rewards may only decrease the agent's total rewards.

To improve the aforementioned problem coming from a fixed $\epsilon$ value, $N$-greedy is introduced where its $\epsilon$ value decreases over the time. The new $\epsilon$ value is determined by the value of the number of actions divided by the number of total iterations. Therefore, the exploration time decreases while the number of total iterations increases, which reduces choosing non-highest rewards actions.

Complexity analysis:

- Space complexity : $\mathcal{O}(n)$
- Time complexity : $\mathcal{O}(n)$

The algorithm only stores the average rewards for each action in order to choose an action in the next iteration. Thus, the space complexity is $\mathcal{O}(n)$. The algorithm exploits the action with the highest average rewards by looping each action once, which results in the time complexity $\mathcal{O}(n)$.

### 2.2.2 Boltzmann Exploration - Softmax

*Softmax* (or Boltzmann Exploration) chooses its action based on a probability of an action. The probability is proportional to action's average rewards. An agent begins to explore available actions in order to obtain average payoffs of each action, which are converted into probabilities. Next, the agent choose an action by these probabilities.

Similar to $\epsilon$-greedy, *Softmax* keeps a possibility to explore actions by using the probability of actions, which enables the algorithm to be more adoptive in dynamic environments, e.g. the average rewards of each action may change over time. However, the trade-off is as same as $\epsilon$-greedy has by not fully exploiting to an optimal action with the highest average rewards. This trade-off between exploration and exploitation is tuned by the parameter $\tau$ that is introduced later. The probability of choosing an action $i$ as the best action in the next step $t+1$ is derived by:

$$p_i(t+1) = \frac{e^{\mu_i(t)/\tau}}{\sum_{i=1}^{K} e^{\mu_i(t)/\tau}} \tag{2.2}$$

where
$K$ : the number of available actions for agent

$\tau$ : a temperature parameter, that is used to determine the randomness of choosing the action.

$\mu_i(t)$ : the average rewards for an action $i$

The pseudo code is given by:

**Algorithm 4** *$Softmax$* algorithm

```
1: function $Softmax(averageReward_{action_i})$
2:     for all $actions$ do                          ▷ calculate a probability of each $action_i$
3:         $probability_{action_i} \leftarrow$ formula 2.2
4:     end for
5:     $rand \leftarrow$ generate a random value between 0 to 1
6:     for all $probability$ do                      ▷ choose an action by action probabilities
7:         $rand = rand - probability_{action_i}$
8:         if rand < 0.0 then
9:             $bestAction \leftarrow action_i$
10:        end if
11:    end for
12:    return $bestAction$
13: end function
```

Based on different $\tau$ values, there are three kinds of behavior of choosing agent's best action as below. A smaller value leads to more exploitation, while a larger value leads to more exploration. In practice, the value is tuned in between by running experiments.

$$\begin{cases} \text{Pure exploitation} & \text{if } \tau = 0 \\ \text{Pure exploration (uniformly choose an action)} & \text{if } \tau \to \infty \\ \text{the mix of exploration and exploitation} & \text{otherwise} \end{cases} \tag{2.3}$$

Complexity analysis:

- Space complexity : $\mathcal{O}(n)$
- Time complexity : $\mathcal{O}(n)$

The information needs to be stored is the average rewards for each action so that it does not have to be recalculated in each iteration. Therefore, the space complexity is $\mathcal{O}(n)$ based on the number of actions. Each iteration, the probability of each action has to be calculated once, which results the time complexity $\mathcal{O}(n)$

### 2.2.3 Q-learning

*Q-learning* chooses its action based on the optimal policy in *Finite Markov Decision Process* (FMDP) by evaluating Q-values by a given pair $(state, action)$, which maximizes total rewards in the next subsequent steps. We define the step of updating Q values as:

$$Q_n(x, a) = (1 - \alpha_n) \cdot Q_{n-1}(x_n, a_n) + \alpha_n \cdot [r_n + \gamma \cdot \max_a Q_{n-1}(x_{n+1}, a)] \tag{2.4}$$

$x_n$ : current state in step $n$

$a_n$ : the action being performed in step $n$

$y_n$ : the subsequent state in step $n$

$r_n$ : received rewards by performing action $x_n$ in step $n$

$\gamma$ : the discount factor, that determines how important the future expected rewards might be. If the $\gamma$ is set to 0, it means that the agent only considers the current rewards. In contrast, the $\gamma$ is set to 1 or close to 1, which leads the agent to only seek for an infinite future rewards.

$\alpha_n$ : the learning rate at the time step $n$, that determines how fast Q value is updated against the change of current received rewards and future expected rewards.

$\max_a Q_{n-1}(x_{n+1}, a)$ : agent chooses the best action $a$ that can receive maximum estimated rewards in the future in the next state $x_{n+1}$

Because our testing game environment is stateless, which views each iteration as the same state. The formula of updating Q-value is reduced to:

$$Q_n(a) = (1 - \alpha_n) \cdot Q_{n-1}(a) + \alpha_n \cdot [r_n + \gamma \cdot 1] \tag{2.5}$$

Finally, with each iteration the agent evaluates all Q-values and then performs the action with the maximum Q value.

In addition to using Q-value to determine the best action, the $\epsilon$-greedy policy is adopted to spend $\epsilon$ time exploring other actions. The $\epsilon$ value can also be decreasing over time, which is controlled by the value of decreasing exploration rate.

Complexity analysis:

- Space complexity : $\mathcal{O}(n)$
- Time complexity : $\mathcal{O}(n)$

The stateless *Q-learning* stores a Q-value array with the same size to available actions. The space complexity is $\mathcal{O}(n)$. Each iteration, it updates single element in the Q-value array and picks the action with the maximum Q-value by looping each Q-value. Thus, the time complexity is $\mathcal{O}(n)$.

### 2.2.4 Satisficing

*Satisficing* chooses its action based on the criteria whether the received rewards meet an attained threshold. If the current received rewards of the best action exceed a given threshold, the agent keeps choosing this action until the condition is violated. In contrast, the violated condition leads the agent to randomly choose other actions that have not been chosen in the previous iteration.

The threshold is called *aspiration level* in *Satisficing*. In *Satisficing*, the agent maintains a tuple $(A_t, \alpha_t)$ at any time step $t$, which is represented by:

$A_t$ : the currently chosen action

$\alpha_t$ : the *aspiration level* that is used as a threshold to enable the agent to select other new actions.

In each iteration, the *aspiration level* is updated by:

$$\alpha_{t+1} = (1-\lambda)\alpha_t + \lambda r_t \tag{2.6}$$

$\lambda$ : the persistence rate that influences how fast to update the *aspiration level* by new coming received rewards.

$r_t$ : the received payoff in the time step $t$

The best action in the next iteration is determined by:

$$A_{t+1} = \begin{cases} A_t, & \text{if } r_t \geq \alpha_t, \\ \text{any other action} & \text{otherwise} \end{cases} \tag{2.7}$$

The current action is chosen if the received rewards do not exceed the *aspiration level*. However, if the current received rewards decrease somehow, the *aspiration level* decreases as well. When the current received rewards are lower than the *aspiration level*, the agent chooses other actions.

Compared to other algorithms who exploit to an action with the highest rewards, the *Satisficing* does not always choose an optimal action. Instead, it only chooses an action that meets its need (higher than *aspiration level*) until the need is not met.

Complexity analysis:

- Space complexity : $\mathcal{O}(1)$
- Time complexity : $\mathcal{O}(1)$

*Satisficing* only maintains a single tuple including a value of chosen action and *aspiration level*, which makes the space complexity $\mathcal{O}(1)$. With each iteration the values in the tuple are updated without any loop so that the time complexity is $\mathcal{O}(1)$.

### 2.2.5 EXP3

*Exponential-weight algorithm for exploration and exploitation* (EXP3) chooses its action based on the probability distribution of available actions. A higher probability represents high expected rewards. The probability distribution is updated by using a weight array with the equal size to available action size. After each iteration, the weight array is updated as follows:

Initially, each element $w_i$ in the weight array is set to 1, which gives all actions the same probability in the first iteration.

$$w_i(t) = 1 \quad \text{Only if } t = 0 \tag{2.8}$$

Each action is chosen by a weighing value in the weight array. A higher weighting value means that the corresponding action has higher probability to be chosen and updated. In the beginning iterations, randomly updating an action more frequently results in higher weighting values of this action. However, this reduces the opportunity to explore other actions that have not been chosen in the beginning iterations. In order to eliminate the imbalanced sampling, a function $p_i(t)$ is used to serve as a probability dense function. The function is used in the later formula of updating weighting values. Basically, higher weighting value $w_i(t)$ results in higher $p_i(t)$, which leads to update less weighting value instead.

$$p_i(t) = (1-\gamma)\frac{w_i(t)}{\sum_{i=1}^{n} w_i(t)} + \frac{\gamma}{N} \tag{2.9}$$

$\gamma$ : the learning rate of updating weighting value

$N$ : the size of all available actions that are chosen by the agent.

Updating weighting values $w_i(t)$ are calculated below. The first formula is to calculate the estimated rewards by the observe rewards $r_i(t)$ divided by the probability density function $p_i(t)$ that has been calculated by the formula above. The second formula then updates the corresponding weighting value $w_i(t)$ in the weight array by the estimated rewards $r_i$.

$$r(t) = \begin{cases} \frac{r_i(t)}{p_i(t)}, & \text{if } a(t) = i \\ 0, & \text{otherwise} \end{cases} \tag{2.10}$$

$$w_i \leftarrow w_i(\gamma r_i / N) \tag{2.11}$$

Complexity analysis:

- Space complexity : $\mathcal{O}(n)$
- Time complexity : $\mathcal{O}(n)$

*EXP3* stores a weighting array with the size as same as action size, which leads the space complexity $\mathcal{O}(n)$. In each iteration, the algorithm generates a new probability distribution based on the weighting array and then pick an action by the probability of weighting value. These calculations are associated with the action size, which results in the time complexity $\mathcal{O}(n)$.

### 2.2.6 UCB1

Upper confidence bound 1(*UCB1*) chooses its action based on both average rewards received in the past and the number of times that an action was played. Using average rewards in the evaluation ensures the agent has a tendency to choose an action with higher average rewards in the past. On the other hand, using the frequency of each action keeps a possibility to explore actions that have been played less often.

To choose the best action, we calculate the values based on both the average rewards and the frequency of each action mentioned above. Then, the best action is picked by the action with the maximum value. The detail calculation is shown by:

$$a(t) = \underset{i=1,\ldots,N}{\arg\max} \quad r_i + \sqrt{\frac{2\ln t}{t_i}} \tag{2.12}$$

$t_i$ : the number of times that action $i$ was played up to time-step $t$

$r_i$ : the average rewards received when action $i$ was played

From the equation above, the algorithm not only exploits the action with the highest average rewards in the past by the first term of equation, but also explores actions that have not been played often by the second term of equation.

The quality of the *UCB1* is evaluated by its expected accumulative bound. According to [8], the expected cumulative regret of *UCB1* after $t$ iterations is bounded by :

$$8 \sum_{i:\mu_i<\mu^*} \frac{\ln t}{\Delta_i} + \left(1 + \frac{\pi^2}{3}\right) \sum_{i=1}^{k} \Delta_i \tag{2.13}$$

where

$\mu^*$ : the expected rewards of the best action

$\mu_i$ : the expected rewards of action $i$

$\Delta(i) = \mu^* - \mu_i$: how suboptimal the action $i$ is

The formula above is separated into two terms. The first term includes accumulative regrets, which means an agent performs suboptimal actions for a logarithmic number of times $t$. It is divided by $\Delta(i)$ indicating how easy the agent distinguish an action from the optimal action. A smaller $\Delta(i)$ means the distinction between optimal and suboptimal action is more difficult, the agent may need more time to explore actions, resulting in an increasing accumulative regret.

The second term is used to cap the first term (the inverse of $\Delta(i)$) when the agent plays a number of suboptimal actions.

The pseudo code of *UCB1* is given by:

**Algorithm 5** *UCB*1 algorithm

```
function UCB1(averageReward)
    if initialState then                                        ▷ play each action once
        bestAction ← initialAction                              ▷ initialAction=0
        if initailAction == actionSize - 1 then
            initialState ← false
        end if
        initialAction ← initialAction + 1
        return bestAction
    end if
    for all actions do                                          ▷ choose best action
        $sum_i$ ← formula 2.11
    end for
    bestAction ← action with maximum sum
    return bestAction
end function
```

Complexity analysis:

- Space complexity : $\mathcal{O}(n)$
- Time complexity : $\mathcal{O}(n)$

*UCB1* stores average rewards of each action, which produces the space complexity $\mathcal{O}(n)$. The time complexity is also $\mathcal{O}(n)$ based on looping each action to find the maximum value in the formula above.

## 2.3 Random

*Random* algorithm chooses an action by an equal probability among all actions, which do not require information from an agent or its opponents. Therefore, it does not belong to MAL or MAB by the definition in Chapter 1. However, it is used as a benchmark against other MAL or MAB algorithms. For example, any algorithm with exploitation characteristic against the *Random* should receive similar average rewards since these algorithms tend to exploit the action with the highest average rewards after a certain number of iterations.

Besides, due to the simplicity of *Random* algorithm, the algorithm is used to evaluate the correctness of other implemented algorithms given that any other algorithm outperforms *Random*. As the performance of any other algorithm is not as expected, the algorithm against Random algorithm helps identify a possible issue.

Complexity analysis:

- Space complexity : $\mathcal{O}(1)$
- Time complexity : $\mathcal{O}(1)$

*Random* chooses its action randomly so that it does not need to save any information for any extra calculation.

## 2.4 Complexity Summary

TABLE 2.1: Complexity of MAL algorithms

| MAL | FP | No-regret |
|---|---|---|
| Space Complexity | $\mathcal{O}(n^2)$ | $\mathcal{O}(n)$ |
| Time Complexity | $\mathcal{O}(n^2)$ | $\mathcal{O}(n)$ |

TABLE 2.2: Complexity of Bandit algorithms

| Bandit | $\epsilon$-greedy | N-greedy | Softmax | QL | Satisficing | EXP3 | UCB1 |
|---|---|---|---|---|---|---|---|
| Space Complexity | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| Time Complexity | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |

TABLE 2.3: Complexity of other algorithm

| Other | Random |
|---|---|
| Space Complexity | $\mathcal{O}(1)$ |
| Time Complexity | $\mathcal{O}(1)$ |

# Chapter 3

# Games

In Chapter 2, we have introduced different algorithms that can be used by the agents. Each algorithm except for Random tries to maximize received payoffs from payoff matrix in a game instance. In this chapter we will talk about different game types that can be used to generate different payoff matrices. Also, we will talk about how to set parameters in all game types in general.

Diverse game types are designed not only to satisfy scientific interests where different algorithms may choose a pure or mixed strategy, but also to evaluate the performance of algorithms against each other in a multi-agent environment. Some algorithms may perform better in different types of games than others. For instance, any algorithm is supposed to perform better than the Random algorithm in most game types. In *Prisoners Dilemma*, for instance, two agents both using *Random* algorithm would perform slightly better than two agents using *Nash-equilibrium*. This indicates various game types can produce different result for a given algorithm pair. Thus, we will evaluate all algorithm pairs in various game types in order to obtain a more general result for algorithm performance.

To utilize game types with scientific interest instead of only using random games, the *GAMUT* [11] is used to work as a game generator to produce game instances in various game types. In this project, we will only use part of game types to evaluate our algorithms in the tournament. This is because one of our research interests is to find out the possible relationship between the number of players and actions and the performance of MAL and MAB algorithms. However, some game types in *GAMUT* can be only played in a fixed number of players or actions, which limits our experiment. Therefore, in our tournament we will evaluate all algorithms by using those game types with more than two players and more than two actions. For other less important experiment, e.g. observing basic algorithm performance in all two players and two actions games, we will run all types of game, which supports two players and two actions.

## 3.1 Games types and parameters

We divide all games into four groups based on whether the game type supports more than two players or more than two actions shown in tables below. In addition, we will not dive into each game type in each group in detail since the description of each game type can be found in the document of *GAMUT*. Here we will only discuss the basic values that are applied to all game types in order to generate different game instances.

Firstly, each game type can generate different payoff matrices, but the range of payoff matrices has to be the same in order to make a fair comparison among different game types. Therefore, we apply normalization to the payoff matrix in each game instance so that the minimum payoff and the maximum payoff will be normalized to a given range (-100, 100).

Secondly, each game type has its game specific parameters. Neither do we test all possible values for these parameters, nor do we tune them in purpose. Instead, in most cases the values of parameters are set by default or by using a built-in random function. This makes algorithms to be agnostic for game types to avoid introducing bias into the experiment.

The first group of game types are listed below, this group includes 12 game types that can support more than two players and more than two actions. The group is the most essential for the experiment since the adjustable players and actions make these game types more flexible to evaluate algorithms in the same game type using different numbers of players and actions.

| Group 1 : >= 2 players and >= 2 actions | |
|---|---|
| G1 | RandomGame |
| G2 | MajorityVoting |
| G3 | DispersionGame |
| G4 | GuessTwoThirdsAve |
| G5 | CovariantGame |
| G6 | BertrandOligopoly |
| G7 | BidirectionalLEG |
| G8 | TravelersDilemma |
| G9 | RandomGraphicalGame |
| G10 | RandomLEG |
| G11 | UniformLEG |
| G12 | MinimumEffortGame |

The following three groups contain the game types with an adjustable number of actions but not players, players but not actions and the types in which neither can be changed. Even though these game types are not used to answer the research question, still, they can be used as a general reference by evaluating the performance of algorithms with a complete set of 29 kinds of game types.

| Group 2: 2 players and >= 2 actions | |
|---|---|
| G13 | RandomZeroSum |
| G14 | GrabTheDollar |
| G15 | LocationGame |
| G16 | ArmsRace |
| G17 | CournotDuopoly |
| G18 | WarOfAttrition |

| Group 3 : >= 2 players and 2 actions | |
|---|---|
| G19 | CollaborationGame |
| G20 | RandomCompoundGame |
| G21 | CoordinationGame |

| Group 4 : 2 players and 2 actions | |
|---|---|
| G22 | Chicken |
| G23 | RockPaperScissors |
| G24 | PrisonersDilemma |
| G25 | BattleOfTheSexes |
| G26 | MatchingPennies |
| G27 | HawkAndDove |
| G28 | ShapleysGame |
| G29 | TwoByTwoGame |

# Chapter 4

# Metrics

The performance of an algorithm is viewed as an indication of which algorithm or which set of algorithms may perform better than others in a tournament. Literature [15] Section 7.1 suggested a more common approach to evaluate the performance by asking whether an algorithm achieves rewards that are *high enough*. The term of *"high enough"* contradicts the concept of *"best response"*, which focuses on the highest rewards. This approach includes three basic properties, *Safety*, *Rationality*, and *No-regret*, which evaluates an algorithm from different aspects. More specifically, *Safety* ensures an secured rewards to an agent by obtaining at least minmax rewards; *Rationality* focuses that an agent plays best response to other opponents that have settled stationary algorithms; and *No-regret* means that an agent pursues rewards that are less than the rewards that an agent could have received by playing one of agent's pure strategy.

These properties can be used to evaluate the performance of algorithms, even categorize an algorithm by examining whether it meets a number of properties. Instead of applying these properties to performance comparison directly, our experiments adopted similar but more quantitative approaches to measure the performance of algorithms. The following sections introduced two major metrics used in experiments: average rewards and average regrets.

## 4.1 Average rewards

The received reward is the most important to an algorithm because each agent all strives to maximize their received reward in each iteration. The metric of the average rewards is introduced to measure the quality of an algorithm, which is based on the average of accumulated rewards of an agent in a sampling period $T = b - a$, as formula 4.1 shows.

$$\text{Average rewards} \equiv \frac{\sum_{t=a}^{b} R(t)}{b - a + 1} \tag{4.1}$$

The sampling period of the metric is supposed to be long enough to measure whether an algorithm tends to converge. Some approaches suggested using an adaptive way to decide the number of total iterations in order to measure the stable state, which represents the variation of average rewards converge to a defined range. However, the different numbers of iterations favor the performance of different groups of algorithms in various environment settings. Instead of using a varied number of iterations, in our experiments, a number of total iterations was set to a fixed value. Though a fixed value may not be optimal to the performance of all algorithms, it maintained the consistency of tests; and more possible values could be tested in other experiments. In addition, some approaches suggested remove the results of earlier iterations in the metric since the results of these iterations are used for action exploration by algorithms. However, no significant performance difference were observed between using all iterations in a game instance and the removal of the results from the earlier iterations. Therefore, the entire sampling period was used in the average rewards. For instance, the average rewards are calculated by sampling 10,000 iterations as a game is run by 10,000 iterations.

## 4.2 Average regrets

Regrets are the difference between the current received rewards and the hypothesized rewards if an agent has had played the best action when opponents' actions kept the same. The average regrets are derived by the average of accumulated regrets in a sampling period $T = b - a$, as formula 4.2 shows. In a time step $t$, regrets are calculated by the current received rewards (i.e. $r(a, a_{-i}^{(t)})$ minus expected rewards (i.e. $\mathbb{E}[r(\sigma_i^{(t)}, a_{-i}^{(t)})]$). More detail can be found in [14].

$$\text{Average regrets} \equiv \frac{max_{a\in A}\sum_{t=a}^{b}[r(a, a_{-i}^{(t)}) - \mathbb{E}[r(\sigma_i^{(t)}, a_{-i}^{(t)})]]}{b - a + 1} \tag{4.2}$$

Average regrets are used to evaluate how much time an algorithm exploits to an action in a game. In Prisoner Dilemma, for instance, an algorithm using Nash Equilibria to determine its action will end up exploiting to the action (Defect, Defect) in all iterations. Because the algorithm does not explore other possible actions with higher rewards, the algorithm will produce higher regrets than other algorithms that explore actions with higher rewards. In contrast, other algorithm, such as $\epsilon$-greedy, spends $\epsilon$ time exploring different actions (C,C) (C,D) (D,C), which generates higher rewards and leads to less regrets.

To sum up, average rewards will mainly be used through the experiment since the metric is directly associated with the performance of any algorithm. Moreover, it was being widely used in all tournament experiments, such as [14]. As for average regrets, the metric is used as a reference instead of in the performance comparison, though it was used in a pure bandit algorithm tournament. [12]

# Chapter 5

# Methodology

To answer our research question, we carried out an experiment by conducting tournaments. A tournament consists of game types, algorithms, and evaluation metrics, as discussed in previous chapters. After running the tournament, we analyzed the tournament results based on algorithm performance. The results served to verify the hypothesis.

A test bed was used to run the tournament, for which we collected results for analysis. The following sections introduce the test framework that was used to run the experiment, discuss how the games and algorithms were chosen in the framework, examine the scale of the tournament, and consider how the data collected in the tournament was analyzed.

## 5.1 Test framework

The main goal of the test framework was to establish a flexible, efficient and robust platform that allowed us to run various experiments in order to answer the research question. The framework needed to be flexible enough to quickly add or remove algorithms and games. Moreover, we made the associated parameters easy to configure. The framework also had to be efficient in order to save time during large comparisons. The more efficient the test framework, the more comparisons it can run. Finally, the robust platform ensured the collection of valid results for data analysis.

The test framework can be broken up into several steps, as depicted below in Figure 5.1. The pipeline begins at the interface of the command line input, which allows a user to specify parameters to configure the test framework in the configuration step. For instance, it can determine the set of game types and the set of algorithms used in the experiment in addition to the number of players and actions used in each game. Based on the configuration, the task generation generates game instances and tasks, which are run in the subsequent computation step.

The computation step is composed by the number of compute units, which is based on the number of CPU cores provided by the hardware. Each compute unit retrieves one task from the task queue and obtains the required game file of the task. The compute units work in parallel in order to minimize total running time. All of the raw data of experiment results is stored in the database, which is used in the next step, data analysis.

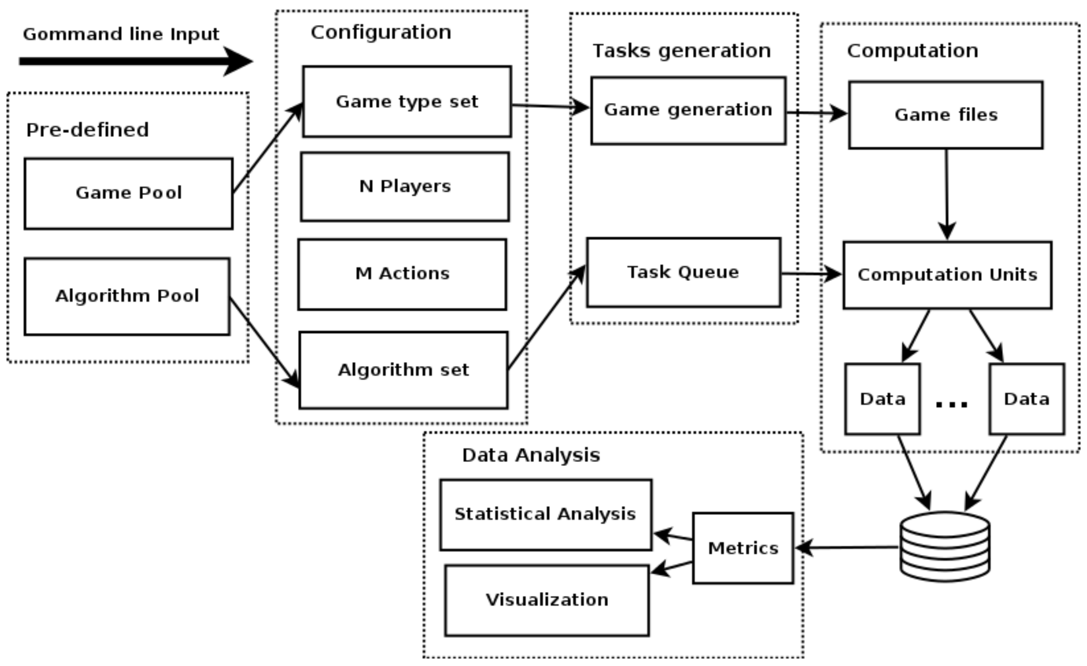

FIGURE 5.1: Test framework structure

### 5.1.1 Choosing tournament game types

In Chapter 3, we introduced all game types that could be used in the tournament. In the test framework, all game types were placed in a pre-defined game pool so that each tournament could pull a set of game types by a given criteria. For instance, the tournament could be run with game types of more than two players. In our experiment, we ran two versions of the tournament by using different sets of game types. The first tournament utilized the set of game types with two players and two actions. The tournament including all game types allowed us to obtain a basic benchmark for each game type played with algorithms. The second version used a small set of game types with more than two players and more than two actions. These types provided us with greater flexibility for examining algorithm performance when the number of players or the number of actions increased.

### 5.1.2 Choosing tournament algorithms

In Chapter 2, we introduced the algorithms. These algorithms were placed in a pre-defined algorithm pool in order to be available by a tournament. As this research focuses on the performance of MAL and MAB algorithms in a multi-player environment, the algorithms were grouped into MAL or MAB, except for the Random algorithm, which remained uncategorized.

After this categorization, the number of algorithms in the MAL and MAB groups was unbalanced. The total number of MAL algorithms was 2, which was unequal to the number of MAB algorithms, 6. However, the two MAL algorithms, FP and *No-Regret*, are more representative than the other MAL algorithms, which are not suitable for $n$-player games. For example, $TitforTat$ and $Bully$ are typically designed for two-player games. *Markov* is suitable for *n*-player games, although it becomes impractical when the algorithm uses greater memory storage for holding exponentially increasing action states. Yet, this framework more easily allows the future addition of more MAL algorithms.

## 5.2 Tournament arrangement

Given the pre-defined pools of game type and algorithm, the tournament was able to pull a subset from each pool in order to run different scales of experiment. The tournament scale was determined by the number of game types, algorithm pairs, players and actions, as demonstrated in formula 5.1.

To identify the tournament boundary, the maximum value of each parameter was used. More specifically, given the number of game types, which has been listed in Chapter 3, we used types which supported more than two players and more than two actions. In addition, each algorithm played against all others so that the total number of algorithm pairs was the square of total number of algorithms. Thus, 10 algorithms and 10 algorithms equaled 100 algorithm pairs. Each *n*-player game created *n* permutations of players in order to eliminate the bias of players who used different payoff matrices in the same game. Lastly, the number of actions for each player did not increase the number of total matches. However, the number of actions did increase the size of each game, which exponentially increased the running time, as demonstrated in Section 6.1.4, thereby adding more search space for each player.

### 5.2.1 Tournament scale

To calculate the number of matches played in a tournament, the formula below is given:

$$Total\ \ matches = (game\ \ types) * (algorithm\ \ pairs) * (player\ \ permutation). \tag{5.1}$$

By using formula 5.1, we derived that tournament scale mainly relied on the number of game types, algorithms and players. Both game types and players were linearly proportional to the total matches, while the algorithms scaled quadratically with the number of matches.

### 5.2.2 Game size

The game size represented as follows:

$$Game\ \ size = Actions^{Players}. \tag{5.2}$$

Although the game size did not increase the total number of matches, it did influence other aspects of the tournament. First, game size can be viewed as a searching space for players in a game. If the game size increases, the player cannot explore all possible actions in the finite iterations as a player can only find the best action if the player has already gone through all possible actions.

For instance, a game size of 100 in a 2-player and 10-action game would cost a player at least 100 iterations to explore each possible action once. If the game was only run for 50 iterations, it is unlikely that a player would be able to identify which action produces the highest reward as some actions with higher rewards may not have been explored. This problem is solvable by increasing the number of iterations. However, the number of iterations should not be infinitely increased as more iterations too quickly increases the running time, which may result in meeting the hardware limitation. Moreover, the scale of tournament is less important once the relation among relevant variables has been observed.

### 5.2.3 Tournament evaluation

To determine whether a correlation exists among the tournament variables, we first evaluated the performance of algorithm pairs in each match by using the metrics discussed in Chapter 4. For instance, each match produced the average payoffs for the main algorithm and its opponents. Thus, the tournament generated equal amounts of raw data for all matches. To derive meaningful data that can represent the performance of MAL and MAB algorithms in multi-player environments, we developed a multi-step evaluation process:

1. Collect all raw data by averaging accumulated payoffs or other metrics in each match.
2. Determine the performance of each algorithm pair by summing up matches with the same algorithm pair but different game types.
3. Determine the performance of MAL and Bandit algorithms by grouping algorithm pairs into two sets.
4. Repeat steps 1 to 3 for different number of players and actions.
5. Perform data analysis in order to answer the research question (refer to Section 5.3.5)

## 5.3 Implementation of test framework

In section 5.1, we introduced the individual components of the test framework that each serve distinct roles, which were assembled into a compact and flexible platform. Now, we discuss practical implementation in terms of the tasks that must be carried out and the relevant techniques for doing so.

Several multi-agent learning frameworks and learning algorithms have been written in various programming languages and already exist online. However, none of these frameworks and algorithms satisfy our requirement to run a large scale tournament with few modifications. Nonetheless, they may be useful as references.

### 5.3.1 Command-line input / configuration

The first task for our framework was to determine possible design choices for each component. In this case, the program provided the command line input, allowing us to configure all parameters in the test framework. However, it was unclear if the GUI interface be better to visualize parameter settings or experiment results. A graphical interface may be more human friendly, but it also increases both computation costs and the complexity of the test framework. Input arguments must include adjustable variables such as the number of players, the number of actions, the number of strategies and the game types.

Using simple command-line input also has advantages. To begin, the information is on the same terminal. There is no need to handle event callbacks and data transformation between text-based data and GUI. The GUI may also provide an easy way to inspect data, such as the game matrix displayed by a table. However, we focused on tasks that were considered necessary to run the experiment. As a result, we designed a command-line input that accepted most programming options. The details of these options are located in Appendix A.

### 5.3.2 Programming language

Running a large scale tournament involved many computations, especially when the number of actions and players increased. Although the tournament could be run once to generate all needed raw data, we were unsure how much computation time the tournament would require in advance. Moreover, the scale may have needed to change over time based on our research interests. If we did not prepare more capabilities to the framework beforehand, overcoming the limitations of the framework would have required more effort. As a result, we used C++, which equips more capabilities in system low-level control in order to write a more efficient application than other programming languages. It allowed us to take a full advantage of hardware cores and equally distribute computation tasks among computation units. In addition, effective memory operations reduced the amount of overhead. However, performance was not the only consideration in the test framework. We started the framework from scratch in order to have an opportunity to choose features that added more flexibility.

### 5.3.3 Task generation / computation

The basic workflow to execute a single is described below. First, a game instance was generated by an external program, GAMUT executable file, which was written in Java. The GAMUT output a single game file storing game matrices. Second, the game file was loaded into the C++ program with a parser, parsing the raw data of the file into payoff matrices. Lastly, we created a task with game settings, including the number of players, number of actions, number of algorithms used by players, and total iterations. A compute unit executed this task in order to generate the results and save them in the database.

The basic workflow functioned smoothly. However, the execution time of the experiment could have easily increased if the scale of tournament increased. Thus, the distributed approach was introduced to ease this problem by using multiple CPU cores to calculate in parallel. In practice, we created 8 threads as workers that could operate independently, based on the maximum number of CPU cores. First, those workers generated all game files and put associated tasks in a task queue. After the workers finished generating the game files, they took tasks from the task queue and executed them until the task queue was empty. Compared to the basic workflow, this approach could effectively reduce execution time by taking advantage of multi-threading. It also provides the possibility to run larger scales of experiment.

### 5.3.4 Limitations of GAMUT

To generate the heatmap of a $m$-action $n$-player tournament presented in Figure 6.3, the 8-player and 9-player 4-action games were not run. One reason for this was the limitation of the test framework discussed in Section 5.2.2, explaining that the present research did not focus on a larger scale tournament when an agent was unable to explore each possible action even once. The other reason was that the game generation relied on GAMUT, a Java program which can only generate a game instance under a fixed game size. The maximum game size is limited by the maximum virtual memory size (8 GB). Although the maximum virtual memory size can be configured to a higher number if the system's physical memory is increased, in this case the available physical memory was viewed as a limitation of running a larger tournament. For visualization reasons in the heatmap, the testing data of 8-player 4-action games was applied to the data of both 9-player 4-action and 10-player 4-action.

### 5.3.5 Data analysis

*sqlite3* was used as a database to store all raw data from the computation results. More specifically, *sqlite3* is a single database file that can be easily created and used to store data in addition to being compatible with SQL syntax. In the database schema, the format of each observation was defined, including game type, algorithm used by each player, accumulated rewards received by players, total iterations, number of actions, and number of players. The format ensured that we were later able to manipulate data.

After the raw data was stored in the *sqlite3* database, we entered it into an *R* program, which is used for statistical analysis. In R, the raw data can be aggregated and processed given the research interests. For instance, we can categorize data based on several criteria, including number of players, number of actions, set of game types, or two algorithm types – MAL and MAB. The basic process for grouping data is to average all data points in the same group. However, this approach may be biased and ignore the distribution of the original data. Therefore, different statistical analyses for distribution comparison are introduced in the next section.

## 5.4 Statistical analysis

Two statistical approaches are introduced in this section. In general, the Kolmogorov-Smirnov test is used to determine whether the difference between two data distributions is significant. This test avoids the disadvantage of using mean values that do not consider the data distribution. For example, two similar mean values of the average rewards may be interpreted as the same performance, even though their data distributions are different. On the other hand, Spearman's rank correlation helps inspect the correlation between two variables. In this case, each approach serves a different purpose and manipulates raw data in different ways. Therefore, we describe how we applied them in our data analysis.

### 5.4.1 Kolmogorov-Smirnov test

To avoid incorrectly interpreting the data results, the Kolmogorov-Smirnov (KS) test is used to identify the magnitude of difference between two distributions. A KS test uses two distributions as inputs and converts them into ECDFs (Empirical Cumulative Distribution Function) that represent data in a probability space. After comparing two ECDFs, the test finds a maximum distance between two ECDFs in order to indicate how large the difference is between two distributions. More detail about the KS test is located in [16].

Below, Figure 5.2 demonstrates a KS test as applied to two data distributions in terms of normalized average rewards. The vertical dashed line represents the maximum distance between two ECDFs. More specifically, a KS test obtains the value of the maximum vertical distance and another $p$-value (significance level) in order to determine if the maximum difference of two distributions is large enough. In fact, Zawadzki [14] has set the significance level as 0.05 in the KS test in order to discern whether two distributions of algorithm performance are significantly different.

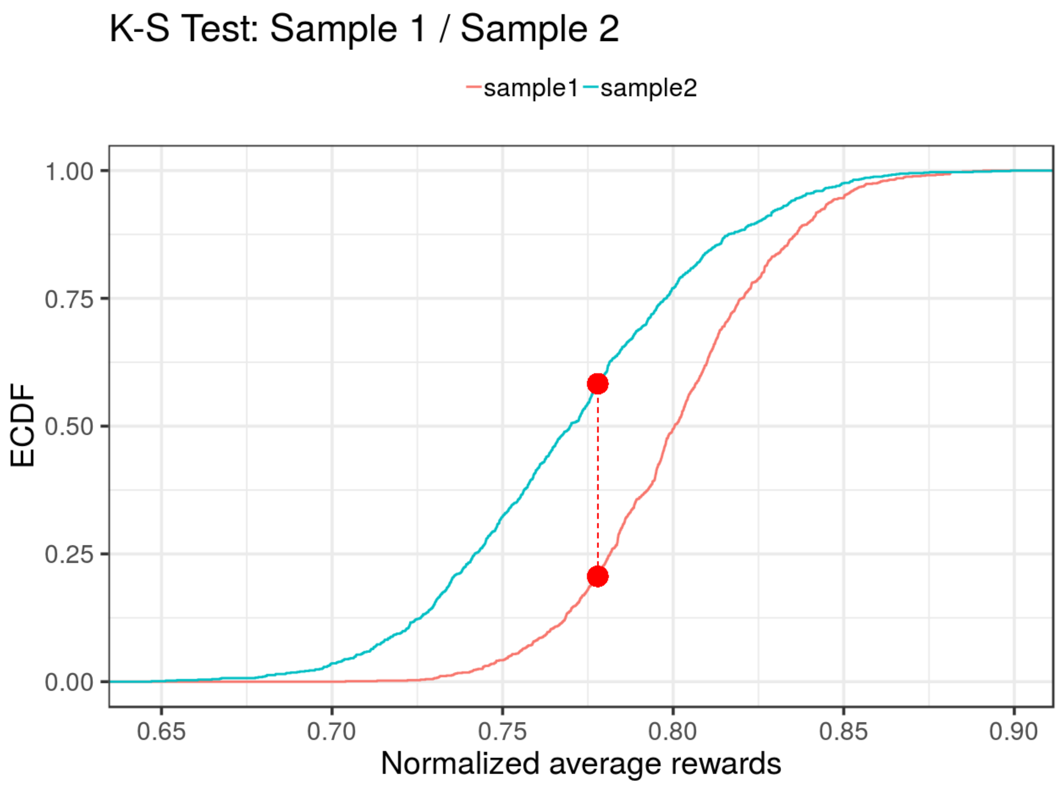

FIGURE 5.2: KS-test example

In our research, the KS test was applied to determine if the data distributions between MAL and MAB were significantly different for various numbers of actions and players. In terms of the test process, for the $m$-action $n$-player tournament, we first separated the results of the tournament, which included average rewards of all games and all algorithm pairs, into the MAL group or the MAB group. Each group of data was comprised of all data subsets, with different numbers of actions and players. Next, we applied KS tests to evaluate any two subsets of data with the same number of actions and players. The results of each KS test generated a value of the maximum distance of data distributions from two subsets as well as a $p$ value of the significance level of the test. Lastly, we analyzed these values by comparing them with the given significance level in order to indicate whether the performance of MAL and MAB were significantly different for various numbers of actions and players.

### 5.4.2 Spearman's rank correlation

Spearman's rank correlation was used to examine whether two variables demonstrated a monotonic relationship. Figure 5.3 illustrates examples of three possible relationships between two variables: monotonically increasing, monotonically decreasing, or non-monotonic. Given a possible kind of relationship, called alternative hypothesis, Spearman's rank correlation calculated how well two variables matched the given relationship. For example, the literature [14] has used Spearman's rank correlation to examine whether a monotonic relationship exists between game size and average rewards of an algorithm. The details of how Spearman's is used in the correlation analysis are located in [21].

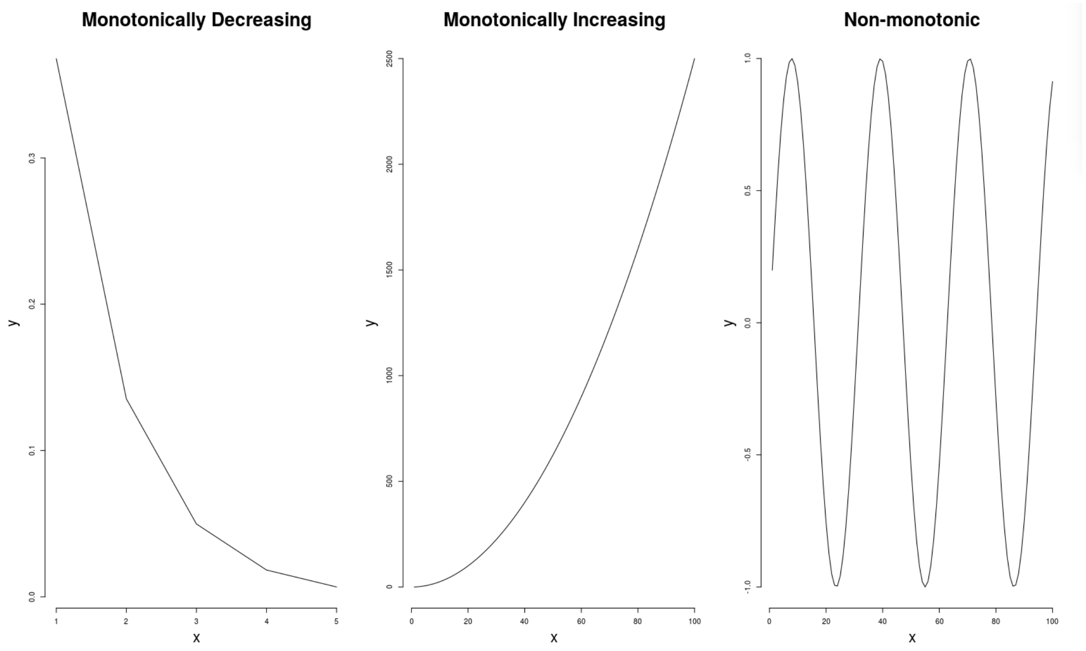

FIGURE 5.3: Examples of monotonic and non-monotonic relationships

In our statistical analysis, we emphasized three hypotheses to test correlations among variables, as noted below. The first and second hypotheses examined whether the number of players or actions served a role in degrading the performance of MAL or MAB. The third hypothesis constitutes our research question.

1. An increasing number of players may lead to decreasing average rewards of MAL and MAB at 2-action, 3-action, or 4-action games.
2. An increasing number of actions may lead to decreasing average rewards of MAL and MAB.
3. An increasing number of players may lead to decreasing the difference of average rewards between MAL and MAB.

In order to run Spearman's rank correlation for each hypothesis, we pre-processed our raw data in the tournament, which was later fed into Spearman's rank correlation. The complete test process of the first hypothesis was as follows:

1. Retrieve a subset of raw data in the $m$-action $n$-player tournament, which only used data in 2-action games.
2. Categorize the subset data into MAL and MAB groups.
3. Average the grouped data based on a numbers of players.
4. Run Spearman's rank correlation with two input variables and the alternative hypothesis "less" (number of players and the average rewards).
5. Repeat step 1 by 3-action and 4-action.
6. Analyze the $p$-value in each test result.

In step 4, we specified the type of alternative hypothesis "less," which indicates testing a monotonically decreasing relationship. In step 6, if the $p$ value was lower than the significance level 0.05, the tested hypothesis held. At the end of the process, we collected six results given the numbers of actions and the groups of MAL and MAB.

Similar to the first hypothesis, in the second hypothesis, we grouped the raw data based on a number of actions and the average rewards in MAL and MAB. However, we did not separately run tests for different numbers of players. Therefore, we only obtained two test results from MAL and MAB.

The last hypothesis is also similar to the first hypothesis test. The added step in this case is the subtraction of average rewards between MAL and MAB. Given the combined data of MAL and MAB, this hypothesis only generated three results by the number of actions.

In summary, we have described Spearman's rank correlation and how we used it to test our hypotheses. Especially for the third hypothesis, the test results are directly associated with our research question. As such, more detailed discussion of this test result is provided in the final chapter.

# Chapter 6

# Results

## 6.1 Performance

This section presents the test results from the two tournaments that were conducted on the testbed. One set of results comes from a tournament with various 2-player and 2-action game types, while the other set includes *n*-player and *m*-action games types. Each tournament included all available algorithms in the test bed.

### 6.1.1 Tournament for 2-player 2-action game types

Figure 6.1 [1]displays a heatmap of the test results from 29 kinds of 2-player, 2-action game types for all algorithms. Each cell indicates the performance of an algorithm played in a game type. Darker cell colors represent lower average payoffs, while brighter cell colors represent higher average payoffs. In fact, the *Random* algorithm performed worse in most game types than the other algorithms.

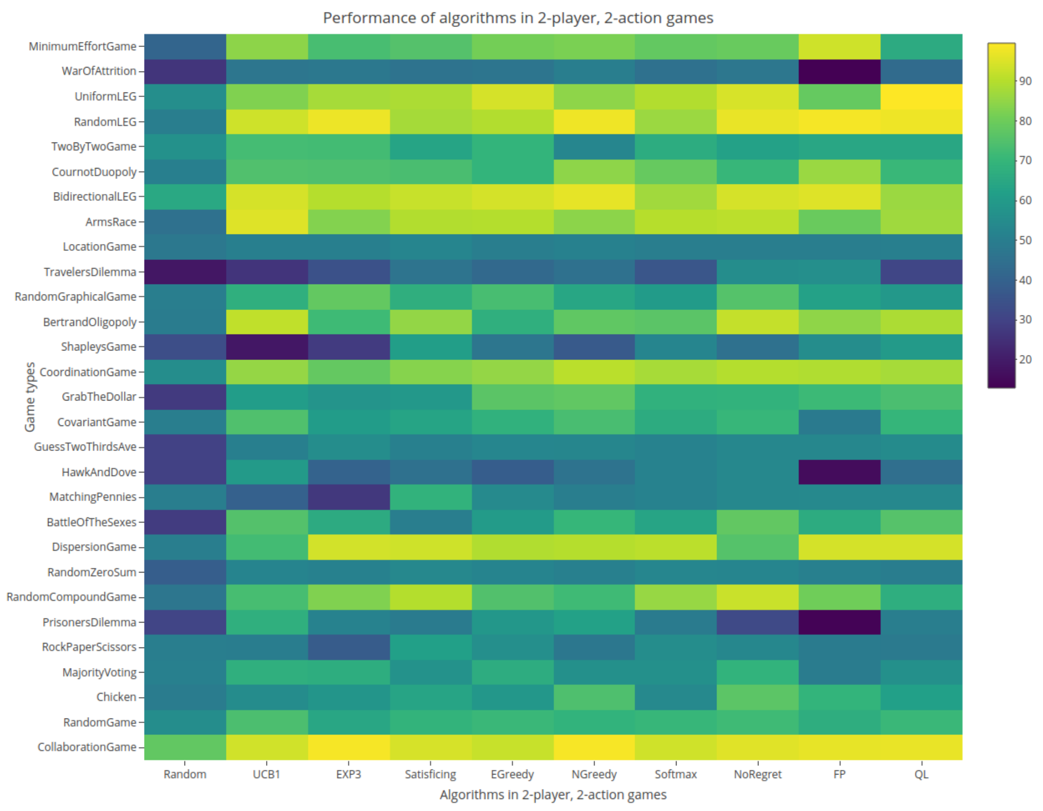

FIGURE 6.1: Performance of all algorithms in the 2-player 2-action tournament

After grouping all algorithms except Random into MAL or MAB algorithms from the previous heatmap, another heatmap was produced. As Figure 6.2 [2]indicates, this heatmap represents the performance of MAL and MAB algorithms. Depending on the game type, MAL and MAB algorithms may outperform each other.

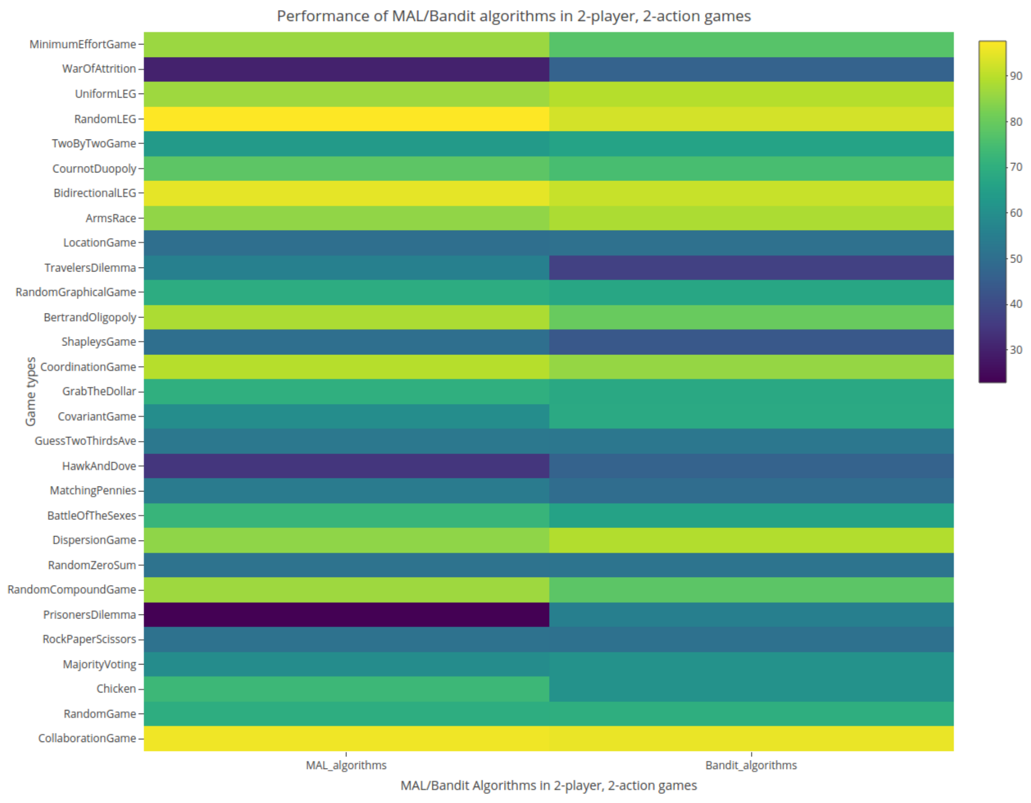

FIGURE 6.2: Performance of MAL and MAB algorithms in the 2-player 2-action tournament

### 6.1.2 The tournament for $n$-player $m$-action game types

This section discusses the experiment results from the tournament for $n$-player, $m$-action game types. Figure 6.3 [3]displays a heatmap for the 12 game types of supported $n$-player, $m$-action games for all available algorithms. Since this research focuses on the performance of algorithms as increasing the number of players and actions, each cell represents the average performance of an algorithm over 12 game types at $n$ players and $m$ actions.

The y-axis of the figure measures the number of players, $n$, ranging from 2 to 10, while the x-axis represents all available algorithms played at the number of actions, $m$, ranging from 2 to 4. The cells at 9-player 4-action and 10-player 4-action games are the same as cells at 8-player 4-action games, which are manually made for visualization purpose. The lack of tests for running these games come from a limitation

[1]The labels on the y-axis in Figure 6.1 can be read off by zooming in. The plot is rendered in a vector-based graphics format.

[2]The labels on the y-axis in Figure 6.2 can be read off by zooming in. The plot is rendered in a vector-based graphics format.

of GAMUT. This limitation is considered in more detail in Section 5.3.4.

The heatmap provides an overview of performance by spotting it along the x-axis or y-axis. For instance, more dark cells appeared at 4-action (ten columns in the rightmost) than at 3-action (ten columns in the middle). Moreover, more dark cells appeared in the middle 10 columns than in the left-most 10 columns. More dark cells along the x-axis indicates that an increasing number of actions may lead to degrading performance for some algorithms. Similarly, the cell color becomes darker from the bottom row to the top row (2-player to 10-player), which suggests that an increasing number of players may lead to degrading performance for some algorithms.

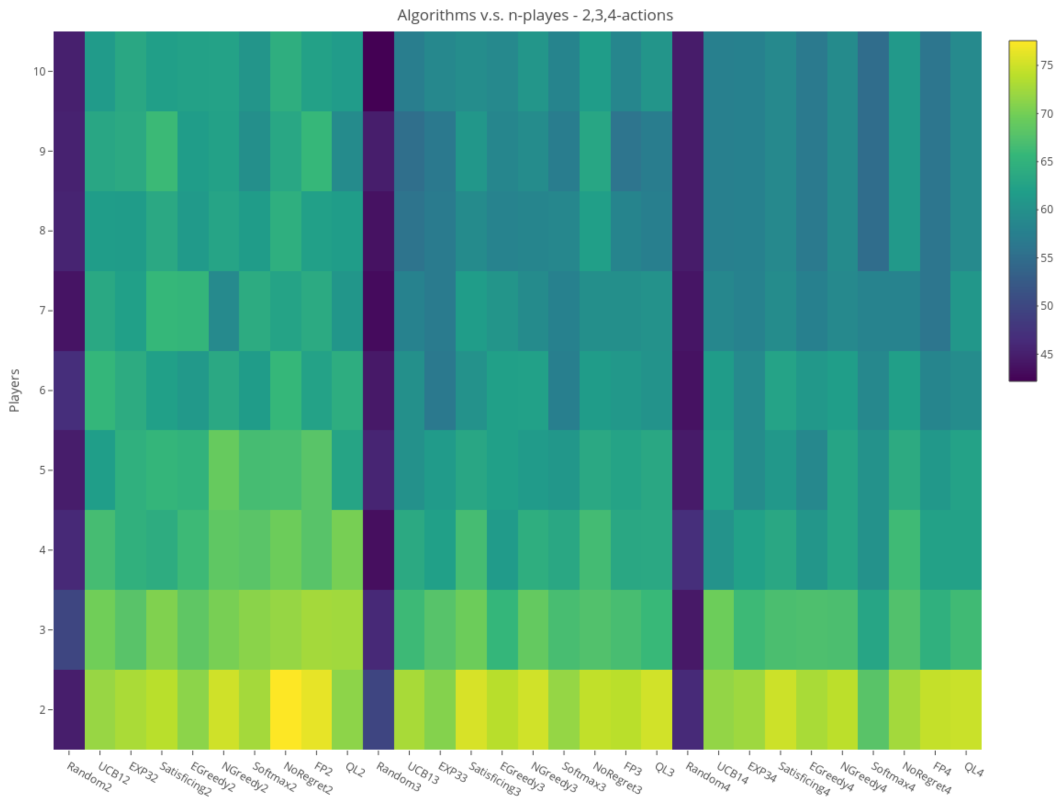

FIGURE 6.3: Performance of all algorithms in the n-player m-action tournament.

[3]The labels on the x-axis in Figure 6.3 can be read off by zooming in. The plot is rendered in a vector-based graphics format.

To precisely compare the performance of algorithms over the $n$-player $m$-action tournament, the raw data of Figure 6.3 is presented in Table 6.1, 6.2 and 6.3 for 2-action, 3-action, and 4-action games.

TABLE 6.1: Performance for n-player, 2-action games

| | Random | UCB1 | EXP3 | Satis. | $\epsilon$-greedy | $N$-greedy | Softmax | NoRe. | FP | QL |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 44.84 | 71.89 | 72.98 | 73.88 | 71.26 | 75.07 | 72.64 | 77.57 | 76.39 | 71.33 |
| 3 | 49.83 | 69.82 | 67.92 | 70.68 | 68.44 | 70.23 | 71.20 | 71.81 | 72.62 | 72.46 |
| 4 | 46.19 | 66.67 | 64.71 | 64.06 | 66.18 | 68.51 | 68.03 | 69.55 | 67.86 | 70.17 |
| 5 | 44.74 | 61.87 | 64.61 | 65.40 | 64.98 | 69.11 | 66.65 | 66.78 | 68.00 | 62.85 |
| 6 | 46.76 | 65.48 | 63.96 | 62.09 | 61.06 | 63.44 | 61.57 | 65.71 | 62.53 | 64.17 |
| 7 | 43.92 | 63.40 | 62.10 | 65.69 | 65.39 | 58.96 | 63.98 | 62.59 | 63.73 | 60.76 |
| 8 | 45.59 | 61.75 | 61.60 | 63.54 | 61.23 | 62.72 | 61.55 | 64.48 | 62.18 | 61.62 |
| 9 | 45.37 | 63.18 | 63.67 | 66.06 | 61.65 | 62.32 | 59.78 | 63.21 | 65.73 | 59.24 |
| 10 | 45.14 | 61.38 | 63.34 | 61.98 | 62.28 | 62.49 | 60.61 | 64.37 | 62.29 | 61.51 |

Raw data in terms of average rewards can be used to investigate the performance change for any specific algorithm at different numbers of players and actions. For example, as Table 6.1 shows, *NoRegret*'s average rewards are from 77.57, 77.81,..., to 64.37 as player numbers are from 2 to 10 in 2-action games. This presents a trend that the performance of *NoRegret* degrades when the number of players increases. This trend is generally applied to other MAL and MAB algorithms except for small amount of data points, e.g. a trend of a decreasing average reward as long as an increasing number of player holds for FP algorithm from 2-player to 8-player, but it was broken at 9-player (65.73), which is higher than average reward at 8-player and 10-player (62.18 and 62.29).

The table can also be used to find top two best algorithms based on their average rewards, which the total number of MAL algorithms is two. At 2-player games in the first row, the top two algorithms are *NoRegret* and FP, which score 77.57 and 76.39 individually. However, the top two algorithms change to *NoRegret* and EXP3 as the player numbers increase to 10-player, which score 64.37 and 63.34. *NoRegret* stays in the top two algorithms in both cases, while FP is replaced with EXP3 at 10-player case. For the rest of cases with different player numbers, the FP with the second highest average rewards is also replaced by one of other MAB algorithms.

In addition, the performance comparison between two single algorithms can be conducted by picking two columns in a table. For instance, given two columns of $\epsilon$-greedy and N-greedy performance in Table 6.1, the average payoffs of N-greedy are higher than those of $\epsilon$-greedy, except for the case of 7-player (65.39, 58.96). This result suggests that N-greedy may dominate $\epsilon$-greedy at 2-action $n$-player games.

TABLE 6.2: Performance for n-player, 3-action games

| | Random | UCB1 | EXP3 | Satis. | $\epsilon$-greedy | $N$-greedy | Softmax | NoRe. | FP | QL |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 49.78 | 72.87 | 70.87 | 75.52 | 73.83 | 75.12 | 71.76 | 74.34 | 73.97 | 75.19 |
| 3 | 46.18 | 66.16 | 67.79 | 69.58 | 65.56 | 68.88 | 66.80 | 67.46 | 66.80 | 65.86 |
| 4 | 43.40 | 63.68 | 62.05 | 66.67 | 61.40 | 64.38 | 63.35 | 66.54 | 63.35 | 63.52 |
| 5 | 45.70 | 59.95 | 61.45 | 63.17 | 62.07 | 61.40 | 60.81 | 63.48 | 62.58 | 63.35 |
| 6 | 44.37 | 59.82 | 56.54 | 60.09 | 62.07 | 62.26 | 57.45 | 61.61 | 60.94 | 60.30 |
| 7 | 43.12 | 57.90 | 56.77 | 61.76 | 60.53 | 59.25 | 57.74 | 59.76 | 59.76 | 60.14 |
| 8 | 43.74 | 55.87 | 56.86 | 59.18 | 57.99 | 58.33 | 58.50 | 62.00 | 58.09 | 57.48 |
| 9 | 44.87 | 55.18 | 56.60 | 60.87 | 58.38 | 59.36 | 57.13 | 63.06 | 55.93 | 57.06 |
| 10 | 42.17 | 57.21 | 58.73 | 59.47 | 58.89 | 60.64 | 58.14 | 61.74 | 58.34 | 60.49 |

A similar analysis is applied to raw data in Table 6.2, which observes if there is a pattern of degraded performance as player number increases and top two algorithms at 3-action games. First, the raw data of each MAL or MAB algorithm shows a trend of a decreasing average rewards as a number of player increases, though the trend may not be applied to a higher number of players. For instance, average rewards of -greedy keep similar values at 8-player (57.99), 9-player (58.38), and 10-player (58.89) games, even there is an opposite trend for N-greedy (58.33, 59.36, 60.64). Secondly, two best algorithms at 2-player games are Satisficing and QL with two highest average rewards, 75.52 and 75.19 individually, while two best algorithms become *NoRegret* and N-greedy at 10-player games. In 3-action games, there is no single algorithm can stay in the top-two algorithm list at any number of players.

TABLE 6.3: Performance for n-player, 4-action games

| | Random | UCB1 | EXP3 | Satis. | $\epsilon$-greedy | $N$-greedy | Softmax | NoRe. | FP | QL |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 46.28 | 71.65 | 72.30 | 74.93 | 72.91 | 74.08 | 67.96 | 72.57 | 74.51 | 74.77 |
| 3 | 44.31 | 69.58 | 66.17 | 67.01 | 67.23 | 67.10 | 62.86 | 67.47 | 64.65 | 66.35 |
| 4 | 46.86 | 60.28 | 62.20 | 63.36 | 60.65 | 62.93 | 60.08 | 66.28 | 62.34 | 62.30 |
| 5 | 44.49 | 62.25 | 59.36 | 61.05 | 58.68 | 62.81 | 60.11 | 63.85 | 60.98 | 62.54 |
| 6 | 43.74 | 61.59 | 59.02 | 62.58 | 61.00 | 61.84 | 58.68 | 61.98 | 58.33 | 59.32 |
| 7 | 44.03 | 58.62 | 57.92 | 59.36 | 57.61 | 58.91 | 58.07 | 58.00 | 56.08 | 60.86 |
| 8 | 44.62 | 57.58 | 57.72 | 58.82 | 56.71 | 59.19 | 54.96 | 61.08 | 56.15 | 59.06 |

It is evident that a trend mentioned before is also applied to the raw data in Table 6.3. For instance, *NoRegret*'s average rewards decrease from 72.57 to 58.00 as a number of player increases from 2-player to 7-player in the table. Given the observations at 2-action, 3-action and 4-action in three tables, they suggest a possible relation between an increasing number of player and decreasing average rewards of an either MAL or MAB algorithm. Yet, these examples only indicate how these tables can be used to demonstrate that possible patterns exist. They do not lead to any conclusion until developing a more comprehensive analysis.

### 6.1.3 Grouped data for MAL and MAB algorithms

To compare the performance between MAL and MAB algorithms, the experiment data in the previous $n$-player $m$-action tournament is grouped into MAL and MAB categories. Columns belonging to the same categories have been combined, such as MAL including *NoRegret* and *FP*, and the values have been averaged for the same number of players. Figure 6.4 displays a heatmap of the average rewards of MAL and MAB in the $n$-player $m$-action tournament. The figure reveals that the upper right cells are darker than the lower left cells, suggesting an increased number of actions or players, all of which may degrade performance for MAL and MAB algorithms. The raw data of this heatmap is included for further analysis in Table 6.4.

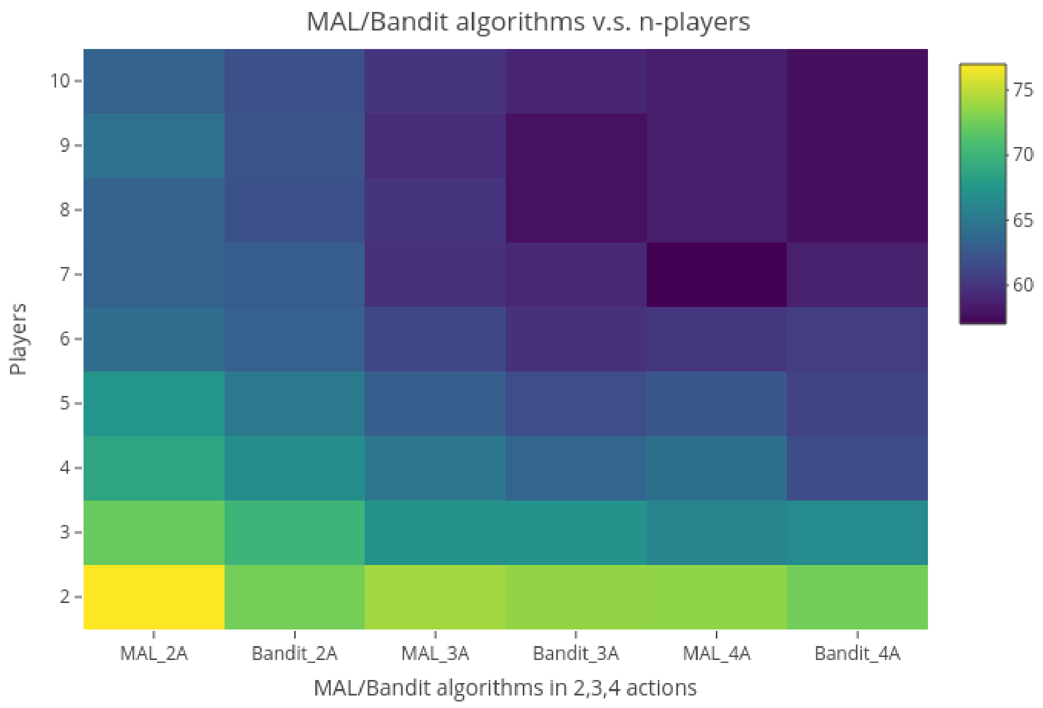

FIGURE 6.4: Performance of MAL and MAB in the n-player m-action tournament

TABLE 6.4: Performance for n-player, m-action games by MAL and MAB

| | MAL_2A | Bandit_2A | MAL_3A | Bandit_3A | MAL_4A | Bandit_4A |
|---|---|---|---|---|---|---|
| 2 | 76.98 | 72.72 | 74.16 | 73.60 | 73.54 | 72.66 |
| 3 | 72.21 | 70.11 | 67.13 | 67.23 | 66.06 | 66.61 |
| 4 | 68.71 | 66.90 | 64.94 | 63.58 | 64.31 | 61.68 |
| 5 | 67.39 | 65.07 | 63.03 | 61.74 | 62.41 | 60.97 |
| 6 | 64.12 | 63.11 | 61.27 | 59.79 | 60.15 | 60.57 |
| 7 | 63.16 | 62.90 | 59.76 | 59.16 | 57.04 | 58.76 |
| 8 | 63.33 | 62.00 | 60.04 | 57.74 | 58.62 | 57.72 |
| 9 | 64.47 | 62.27 | 59.50 | 57.80 | 58.62 | 57.72 |
| 10 | 63.33 | 61.94 | 60.04 | 59.08 | 58.62 | 57.72 |

To determine the exact difference between MAL and MAB performance, a subtraction operation was made for any two cells with the same condition in Table 6.4. Based on thesubstration results, Figure 6.5, 6.6, and 6.7 present not only the raw data of Table 6.4, but also the difference between MAL and MAB for 2-action, 3-action and 4-action, respectively. In these three figures, the x-axis represents the number of players. The left y-axis represents the average payoffs of MAL and MAB, while the right y-axis represents the difference payoffs between MAL and MAB. Therefore, each figure shows the performance change of both MAL and Bandit algorithms and their difference over an increasing number of players. A detailed discussion of each figure and the relation behind them is provided in Section 7.1.1.

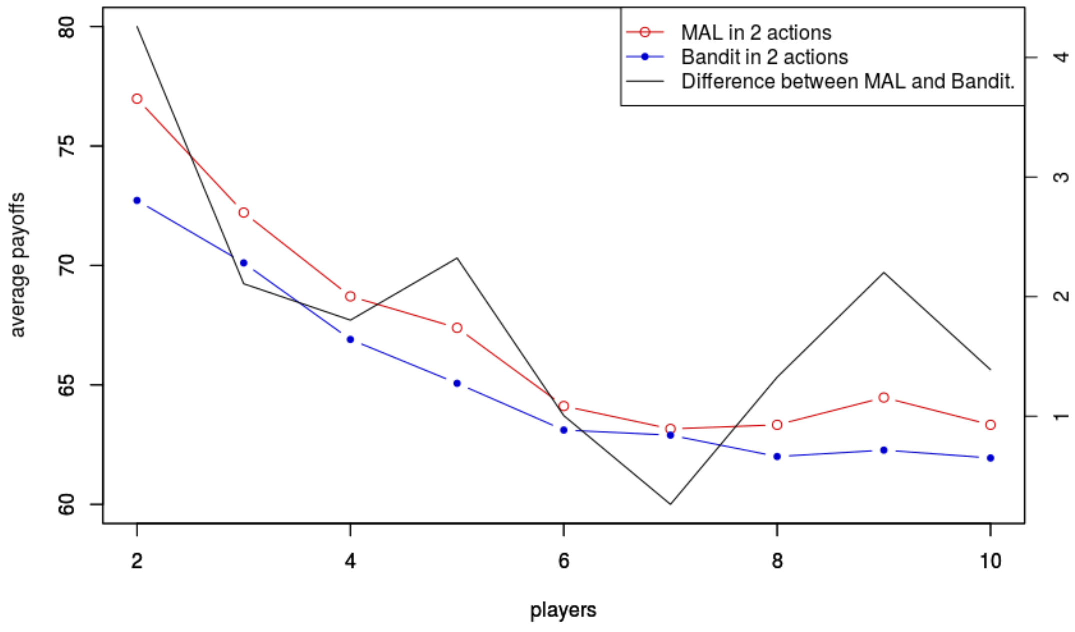

FIGURE 6.5: Performance comparison between MAL and MAB for n-player, 2-action games

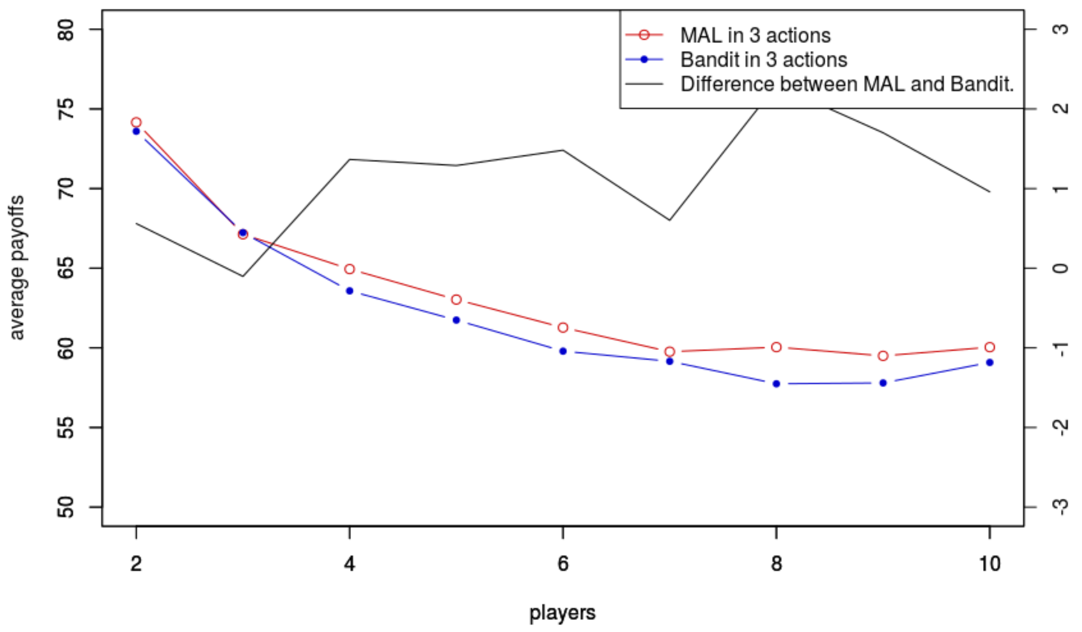

FIGURE 6.6: Performance comparison between MAL and MAB for n-player, 3-action games

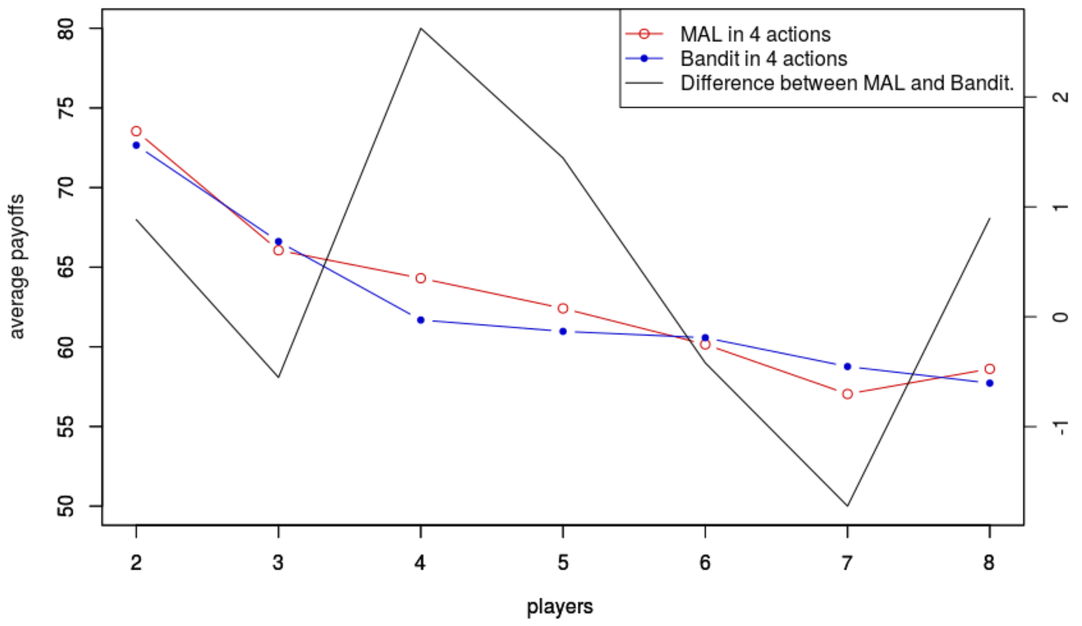

FIGURE 6.7: Performance comparison between MAL and MAB for n-player, 4-action games

### 6.1.4 Running time in the $n$-player $m$-action tournament

To evaluate the amount of computing time required for running an $n$-player $m$-action tournament, we tested the relation between the number of game instances in the tournament and their execution time. In general, the more instances a tournament ran, the more execution time was needed. Furthermore, higher numbers of actions increased the execution time as larger game sizes lead to more calculations for algorithms. Figure 6.8 depicts the execution time as increasing exponentially when the number of players increases. In addition, more actions required more execution time. This result can be used to estimate the required execution time for different tournament scales in the future.

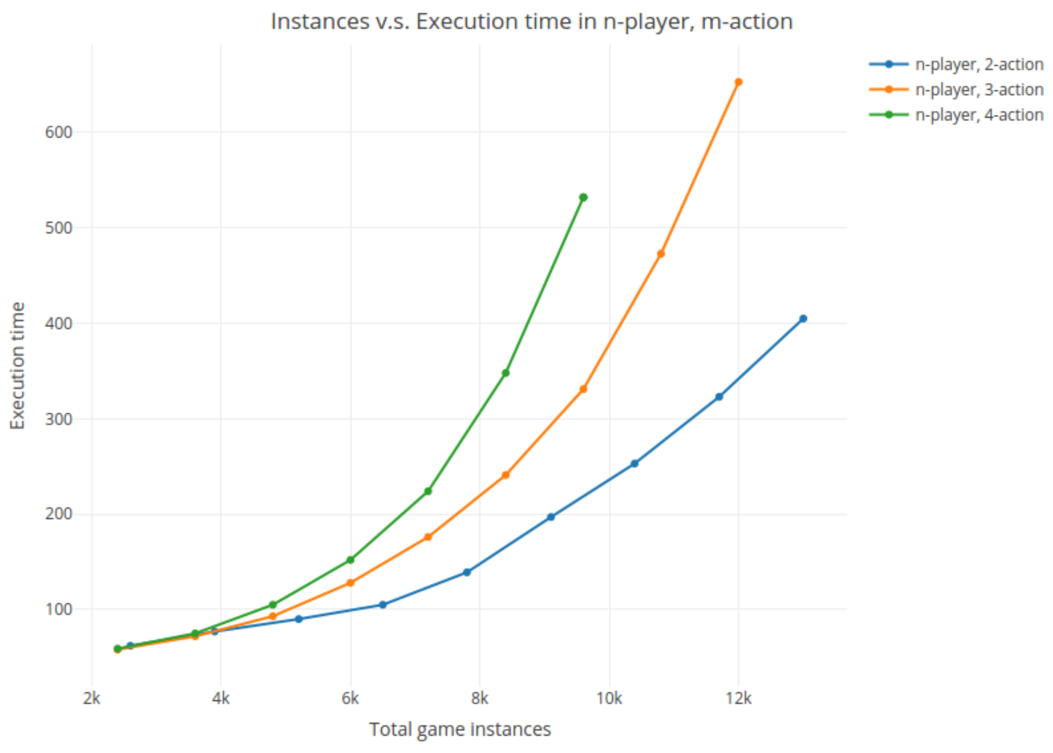

FIGURE 6.8: Running cost for a n-player m-action tournament

## 6.2 Statistical analysis

### 6.2.1 Kolmogorov–Smirnov test

After conducting the KS test analysis, as described in Section 5.4.1, Table 6.5 provides the analysis results based on testing data of the $m$-action $n$-player tournament for MAL and MAB algorithms. Each KS test evaluated two distributions in terms of the average rewards of MAL and MAB algorithms at any number of actions and players. A single test generated results including *statistics* and $p$-value. The $p$-value is used to examine whether two distributions are significantly different when the $p$-value is less than the significant level of 0.05. In this case, the results indicate whether two distributions of MAL and MAB algorithms are significantly different.

TABLE 6.5: KS-test for average rewards of MAL and MAB in n-player m-action tournament

| | 2-action | | 3-action | | 4-action | |
|---|---|---|---|---|---|---|
| | statistic | p.value | statistic | p.value | statistic | p.value |
| 2 | 1.876E-01 | 8.564E-13 | 1.119E-01 | 1.739E-04 | 1.199E-01 | 4.326E-05 |
| 3 | 1.115E-01 | 5.564E-07 | 1.008E-01 | 2.288E-05 | 8.472E-02 | 6.451E-04 |
| 4 | 9.231E-02 | 2.063E-06 | 8.185E-02 | 9.050E-05 | 7.738E-02 | 2.616E-04 |
| 5 | 8.319E-02 | 1.673E-06 | 5.250E-02 | 1.166E-02 | 7.119E-02 | 1.557E-04 |
| 6 | 1.012E-01 | 3.234E-11 | 6.339E-02 | 2.464E-04 | 8.968E-02 | 2.997E-08 |
| 7 | 9.513E-02 | 1.490E-11 | 6.378E-02 | 4.840E-05 | 4.821E-02 | 4.599E-03 |
| 8 | 9.138E-02 | 3.719E-12 | 7.760E-02 | 3.086E-08 | 6.019E-02 | 1.588E-14 |
| 9 | 8.181E-02 | 5.266E-11 | 6.772E-02 | 4.055E-07 | 6.019E-02 | 1.588E-14 |
| 10 | 6.626E-02 | 3.877E-08 | 7.244E-02 | 6.205E-09 | 6.019E-02 | 1.588E-14 |

As Table 6.5 displays, all $p$-values in all KS-tests are less than the significant level, which indicates two distributions of average rewards between MAL and MAB algorithms are significantly different. If the raw data in the previous Table 6.4 is examined again, in some cases two mean values in terms of average rewards between MAL and MAB algorithms are close, such as 59.76 (7-player, MAL 3A) and 59.16 (7-player, MAB 3A). However, the $p$-value of the corresponding KS test at 3-action 7-player is 4.84E-05. Therefore, while the test results of MAL and MAB algorithms in terms of mean values are close, their distributions are different and can be inspected through other analytical methods, e.g. comparing the ECDF of two distributions.

### 6.2.2 Spearman's rank correlation

In Section 5.4.2, three kinds of hypothesis tests were introduced for evaluation by Spearman's rank correlation. The analysis results for each hypothesis are displayed in Table 6.6, 6.7, and 6.8. Following the significance level suggests that the first hypothesis holds, the second hypothesis is rejected, and the third hypothesis partially holds. As these results can be used to examine the research questions, they are discussed in detail in Section 7.1.2.

TABLE 6.6: Spearman's rank tested the alternative hypothesis that an increasing number of players results in a decreasing average rewards of MAL and MAB at the 2-action, 3-action and 4-action games.

| | statistic | p.value | estimate |
|---|---|---|---|
| MAL, 2-action | 2.160E+02 | 6.914E-03 | -8.000E-01 |
| MAB, 2-action | 2.380E+02 | 2.480E-05 | -9.833E-01 |
| MAL, 3-action | 2.300E+02 | 6.559E-04 | -9.167E-01 |
| MAB, 3-action | 2.320E+02 | 3.748E-04 | -9.333E-01 |
| MAL, 4-action | 1.100E+02 | 1.389E-03 | -9.643E-01 |
| MAB, 4-action | 1.120E+02 | 1.984E-04 | -1.000E+00 |

TABLE 6.7: Spearman's rank tested the alternative hypothesis that an increasing number of actions results in a decreasing average rewards of MAL and MAB.

| | statistic | p.value | estimate |
|---|---|---|---|
| MAL | 8.000E+00 | 1.667E-01 | -1.000E+00 |
| MAB | 8.000E+00 | 1.667E-01 | -1.000E+00 |

TABLE 6.8: Spearman's rank tested the alternative hypothesis that an increasing number of players leads to a decreasing difference of average rewards between MAL and MAB at 2-action, 3-action, and 4-action games.

| | statistic | p.value | estimate |
|---|---|---|---|
| 2-action | 1.000E+02 | 2.401E-02 | -7.857E-01 |
| 3-action | 1.600E+01 | 9.669E-01 | 7.143E-01 |
| 4-action | 6.200E+01 | 4.198E-01 | -1.071E-01 |

# Chapter 7

# Discussion and Conclusion

## 7.1 Discussion

### 7.1.1 Observation of the $n$-player $m$-action tournament

Figures 6.5, 6.6, and 6.7 display the individual test results of both MAL and MAB algorithms in $n$-player 2-action, 3-action and 4-action games. By comparing the data points at each action number, possible patterns were identified as described below.

- In $n$-player 2-action games, an increasing number of players leads the convergence of performance between MAL and MAB algorithms from 2-player to 7-player, even though the difference began to diverge from 8-player to 10-player. Among these games, MAL always outperformed MAB.

- In $n$-player 3-action games, MAL outperformed MAB in all cases. However, if the small difference data point is defined as the data point with a difference less than 1, 4 data points with players at (2,3,7,10) satisfy the criteria. This result differs from the result of 2-action games where the data points with players at (7,8,9,10) with small differences only appear in more player games.

- In $n$-player 4-action games, most data points are in the small difference except for the single data point with player number at (2). In addition, the test results are the first indication of 3 data points with player numbers at (3,6,7) demonstrating that MAB outperformed MAL.

Despite the observations for 2-action, 3-action, and 4-action games, it cannot yet be concluded that MAB outperforms MAL when the number of players and the number of actions are increased. However, increasing the number of players does have a positive effect on leading the performance of MAB to converge with the performance of MAL, as evidenced by the first observation in 2-action games, particularly in the range of player numbers 2 to 7.

### 7.1.2 Statistical analysis

Based upon the KS-test, Table 6.5 reveals that all $p$-values were less than the significant level of 0.05. This test result suggests that the distribution of average rewards in MAL and MAB algorithms are significantly different in the $m$-action $n$-player tournament, although the mean values may be similar. However, this result can be explained when each mean value (average rewards) comes from the average result by playing an algorithm in many game types and encountering different algorithms in the tournament. Hence, the distribution of an algorithm is different from the distribution of any other algorithm.

Given the Spearman's rank correlation analysis, the results of three alternative hypotheses are presented in the Tables 6.6, 6.7, and 6.8. For the first hypothesis test, all $p$-values in Table 6.6 were less than the significant level 0.05 at 2-action, 3-action, and 4-action games. This test result indicates that an increasing number of players leads to a decreasing average reward for both MAL and MAB algorithms at $m$-action games. Next, the second hypothesis did not hold because the higher $p$-values in Table 6.7 exceeded the significant level. This result refutes its hypothesis, which held that an increasing number of actions led to a decreasing average reward in both MAL and MAB. In terms of the last hypothesis, Table 6.8 revealed that the hypothesis only held at 2-action games instead of 3-action and 4-action games.

From the results above, we derive that the average rewards of MAL and MAB decrease as the number of players increases. A decreasing performance is expected when an increasing number of players lead an agent to explore more actions that are more difficult for MAL and MAB algorithms to find an action with higher rewards. However, an increasing number of actions did not demonstrate a high correlation with decreasing performance, which may be a result of the tournament only testing the number of actions from 2 to 4, which was insufficient to find high correlation.

In fact, the third hypothesis, which suggested that the difference of average rewards between MAL and MAB algorithms became smaller when the number of players increased, was only confirmed at 2-action games. This outcome is matched with the previous observation about Figure 6.5, that the performance of MAB converges with the performance of MAL in 2-action games from players 2 to 7.

## 7.2 Conclusion

For the purpose of this research, we have designed a test bed to conduct an $m$-action $n$-player tournament for MAL and MAB algorithms. By combining the observations of the $m$-action $n$-player tournament in Section 7.1.1 and its statistical analysis in Section 7.1.2, we conclude that the results partially support our main research hypothesis, as presented in Table 1.1.

First, the average rewards of MAL algorithms are larger than MAB algorithms at 2-player games (see Figures 6.5, 6.6, and 6.7). To be more precise, the raw data from $n$-player $m$-action tournament, as detailed in Table 6.4, indicates that MAL algorithms outperform MAB algorithms given their average payoffs at 2-player (the first row). This result matches the description in the first row of Table 1.1. Thus, at 2-player games MAL algorithms, by using opponent information, help agents receive higher average rewards than MAB algorithms.

When the number of players increases in these figures, the difference of average rewards decreases between MAL and MAB algorithms, or the performance of MAB algorithms degrades more slowly than MAL. However, this pattern was only satisfied at 2-action games and was limited to the player range of 2 to 7. For 3-action and 4-action games, we observe the performance of MAL and MAB algorithms as similar when they outperformed each other based on different numbers of players. However, the evidence is insufficient to suggest that the performance of MAL and MAB are similar or exactly the same for these cases in 3-action and 4-action games.

Therefore, we conclude that the hypothesis is not held for any of the conditions but does hold under a given number of actions and a specific range of players. As such, MAB algorithms remain promising for application in a multi-agent environment when the boundary of application is known in advance.

## 7.3 Future research

Potential topics to be investigated in the future include a more accurate boundary, algorithm level comparison, and game categories. To begin, this research has evaluated a defined scale of tournament and identified a valid range to support the hypothesis. However, many variables have yet to be tested, such as different numbers of iterations or algorithm-specific parameters. Thus, future research should examine more combinations of variables in order to identify a clearer hypothesis boundary.

In addition, although both MAL and MAB grouped data have been analyzed, the comparison can be changed from group level to algorithm level. For example, the best performance algorithm in the MAL group compares with the best performance algorithm in the MAB group. Even though this comparison has bias toward specific algorithm comparisons, the comparison helps determine how well the hypothesis fits with a particular algorithm. Lastly, this research has not separated game types into different categories, such as cooperation games or zero-sum games. This categorization of game types may help identify which multi-agent environments would be more suitable to apply the hypotheses.

# References

[1] G. W. Brown. “Iterative solution of games by fictitious play”. In: *Activity Analysis of Production and Allocation* 374–376 (1951).

[2] J. Robinson. “An iterative method of solving a game”. In: *Annals of Mathematics* 44 (1951), pp. 296–301.

[3] V. Thathachar and P. S. Sastry. “A class of rapidly converging algorithms for learning automata”. In: *IEEE Transactions on Systems, Man and Cybernetics* 15 (1985), pp. 168–175.

[4] C. Watkins. “Learning from Delayed Rewards”. PhD thesis. University of Cambridge, Cambridge, England, 1989.

[5] C. Watkins and P. Dayan. “Q-learning”. In: *Machine Learning* 8 (1992), pp. 279–292.

[6] Sergiu Hart and Andreu MasColell. “A simple adaptive procedure leading to correlated equilibrium”. In: *Econometrica* 68.5 (2000), pp. 1127–1150.

[7] M. A. Wiering. “Multi-agent reinforcement learning for traffic light control”. In: *Machine Learning: Proceedings of the Seventeenth International Conference (ICML'2000)* 2000 (2000).

[8] P. Auer, N. Cesa-Bianchi, and P. Fischer. “Finite-time analysis of the multiarmed bandit problem”. In: *Machine Learning* 47.2-3 (2002), pp. 235–256.

[9] P. Auer et al. “The nonstochastic multiarmed bandit problem”. In: *SIAM Journal on computing* 32.1 (2002), pp. 48–77.

[10] J. L. Stimpson and M. A. Goodrich. *Learning to cooperate in a social dilemma: a satisficing approach to bargaining*. In ICML, 2003.

[11] Eugene Nudelman et al. “Run the GAMUT: A comprehensive approach to evaluating game-theoretic algorithms”. In: *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*. Ed. by IEEE Computer Society. 2004.

[12] Joannes Vermorel and Mehryar Mohri. “Multi-armed bandit algorithms and empirical evaluation”. In: *European conference on machine learning*. Springer, 2005, pp. 437–448.

[13] S. Airiau, S. Saha, and S. Sen. “Evolutionary tournament-based comparison of learning and non-learning algorithm for iterated games”. In: *Journal of Artificial Societies and Social Simulation* 10.3 (2007), pp. 1–7.

[14] Erik P. Zawadzki. *Multiagent learning and empirical methods*. University of British Columbia, 2008.

[15] Yoav Shoham, Kevin Leyton-Brown, et al. “Multiagent systems”. In: *Algorithmic, Game-Theoretic, and Logical Foundations* (2009).

[16] Gerhard Bohm and Günter Zech. “Introduction to Statistics and Data Analysis for Physicists”. In: (Jan. 2010), p. 277.

[17] Bruno Bouzy and Marc Métivier. "Multi-agent learning experiments on repeated matrix games". In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 2010.

[18] Michel Tokic. "Adaptive $\varepsilon$-greedy exploration in reinforcement learning based on value differences". In: *Annual Conference on Artificial Intelligence*. Springer, 2010, pp. 203–210.

[19] Volodymyr Kuleshov and Doina Precup. *Algorithms for multi-armed bandit problems*. Tech. rep. 2014.

[20] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. *Safe, multi-agent, reinforcement learning for autonomous driving*. preprint. arXiv, 2016. arXiv: 1610.03295.

[21] Kathleen F.. Weaver et al. "An Introduction to Statistical Analysis in Research: with Applications in the Biological and Life Sciences". In: (2018), pp. 435–471.

# Appendix A

# AppendixA

## A.1 Command line options

To configure the test bed in the C++ program, available command line options are listed as follows:

| Flag | Description | Default |
|---|---|---|
| -h [ –help ] | print usage message | |
| -g [ –gametype ] | the game type for normal single game | (Random) |
| -p [ –players ] | the number of players in a game | (2) |
| -a [ –actions ] | the number of actions for each player | (2) |
| -r [ –iterations ] | the number of iterations in a game | (10000) |
| -s [ –strategy ] | set main strategy in a single game for comparison | (0) |
| -e [ –opp_strategy ] | set opponent strategy for comparison, in 2 player game | (1) |
| -t [ –print_top ] | print top n iterations info | (3) |
| -l [ –print_last ] | print last n iterations info | (1) |
| -z [ –print_flag ] | true to print more info. | (true) |
| -y [ –permute ] | run permutation of payoffs. | (false) |
| -o [ –tournament ] | run tournament w/ single game in all algorithm pairs. | (false) |
| -q [ –tournament_all_games ] | run tournament w/ all game types and all algorithms. | (false) |
| -m [ –enable_multithreading ] | enable multithreading. | (true) |
| -v [ –total_stratagies ] | total strategies for tournament mode. | (10) |

# Appendix B

# AppendixB

## B.1 Implementation details

The software includes two parts. The first part of the system is the test framework that was implemented by C++ programming language, which is used to run different scales of tournaments. To improve running time of tournaments, C++ program adopted multi-threading to reduce computation time. In addition, two third-party libraries were used: *sqlite*3 library was used for the access of a sqlite database, while *Boost* library was used for parsing command line input arguments. The another part of the system was R scripts in order to analyze raw data stored in a database. These scripts can be used directly after the installation of external R libraries, e.g. *RSQLite*, *ggplot*2, and *plotly*

## B.2 How to build and run the C++ program

The C++ source code and R scripts are stored in a remote repository. The C++ source code has to be built into a executable program, e.g. gcc or clang compiler. To build the C++ program, Section *Build dependencies* in README.md in the repository detailed instructions to install dependencies and build the program. After successfully building the program, Section How to use in README.md described the basic usage of the program, e.g. run a 3-action 5-player tournament. To customize the program, more command line flags were listed in Appendix A.

## B.3 Where to find the program

The C++ source files and R scripts are maintained in: https://github.com/ryanpig/mal-tournament.