Lancaster University

On-line planning and learning in type-based ad-hoc teamwork

Elnaz Shafipour Yourdshahi, BSc (Software Engineering)

School of Computing and Communications

Lancaster University

A thesis submitted for the degree of

Doctor of Philosophy

August, 2021

Declaration

I declare that the work presented in this thesis is, to the best of my knowledge and belief, original and my own work. The material has not been submitted, either in whole or in part, for a degree at this, or any other university. This thesis does not exceed the maximum permitted word length of 80,000 words including appendices and footnotes, but excluding the bibliography. A rough estimate of the word count is:

23972 (errors:2)

Elnaz Shafipour Yourdshahi

On-line planning and learning in type-based ad-hoc teamwork

Elnaz Shafipour Yourdshahi, BSc (Software Engineering).

School of Computing and Communications, Lancaster University

A thesis submitted for the degree of *Doctor of Philosophy.* August, 2021

List of Publications

Contributing publications

On-line estimators for effective ad-hoc task allocation in full and partial observability **Shafipour Yourdshahi, E.**, Soriano Marcolino, L., Alves, M., Angelov, P. *11th International Workshop on Optimization and Learning in Multiagent Systems. 2020*

Decentralised Task Allocation in the Fog: An On-line Genetic Estimator for Ad-hoc Teamwork (Extended Abstract) **Shafipour Yourdshahi, E.**, Soriano Marcolino, L., Alves, M., Angelov, P. *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020)*

Towards Large-Scale Ad-hoc Teamwork **Shafipour Yourdshahi, E.**, Pinder, T., Dhawan, G., Soriano Marcolino, L., Angelov, P. *P 22/06/2018 In: Proceedings of The 3rd International Conference on Agents (ICA 2018)*

Contributing publications under submissions

On-line Estimators for Ad-hoc Task Allocation – Learning types and parameters of teammates for effective teamwork **Shafipour Yourdshahi, E.**, Soriano Marcolino, L., Alves, M., Ueyama,J., Angelov, P. *Journal of Autonomous Agents and Multi-Agent Systems*

Additional publications

A Novel Self-Organizing PID Approach for Controlling Mobile Robot Locomotion Xiaowei Gu, Muhammad Aurangzeb Khan, **Shafipour Yourdshahi, E.**, Plamen Angelov, Bikash Tiwary *IEEE World Congress on Computational Intelligence (IEEE*

WCCI 2020)

Foreign Currency Exchange Rate Prediction using Neuro-Fuzzy Systems Yong, Y. L., Lee, Y., Gu, X., Angelov, P. P., Ling Ngo, D. C., **Shafipour Yourdshahi, E**. *Procedia computer science 144 (2018): 232-238.*

Towards Evolving Cooperative Mapping for Large-Scale UAV Teams **Shafipour Yourdshahi, E.**, Angelov, P. P., Soriano Marcolino, L., Tsianakas, G. *1/09/2018 In Proceedings of the 2018 IEEE Symposium Series on Computational Intelligence (IEEE SSCI 2018).*

Acknowledgements

I would like to thank my supervisors, Dr Leandro Soriano Marcolino and Professor Plamen Angelov for providing guidance and feedback throughout this project and their thoughtful comments and recommendations. Also, thanks to my husband and parents for their unconditional support that they gave me.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

Agents usually follow previously specified coordination and communication protocols for the sake of working together towards solving various problems. Employing these rules, however, is challenging due to environmental and technological constraints. There are circumstances where communication channels are unreliable, and agents cannot fully trust them to send or receive information. Moreover, particular situations require the design of agents (e.g., robots) from various parties aiming to solve a problem urgently, but constructing and testing communication and coordination protocols for all different agents can be unfeasible given the time constraints [5], [22]. For instance, imagine a natural disaster and hazardous situation where autonomous robots (agents) have been deployed from different countries or different organisations for handling the emergency situations, and they need to do it quickly to save lives by avoiding delays and funding usage but there is no time to construct and test communication/coordination protocols.

Another example would be planetary rovers which are small, unmanned vehicles that explore the surface of a planet, taking pictures and performing experiments. One of the most important and interesting research topics related to space rovers is the decision-making issue [49]. This problem is beyond simply developing robust navigation strategies for the rovers. However, completing a mission by a rover might take many years and lots of money to accomplish. Also, space is an unknown environment, and there might be many harsh, dangerous and unpredictable situations that cause the rover to fail and be unable to achieve its goals. A solution to mitigate these concerns is having a team of artificially intelligent planetary rovers, which must perform a wide variety of tasks with a wide variety of potential team-mates in uncertain and unsafe environments. A team of rovers can allow us to have a huge potential for space exploration, reduce cost and increase flexibility and reliability. However, having these multiple autonomous rovers which are acting simultaneously causes a coordination challenge. To achieve the best results, they should work together, and it is not a simple task due to the large distances and harsh environments. Furthermore, there are situations where different countries send robots to a specific planet.

These kinds of scenarios define the context denominated as *ad-hoc teamwork* in the multi-agent systems community, where agents intend to coordinate and cooperate to reach common goals, without the definition of any prior communication or coordination protocols. The agents, aware that other agents may follow different standards for coordination and communication, will try to learn about the behaviour and capabilities of their team-mates. As a result of their intelligent coordination, the agents must be able to accomplish shared goals efficiently, even though they face the lack of previous information about each other.

In particular, in many relevant domains the coordination can be modelled as a set of tasks to be accomplished in a distributed fashion (e.g., victims to be rescued from a hazard, mines that must be cleared, etc). In this way, I present a novel ad-hoc teamwork method in this research that handles problems where agents are supposed to complete several tasks in an environment, cooperatively. I denominate this ad-hoc team situation as *Task-based Ad-hoc Teamwork*. I define it as a decentralised distributed system where agents decide their tasks autonomously, without previous knowledge of each other, in an uncertain environment. Hence, there is no centralised mechanism to allocate tasks to individual agents, or manage their actions to accomplish the objectives. Agents need to decide, autonomously, which task they should pursue [23]. The *decentralised* allocation is quite natural in ad-hoc teamwork, as we cannot assume that other agents would be programmed to follow a centralised controller. Creating such partnerships among agents can support the accomplishment of missions that are hard to deal with individually, reducing the necessary completion time to achieve all tasks, and minimising the costs related to the process.

In my research, I assume that every agent follows an algorithm to reach its goal. Each of these algorithms depends on some parameters, which can be considered as properties of the agents. I call these algorithms as agent types. Therefore, instead of developing methods that could learn from scratch any possible policy, a common approach in the ad-hoc teamwork literature is to consider a set of possible agent types and parameters, reducing the problem to estimating those [2], [3], [21]. This approach is more applicable than learning models from scratch, as it does not require such a large number of observations, allowing learning and acting to happen simultaneously in an on-line fashion, in a single execution. Types could be built based on previous experiences [18], [19] or derived from the domain [1]. Moreover, the introduction of parameters for each type allowed more fine-grained models [2]. However, the previous works that learn types and parameters in ad-hoc teamwork are not specifically designed for decentralised task allocation, missing an opportunity to obtain better performances in this relevant scenario for multi-agent collaboration.

Other lines of works focus on neural network-based models and learn the policies of other agents after thousands (even millions) of observations [46], [71]. These methods, however, would be costly to be applied, especially when domains get more extensive and more complicated. Similarly, I-POMDP based models [29], [35], [42], [48] could be employed for reasoning about the model of other agents from scratch, but utilising such models to larger problems is non-trivial.

On the other hand, some approaches in the literature have also applied a task-based point of view, inferring about agents pursuing tasks to predict their behaviour [30]. Although I share some similarities, they have not yet handled learning types and parameters of agents in ad-hoc teamwork, in a system where multiple agents may need to help each other to complete a single task.

Meanwhile, a Monte Carlo Tree Search (MCTS) approach is usually employed to estimate best actions, given the current type estimations [2], [21]. However, the uncertainty over actions of the team-mates leads to a combinatorial explosion on the number of potential next states, leading to an exponential number of possible children for any given node in the search tree.

Consequently, in this work, I present a new method for estimating future behaviour of the team-mates in the ad-hoc team which is task-based, *Online Estimators for Ad-hoc Task Allocation*. Additionally, I introduce the novel on-line planning technique, *UCT-H*, which helps to do planning in larger teams.

1.1 Summary of Contributions

1.1.1 Online Estimators for Ad-hoc Task Allocation

My main contribution in this research is presenting a *novel algorithm* for estimating the team-mates types and parameters in decentralised task allocation, which is called *On-line Estimators for Ad-hoc Task Allocation* (OEATA). This algorithm is light-weight, enabling running estimations *from scratch* at every single run, instead of employing pre-trained models or carrying knowledge between executions. The main idea of the algorithm is to observe how the team-mates accomplish their tasks and keep them as a history of information about other agents. Afterwards, it applies this information to assess the sets of *estimators*, to have a better prediction of the team-mates types and parameters.

I prove theoretically that my algorithm converges to a perfect estimation when the number of tasks to be performed gets larger. Additionally, I run experiments in a collaborative foraging domain, considering both full and partial observability scenarios, where agents collaborate to collect "heavy" boxes together. I show that OEATA can obtain a lower error in parameter and type estimations in comparison with the state-of-the-art, leading to significantly better performance in task execution. I also run a range of different scenarios and find that OEATA still outperforms previous approaches as the number of agents, scenario sizes, and the number of items gets larger. Furthermore, I evaluate the impact of increasing the number of possible types and find that my approach scales better than other algorithms. Finally, I run experiments where my learning agent does not have the correct type of the other agents in its pool of possible agent types. In such challenging situations, the performance of OEATA is still better than the state-of-the-art in several cases.

For this contribution, I published two papers, and one paper is under review. One of them is *On-line Estimators for Ad-hoc Task Allocation: Extended Abstract* [85] which was accepted in Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020). In this paper, I presented *OEATA* in a fully observable environment. My other paper was presented at 11th International Workshop on Optimization and Learning in Multiagent Systems, called *Decentralised Task Allocation in the Fog: Estimators for Effective Ad-hoc Teamwork* [84]. In this paper, I considered applying *OEATA* for an agent that has a partial observation of the environment. Recently, I submitted another paper to the Journal of Autonomous Agents and Multi-Agent Systems. In this paper, I explained

OEATA in more detail with more results and a larger number of tasks.

1.1.2 History-based UCT

My other contribution in this thesis is proposing UCT-H, a new version of UCT Monte Carlo Tree Search, adopting a history-based compact representation. After estimating types and parameters of the team-mates, the learning agent needs to take the best action to enhance the performance of the team where a Monte Carlo Tree Search (MCTS) approach [2], [21] is usually employed to estimate best actions. Nevertheless, the uncertainty over the actions of the team-mates leads to a combinatorial explosion on the number of potential next states, leading to an exponential number of possible children for any given node in the search tree.

Accordingly, to enable large-scale ad-hoc teamwork, I first formalise the problem as a Markov Decision Process (MDP) and then solve it with UCT-H. Again, I evaluate my approach in the level-based foraging domain, with larger team sizes than what has been explored before[2], [6]. I evaluate overall task performance, computational time, and memory usage. I find that my compact representation achieves better results than the previous MCTS approaches for any team size, and scales better with the number of agents. After evaluation, I show that the difference in performance between UCT-H and UCT tends to increase as the number of agents grows, reaching 65% better performance with ten agents; and the memory usage of UCT-H is roughly constant, while memory usage for UCT rises exponentially. I present this method in my paper called *Towards Large Scale Ad-hoc Teamwork* [102], which was accepted in the 2018 IEEE International Conference on Agents (ICA).

1.2 Guide to Thesis

I organise the thesis as follows: Chapter 2 reviews some background for a better understanding of the main contributions of the research as well as the literature review. In Chapter 3, I explain how I define my task-based ad-hoc team as well as the state-of-the-art for estimating type and parameters of the team-mates. Then, Chapter 4 is focused on my novel approach, *UCT-H*, and its evaluations. Afterwards, in Chapter 5, I focus more on my other contribution, *OEATA* following with its evaluations. Finally, Chapter 6 presents my conclusions and discussions for future work.

Chapter 2

Background

2.1 Reinforcement Learning

There are different areas in machine learning [11], [61], such as supervised learning [28] and unsupervised learning [16], in which there are training and testing dataset to help learning and obtaining results. However, in reinforcement learning [55], [94], which is another area in machine learning, there is no dataset, and all the learning processes happen based on experience and interaction to achieve a goal. In this area, the learner and decision-maker are called the agent, which wants to learn optimal behaviour in an environment. However, the environment is everything outside the agent, which it tries to interact with. These interactions occur continuously, with the agent choosing actions and the environment responding to those actions and rendering new situations to the agent. Hence, in reinforcement learning, there is one or multiple agents in an environment, and they receive positive and negative rewards based on their actions (Figure 2.1). The goal of agents is to find the best actions to maximise the total reward they receive.

As I explained earlier, in reinforcement learning, there is no batch of data like in supervised learning. When the agent moves around the environment starts

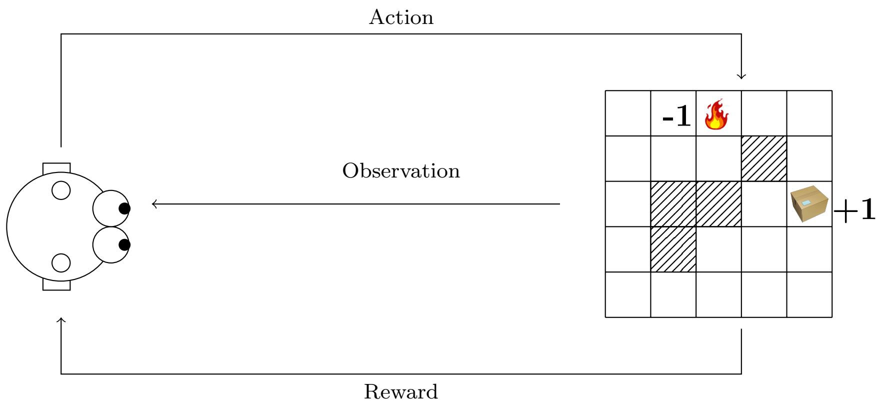


Figure 2.1: The interaction of agent and environment in reinforcement learning

gathering data and the actions that the agent takes affects the data that it observes. Therefore, one of the fundamental dilemmas in reinforcement learning is *exploration and exploitation* [12], [101]. In exploration, the agent keeps searching for new strategies and gathering more information that might lead us to better decisions in the future. However, in exploitation, the agent makes the best decision given current information and chooses the best strategies found thus far.

2.2 Multi Arm Bandit

A good model for *exploration and exploitation* is the multi-armed bandit problem. The multi-armed bandit is a problem [77] in which we need to allocate many alternatives between a fixed limited set of resources. At the time that we are doing the allocation, the property of each choice is partially known, but it may become better understood as time passes or by allocating resources to that choice. However, we are supposed to find a way that maximises their expected reward. Multi-armed bandit is a classic reinforcement learning problem that exemplifies the exploration-exploitation trade-off dilemma.

The name comes from imagining a gambler at a row of slot machines (sometimes known as "one-armed bandits"). In this game, the gambler should decide which machines to play. Additionally, he should resolve how many times to play each one as well as the order of playing them. Furthermore, he should think about whether to continue with the current machine or try a different one [41]. The multi-armed bandit problem also falls into the broad category of stochastic scheduling. Imagine that you are in a casino where there are many slot machines that you might want to play and get a reward (Figure 2.2). However, you do not have any information about how each is configured. Moreover, we need to know how big is the reward that you get from a slot machine at each play. Hence, you might think about what would be the best strategy to achieve the highest long-term rewards.

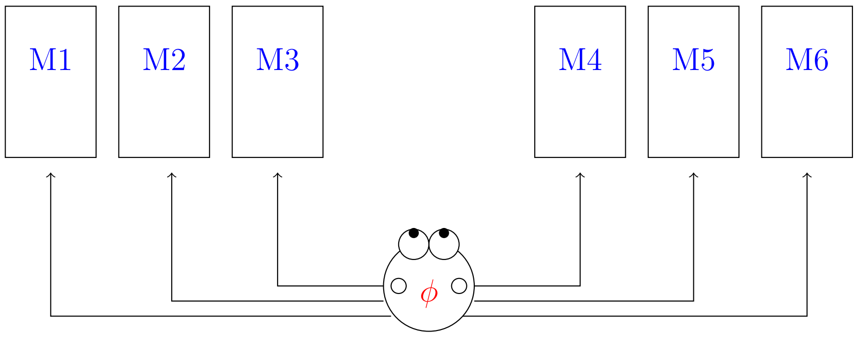


Figure 2.2: Multi Armed Bandit

In the k-armed bandit problem [105], there are k slot machines with reward probabilities, $\{p_1, \ldots, p_K\}$, over each arm. Therefore, the *Multi Armed Bandit* is defined as a tuple $(\mathcal{A}, \mathcal{R})$, where:

- $\mathcal{A}$ is a set of actions (arms)
- $\mathcal{R} = \mathbb{P}[R = r|A = a]$ is an unknown probability distribution over rewards.

Hence, at each time step t, by taking any action a on a slot machine, a reward r is received.

2.3 Upper Confidence Bound

As I mentioned before, there is a dilemma when we want to have a balance between exploration and exploitation, as the agent cannot choose to both explore and exploit at the same time. Accordingly, one solution for it is to apply the Upper Confidence Bound algorithm [39]. With UCB, the arms will be chosen in a way that keeps a balance between exploring the less frequently simulated actions (arms) and exploiting the already promising ones.

There are different variants of the UCB algorithms but in this research, I apply the UCB1 algorithm[13].

At each given round of t trials, the UCB value of all arms (actions) are represented by the following:

$$Q_t(a) + c\sqrt{\frac{\log(t)}{\mathcal{N}_t(a)}} \tag{2.1}$$

In this equation, $Q_t(a)$ denotes the estimated value of action a at time t, where $\log(t)$ denotes the natural logarithm of t, $\mathcal{N}_t(a)$ denotes the number of times that action a has been selected at t trial, and the number $c > 0$ controls the degree of exploration. Therefore, the selected action or arm would be the action that has the maximum UCB value:

$$\mathcal{A}_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left[Q_t(a) + c\sqrt{\frac{\log(t)}{\mathcal{N}_t(a)}} \right] \tag{2.2}$$

2.4 Markov Decision Process

Markov Decision Process (MDP) [99] provides a mathematical framework to formalise sequential decision making. This formalisation is the basis for structuring problems that we can solve with reinforcement learning. It formally describes a framework used to help for making decisions in a stochastic environment where the environment is fully observable. The goal is to find a solution, with the given information and optimal actions on each state of the environment. In practice, there are some situations where given a certain action, the next state is stochastic, which is caused by uncertainty. For example, we can consider the agent as a robot which is moving around a room. Accordingly, based on the stochastic nature of the environment, some unexpected outcomes might happen during execution time. There might be a possibility that the actual action that the robot takes is going north, but it ends up going west because of the dynamics of the environment and uncertain situations that might be arising. It can occur even if the state of the environment is fully observable. There are some initial concepts that I need to explain before explaining *MDP*.

2.4.1 Markov Property

The central idea for *MDP* is the *Markov Property*, which means that the future is independent of the past given the present. In other words, a state is *Markovian* if and only if:

$$\mathbb{P}[\mathcal{S}_{t+1}|\mathcal{S}_t] = \mathbb{P}[\mathcal{S}_{t+1}|\mathcal{S}_1, ...\mathcal{S}_t] \tag{2.3}$$

Therefore, the next state $\mathcal{S}_{t+1}$ is only dependent on the current state $\mathcal{S}_t$ and not any of the other previous states.

2.4.2 Markov Process / Markov Chain

Markov Process is a memoryless random process, which is a sequence of random states $\mathcal{S}_1, \ldots, \mathcal{S}_t$, with the *Markov Property*, and is defined as a tuple $(\mathcal{S}, \mathcal{P})$ where $\mathcal{S}$ is a finite set of states and $\mathcal{P}$ is a state transition matrix which defines the transition probability for all $\mathcal{S}_t$ to all possible successor states $\mathcal{S}_{t+1}$. Therefore, for a Markovian state s and successor state s', the state *transition probability* is defined by :

$$\mathcal{P}_{ss'} = \mathbb{P}[\mathcal{S}_{t+1} = s' | \mathcal{S}_t = s] \tag{2.4}$$

Where $\mathcal{P}_{ss'}$ is a state transition matrix which defines the transition probability for each s to all possible successor states s'.

2.4.3 Markov Reward Process

MRP is a Markov Process with a value judgement that says how much reward we accumulate in a particular sequence that is sampled from a Markov Process. *MRP* is defined as a tuple $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$ where $\mathcal{S}$ and P are the same as Markov Chain and the new element $\mathcal{R}$ is a reward function:

$$\mathcal{R}_s = \mathbb{E}[\mathcal{R}_{t+1} | \mathcal{S}_t] \tag{2.5}$$

which tells us how much reward the agent will get immediately after transitioning from $\mathcal{S}_t$ to $\mathcal{S}_{t+1}$. However, the main important factor in reinforcement learning is to maximise the cumulative reward in the whole process and not only the reward that the agent gets in one time step. Additionally, $\gamma \in [0, 1]$ is the discount factor which presents the value of the future rewards in relation to the present.

2.4.4 Return

The *return* G_t is the total *discounted* reward from time step t summing up with the future rewards, and the main goal of the decision-maker is to maximise the *return* value. The total reward is defined as the following equation:

$$G_t = \mathcal{R}_{t+1} + \gamma \mathcal{R}_{t+2} + \gamma^2 \mathcal{R}_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k \mathcal{R}_{t+k+1} \tag{2.6}$$

As it is shown in the equation, the return value is the accumulation of rewards of the whole process where the reward is discounted in each time step by the factor γ. In this equation, the value of γ shows how much we care now about the rewards that we get in the future. Therefore, the reward which is gotten in the next step $t+1$ has the least discount factor value. Additionally, $\mathcal{R}_{t+2}$ shows the reward which the agent obtains when it goes from the state s_{t+1} to the state s_{t+2}, which will be reduced by the γ value. Consequently, the value of receiving reward $\mathcal{R}$ after $k+1$ time-step is $\gamma^k\mathcal{R}$.

2.4.5 Policy

The other concept in MDP is the policy π, which is a distribution over actions given states, and defines the behaviour of an agent.

$$\pi(a|s) = \mathbb{P}[\mathcal{A}_t = a|\mathcal{S}_t = s]$$

2.4.6 Value Function

Value function estimates how good it is for the agent to be in a given state or how good it is to perform a given action in a given state. In other words, how much will be the future rewards that can be expected, or, to be precise, in terms of expected return. Of course, the rewards the agent can expect to receive in the future depends on what actions it will take. Accordingly, value functions are defined concerning particular policies π.

The state value function $\mathcal{V}_\pi(s)$ of an MRP is the expected return starting from state s for policy π, which gives us the long term value of the state s.

$$\mathcal{V}_\pi(s) = \mathbb{E}[G_t|\mathcal{S}_t = s] = \mathbb{E}[\sum_{k=0}^{\infty}\gamma^k\mathcal{R}_{t+k+1}|\mathcal{S}_t = s] \tag{2.7}$$

Likewise, we define the value of taking action a in state s under a policy π:

$$Q_\pi(s,a) = \mathbb{E}[G_t|\mathcal{S}_t = s, \mathcal{A}_t = a] = \mathbb{E}[\sum_{k=0}^{\infty}\gamma^k\mathcal{R}_{t+k+1}|\mathcal{S}_t = s, \mathcal{A}_t = a] \tag{2.8}$$

2.4.7 Bellman Equation for MRPs

The value function depends on the policy by which the agent picks actions to perform. The Value Function can be decomposed into two parts:

- R_{t+1} is the immediate reward
- $\gamma\mathcal{V}(s_{t+1})$ is the discounted value of successor states

Therefore, we can replace G_t with the sequences of rewards, so we would have:

$$\mathcal{V}(s) = \mathbb{E}[\mathcal{R}_{t+1} + \gamma\mathcal{R}_{t+2} + \gamma^2\mathcal{R}_{t+3} + \ldots |\mathcal{S}_t = s],$$

$$\mathcal{V}(s) = \mathbb{E}[\mathcal{R}_{t+1} + \gamma(\mathcal{R}_{t+2} + \gamma\mathcal{R}_{t+3} + \ldots)|\mathcal{S}_t = s],$$

$$\mathcal{V}(s) = \mathbb{E}[\mathcal{R}_{t+1} + \gamma\mathbf{G}_{t+1}|\mathcal{S}_t = s],$$

$$\mathcal{V}(s) = \mathbb{E}[\mathcal{R}_{t+1} + \gamma\mathcal{V}(\mathcal{S}_{t+1})|\mathcal{S}_t = s],$$

$$\mathcal{V}(s) = \mathcal{R}_s + \gamma\sum_{s'\in\mathcal{S}}\mathcal{P}_{ss'}\mathcal{V}(s')$$

Finally, the Markov Decision Process is a Markov Reward Process with decisions in which all states are Markovian. Accordingly, we can define it as tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where:

- $\mathcal{S}$ is a finite set of *states* $\mathcal{S} = \{s_0, s_1, s_2, s_3, \ldots, s_n\}$, with s_0 being the initial state
- $\mathcal{A}$ is a finite set of possible actions in each state, $\mathit{Action}(s) = a \in \mathcal{A}$;

- $\mathcal{P}$ is the state transition probability matrix (function), and now the action will be added to the following equation:

$$\mathcal{P}^a_{ss'} = \mathbb{P}[\mathcal{S}_{t+1} = s' | \mathcal{S}_t = s, \mathcal{A}_t = a]$$

- $\mathcal{R}$ is the *Reward function*

$$\mathcal{R}^a_s = \mathbb{E}[\mathcal{R}_{t+1} | \mathcal{S}_t = s, \mathcal{A}_t = a]$$

In detail, at each time step, the process is in some state s, and the agent may choose any action $a \in Action(s)$ that is available in the state s. The agent follows a stochastic way of thinking where, by taking action a, with a specific probability, the next state will be s'. The probability that the process moves into its new state s', $\mathcal{P}^a_{ss'}$, is influenced by the chosen action a. Thus, the next state s' depends on the current state s and the agent's action a. However, given s and a, it is conditionally independent of all previous states and actions; in other words, the state transitions of a *MDP* satisfies the Markov property. Regarding computing applications, every *MDP* must have a "final goal", i.e., final states/terminals. It is mandatory because it is required to evaluate the decisions throughout the process and it is done by assigning *rewards* directly or indirectly linked to states until the tasks or goals are accomplished.

2.4.8 Optimal Value Function

Solving a reinforcement learning task means to find a policy that achieves a lot of reward over the long run.

A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $\mathcal{V}_\pi(s) \geq \mathcal{V}_{\pi'}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies which is called an optimal policy. Therefore, the optimal value function specifies the best possible performance in the MDP, and once we find the optimal value function, the MDP is solved. Although there may be more than one, we denote all the optimal policies by π_*. They share the same state-value function, called the optimal state-value function, denoted $\mathcal{V}_*$, and defined as

$$\mathcal{V}_*(s) = \max_{\pi} \mathcal{V}_\pi(s) \tag{2.9}$$

Additionally, the optimal action-value function is the maximum action-value function over all policies.

$$Q_*(s,a) = \max_{\pi} Q_\pi(s,a) \tag{2.10}$$

Consequently, the optimal policy π_* can be found by maximising over $Q_*(s,a)$:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if a =}\operatorname*{argmax}_{a\in\mathcal{A}} Q_*(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{2.11}$$

Therefore, if we know the $Q_*(s,a)$, we immediately have the optimal policy.

2.5 Monte Carlo Tree Search

One of the techniques to solve an MDP problem (described in Section 2.4) is called the *Monte Carlo Tree Search (MCTS)* [26], [32], [56], which is a heuristic driven search algorithm for making an optimal decision. MCTS is a combination of classic tree search and reinforcement learning. In 2006, MCTS was introduced for computer Go [87]. Other board games like chess and shogi [88] games with incomplete information such as bridge [80] and poker [79], used MCTS as well.

MCTS is an online planning model which is aiming to find the most promising actions in the current state by expanding the search tree based on a random sampling of the search space. Therefore, it helps *planning ahead* to reach goals and avoid failures.

The MCTS algorithm keeps evaluating any possible action periodically by executing them in simulation, and there is always an exploration-exploitation trade-off. It exploits the best actions, and strategies found so far and at the same time, proceed with exploring the local space of alternative decisions and discover if they could replace the new best path. In other words, the goal is to find the unexplored parts of the tree, which leads to identifying a more optimal path. In MCTS, nodes are the building blocks of the search tree. These nodes are formed based on the outcome of several simulations.

MCTS solves MDP problems, therefore, in this algorithm there is a set of *states* $\mathcal{S} = \{s_0, s_1, s_2, s_3, ..., s_m\}$, with s_0 being the initial state and in each state there are a set of actions $\mathcal{A} = \{a_0, a_1, a_2, a_3, ..., a_n\}$. The agent needs to get the best *action* which gives it the highest *return value*. When the *MCTS* tackles the *MDP*s environment, each node in the tree search holds a *Q-table*, where the average value of each action across all simulations is stored and consists of a tuple $(s, Q, \mathcal{N})$. In this tuple,

- s is the state,
- $Q(s, a)$ is the value that indicates how good or bad is a state-action pair or evaluating the action a when an agent takes in this state s. The *value* associated to the node is estimated by the mean cumulative discounted reward of all simulations for the state s and action a where the action a was selected from state s.
- $\mathcal{N}(s, a)$ is a *visitation count* which indicates how many times a node is visited in the state s by taking the action a.

For each node, values for $Q(s, a)$, $\mathcal{N}(s, a)$ are initialised to 0.

The four distinct steps of the Monte Carlo Tree Search process are *Expansion, Selection, Simulation, Backpropagation.* Details of these steps are as below:

2.5.1 Selection

In this process, the MCTS algorithm traverses the current tree from the root node using a specific strategy (Figure 2.3). The strategy uses an evaluation function to optimally select child nodes with the greatest estimated value. Tracing the tree by selecting the child node will continue until reaching a leaf node or reaching the maximum depth of the tree. By selecting a node, the $\mathcal{N}(s, a)$ of the selected node will increase.

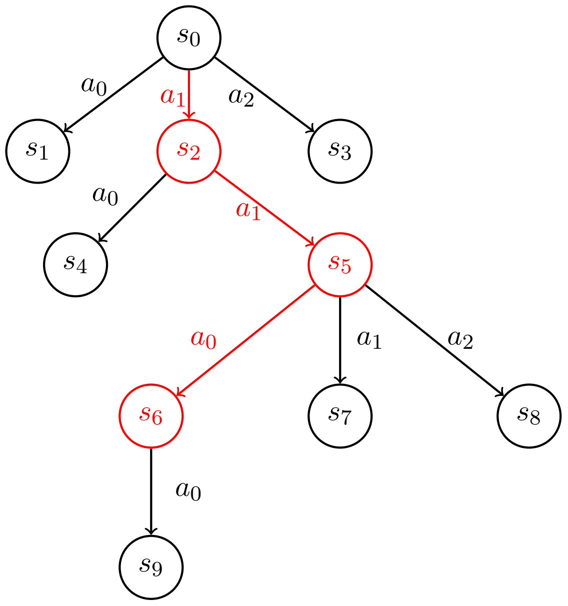


Figure 2.3: The process of selecting an action while tracing the tree.

2.5.2 Expansion

In the selection step, we optimally reached a node, and now it is time to expand it (Figure 2.4). Therefore, an action a_i will be selected among the list of actions which is not selected for that specific node. Accordingly, a new child node is added to the tree. Then the selected action a_i will be applied to the state s_j, which is the state of the last node and get the immediate reward by taking action a_i in the state s_j.

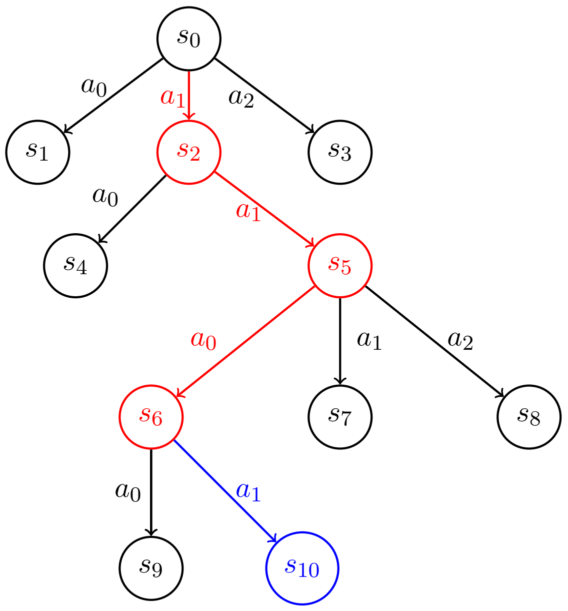


Figure 2.4: Expanding the tree by adding non-selected actions from the specific node

2.5.3 Simulation

Next step after expanding is the simulating step. In the simulating step (Figure 2.5), random actions will be taken from the state of the expanded node. In each iteration, after taking an action, there might be a reward. At each iteration, the reward is multiplied with a discount factor γ. As I mentioned earlier (Section 2.4.4), the reason is to reduce the value of the rewards which are taken in further states. This process will be repeated until n iterations. But sometimes, in turn-based games, taking a certain path or branch could result in losing. In the long run, this is due to a large number of combinations and each node might not be visited enough times to grasp its outcome. Additionally, in order to be able to determine the most efficient path, MCTS algorithm needs a large number of iterations. The speed is a bit slow.

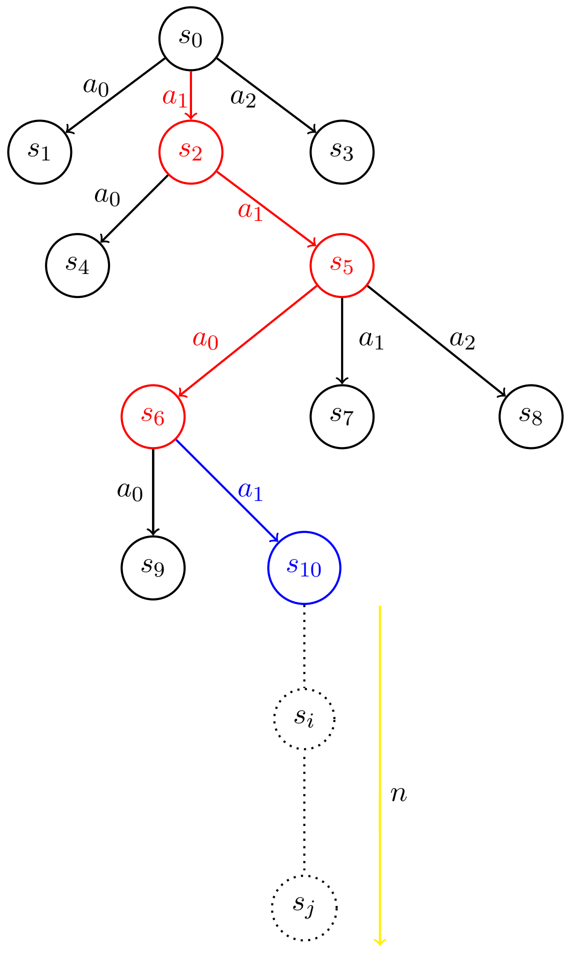


Figure 2.5: Simulating random actions to evaluate the current state

2.5.4 Backpropagation

After the simulation phase, a result is returned. Therefore, the simulation result will be added to all nodes' value, from the last expanded node up to the root (Figure 2.6). Moreover, the count of visits at each node will increase.

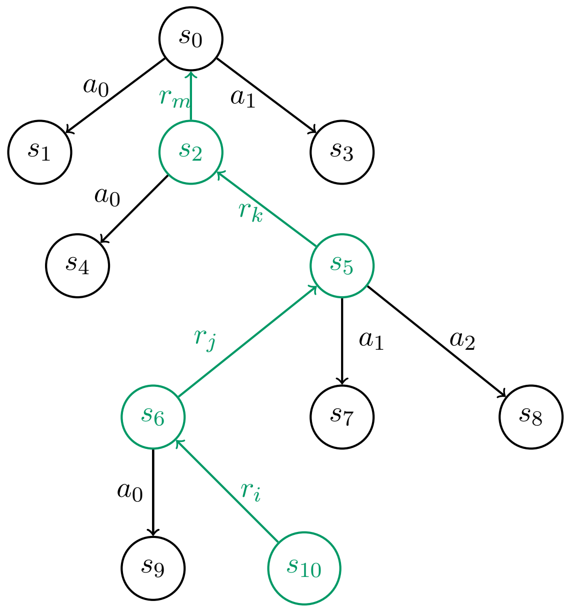


Figure 2.6: Backpropagation phase to return the accumulated rewards

2.6 UCT

The literature suggests the application of UCT [56] algorithm for improving the MCTS, which is an application over the multi-armed bandit. With this in mind, each state at the search tree is viewed as a multi-armed bandit taking an action chosen by the *Upper Confidence Bound 1 (UCB1)* [13] algorithm. The *UCB1* follows the Equation 2.12, that tries to maximise the value of the experienced action attaching bonus reward for each tried action at the current state.

$$Q^{\oplus}(s,a) = Q(s,a) + c\sqrt{\frac{\log\left(\mathcal{N}(s)\right)}{\mathcal{N}(s,a)}} \tag{2.12}$$

The scalar constant $c \in [0,1]$ determines the relative ratio of exploration to exploitation, where if the constant is equal to 0, the *UCT* algorithm acts greedily within the tree. Once all actions from state s are represented in the search tree, the tree policy selects the action maximising the augmented action-value, $\operatorname{argmax}_a Q^{\oplus}(s, a)$. For a suitable choice of c, the value function constructed by UCT converges in probability to the optimal value:

$$Q(s, a) \xrightarrow{p} Q^*(s, a),$$

Although Albrecht and Stone (2017) did not explicitly formalise the ad-hoc teamwork problem as an MDP [2], they employed a traditional UCT Monte Carlo Tree Search [56] (which are used for solving MDPs in an online fashion). As I explained in Section 2.5, in *Selection* step of the *MCTS*, the best child node should be selected. There might be a different selecting function. However, to decide an action to simulate at each node, the UCB1 algorithm [13] is employed. After all x simulations are performed, the agent can estimate the best action to take by considering the *Q-table* at the root node. Once a new state is reached, the whole algorithm is repeated, to decide the next action in that new state. A pseudo-code of the original UCT algorithm is shown in Algorithm 1.

Algorithm 1 UCT algorithm

```
function UCT($state$)                                    ▷
    repeat
        $Search(state, 0)$
    until $Timeout$
    return bestAction(state,0)
end function
function SEARCH($state, depth$)                          ▷
    if $Terminal(state)$ then
        return 0
    end if
    if $Leaf(state, depth)$ then
        return $Evaluate(State)$
    end if
    $action \leftarrow selectedAction(state, action)$
    $(nextState, reward) \leftarrow simulateAction(state, action)$
    $q \leftarrow reward + \gamma\ Search(nextState, depth + 1)$
    return $q$
end function
```

2.7 POMDP

Unlike the MDP, in *Partial Observable Markov Decision Process (POMDP)* [54] approach, the agent cannot fully observe the environment and current state directly, so there is not enough knowledge of the current state when the agent has a partial observation.

A POMDP can be defined as a tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{O}, \mathcal{Z})$, where like MDP, $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, $\mathcal{R}$ is the reward function and $\mathcal{P}$ is the transition model. The only differences are $\mathcal{O}$, $\mathcal{Z}$, where $\mathcal{O}$ is the observations that agent receives and $\mathcal{Z}$ is the observation probabilities that is equal to:

$$\mathcal{Z}^a_{s'o} = \mathbb{P}(\mathcal{O}_{t+1} = o | \mathcal{S}_{t+1} = s', \mathcal{A}_t = a),$$

Where the initial state $s_0 \in \mathcal{S}$ is determined by a probability distribution $\mathcal{I}_s = \mathbb{P}(s_0 = s)$. Also, in POMDP there is a *History* which is a combination of action and observation $h_t = \{a_1, o_1, a_2, o_2, ..., a_t, o_t\}$ or $h_t a_{t+1} = \{a_1, o_1, ..., a_t, o_t, a_{t+1}\}$, where $a \in \mathcal{A}$ and the observation $o \in \mathcal{O}$ represents the action taken at time t and the corresponding observation that the agent receives from the environment.

Consequently, the agent never receives its exact current state and it builds a belief state based on its history. Hence, the belief state is the probability distribution over states given history h, and we can define a probabilistic belief state for each agent as:

$$\mathcal{B}(s, h) = \mathcal{P}_{h,s} = \mathbb{P}[\mathcal{S}_t = s | \mathcal{H}_t = h]$$

The *reward* for a partially observable environment is calculated in the same way as MDP:

$$\mathcal{R}_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_k,$$

The *policy* of selecting action a given the history h follows a probability over the actions:

$$\pi(h, a) = \mathbb{P}(\mathcal{A}_{t+1} = a | \mathcal{H}_t = h),$$

being the $\pi^*(h, a)$ the optimum policy. And the *value function* is:

$$\mathcal{V}_\pi(h) = \mathbb{E}[R_t | \mathcal{H}_t = h]$$

which is the expected return value from the state s when following policy π. The optimal value function is the maximum value function achievable by any policy.

2.8 POMCP

Partially Observable Monte-Carlo (POMCP) [89] is a combination of a Monte-Carlo update of the agent's belief state and a Monte-Carlo tree search from the current belief state. With regards to search approaches, POMCP is a very famous extension of the traditional UCT Monte Carlo Tree Search, when considering *partially observable* environments. In POMCP, Monte Carlo sampling is used both during belief state updates and during planning. In addition, instead of having explicit probability distribution, only a black box simulator of the POMDP is required. These specifications of POMCP help to solve larger POMDP problems.

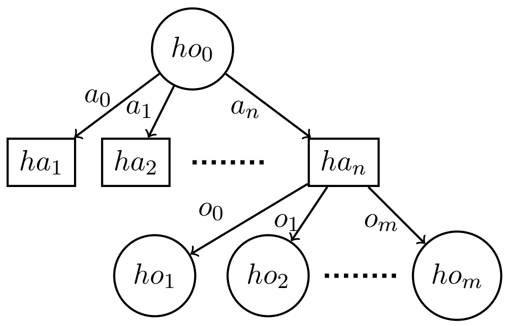


Figure 2.7: An illustration of POMCP search tree

Partially Observable Monte-Carlo Planning (POMCP) consists of a UCT search that selects actions at each time-step; and a particle filter that updates the agent's belief state. The search tree contains a node $T(h) = (\mathcal{V}(h), \mathcal{N}(h), \mathcal{B}(h))$ for each represented history h where:

- $\mathcal{N}(h)$ counts the number of times that history h has been visited.
- $\mathcal{V}(h)$ is the value of history h, estimated by the mean return of all simulations starting with h.
- $\mathcal{B}(h)$ contains a set of particles. The most important change in the *POMCP*

algorithm is the idea to use an unweighted particle filter to approximate the belief state at each node in the UCT algorithm.

The search procedure, which is explained in detail in Algorithms 2, is called from the current history h_t. Each simulation begins from a start state that is sampled from the belief state $\mathcal{B}(h_t)$ (Line 6).

Algorithm 2 POMCP-Search Algorithm

```
1:  procedure SEARCH($h$)
2:      repeat
3:          if $h$ is empty then
4:              $s \sim I$
5:          else
6:              $s \sim \mathcal{B}(h)$
7:          end if
8:          $Simulate(s, h, 0)$
9:      until $Timeout()$
10:     return $\underset{b}{\operatorname{argmax}}(\mathcal{V}(hb))$
11: end procedure
```

In the Simulation stage (Algorithm 3), similar to the MCTS, the simulations are divided into two stages. In the first stage of simulation, when child nodes exist for all children, actions are selected by UCB1 based on following equation,

$$\mathcal{V}^{\oplus}(ha) = \mathcal{V}(ha) + c\sqrt{\frac{\log(\mathcal{N}(h))}{\mathcal{N}(ha)}}$$

and the action that maximises this augmented value, $\underset{a}{\operatorname{argmax}}\,\mathcal{V}(ha)$ will be selected. In the next stage of simulation, actions are selected by a history based rollout policy (Algorithm 4) $\pi_{rollout}(h, a)$ (e.g. uniform random action selection). After each simulation, precisely one new node is added to the tree, corresponding to the first new history encountered during that simulation.

The agent uses a simulator $\mathcal{G}$ (Line 12 in Algorithm 3) as a generative model of the POMDP. The simulator provides a sample of a successor state, observation and reward, given the current state and action, $(s_{t+1}, o_{t+1}, r_{t+1}) \sim \mathcal{G}(s_t, a_t)$, and can also be reset to a start state s.

Algorithm 3 POMCP-Simulate Algorithm

```
1: procedure SIMULATE($s, h, depth$)
2:     if $\gamma^{depth} = \epsilon$ then
3:         return 0
4:     end if
5:     if $h \notin T$ then
6:         for all $a \in \mathcal{A}$ do
7:             $T(ha) \leftarrow (\mathcal{N}_{init}(ha), \mathcal{V}_{init}(ha), \emptyset)$
8:         end for
9:         return Rollout($s, h, depth$)
10:    end if
11:    $a \leftarrow \underset{b}{\operatorname{argmax}} \mathcal{V}(hb) + c\sqrt{\frac{\log(\mathcal{N}(h))}{\mathcal{N}(ha)}}, \forall b \in \mathcal{A}$
12:    $(s', o, r) \sim \mathcal{G}(s, a)$
13:    $\mathcal{R} \leftarrow r + \gamma.Simulate(s', hao, depth + 1)$
14:    $\mathcal{B}(h) \leftarrow \mathcal{B}(h) \cup \{s\}$
15:    $\mathcal{N}(h) \leftarrow \mathcal{N}(h) + 1$
16:    $\mathcal{N}(ha) \leftarrow \mathcal{N}(ha) + 1$
17:    $\mathcal{V}(ha) \leftarrow \mathcal{V}(ha) + \frac{\mathcal{R} - \mathcal{V}(ha)}{\mathcal{N}(ha)}$
18:    return $\mathcal{R}$
19: end procedure
```

For every history h, encountered during simulation, the belief state $\mathcal{B}(h)$ is updated to include the simulation state. When the search is complete, the agent selects the action a_t with the highest value and receives a real observation, o_t, from the world. At this point, the node $T(h_t a_t o_t)$ becomes the root of the new search tree, and the belief state $\mathcal{B}(h_t ao)$ determines the agent's new belief state. The remainder of the tree is pruned, as all other histories are now impossible.

Algorithm 4 POMCP-Rollout Algorithm

```
1: procedure ROLLOUT($s, h, depth$)
2:     if $\gamma^{depth} = \epsilon$ then
3:         return 0
4:     end if
5:     $a \sim \pi_{rollout}(h)$
6:     $(s', o, r) \sim \mathcal{G}(s, a)$
7:     return $r + \gamma\ Rollout(s', hao, depth + 1)$
8: end procedure
```

More specifically, each time we start a search procedure with the tree, a state is sampled from the belief state of the root node h_t. Defining the current state, an action a is selected, so the simulator samples the next state s' and the observation o (Figure 2.7). The pair ao defines the next node in the search tree, and for the current iteration, the state of the node will be assumed to be s'. This sampled state s' is added to tree *particle filter*, and the process repeats recursively down the tree.

Chapter 3

Literature Review

3.1 Multi-Agent Systems

There are many works towards multi-agent systems as the world is moving towards "smart systems" which rely on some form of intelligent agent technology. These agents can autonomously collect information from their surrounding environment and act upon it. Examples include connected autonomous vehicles that gather information from adjacent ones and act upon it to improve the efficiency and safety of the transportation systems [52]. Moreover, virtual personal assistants that can keep track of users' behaviours and preferences to make recommendations and assist the users in several tasks [62]. As technology evolves, so will the autonomy and perceptual/actuation capabilities of such agents, prompting the need for autonomous agents that can coexist with other (different) agents and eventually engage in some form of teamwork towards the completion of some common task.

The literature introduces a multi-agent system (MAS) [86] as a system which is composed of multiple interacting intelligent agents [80], which can solve problems that are difficult or impossible for an individual agent to solve. Therefore, multi-agent systems can solve complicated problems by dividing them into tasks [34], where the individual tasks are allocated to agents. Each agent chooses a proper action to solve the task, handling multiple inputs, e.g., history of actions, interactions with its neighbouring agents, and its goal. There are many problems in engineering and technology [65] that are solved by applying MAS. Additionally, multi-agent systems research may deliver an appropriate approach to many applications including online trading [78] disaster response [40], [83], target surveillance [51] and social structure modelling [93]. The key objective of forming the team of agents is the cooperation and collaboration of the autonomous agents, which help them to achieve goals that they cannot deal with individually and reduces the completion time of a target that might take a significant long time to finish if they are alone. Accordingly, a core area of research in modern artificial intelligence (AI) is the development of autonomous agents that can interact effectively with other agents.

Working agents as a team is a principal subject of research in the multi-agent systems literature [100]. To develop MAS, addressing a diverse range of complex challenges such as coordination among agents [106], learning [25] and security [100] is required. Many theoretical frameworks of teamwork have been developed for MAS. For example, Cohen and Levesque (1991) [31], introduced the Joint Intentions theory, which defined that a team has a joint mental state. All agents work to achieve a certain objective in the joint mental state. If one of the agents discovers that the objective has been achieved, or became irrelevant/impossible, then it must communicate with its team-mates to pass this knowledge to the joint mental state. In the SharedPlans [43] framework, there is a set of possible recipes for achieving one action, which is composed by subactions, forming a hierarchy; and agents may have individual plans to complete some of the subactions. These ideas are combined in a real implemented framework in STEAM [95], where agents build a hierarchy of joint intentions when performing tasks in three different domains. STEAM is further extended in Scerri and Pynadath (2003) [82], where a Markov Decision Process (MDP) model is proposed, enabling agents to autonomously decide when to transfer control (i.e., decision-making) to humans or other agents.

3.2 Task Allocation

Task allocation is a principal approach to coordinate a team of agents [57], which refers to the allocation of tasks to agents considering the associated cost, time, and (communication and processing) overhead [38], [96]. The contract net protocol [90] is a common technique for task allocation, where agents can be managers and contractors. A manager receives bids and allocates a task to the most appropriate agent. Upon being allocated a task, the agent (contractor) must execute it, but it can divide those into subtasks and also act as a manager to allocate those. A similar approach is the auction-based task allocation mechanism [24], where agents submit bids to compete for tasks, like in actual auctions.

Task allocation can be centralised or decentralised [57]. Dos Santos and Bazzan (2002) [33] suggest a hybrid approach by organising the agent system into multiple clusters. In each cluster, one node (known as cluster head) allocates tasks to the members of the cluster. Task allocation has diverse applications, including allocating sensing tasks to heterogeneous agents and allocating rescue missions to ambulances [72]. A complete survey on task allocation is given in Krothapalli and Deshmukh (2002) [57].

The remarkable features of MAS, including efficiency, low cost, flexibility, and reliability, make it a powerful solution to solve complex tasks. Their efficiency arises from the division of labour inherent in MAS whereby a complex task is divided into multiple smaller tasks, each of which is assigned to a distinct agent [75].

As mentioned before, my main idea is to concentrate on decentralised task allocation problems in ad-hoc teamwork. Chen et al. (2019) [30] present a related work, where they focus on estimating tasks of team-mates, instead of learning their model. While related, they focus on task inference in a model-free approach, considering that each task must be performed by one agent, and the ad-hoc agent goal changes to identifying tasks that are not yet allocated. My work, on the other hand, combines task-based inference with model-based approaches and allows for tasks to require an arbitrary number of agents. Additionally, their experiments are on small 10×10 grids, with a lower number of agents than us.

Another work attempts to identify the task being executed by a team, from a set of potential tasks [60]; or an agent's strategy for solving a repetitive task, enabling the learner to perform collaborative actions [97]. My work, however, is fundamentally different, since I focus on a set of (known) tasks which must all be completed by the team.

3.3 Modelling Team-mates

A crucial feature of MAS is the capability to reason about the behaviours, goals, and beliefs of the other agents. This reasoning occurs by forming models of the other agents. Generally, a model is a function which takes as input some portion of the observed interaction history and returns a prediction regarding the modelled agent. The interaction history may contain information such as the past actions that the modelled agent took in several conditions. The most important part of the modelling of the autonomous agent is to discover its decision-making process.

Modelling agent is not only applied in informing decisions, and it can be utilised for other purposes. An example can be an intelligent coaching system which may use a model of a specific human player in games such as Chess to recognise and point out weaknesses in the human's play [53]. The process of creating models of other agents is sometimes referred to as agent modelling. However, learning of the model can be based on information observed from the current interaction and possibly data collected in past interactions. It is also possible that an agent may model another decision making of the agent as a deterministic finite automaton and learn the parameters of the automaton (e.g. nodes, edges, labels) during the interaction [27]. Likewise, an agent may endeavour to classify the policy of another agent by employing classifiers which were trained with statistical machine learning on data obtained from recorded interactions [98].

There are several pieces of research for opponent modelling for particular domains. Pourmehr and Dadkhah (2012) [69] provides a summary of modelling methods used in 2D simulated robot soccer, in which two teams of agents compete in a soccer match. Rubin and Watson (2011) [79] has a survey in Poker playing agents which has a separate section about opponent modelling methods. Baarslag et al. (2016) [14], gives a study of the opponent modelling in bilateral negotiation settings, in which two agents adjust the values of one or more "issues" in an exchange. Bakkes et al. (2012) and Karpinskyj et al. (2014) [15] survey methods for player modelling in commercial video games, where the objective of modelling is to improve the strength and satisfaction of the player.

There is another survey that Lasota et al. (2014) [58] did in safe human-robot interaction. This survey has a section on methods which predict the motions and actions of humans. Additionally, there are other several articles that survey work in trust and reliability modelling in multi-agent systems (e.g. Pinyol and Sabater-Mir (2013) [68]; Yu et al. (2013) [104]; Ramchurn et al. (2004)) [74]. Other surveys of opponent modelling include van den Herik et al. (2005) [47], Olorunleke and McCalla (2005) [66].

3.4 Modelling in ad-hoc team

The literature introduces ad-hoc teamwork as a principled approach to handle multi-agents systems [9], [92]. This approach presents the opportunity to achieve objectives of the multiple agents in a collaborative-manner that surpasses the requirement of designing a communication channel for information exchange between the agents or the need for an application to do prior coordination. The principal aspect of ad-hoc teams is the capability to analyse the behaviours, aims, and beliefs of the other agents in the team. This reasoning can proceed by assembling models of the other agents.

Several works addressed this problem by introducing methods which utilise beliefs over a set of possible behaviours for the other agents [4], [7], [20], [21], [36], [91]. Behaviours in this approach are specified as types, which are mappings from interaction histories to probability distributions over actions. If the types are adequately representative of the true behaviours of other agents, then this method can lead to speedy adaptation, and effective interaction [8], [20]. Therefore, considering *type-based reasoning* and *parameter learning*, we can solve this problem using fine-grained models, which evaluate the observations and estimate each agent's type and parameters in an on-line manner [1], [3], [18], [19], [21]. These lines of works propose the approximation of agents' behaviour to a set of potential types to improve the ad-hoc agents' decision-making capabilities, allowing the agents' algorithms to be quickly estimated on-line, without requiring a massive training for learning their policies from scratch. However, if a set of potential types is not given by domain knowledge, then they would have to be learned from previous interactions (e.g., [19]).

Albrecht and Stone (2017) [2], in particular, introduced the AGA and ABU algorithms for *type-based* reasoning of team-mates parameters in an online manner. These methods sample sets of parameters for *gradient ascent* and *Bayesian* estimation, and were my main inspiration for this work.

On the other hand, Hayashi et al. (2020) [46] propose an enhanced particle reinvigorating process that leverages prior experiences encoded in a recurrent neural network (RNN), acting into a partial observable scenario in their ad-hoc team. However, they need thousands of previous experiences for training the RNN, while still requiring knowledge of the potential types. My approach, however, starts from scratch at every single run, with no pre-training.

Rabinowitz et al. (2018) [71] introduce a "Machine Theory of Mind", where neural networks are trained in general populations to learn agent types, and the current agent behaviour is then estimated on-line. Similarly to learning policies from scratch, however, their general models require thousands (even millions) of observations to be trained. Besides, they used a small 11×11 grid in their experiments, while I scale all the way to 45×45 to estimate the behaviour of several unknown and distinct team-mates. On the other hand, if a set of potential types is not given by domain knowledge, then their work serves as another example that types could be learned.

A different approach to learn team-mates' models and reason about their behaviour in planning is given by I-POMDP based models [29], [35], [42], [48]. However, they are computationally expensive, assuming all agents are learning about others recursively, and they consider agents with individual rewards.

On the other hand, Eck et al. (2019) recently proposed a scalable approach using the I-POMDP-Lite Framework [37] in order to consider large open agent systems. In their approach, an agent considers a large population by modelling a representative set of neighbours. They focus on estimating how many agents perform a particular action, hence their approach is not applicable to the task-based problems that I consider in this work. Additionally, although they present a scalable approach in terms of team size, they still consider only small 3×3 scenarios.

Rahman et al. (2020) also handle open agent problems, and propose the application of a Graph Neural Network (GNN) for estimating agents behaviours [73].

Similarly to other neural network-based models, it needs a large amount of training, and their results are limited to a 10×10 grid world with 5 agents. Their agent parametrisation is also more limited, with only 3 possible levels in the level-based foraging domain, which is directly given as input for each agent (instead of learned).

Recently, Panella and Gmytrasiewicz (2017) [10] proposed an extension of POMCP for ad-hoc teamwork, when agents can be represented by probabilistic deterministic finite state controllers. However, their approach still does not scale easily to a large number of agents. In fact, their results are limited to only two agents (the main planning agent, and a single unknown agent).

Another possible approach for scalability in ad-hoc teamwork is to learn a single model for a *team* of agents, instead of individual models for each agent. For instance, in the RoboCup soccer domain, Riley and Veloso (2002) [76] propose a method to identify the type of an adversarial team, which defines probabilities for agents locations in the field. Similarly, Barrett and Stone (2015) [17] assume a series of previous games with potential teams, which are used to train team policies. Then, at execution time, the most likely current team is estimated, and its corresponding policy executed. Obviously, however, a single team model is less flexible than learning models for each individual agent.

Other works directly try to learn a transition function. For instance, Guez et al. (2013) proposed a Bayesian MCTS, sampling different potential MDP models [44]. Our planning approach (inspired by [2], [18]) is similar, as I sample different agent models from my estimations. However, instead of directly working on the complex transition function space, I learn agents types and parameters, which would then translate to a particular transition probability for the current state or belief state.

3.5 Planning

Regarding on-line planning, Barrett et al. (2011) [21] introduced the idea of sampling types for each agent, based on the current beliefs, at each roll-out iteration of the UCT Monte Carlo Tree Search method [56]. Albrecht and Stone (2017) [2] employ a similar search technique, but they consider that parameters may affect the behaviour of each type, and they introduced techniques for dynamically estimating these parameters.

Concerning task allocation, MDP-based models are commonly applied [63], [64]. For instance, it can be framed as a multi-agent team decision problem [81], where a global planner calculates local policies for each agent. Auction-based approaches are also common, assigning tasks based on bids received from each agent [59]. These approaches, however, require pre-programmed coordination strategies, while I employ on-line learning and planning for ad-hoc teamwork in decentralised task allocation, enabling agents to choose their tasks without relying on previous knowledge of the other team members, and without requiring centralised planners/controllers.

Additionally, Pelcner et al. (2020) [67] recently proposed an on-line learning and planning approach for an agent to make decisions in environments containing previously unknown swarms. Similarly to us, they also learn from scratch at every run, but they focus on learning a single model for a whole swarm, while I learn a model for each agent, also considering potentially different types.

Consequently, regarding estimating team-mates parameters and types, my main novelty comes from focusing on decentralised task allocation in ad-hoc teamwork, allowing us to outperform previous algorithms. Additionally, I do not rely on neural network-based models nor I-POMDP based planners, allowing us to develop a light-weight approach that can learn from scratch at every run. On the other hand, open agent systems are not in my scope, and I do require a set of potential agent types, which may have to be pre-trained.

3.6 Summary

According to the state-of-the-art review I provided in this chapter, ad-hoc teamwork is not commonly practised. Since there is no communication or pre-coordination in ad-hoc teams, quicker and better decisions lead to improved performance. To make a faster decision, in this research, I propose a modification for the UCT Monte Carlo Tree Search algorithm, UCT-H, inspired by the representation strategy used in POMCP [89]. However, my compact representation is aimed at scalability in ad-hoc teamwork, instead of handling partial observability. Additionally, in POMCP it is assumed full knowledge of the transition function (embedded in a "black-box simulator"), while in my work the states are sampled from an estimated transition function, according to the current estimations of types and parameters for each agent. My approach for on-line planning is used for finding optimal actions while dynamically learning types and parameters, as in Albrecht and Stone (2017) [2], but leads to a significantly better performance, and scales better in terms of memory usage with team size.

Making better decisions requires predicting the future behaviours of team members. To do so, we need to learn their parameters and types. Therefore, by focusing on decentralised task allocation in ad-hoc teams, my *novel* method *OEATA* surpasses their parameter and type estimations, and consequently leads to better team performance. I also extend their work by adding partial observability to all team members.

Chapter 4

Task-based Ad-hoc Team

In this chapter, I will introduce the ad-hoc team I will be working with in this study. In addition, I will discuss the models of agents in the team. To have a better understanding of ad-hoc teamwork, we can define it as a domain where agents intend to cooperate with their team-mates and coordinate their actions to reach common goals. The agents in ad-hoc teamwork domains do not have prior communication nor coordination protocols, so learning and reasoning about the current context are mandatory to improve the team's performance. However, if agents are aware of some pre-existing standards for coordination and communication, they can try to learn about their team-mates with limited information [19]. As a result of such intelligent coordination in the ad-hoc teams, they could accomplish shared goals more efficiently.

In many domains, agents have to coordinate to handle sets of tasks that are distributed in the environment. Hence, I describe the ad-hoc team that is introduced in this work, *Task-based Ad-hoc Teamwork*, as a decentralised distributed system. In this system, there are multiple tasks to be accomplished in an uncertain environment with no centralised mechanism to allocate tasks. Therefore, agents are not managed to perform their tasks, and they autonomously decide which one to complete, without being directly allocated [23]. The *decentralised* allocation is quite natural in ad-hoc teamwork, as we cannot assume that other agents would be programmed to follow a centralised controller.

Considering this ad-hoc teamwork definition, in this chapter, I will now describe my model in detail to clarify my approaches.

4.1 Task-based Ad-hoc Teamwork

In task-based ad-hoc teamwork, there is one learning agent ϕ, that acts in the same environment as a set of non-learning agents $\omega \in \mathbf{\Omega}$, where $\phi \notin \mathbf{\Omega}$. In this ad-hoc team, the objective of the agent ϕ is to maximise the performance of the team. However, all non-learning agents are unknown to the agent ϕ. Hence, the agent ϕ must estimate and understand their model as times progresses (Figure 4.1).

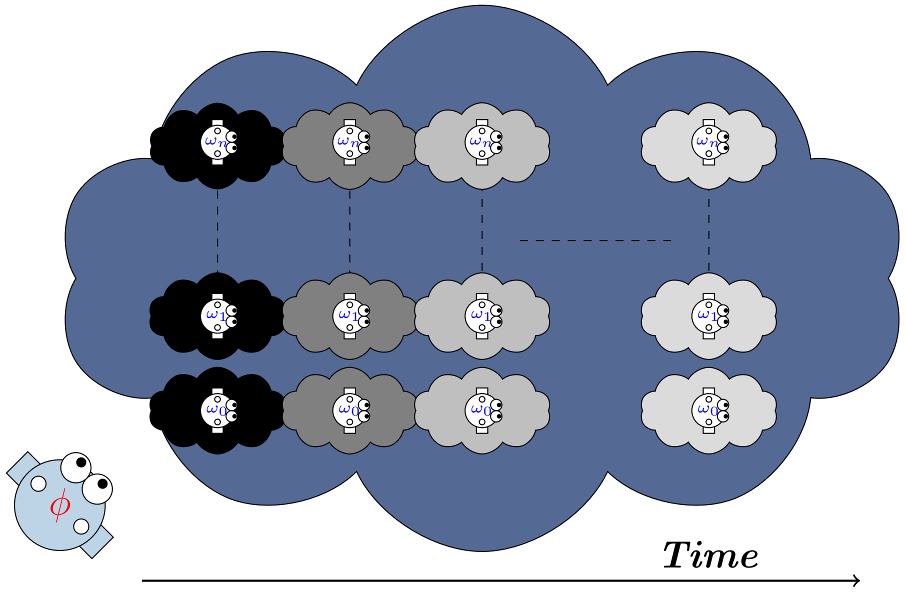


Figure 4.1: Agent ϕ tries to understand the behaviours of agents $\omega \in \mathbf{\Omega}$, which are quite vague at the beginning, but agent ϕ is able to have a better understanding of them as time goes by.

Besides, there is a set of tasks ($\mathcal{T}$) which all agents in the team endeavour to accomplish autonomously. A task $\tau \in \mathcal{T}$ may require multiple agents to perform it successfully. Additionally, the task requires many time steps to be completed. For instance, in a foraging problem, a heavy item may require two or more robots to be collected, and the robots would need to move towards the task location, taking multiple time steps to move from their initial position.

4.2 Model of Non-Learning Agents

All non-learning agents aim to finish the tasks in the environment autonomously. However, choosing and completing each task τ by each ω agent is dependent on its internal algorithm and its capabilities. Nonetheless, the algorithm of the ω agent can be one of the potential algorithms defined in the system, which might be learned from previous interactions with other agents [18]. Therefore, I suppose that there is a set of potential algorithms in the system, and I see them as a set of possible types $\boldsymbol{\Theta}$ for all $\omega \in \boldsymbol{\Omega}$, as in previous works [2]. I also assume that all these algorithms have some inputs, which I denominate as *parameters*.

Hence, the types are all parameterised, which affects agents behaviour and actions. Considering the existence of these types' parameters allows the agent ϕ to use more fine-grained models when handling new unknown agents.

According to these assumptions, I define each $\omega \in \boldsymbol{\Omega}$ as a tuple $(\theta, \mathbf{p})$, where $\theta \in \boldsymbol{\Theta}$ is ω's type and $\mathbf{p}$ represents its parameters, which is a vector $\mathbf{p} =< p_1, p_2, ..., p_n >$. Also, each element p_i in the vector $\mathbf{p}$ is defined in a fixed range $[p_i^{min}, p_i^{max}]$ [2]. These parameters can be the abilities and skills of an agent. For instance, a robot can be quite different depending on its hardware – for a robot, it can be vision radius, the maximum battery level or the maximum velocity. The parameters could also be hyper-parameters of the algorithm itself. Consequently, each $\omega \in \boldsymbol{\Omega}$, based on its type θ and parameters $\mathbf{p}$ will choose a target task. Selecting a new task (considered as the agent's "target") happens in the very first state, and whenever the agent ω finishes a task. I call these states as *Choose Target State* ($\mathfrak{s}$).

4.3 MDP Model

First, I introduce task-based ad-hoc teamwork under full observability and formalise the problem as a *Markov Decision Process (MDP)*. Although there are multiple agents in the team, I define the model *under the point of view of an agent* ϕ and apply a *single agent MDP model*, as in previous works [2], [102]. Therefore, I consider a set of states $\mathcal{S}$, a set of actions $\mathcal{A}_\phi$, a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A}_\phi \times \mathcal{S} \to [0, 1]$, and a transition probability function $\mathcal{P} : \mathcal{S} \times \mathcal{A}_\phi \times \mathcal{S} \to [0, 1]$, where the actions in the model are only the agent ϕ's actions and not any of others. Hence, the agent ϕ can only decide its own actions and has no control over the actions of agents in the set $\mathbf{\Omega}$. All ω in $\mathbf{\Omega}$ are modelled as the *environment*, as their actions indirectly affect the next state and the obtained reward, but they are not directly represented in the MDP model. Accordingly, in the *actual* problem, the next state depends on the actions of all agents. However, the agent ϕ is unsure about the following action of the non-learning agents. For this reason, I consider that given a state s, an agent $\omega \in \mathbf{\Omega}$ has a (unknown) probability distribution (pdf) across a set of actions $\mathcal{A}_\omega$, which is given by the agent ω's internal algorithm (θ, $\mathbf{p}$). Therefore, the uncertainty in the MDP model comes from the randomness of the actions of the ω agents in the team as well as the stochasticity of the environment.

This model allows us to employ single-agent on-line planning techniques, like UCT Monte Carlo Tree Search [56]. Consequently, in the tree search process, the probability distribution function (pdf) of each agent defines the transition function. At each node transition, agent ϕ samples ω agents' actions from their (estimated) pdfs, and that will determine the next state s' for the next node. However, in UCT Monte Carlo Tree

Search, the search tree increases exponentially with the number of agents. Hence, I apply the history-based version of UCT Monte Carlo Tree Search called *UCT-H*. It employs a more compact representation than the original algorithm, which helps to trace the tree in larger teams in a simpler and faster fashion (Chapter 5) [102].

As mentioned earlier, in this task-based ad-hoc team, the agent ϕ attempts to help the team to get the highest possible achievement. For this reason, the agent ϕ requires to find the optimal value function, which maximises the expected sum of discounted rewards $E[\sum_{j=0}^{\infty} \gamma^j r_{t+j}]$, where t is the current time, r_{t+j} is the reward ϕ receives at j steps in the future, $\gamma \in (0, 1]$ is a discount factor. Also, I consider that we obtain the rewards by solving the tasks $\tau \in \mathcal{T}$ of the team. That is, I define the agent ϕ's reward as $\sum r_\tau$, where r_τ is the reward obtained after the task τ completion. Note that the sum of rewards is not only across the tasks accomplished by the agent ϕ but all of them completed by any set of agents in a given state. Furthermore, there might be some tasks in the system that cannot be completed without cooperation between the agents. Accordingly, the number of required agents for finishing a task τ depends on each specific task and the set of agents that are jointly trying to complete it.

Note that the agents' types and parameters are actually not observable, but in my MDP model that is not directly considered. The estimated types and parameters are used during on-line planning, affecting the current transition function. More details are available in the next section.

4.4 Learning Team-mates

In order for the agent ϕ to be able to maximise the performance of the team, it needs to know the target task of its team-mates. Moreover, we know that based on the agent ω's type and vector of parameters, it will choose a task τ and will try to finish it by applying various actions $a \in \mathcal{A}_\omega$.

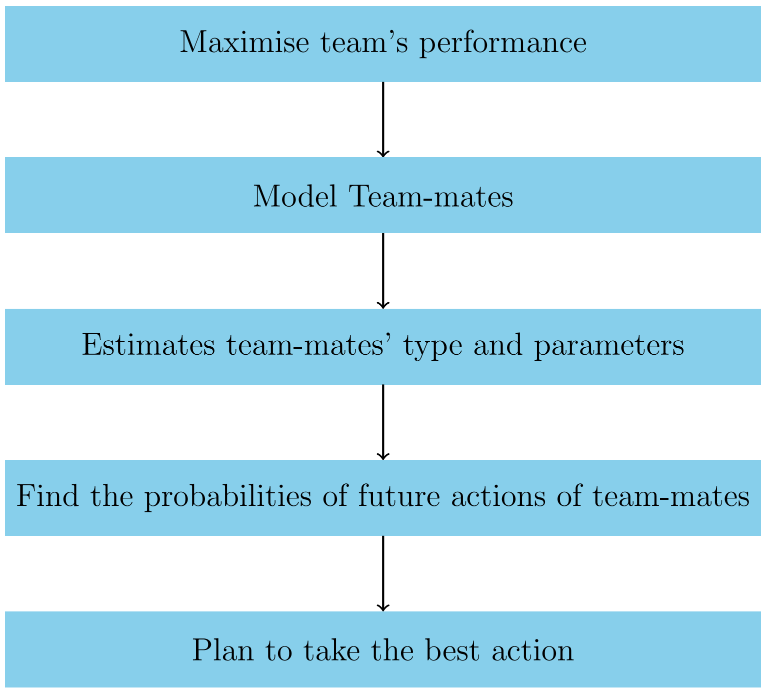

Maximise team's performance
Model Team-mates
Estimates team-mates' type and parameters
Find the probabilities of future actions of team-mates
Plan to take the best action

Figure 4.2

4.4.1 Estimating team-mates behaviour

Since the agent ϕ does not have information about each agent ω's true type θ^* and true parameters $\mathbf{p}^*$, it will not know how they may behave at each state. Hence, the agent ϕ attempts to have an appropriate estimation for type θ and parameter $\mathbf{p}$ of each non-learning agent in order to have better decision-making. At the end of the estimation process, agent ϕ will learn a probability for each type, as well as a corresponding estimated parameter vector.

Algorithm 5 gives more details about the process of how the agent ϕ estimates models for all $\omega \in \mathbf{\Omega}$. I assume that the agent ϕ does not have enough previous information about the type and parameters of non-learning agents. Therefore, for each $\omega \in \mathbf{\Omega}$, I use uniform distributions for initialising the probability of having each type $\theta \in \mathbf{\Theta}$. Accordingly, I randomly initialise each parameter in the parameter vector $\mathbf{p}$ based on their corresponding value ranges. However, given some domain knowledge, it could be sampled from a varied distribution both for types and for parameters. Hence, for each agent ω, the agent ϕ produces a parameter vector $\mathbf{p}$ for each $\theta \in \mathbf{\Theta}$, and each element of the vector, p_i, is generated randomly in its corresponding fixed range.

In the further steps, as the agent ϕ observes the behaviour of all $\omega \in \mathbf{\Omega}$, it notices their actions and the tasks that they accomplish. Therefore, it keeps updating all the estimated parameter vectors $\mathbf{p}$, and the probability of each type $\mathsf{P}(\theta)_\omega$, based on the current state. The way these estimations are updated depends on which on-line learning algorithm is employed.

Hence, to improve the ad-hoc agent ϕ's decision-making, I introduce a novel algorithm *Online Estimators for Ad-hoc Task Allocation (OEATA)* for parameter and type estimation of the team-mates where the task allocation is decentralised. I will describe OEATA in more detail in Chapter 6. I compare my method to the state of the art, Approximate Gradient Ascent, Approximate Bayesian Update [2] and POMCP [89] which will be explained in more detail in further sections (Sections 4.5.2, 4.5.1, 4.5.3).

Algorithm 5 Learning agent estimates the non-learning agent

```
1: procedure ProcessEstimation($\omega$,$s_c$)                    ▷ $s_c$ is the Current State
2:     for all $\theta_i \in \mathbf{\Theta}$ do
3:         if First Step then                               ▷ Initialisation in the first step
4:             for all $p_j \in \mathbf{p}_{\theta_i}$ do
5:                 $p_j \leftarrow$ random value from Uniform Distribution        ▷ Each $p_j$ is uniformly sampled from the parameter range.
6:             end for
7:             $\mathsf{P}(\theta_i)_\omega = \frac{1.0}{|\mathbf{\Theta}|}$
8:         else
9:             $\mathbf{p}_{\theta_i} = Estimate\ NewParameterVector(\omega, s_c, \theta_i)$
10:            $\mathsf{P}(\theta_i)_\omega = Update\ Type\ Probability\ (\theta_i, \mathbf{p}_{\theta_i})$
11:        end if
12:    end for
13:    Normalise Probabilities Of Types($\mathbf{\Theta}$)
14:    return probability vector and related parameters
15: end procedure
```

4.4.2 Planning of the learning agent

The current estimated models of the non-learning agents are used for on-line planning, allowing the agent ϕ to estimate its best actions. In particular, in this work, I employ UCT-H (more details in Chapter 5) for the agent ϕ's decision-making method. As previously stated, UCT-H is similar to UCT, but using a history-based compact representation. I will explain UCT-H in more detail in the next Chapter. I verify that this modification leads to better results in ad-hoc teamwork problems (Section 5.3).

As in previous works [2], [102], I sample a type for each non-learning agent from the estimated type probabilities each time I re-visit the root node during the tree search process. I utilise the newly estimated parameters for the corresponding sampled type. Consequently, the higher the quality of the type and parameter estimations, the better will be the result of the tree search process. As a result, the agent ϕ decides which action to take.

4.4.3 Wrong type

Note that the actual non-learning agents may be using various algorithms than the ones available in our set of types $\boldsymbol{\Theta}$. Nonetheless, the agent ϕ would still be able to estimate the best type θ and parameters $\mathbf{p}$ to approximate agent ω's behaviour. Additionally, ω agents may or may not run algorithms that explicitly model the problem as decentralised task allocation, but I only need the agent ϕ to be able to model the problem as such.

4.5 Estimation Methods

I apply the state-of-the-art methods from the literature (Approximate Gradient Ascent, Approximate Bayesian Estimation, POMCP) besides my novel method to be able to compare it with them. Therefore, first, I will review the other algorithms, and then I will introduce my novel algorithm.

Approximate Gradient Ascent, and Approximate Bayesian Estimation are introduced in Albrecht and Stone (2017) [2]. In that work, the probability of taking the action a^t_ω at time step t, for agent ω, is based on $\mathsf{P}(a^t_\omega|H^t_\omega, \theta_\omega, \mathbf{p})$ where $H^t_\omega = (s^0_\omega, ..., s^t_\omega)$ is the ω agent's history of observations at time step t, θ_ω is a type in Θ, and $\mathbf{p}$ is the parameter vector which is estimated for type θ_ω. For estimation method, a function f is defined as $f(\mathbf{p}) = \mathsf{P}(a^{t-1}_\omega|H^{t-1}_\omega, \theta_\omega, \mathbf{p})$ where $f(\mathbf{p})$ represents the probability of the agents previous action a_ω^{t-1}, given the history of observations of ω agent in previous time step, H_ω^{t-1}, type θ_ω and its corresponding parameter vector $\mathbf{p}$.

After estimating the parameter for ω agent for the selected type θ_ω, the probability of having type θ_ω is updated like below:

$$\mathsf{P}(\theta_\omega | H_\omega^t) \propto \mathsf{P}(a_\omega^{t-1} | \theta_\omega, \mathbf{p}_\omega^t) \times \mathsf{P}(\theta_\omega | H_\omega^{t-1}) \tag{4.1}$$

4.5.1 Approximate Gradient Ascent

The main idea of this method is to update the estimated parameters of the ω agent by following the gradient of a type's action probabilities based on its parameter values. Algorithm 6 provides a summary of this method.

Algorithm 6 Approximate Gradient Ascent

```
1: procedure AGA Estimation($\mathbf{p}^{t-1}, d$)
2:     Collect samples $\mathbf{D} = (\mathbf{p}^{(l)}, f(\mathbf{p}^{(l)}))$
3:     Fit polynomial $\hat{f}$ of degree $d$ to $\mathbf{D}$
4:     Compute gradient $\nabla\hat{f}(\mathbf{p}^{t-1})$ and step size $\lambda^t$
5:     Update estimate $\mathbf{p}^t = \mathbf{p}^{t-1} + \lambda^t \nabla\hat{f}(\mathbf{p}^{t-1})$
6: end procedure
```

First of all, the method collects samples $(\mathbf{p}^{(l)}, f(\mathbf{p}^{(l)}))$, and stores them in a set $\mathbf{D}$ (Line 2). The method for collection could be, for example, using a uniform grid over the parameter space that includes the boundary points. After collecting a set of samples, the algorithm, in Line 3, fits a polynomial $\hat{f}$ of some specified degree d according to the collected samples. By fitting $\hat{f}$, the gradient $\nabla\hat{f}$ with some suitably chosen step size λ^t is calculated in the next Line 4. At the end, in Line 5, the estimated parameter is updated as presented in Equation 4.2.

$$\mathbf{p}^t = \mathbf{p}^{t-1} + \lambda^t \nabla \hat{f}(\mathbf{p}^{t-1}) \tag{4.2}$$

These steps define the AGA algorithm to estimate the agent's parameters and type iteratively. For further details, I recommend reading Albrecht and Stone (2017) [2].

4.5.2 Approximate Bayesian Update

In this method, rather than using $\hat{f}$ to perform gradient-based updates, Albrecht and Stone use $\hat{f}$ to perform Bayesian updates that retain information from past updates. Hence, in addition to the belief $\mathsf{P}(\theta_\omega|H^t_\omega)$, agent ϕ now also has a belief $\mathsf{P}(\mathbf{p}|H^t_\omega, \theta_\omega)$ to quantify the relative likelihood of parameter values $\mathbf{p}$, for agent ω, when considering type θ_ω. This new belief is represented as a polynomial of the same degree d as $\hat{f}$. Algorithm 7 provides a summary of the Bayesian update.

Algorithm 7 Approximate Bayesian

```
1: procedure ABU Estimation($\mathbf{p}$)
2:     Fit $\hat{f}$ to $f$ as in Algorithm 6
3:     Compute polynomial product $\hat{g} = \hat{f} \cdot \mathsf{P}(\mathbf{p}|H^{t-1}_\omega, \theta_\omega)$
4:     Collect samples $D = (\mathbf{p}^{(l)}, \hat{g}(\mathbf{p}^{(l)}))$
5:     Fit new polynomial $\hat{h}$ of degree $d$ to $D$
6:     Compute integral $I = \int_{\mathbf{p}^{min}}^{\mathbf{p}^{max}} \hat{h}(\mathbf{p})dp$
7:     Set new belief $\mathsf{P}(\mathbf{p}|H^t_\omega, \theta_\omega) = \hat{h}/I$
8:     Extract estimate $\mathbf{p}^t$ from $\mathsf{P}(\mathbf{p}|H^t_\omega, \theta_\omega)$
9: end procedure
```

After fitting $\hat{f}$ (Line 2), the convolution polynomial of $\mathsf{P}(\mathbf{p}|H^{t-1}_\omega, \theta_\omega)$ and $\hat{f}$ results in a polynomial $\hat{g}$ of degree greater than d (Line 3). Afterwards, in Line 4, a set of sample points is collected from the convolution $\hat{g}$ in the same way that is done in Approximate Gradient Ascent. Afterwards, a new polynomial $\hat{h}$ of degree d is fitted to the collected set in Line 5. Finally, the integral of $\hat{h}$ under the parameter space, and the division of $\hat{h}$ by the integral is calculated, to obtain the new belief $\mathsf{P}(\mathbf{p}|H^t_\omega, \theta_\omega)$. This new belief can then be used to obtain a parameter estimation, e.g., by finding the maximum of the polynomial or by sampling from the polynomial. For further details, I also recommend reading Albrecht and Stone [2] work.

4.5.3 POMCP-based Estimation

Although in the MDP model, agent ϕ has the full observation of the environment, it cannot observe the type and parameters of its team-mates. Therefore, I can employ POMCP [89], a state-of-the-art on-line planning algorithm for POMDPs (Partially Observable Markov Decision Process) [54]. POMCP stores a particle filter at each node of a Monte Carlo Search Tree. In this case, the unobservable part is the types and parameters of the other agents, rather than the fully observable environment. Consequently, the particles are defined as different combinations of the types and parameters for all agents in $\mathbf{\Omega}$. I.e., $[(\theta_4, \mathbf{p}_1)$, $(\theta_2, \mathbf{p}_2)$, ..., $(\theta_1, \mathbf{p}_n)]$, where each $(\theta, \mathbf{p})$ corresponds to one non-learning agent.

In the very first root, when the particles are created, I randomly assign types and parameters for each agent at each particle. Therefore, at every iteration, I sample a particle from the particle filter of the root and based on it, the estimated type and parameters of the agents will be changed. As in the POMCP algorithm, the root gets updated once a real action is taken, and a real observation is received. Therefore, for having a type probability $\mathsf{P}(\theta)_\omega$ for a certain agent ω, I calculate the frequency that the type θ is assigned to agent ω in the current root's particle filter. Additionally, for the parameter estimation, I will consider the average across the particle filter (for each type and agent combination). For further explanations about the POMCP algorithm, I recommend reading Silver and Veness (2010) [89].

4.6 Level-based Foraging Domain

The level-based foraging domain is a common problem for evaluating ad-hoc teamwork [2], [6], [102]. In this domain, a set of agents collaborate to collect items displaced in a rectangular grid-world environment in a minimum amount of time (Figure 4.3). In this foraging domain, items have a certain weight, and agents have a certain skill level, which defines how much weight they can carry. Hence, agents may need to collaborate to pick up a particular heavy item.

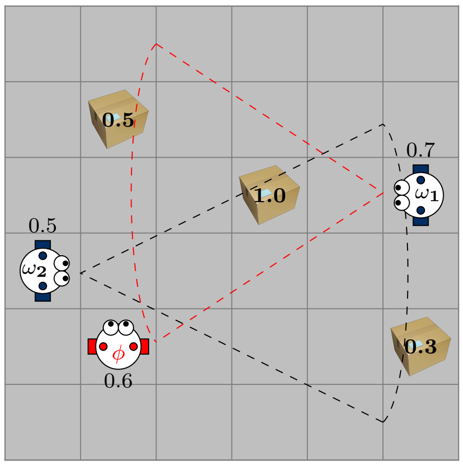


Figure 4.3: Level-based foraging domain. The number next to the boxes indicate their weight, and the one next to agents indicate their skill levels.

4.6.1 Agent's Parameters

Each agent has a visibility region and can only choose items as a target which are in its visibility cone. Therefore, to know which items are in the visibility area of each agent, I need to have the *View Angle* and the maximum *View Radius* of the agents.

Additionally, each agent has a *Skill Level* which is defining its ability to collect items. Also, each item has a certain weight, so each agent can collect items that have a weight below their *Skill Level* or equal to it. Based on what I described above, each agent can be defined by three parameters:

- l, which specifies the *Skill Level* and $l \in [0, 1]$;
- a, which is referring to *View Angle*. The actual angle of the visibility cone is given by the formula $a * 2\pi$. Additionally, it is assumed that $a \in [.1, 1]$;
- r, which is referring to the *View Radius* of the agent. The actual *View Radius* is given by $r\sqrt{w^2 + h^2}$, where w and h are the width and height of the grid. Also, the range of the radius is $r \in [.1, 1]$.

All of these parameters are applicable to all $\omega \in \mathbf{\Omega}$. Agent ϕ has the parameter *Skill Level* when it has either full or partial observability, but the *View Angle* and *View Radius* parameters are only applicable when it has partial observability.

4.6.2 Agent's Type

Concerning types of non-learning agents, I took inspiration from Albrecht and Stone (2017) [2] type definitions in the foraging domain. They considered four possible types for the agents in $\mathbf{\Omega}$: two "leader" types, which choose items in the environment to move towards, and two "follower" types, which attempt to go towards the same items as other agents, in order to help them load items. However, "follower" agents may also choose other agents as targets, while in my work I handle agents that choose tasks as targets. Therefore, I only consider "leader" agents in my work. Hence, based on agent ω's type and parameter values, a target item will be selected, and the agent's internal state (memory) will be set to the position of that target. Afterwards, the agent will move towards the target using the A^* algorithm [45]. Here is the detail for how the different types choose their targets:

- $L1$: if there are items visible, return the furthest item that has a lower weight then the agent's level; else, return $\varnothing$.
- $L2$: if there are items visible, return the item with highest weight below own level, or item with the highest weight if none are below own level; else, return $\varnothing$.
- $L3$: if there are items visible, return the closest item that has a lower weight than the agent level; else, return $\varnothing$.
- $L4$: if there are items visible, return the item with the lowest weight; else, return $\varnothing$.
- $L5$: if there are items visible, return the item with the highest weight above its own level; else return $\varnothing$.
- $L6$: if there are items visible, return an item with a lower weight than the agent's level, in the highest distance; else return $\varnothing$.

Types $L1$ and $L2$ are defined in Albrecht and Stone (2017) [2]. The other types are defined by us.

4.6.3 Actions

Each agent has five possible actions in the grid: *North*, *South*, *East*, *West*, *Load.* The first four actions will move the agent towards the selected direction, if the destination cell is empty or it is inside the grid.

The fifth action, *Load*, helps the agent to load its target item. The only time that an agent can collect an item is when the item is next to the agent, and the agent is facing it. Also, for loading the item, the *Skill Level* of the agent should be equal or higher than the items' weight. If the agent does not have enough *Skill Level* to collect the item, then a group of agents can do the job if the sum of the *Skill Levels* of the agents that surround the target is greater than or equal the item's weight. Therefore, the item can be "loaded" by a set of agents or just one agent. In the situation when the agent does not have enough ability to collect the target item, it will stand still in the same place when issuing the *Load* action. In case of collecting an item, the team of agents receives a reward of 1 and it will be removed from the grid.

4.6.4 Foraging Process

The process of foraging and choosing a target for agents ω is described in Algorithm 8.

Algorithm 8 Foraging

```
1: procedure $MoveOmega$ $(SkillLevel, ViewRadius, ViewAngle, Type)$
2:     if item in $Mem$ is collected then
3:         $Mem \leftarrow \emptyset$                                ▷ Memory to keep target
4:     end if
5:     $Loc \leftarrow location\ of\ \omega$; $Dest \leftarrow \emptyset$
6:     if $Mem \neq \emptyset$ then
7:         $Dest \leftarrow Mem$
8:     else                                                   ▷ Choose new target
9:         $I \leftarrow$ VisibleItems$(Loc, ViewRadius, ViewAngle)$
10:        $Targ \leftarrow$ ChooseTarget $(SkillLevel, Type, I)$
11:        if $Targ \neq \emptyset$ then
12:            $Dest \leftarrow Targ$
13:        end if
14:    end if
15:    $Mem \leftarrow Dest$
16:    if $Dest = \emptyset$ then
17:        Assign probability 0.2 to each action
18:    else
19:        if $Loc$ is next to $Dest$ then
20:            Assign probability 0.96 to $Load$ action
21:        else
22:            Use $A^*$ to find path from $Loc$ to $Dest$
23:            Assign probability 0.96 to first move action in the path
24:        end if
25:        Add probability 0.01 to each move action
26:    end if
27:    Return pdf over actions
28: end procedure
```

In the very first step, as agent ω has not chosen any target, the Mem, which holds the target item, is initialised to $\emptyset$. In Line 9, the *VisibleItems* routine is called, which gets the agent ω's parameters, *View Angle* and *View Radius*, and returns a set containing the visible items. In Line 10, the *ChooseTarget* routine gets the *Skill Level* and *Type* of the ω agent, and the list of visible items, returned from *VisibleItems* routine as input. The output of this routine is the target item that agent ω should go towards.

As it is shown in Line 16, there might be cases where agent ω is not able to find any target task. In these cases, all actions would get equal probabilities and consequently, it will perform actions uniformly randomly until it is able to choose a task.

I should mention that, this is an algorithm template that I assume non-learning agents are following. I use the same template in my simulations, but in practice agents ω could follow different algorithms. Hence, in Section 6.6, I will also evaluate the case where the agents *do not* follow the same algorithm as in my template.

4.7 Conclusion

In this chapter, I focused on a type-based ad-hoc team of agents attempting to complete tasks in a decentralised manner. I described how a learning agent estimates the other non-learning team-mates parameters and types to reason their future behaviour. The result will be better decisions that lead to better team performance.

Moreover, I explained the state-of-the-art methods, AGA, ABU, POMCP-based estimation, which applied for estimating parameters and types in previous works. After that, I discussed the evaluative domain, level-based foraging, in which I applied my novel methods.

Chapter 5

History-based UCT

As mentioned in Chapter 4, the ad-hoc team that is defined in this research has a learning agent ϕ. The agent ϕ attempts to make the best decision based on the estimated future behaviour of the team-mates. In this chapter, I introduce my novel algorithm called *History-based UCT (UCT-H)*. In this algorithm, the search tree will be smaller than the original UCT. Accordingly, a node would have a lower number of children, which will assist the agent ϕ to make a quicker decision when the team gets larger.

5.1 UCT-H

In this section, I propose *UCT-H*, a modification over the original UCT algorithm (described in Section 2.6) for large-scale ad-hoc teamwork. In this research, I apply the *UCT-H* for task-based ad-hoc teamwork. UCT-H can solve any ad-hoc teamwork model where there are probabilities over actions given estimations of team-mates models. In UCT, every time we start to trace the search tree, by taking the same action a, there might be separate nodes with different states, because of the uncertainty in the environment. Consequently, having multiple nodes for the same action will cause a big increase in the size of the tree.

Therefore, my main idea is to represent a *history* of states at every node n for each action (Figure 5.2). That is, instead of a node n representing a specific state s, it will represent multiple states by taking a sequence of actions, $a_0, a_1, \ldots, a_{d-1}$, from the root up to the current depth d. Accordingly, all possible states reachable from the root by the sequence of actions, $a_0, a_1, \ldots, a_{d-1}$, will be represented by exactly the same node n. Note that the root node still represents a unique state s_0. Each time I simulate taking an action a from the root towards a child node n', I will sample the next state s' by simulating taking action a in the state s_0. Similarly, each time, and I go down from a node n to a child node n', by taking action a, I will sample the next state s' by simulating taking action a in the state s (which will be fully determined by the current sequence of action simulations up to n). Afterwards, I re-start the process each time I go back to the root node for a new simulation. Hence, at each simulation, the same node may represent different states. Consequently, instead of each node storing a *Q-Table* with action-value pairs $Q(s, a)$ for a certain state s, I will store action-values $Q(h, a)$ for each *history* h.

For a more detailed understanding of UCT-H I present it in Algorithm 10 and 11. For an easier comparison of UCT and UCT-H, I mention the algorithm for the *Search* function of UCT here in Algorithm 9 one more time. In Line 9 of both algorithms, after selecting the *next action*, the *next state* is simulated in the *simulateAction* function. Consequently, the *next action* will lead us from the current node to a child node. The difference between them appears in line 10 in both algorithms The purpose of this line of the code is to either expand the tree by adding a child node to the parent node or choose an existing child node. However, the way of finding or adding a new node is varied in each algorithm. In UCT, the child only depends on the state, and for any number of actions, there will be a separate node for each state. However, in UCT-H, for each action, there is only one node.

Algorithm 9 Search in UCT

```
1: function SEARCH($state, depth$)  ▷
2:     if $Terminal(state)$ then
3:         return 0
4:     end if
5:     if $Leaf(state, depth)$ then
6:         return $Evaluate(State)$
7:     end if
8:     $action \leftarrow selectedAction(state, action)$
9:     $(nextState, reward) \leftarrow simulateAction(state, action)$
10:    $nextNode \leftarrow child(node, nextState)$
11:    $q \leftarrow reward + \gamma\ Search(nextNode, depth + 1)$
12:    $UpdateValue(node, action, q, depth)$
13:    return $q$
14: end function
```

Algorithm 10 History-based UCT

```
1: procedure UCT-H($state$)  ▷
2:     $root \leftarrow$ new $Node$
3:     repeat
4:         $Search(state, 0)$
5:     until $Timeout$
6:     return bestAction(root,0)
7: end procedure
```

Algorithm 11 Search in History-based UCT

```
1: procedure SEARCH($state, node, depth$)   ▷
2:     if $Terminal(state)$ then
3:         return 0
4:     end if
5:     if $Leaf(state, depth)$ then
6:         return $Evaluate(State)$
7:     end if
8:     $action \leftarrow selectedAction(node, depth)$
9:     $(nextState, reward) \leftarrow simulateAction(state, action)$
10:    $nextNode \leftarrow child(node, nextState, action)$
11:    $q \leftarrow reward + \gamma\ Search(nextNode, depth + 1)$
12:    $UpdateValue(node, action, q, depth)$
13:    return $q$
14: end procedure
```

Note, however, that in my case, I do not have the true MDP model, as mentioned in the previous chapter. Hence, the simulator utilised in the search tree (Line 9 for both UCT and UCT-H, respectively) does not match the true problem, for both UCT and UCT-H. It happens because the transition probability and reward functions ($\mathcal{P}$, $\mathcal{R}$) depend on the pdfs over actions given by the agents in $\mathbf{\Omega}$. These pdfs, however, are a function of the type, parameter and internal state of each $\omega \in \mathbf{\Omega}$, which are unknown.

As in Albrecht and Stone (2017) [2], each time I restart a simulation from the root node, I sample a type for each agent from my estimated type probabilities, which remains fixed for each agent for that simulation (i.e., until I reach the limited horizon l), and is re-sampled next time a simulation is re-started from the root node. Given a type, I use the currently estimated parameters when sampling the agents' pdfs to simulate the reward r and the next state s'.

5.2 Example

I clarify the differences between UCT and UCT-H with an example in this section to give readers a better understanding of the two algorithms. For this purpose, I assume a problem with two possible actions, a_0, a_1, and two possible next states per action. In Figure 5.1 and Figure 5.2, I show the root and the nodes for two levels below the root for both algorithms. Figure 5.1 demonstrates the original UCT. As there are two possible states after taking each action, therefore, after expanding the tree, the first row of the tree will look like Figure 5.1 (a), in which there are four separate nodes. Accordingly, the number of nodes increases exponentially to 16 in the second level.

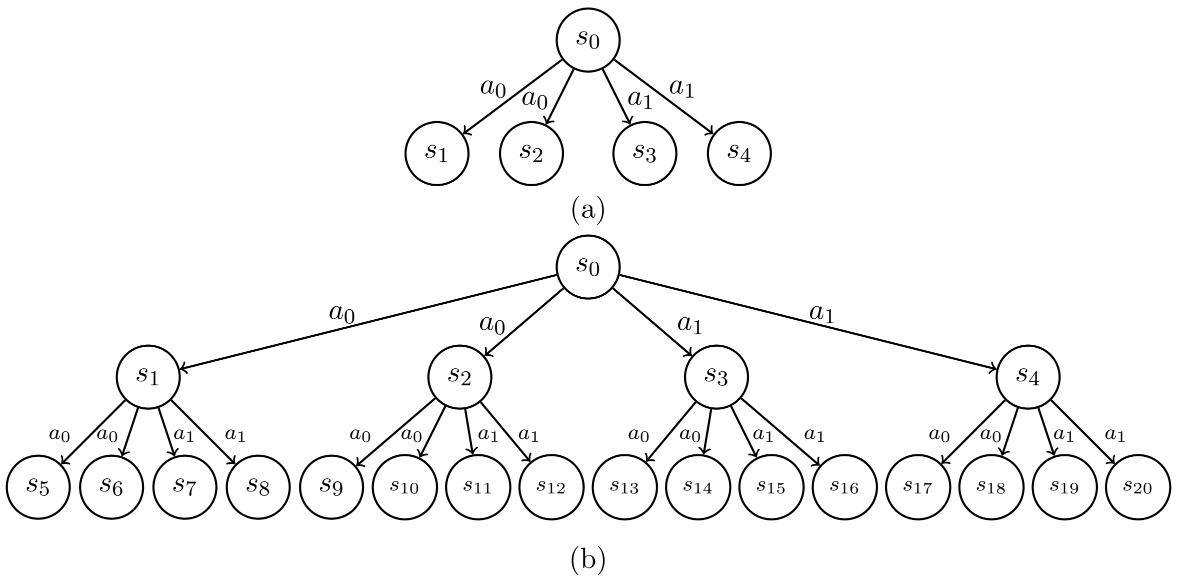

s_0
a_0 a_0 a_1 a_1
s_1 s_2 s_3 s_4
(a)
s_0
a_0 a_0 a_1 a_1
s_1 s_2 s_3 s_4
s_5 s_6 s_7 s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{14} s_{15} s_{16} s_{17} s_{18} s_{19} s_{20}
(b)

Figure 5.1: Illustration of original UCT in which the same action may lead to different states.

On the other hand, Figure 5.2 illustrates the same situation in UCT-H. As it is shown in Figure 5.2 (a), there are only two nodes after the root node, instead of four. Each node is related to each action. Accordingly, in the second level, as shown in

Figure 5.2 (b), there are only 4 nodes rather than 16.

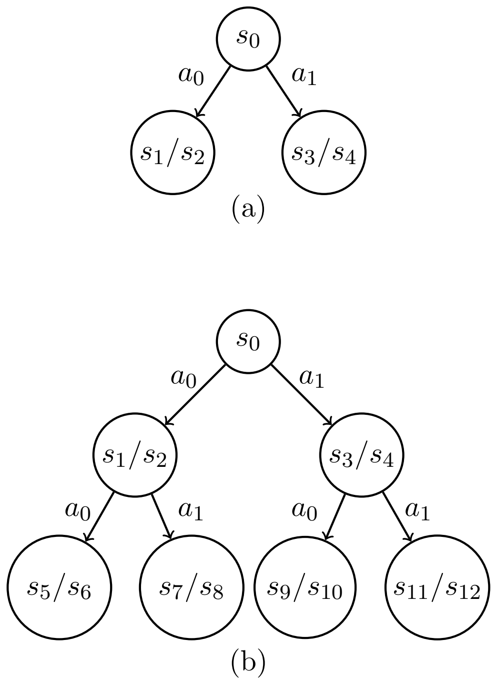
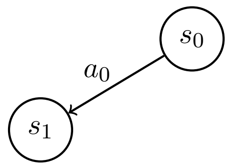


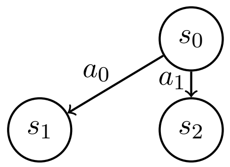
Figure 5.2: Illustration of UCT-H, which shows having multiple states in the same node for each action.

In Figure 5.3 and 5.4, I explain every step of expansions in both UCT and UCT-H with their respective *Q-Tables*. These figures show the expansion of the tree by applying both algorithms with their regarding Q-tables step by step. In the Q-table, as we see, there are 3 values for each action and each state. $\mathcal{R}$ is the cumulative reward for the corresponding state by taking the action. $\mathcal{N}$ is the total number of times that this node was visited. $\mathcal{Q}$ is the Q-Value which is the division of cumulative reward and the total number of visits: $\mathcal{Q} = \frac{\mathcal{R}}{\mathcal{N}}$.



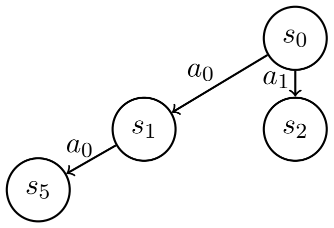
	a_0			a_1		
s_0	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$
	0.9	1	0.9	0	0	0

(a)



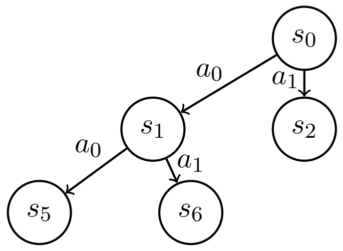
	a_0			a_1		
s_0	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$
	0.9	1	0.9	0.2	1	0.2

(b)



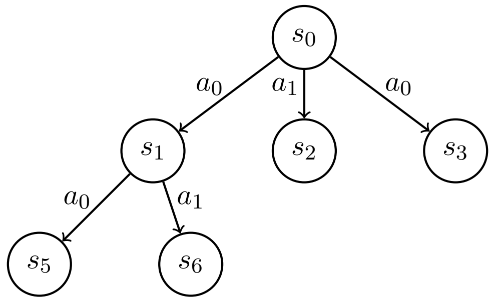
	a_0			a_1		
s_0	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$
	0.9+0.6	2	1.5 / 2	0.2	1	0.2
s_1	0.6	1	0.6	0	0	0

(c)



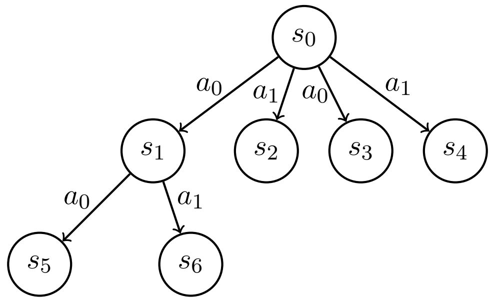
	a_0			a_1		
s_0	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$
	1.5+0.4	3	0.63	0.2	1	0.2
s_1	0.6	1	0.6	0.4	1	0.4

(d)



	a_0			a_1		
s_0	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$
	1.9+0.7	4	0.65	0.2	1	0.2
s_1	0.6	1	0.6	0.4	1	0.4

(e)



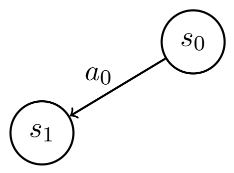
	a_0			a_1		
s_0	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$
	2.2	3	0.73	0.2+0.3	2	0.25
s_1	0.6	1	0.6	0.4	1	0.4

(f)

Figure 5.3: A step-by-step traced tree with their associated Q-tables for UCT

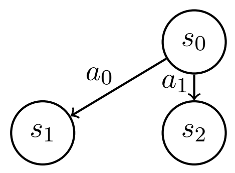
Figure 5.3, outlines how the growth of the tree happens in UCT, and Figure 5.4 demonstrates the steps in UCT-H. The way for expanding the tree and their corresponding Q-tables are the same in four initial steps. However, the difference appears in the fifth step. In this step, the action a_0 is taken for the second time from the root node.

As we see in Figure 5.3 (e), taking the action a_0 leads to a different state, s_3, and not the same as before, which was s_1. In this situation, UCT creates a new node for this new state. However, in UCT-H, despite having a new state, s_3, for a previously taken action, a new node is not created (Figure 5.4 (e)). Instead, the same node for that specific action is visited for the second time. It happens in the same way in Figure 5.3 (f) and 5.4 (f) as well.



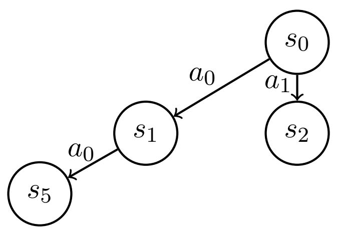
	a_0			a_1		
s_0	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$
	0.9	1	0.9	0	0	0

(a)



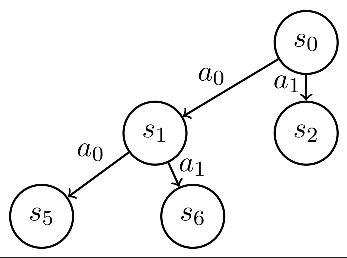
	a_0			a_1		
s_0	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$
	0.9	1	0.9	0.2	1	0.2

(b)



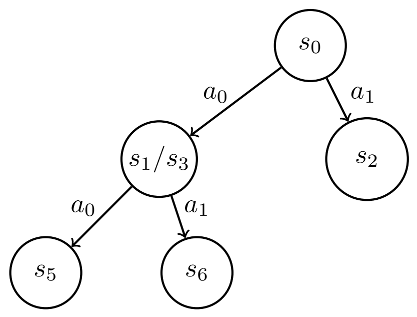
	a_0			a_1		
s_0	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$
	0.9+0.6	2	1.5 / 2	0.2	1	0.2
s_1	0.6	1	0.6	0	0	0

(c)



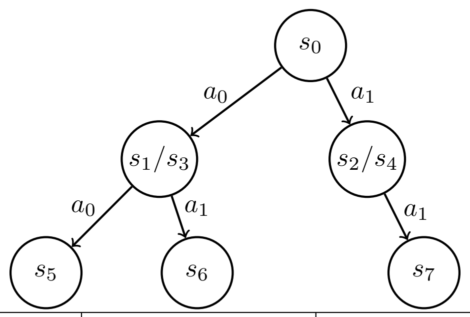
	a_0			a_1		
s_0	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$
	1.5+0.4	3	0.63	0.2	1	0.2
s_1	0.6	1	0.6	0.4	1	0.4

(d)



	a_0			a_1		
s_0	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$
	1.9+0.7	4	0.65	0.2	1	0.2
s_1/s_3	0.6+0.7	2	0.65	0.4	1	0.4

(e)



	a_0			a_1		
s_0	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{Q}$
	2.6	4	0.65	0.2+0.35	2	0.275
s_1/s_3	1.3	2	0.65	0.4	1	0.4
s_2/s_4	0	0	0	0.35	1	0.35

(f)

Figure 5.4: Tracing the tree step by step with related Q-tables for UCT-H

5.3 Evaluation

In this section, I evaluate the overall performance, computational time and memory usage of UCT and UCT-H. I ran experiments in the level-based foraging domain, as I explained in Section 4.6. For these experiments, I only considered two types, $L1, L2$, for the non-learning agents. Regarding their parameters, all assigned random values in defined range as mentioned in Section 4.6.1. I evaluate each execution of the algorithms in randomly generated scenarios. I run 15 executions per experiment and plot the average results. Error bars show the 90% confidence interval. Additionally, when I say that one result is "significantly better" than another, I mean better with statistical significance, considering $\rho < 0.1$.

I evaluated the performance across several numbers of agents ($|\mathbf{\Omega}|$), with the scenario size fixed to 20×20. I consider "performance" as the number of time steps required to collect all items in the scenario (hence, the lower the better). For both UCT and UCT-H, I performed 100 simulations for each state, and considered a limit horizon 100. Additionally, I used discount factor 0.95, and UCB1 exploration constant $\frac{0.5}{\sqrt{2}}$.

I show results for UCT and UCT-H using two different parameter estimation approaches: Approximate Gradient Ascent (AGA) and Approximate Bayesian Updating (ABU), from [2]. I do not consider the Exact Global Optimisation approach since it is significantly more computationally expensive than the other two.

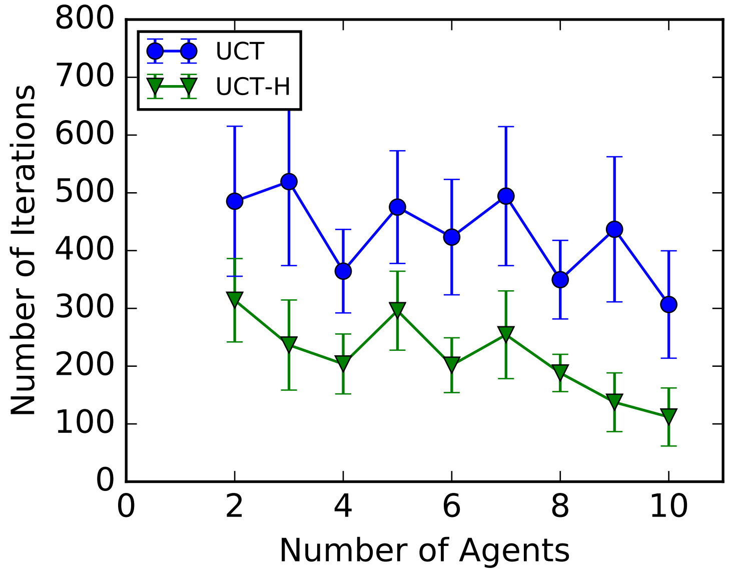


Figure 5.5: Performance of different MCTS algorithms as the number of agents increases (the lower the better).

In Figure 5.5, I show the results for an increasing number of agents ($|\mathbf{\Omega}|$). Evidently, UCT-H has always a significantly better performance than UCT. Additionally, the difference between UCT and UCT-H seems to increase with $|\mathbf{\Omega}|$: I can observe that UCT-H is around 35% better than UCT with 2 agents, but around 65% better with 10 agents.

In Figure 5.6, I evaluate the computational time per time step for each algorithm (as I limit the time of the MCTS by the number of simulations). As I can see, the difference in computational time is not significant between both algorithms. Hence, UCT-H uses about the same computational time as UCT but achieves a better performance.

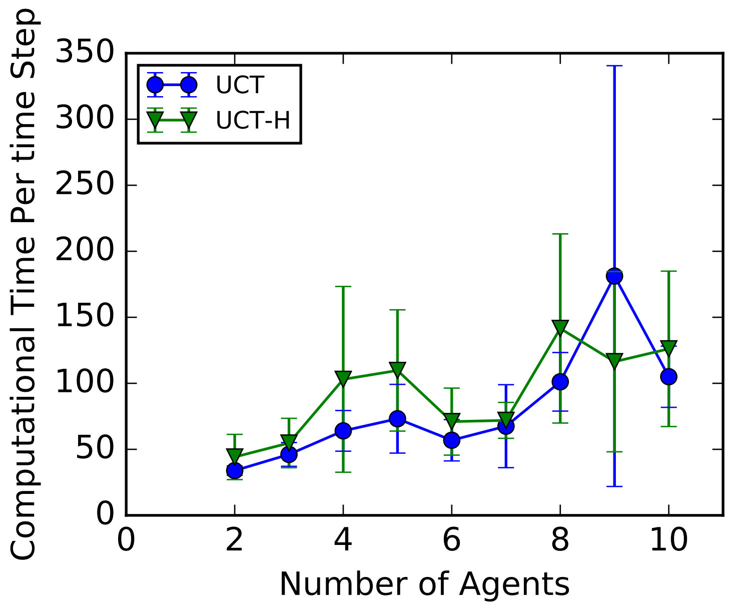


Figure 5.6: Computational time of MCTS algorithms as the number of agents increases.

I also evaluate the memory usage of both algorithms, in Figure 5.7. As it can be seen, both UCT and UCT-H tended to use a similar amount of memory up to 8 agents, although UCT tended to use more memory than UCT-H (up to 8 agents, the difference is only significant with 3 agents). For more than 8 agents, however, *UCT uses a significantly higher amount of memory.* In fact, it can be noted that UCT-H memory usage tends to remain constant with $|\boldsymbol{\Omega}|$, while UCT tends to increase exponentially as the number of agents increases. Therefore, not only UCT-H achieves a better overall performance than UCT, but it is also more scalable in terms of memory usage as the number of agents in the system grows.

Additionally, it is evident that UCT had a much larger variance than UCT-H in terms of memory usage, especially for a larger number of agents. Therefore, when using UCT-H one can have a better expectation of the amount of memory necessary to run the system.

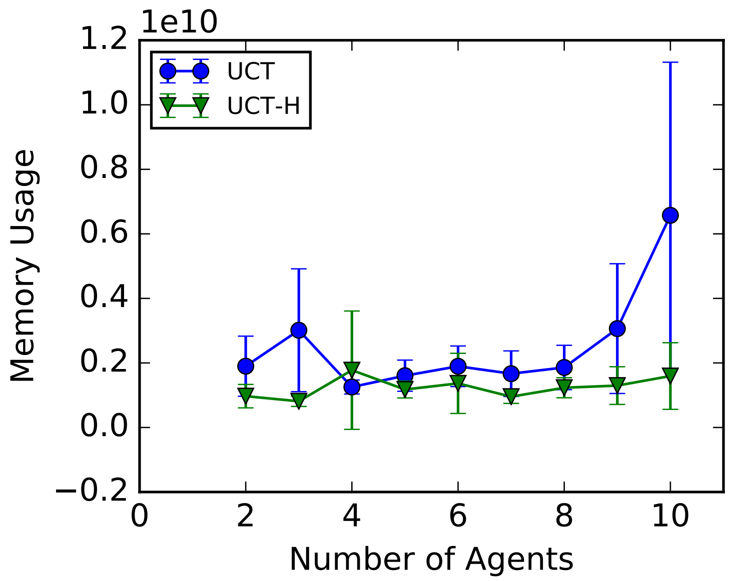


Figure 5.7: Memory usage of MCTS algorithms as the number of agents increases.

5.4 Conclusion

In this chapter, I presented a novel method, *UCT-H*, which is a lighter online planning technique for ad-hoc teamwork. My approach introduces a more compact representation, by representing each node as a *history* instead of a *state*. I have conducted several experiments in the domain of level-based foraging, a problem that requires close cooperation between agents, and as such is very well suited to the evaluation of ad-hoc teamwork. Based on my research, I have shown that my approach has better performance than existing state-of-the-art approaches, and is more efficient using roughly the same amount of computing time. The difference between my approach and the current state-of-the-art gets larger as the number of agents increases.

In addition, I assess the memory usage of my approach compared with the state-of-the-art algorithms. In my experiment, I found that my method tends to use a roughly constant amount of memory, whereas the state-of-the-art method increases exponentially as the number of agents increases. As a result, my approach is more scalable and should better handle larger teams.

Chapter 6

OEATA

In this chapter, I present my novel algorithm, *Online Estimators for Ad-hoc Task Allocation (OEATA)*, which assists the ad-hoc agent ϕ to figure out the parameters and types of non-learning team-mates autonomously. The primary conception of the algorithm is to observe each non-learning agent ($\omega \in \mathbf{\Omega}$) and record all tasks ($\tau \in \mathcal{T}$) that any one of the agents accomplishes, to compare them with the predictions of sets of *estimators*.

OEATA is inspired by Genetic Algorithms (GA) [50], since the main idea is to keep a set of *estimators*, generating new ones either randomly or using information from previously selected *estimators*. However, GAs evaluate all individuals simultaneously at each generation, and usually, they are selected to stay in the new population or for elimination according to its fitness function. Our *estimators*, on the other hand, are evaluated per agent at every task completion, and survive according to the success rate. The proportion of survived *estimators* are then used for type estimation, and new ones are generated using a different approach than the usual GA mutation/crossover.

In OEATA, there are some fundamental concepts applied during the process of evaluating parameters and types of team-mates. Therefore, I will introduce the basics of the method first and, then, explain the algorithm in detail.

6.1 OEATA Fundamentals

Sets of Estimators

In OEATA, there are sets of *estimators* $\mathbf{E}_\omega^\theta$ for each type θ and each agent ω (Figure 6.1), considering that each set $\mathbf{E}_\omega^\theta$ has a fixed number of N *estimators*. Therefore, the total number of sets of *estimators* for all agents are $|\mathbf{\Omega}| \times |\mathbf{\Theta}|$. Figure 6.1 presents this idea, relating for each agent, all possible types and corresponding *estimators*.

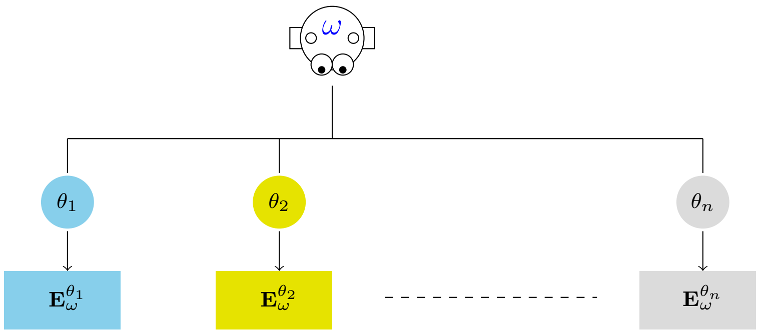

ω
θ_1
θ_2
θ_n
$\mathbf{E}_\omega^{\theta_1}$
$\mathbf{E}_\omega^{\theta_2}$
$\mathbf{E}_\omega^{\theta_n}$

Figure 6.1: For each ω agent there is a set of *estimators* for each type.

An *estimator* e of $\mathbf{E}_\omega^\theta$ is a tuple: $\{\mathbf{p}_e, \mathfrak{s}_e, \tau_e, c_e, f_e\}$, where:

- $\mathbf{p}_e$ is the vector of estimated parameters for the agent ω, and each element of the parameter vector is defined in the related element range;
- $\mathfrak{s}_e$ is the initial state or the last *Choose Target State*, where the agent ω completed a task and wants to find a new task;
- τ_e is the task that the agent ω would attempt to achieve, assuming type θ and parameters $\mathbf{p}_e$. By having estimated parameters $\mathbf{p}_e$ and type θ, I presume it is straightforward to predict the target task of the agent ω when it is at $\mathfrak{s}_e$;
- c_e holds the number of times that e was successful in predicting the next task for the agent ω;

- f_e keeps the count of the *consecutive* failures.

All *estimators* are initialised at the beginning of the process and evaluated whenever a task is accomplished. The *estimators* that are not being able to make accurate predictions after some trials are removed and replaced by *estimators* that are created using successful ones as a basis, or purely random, in a fashion inspired by genetic algorithms [50].

History of Tasks

In this method, besides having sets of *estimators* for each non-learning agent ($\omega \in \mathbf{\Omega}$), the agent ϕ keeps track of the tasks completed by each agent ω, as *History of Tasks.* Hence, *History of Tasks* is defined as $\mathbf{H}_\omega = \{(\mathfrak{s}^0, \tau^0), \ldots, (\mathfrak{s}^n, \tau^n)\}$, where $\mathfrak{s}^i$ is the ith *Choose Target State*, where the agent ω intends to identify a new target, and τ^i is the actual task that the same agent completes afterwards. As previously stated, *Choose Target State* is the initial state or the state where the agent ω accomplishes a task and wants to choose a new one.

However, the states that the agent ϕ considers as the agent ω's *Choose Target State* might not be correct because there are some situations that a specific task τ is completed by any other agent (including the agent ϕ), which could have been the target of the agent ω. In these cases, when the agent ω notices that its target is not existing anymore, it would choose a new target, and the *Choose Target State* would not be the same state when the last task was done by the agent ω (nor the initial state). Hence, as the internal state of the agent ω is not observable by the agent ϕ, there will be an estimated *Choose Target State* instead of the true one in $\mathbf{H}_\omega$. More details will be in the Section 6.2.

During the process of *OEATA*, new parameter vectors $\mathbf{p}_e$ are created in various phases. Keeping *History of Tasks* will facilitate the calculation of the success rate of the created $\mathbf{p}_e$ from the initial step. For this purpose, I define a function called

CheckHistory (Algorithm 12), which receives an agent ω, and the type θ and parameter vector $\mathbf{p}_e$ as inputs. In this function, all elements of the $\mathbf{H}_\omega$ will be evaluated to discover how many of the previously accomplished tasks τ^i can be correctly estimated, supposing that the agent ω has the parameter $\mathbf{p}_e$ and type θ. Accordingly, each element of the $\mathbf{H}_\omega$ is extracted in Line 3. Afterwards, the function *FindTarget* is called in Line 4, which aims to figure out the target τ of the agent ω, assuming that the current state is the *Choose Target State* $\mathfrak{s}^i$ and ($\mathbf{p}_e$, θ) are its parameter vector and type. If τ is equal to τ^i, the success rate will increase, and at the end, the final result is the count of correct task predictions across the whole history from the initial step.

Algorithm 12 Check History

```
1: procedure CHECKHISTORY($\omega, \theta, \mathbf{p}_e$)
2:     $Success\ Count \leftarrow 0$;
3:     for all $(\mathfrak{s}^i, \tau^i) \in \mathbf{H}_\omega$ do
4:         $\tau \leftarrow FindTarget(\omega, \mathfrak{s}^i, \mathbf{p}_e, \theta)$
5:         if $\tau^i = \tau$ then
6:             $Success\ Count \leftarrow Success\ Count + 1$;
7:         end if
8:     end for
9:     return $Success\ Count$ ;
10: end procedure
```

Bags of successful parameters

Given the vector of parameters $\mathbf{p}_e =< p_1, p_2, ..., p_n >$, if any *estimator* e succeeds, I keep each element of the parameter vector $\mathbf{p}_e$ in bags of successful parameters to use them in the future into the new parameter vectors creation (more details in Section 6.2). Accordingly, there are bags $\mathbf{B}_\omega^{\theta,i}$, for each parameter p_i in vector $\mathbf{p}_e$ and each $\mathbf{E}_\omega^\theta$. Therefore, the total number of bags are $n \times |\mathbf{\Omega}| \times |\mathbf{\Theta}|$. These bags are not erased between iterations, and hence they may increase in size at each iteration.

6.2 Process of Estimation

After presenting the fundamental elements of *OEATA*, I will explain how I define the process of estimating the parameters and type for each non-learning agent. The algorithm has five steps: (i) *Initialisation*; (ii) *Evaluation*; (iii) *Generation*; (iv) *Estimation*. Additionally, an (v) *Update* step is executed for all agents in $\boldsymbol{\Omega}$, any time a task is completed by any agent of the team, including the agent ϕ. These steps are described below:

Initialisation

At the very first step, all *estimators* should be generated and initialised. Therefore, the agent ϕ creates N *estimators* for each type $\theta \in \boldsymbol{\Theta}$ and each $\omega \in \boldsymbol{\Omega}$. If there is a lack of prior information, the parameter vectors $\mathbf{p}_e$ of each *estimator* can be initialised with a random value from the uniform distribution, in each parameter's range. For all *estimators*, in the initialisation phase, the initial state of the environment is set as the *Choose Target State* $\mathfrak{s}_e$. Since each *estimator* has a specific type θ and a particular parameter vector $\mathbf{p}_e$, it allows the agent ϕ to estimate the agent ω's task decision process in the initial state. The estimated chosen task is assigned as τ_e in the respective *estimator*. Finally, both c_e and f_e are initialised to zero.

Evaluation

The evaluation of all sets of *estimators* $\mathbf{E}^\theta_\omega$ for a particular agent ω starts when it completes a task τ_ω. The key objective of this step is to find the *estimators* that could correctly estimate the real task τ_ω that the agent ω just completed. Algorithm 13 presents the process for evaluating *estimators*.

Algorithm 13 Evaluating Estimator

```
1: procedure EVALUATION($\tau_\omega$, $\omega$, $s_c$)    ▷ $s_c$ is the Current State
2:     $e' \leftarrow \text{argmax}_{e \in \bigcup_\theta \mathbf{E}_\omega^\theta} c_e$    ▷ Get the estimator with highest success rate
3:     $\mathfrak{s} \leftarrow s_{e'}$;    ▷ Last estimated Choose Target State
4:     for each $\theta_i \in \mathbf{\Theta}$ do
5:         for each $e \in \mathbf{E}_\omega^{\theta_i}$ do
6:             if $\tau_\omega = \tau_e$ then
7:                 for each $p_i \in \mathbf{p}_e$ do
8:                     $\mathbf{B}_\omega^{\theta,i} \leftarrow \mathbf{B}_\omega^{\theta,i} \cup p_i$;    ▷ Parameters are added with repetition.
9:                 end for
10:                $c_e \leftarrow CheckHistory(\omega, \theta_i, \mathbf{p}_e) + 1$;
11:                $f_e \leftarrow 0$;
12:            else
13:                $f_e \leftarrow f_e + 1$;
14:                $c_e \leftarrow c_e - 1$;
15:                if $f_e > \xi$ then
16:                    $remove\ e\ from\ \mathbf{E}_\omega^\theta$;
17:                end if
18:            end if
19:            $\mathfrak{s}_e \leftarrow s_c$;
20:            $\tau_e \leftarrow FindTarget(\omega, \mathfrak{s}_e, \mathbf{p}_e, \theta_i)$
21:        end for
22:    end for
23:    $UpdateHistory(\tau_\omega, \mathfrak{s})$
24: end procedure
```

Consequently, for each type $\theta \in \mathbf{\Theta}$, then for every e in $\mathbf{E}_\omega^\theta$, the algorithm checks if the τ_e (the estimated task by assuming $\mathbf{p}_e$ to be agent ω's parameters with type θ in state $\mathfrak{s}_e$) is equal to τ_ω or not (Line 6 of the Algorithm 13). If they are equal then the *estimator* e is considered as successful *estimator* and each p_i in the $\mathbf{p}_e$ vector is stored in a respective bag $\mathbf{B}_{\omega}^{\theta,i}$, which is mentioned in Line 7 and 8 of the Algorithm 13. The union ($\cup$) which is applied in the equation means that new parameters would be added to the bag with repetition. If a parameter succeeds many times, it will appear in the bag with the same numbers of successes, so the chance of selecting it would be higher.

Moreover, when the estimated task τ_e is equal to the real task τ_ω, f_e is set to zero and c_e increases. However, the increment is not simply done by $c_e \leftarrow c_e + 1$ and the *CheckHistory* function (Algorithm 12) is applied for updating the c_e to find the number of successes across the whole history of agent ω's task completion so far (as shown in Line 10 of the Algorithm 13). The reason is, c_e decreases when there is a failure (Line 14). However, the lack of success might be an accident given the stochastic behaviour of non-learning agents. Thus, when a correct prediction is made with the same *estimator e*, the value of c_e is restored to the total number of successes, which can be easily done through the history $\mathbf{H}_\omega$. Conversely, if τ_e is not equal to τ_ω, then f_e is increased (Line 13) and c_e is decreased (Line 14). The first failure for the *estimator e* would not be the reason to remove it and it will be given more chances since it may still hold correct parameters. Consequently, there would be a *threshold* ξ for the removal, and if f_e is greater than ξ, the *estimator e* will be erased from its belonging set. This penalisation of *estimators* for successive failures aids us later in the type estimation.

In this step, after finding successful and failing *estimators*, the $\mathfrak{s}_e$ and τ_e of all survived *estimators* of the sets $\mathbf{E}_{\omega}^{\theta}$ will be updated. Every $\mathfrak{s}_e$ is replaced with the current state s_c, and the τ_e with the new predicted task (Line 20), by considering the current state s_c as the *Choose Target State* and assuming $\mathbf{p}_e$ as parameter vector of the ω agent, and θ as its type. Additionally, at the end of this step, as a task has just been completed, the history $\mathbf{H}_\omega$ of the corresponding ω agent is updated as well, to apply it for the coming evaluations. Notice that the agent ϕ has no access to the *true Choose Target State* of the ω agent. Even though when a non-learning agent completes a task, the *Choose Target State* of all *estimators* would be the same $\mathfrak{s}_e$ (the state where the task has just been completed), these can later change during the execution. Therefore, the *estimators* are used in the $\mathbf{E}^{\theta}_{\omega}$ sets to estimate the *Choose Target State*. That is, the *Choose Target State* is set to the one, stored in the *estimator* e with highest c_e across all sets $\mathbf{E}^{\theta}_{\omega}$. I.e., $\text{argmax}_{e \in \bigcup_{\theta} \mathbf{E}^{\theta}_{\omega}} c_e$. Accordingly, to obtain the previously estimated *Choose Target State*, OEATA finds the *estimator* in all sets $\mathbf{E}^{\theta}_{\omega}$ with highest c_e value (Line 2 of the Algorithm 13) and then it assign the last estimated *Choose Target State* with the $\mathfrak{s}_e$ of the selected *estimator* (Line 3). Afterwards, $(\mathfrak{s}, \tau_{\omega})$ is added to the history, where τ_{ω} is the task just completed and $\mathfrak{s}$ is the latest estimated *Choose Target State* of agent ω. Note that the process of finding the last estimated *Choose Target State* $\mathfrak{s}$ is done before the evaluation of the *estimators*, to avoid the process being affected by the changed value of the updated *estimators*.

Generation

Let's suppose that $\mathbf{E}'^{\theta}_{\omega}$ is the new set with only the surviving *estimators* for the agent ω and type θ that were not removed in the *Evaluation* step (Figure 6.2).

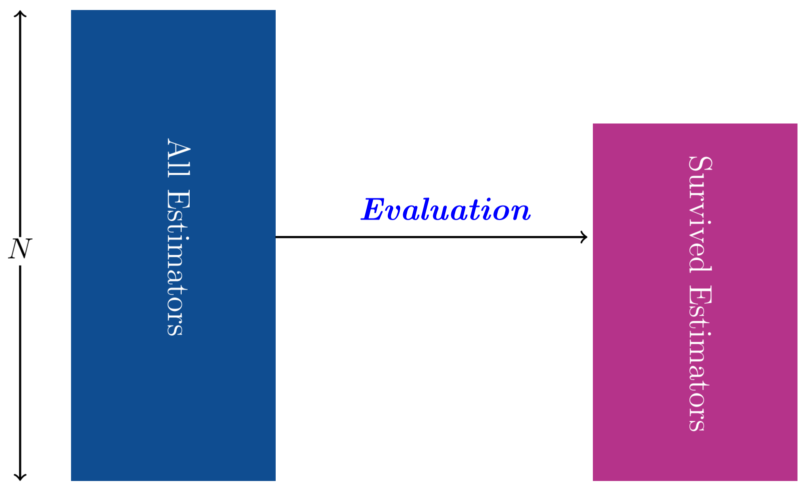


Figure 6.2: Some of the *estimators* of each set $\mathbf{E}_{\omega}^{\theta}$ will be removed after *evaluation* step.

In this step, the aim is to generate new *estimators*, in order to have the size of the sets $\mathbf{E}_{\omega}^{\theta}$ equal to N again, which means $N - |\mathbf{E}'^{\theta}_{\omega}|$ new *estimators* should be generated. Unlike the *Initialisation* step, new *estimators* are not only created with random parameters, but a proportion of them are generated using previous successful parameters from the *bags* $\mathbf{B}_{\omega}^{\theta,i}$. Accordingly, a new combination of parameters that had at least one victory in the previous steps can be utilised in generating new *estimators*. As the number of copies of the parameter p_i in the bag $\mathbf{B}_{\omega}^{\theta,i}$ is equivalent to the number of successes of the same parameter in previous steps, the chance of choosing very successful parameters will increase. Figure 6.3 shows how newly generated *estimators* are divided into two parts.

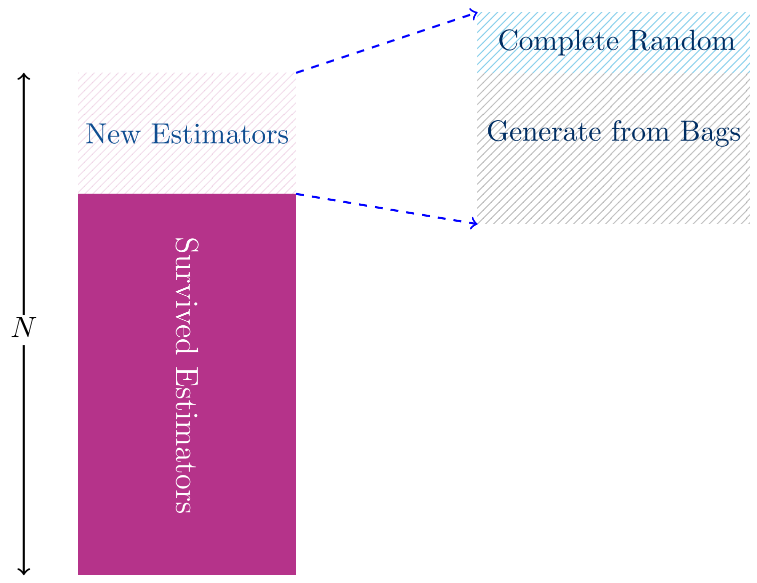


Figure 6.3: A proportion of new *estimators* are generated as a new combination of saved parameters from the respective bags and the others are randomly generated.

More detail of the process of generating new *estimators* is indicated in Algorithm 14. The main part of producing new *estimators* is creating a new parameter vector $\mathbf{p}'$, and then updating the other elements of the *estimator* accordingly. The process of creating all new parameters $\mathbf{p}'$ are shown in Lines 6 to 11 of the Algorithm 14. Parameters for a portion $(N - |\mathbf{E}'^{\theta}_{\omega}|) \times \frac{1}{m}$ (where $m > 1$) of the new *estimators* will be randomly sampled from a distribution (e.g., uniform within the parameters range, if there is no domain knowledge). The other portion $(N - |\mathbf{E}'^{\theta}_{\omega}|) \times (1 - \frac{1}{m})$ will be generated as a new combination from the corresponding bags, which are holding previously victorious parameters.

Algorithm 14 GenerateNewEstimators

```
1: procedure GENERATION($\omega$, $s_c$)                    ▷ $s_c$ is the Current State
2:     for all $\theta_i \in \mathbf{\Theta}$ do
3:         $n \leftarrow 0$;
4:         $number\ of\ mutations \leftarrow (N - |\mathbf{E}'^{\theta_i}_{\omega}|) \times \frac{1}{m}$
5:         while $|\mathbf{E}'^{\theta_i}_{\omega}| < N$ do
6:             for all $p'_i \in \mathbf{p}'$ do
7:                 if $n < number\ of\ mutations$ then
8:                     $p'_i \leftarrow random\ value\ from\ Uniform\ Distribution$    ▷ Each $p'_i$ is
   uniformly sampled from the parameter range.
9:                 else
10:                    $p'_i \leftarrow random\ value\ from\ \mathbf{B}^{\theta,i}_{\omega}$;
11:                end if
12:            end for
13:            $hist_{success} \leftarrow CheckHistory(\omega, \theta_i, \mathbf{p}')$;
14:            if $hist_{success} > 0$ then
15:                $\mathbf{p}_{e'} \leftarrow \mathbf{p}'$;
16:                $\mathfrak{s}_{e'} \leftarrow s_c$;
17:                $\tau_{e'} \leftarrow FindTarget(\omega, \mathfrak{s}_{e'}, \mathbf{p}_{e'}, \theta_i)$;
18:                $c_{e'} \leftarrow hist_{success}$;
19:                $f_{e'} \leftarrow 0$;
20:                $Add\ e'\ to\ \mathbf{E}'^{\theta_i}_{\omega}$;
21:                $n \leftarrow n + 1$;
22:            end if
23:        end while
24:    end for
25: end procedure
```

That is, each position p'_i of the parameter vector $\mathbf{p}'$ of the new *estimator* is populated by randomly sampling from the corresponding bag $\mathbf{B}^{\theta,i}_{\omega}$ (Figure 6.4). If the corresponding bag $\mathbf{B}_{\omega}^{\theta,i}$ is empty, then that position of the parameter vector will be randomly generated. If all bags are empty, then all parameters will be random.

Before creating a new *estimator* e', in Line 13 and 14 of the Algorithm 14, the *CheckHistory* function (Line 13) is employed here to check if the recently generated parameter $\mathbf{p}'$ would have at least one success across the history so far. Checking the history improves the algorithm since it decreases the likelihood of wasting an *estimator* with a parameter $\mathbf{p}'$ that would not be able to make any correct prediction in the previous steps. As a result, if the output of the function is zero, $\mathbf{p}'$ will be discarded. Otherwise, it will be considered as the parameter vector $\mathbf{p}_{e'}$ of the new *estimator* e'.

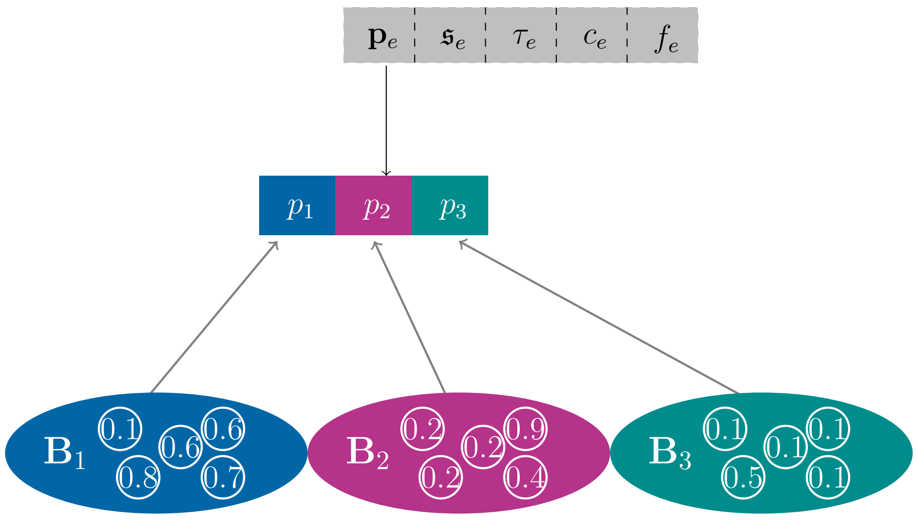


Figure 6.4: Each element p_i of new parameter vector $\mathbf{p}'$ is randomly selected from the corresponding bag $\mathbf{B}_{\omega}^{\theta,i}$.

Now, the other elements of the *estimator* e' tuple should be created. Hence, $c_{e'}$ will be assigned with the output of the *CheckHistory* function (number of successes in the *History of Tasks* $\mathbf{H}_{\omega}$), $\mathfrak{s}_{e'}$ will be set by the current state. Moreover, by assigning $\mathbf{p}'$ and θ to agent ω, the new target will be $\tau_{e'}$ (Line 15 to Line 21). At the end, the created e' will be added to $\mathbf{E}_{\omega}'^{\theta}$, and the process repeats until $|\mathbf{E}_{\omega}'^{\theta}| = N$.

Estimation

To assist the ϕ agent to have better decision-making, it is required to estimate a parameter vector and type for each $\omega \in \mathbf{\Omega}$. Therefore, at each iteration, after doing *evaluation* and *generation*, it is time to do the *estimation* step. First, based on the current sets of *estimators*, the probability distribution over the possible types is measured. For calculating the probability of agent ω having type θ, $\mathsf{P}(\theta)_\omega$, the success rate c_e of all *estimators* of the corresponding type θ is applied. That is, for each $\omega \in \mathbf{\Omega}$, the non-negative success rates c_e of all *estimators* in $\mathbf{E}^\theta_\omega$ of each type θ are added up:

$$k^\theta_\omega = \sum_{e \in \mathbf{E}^\theta_\omega} \max(0, c_e) \tag{6.1}$$

It means that I want to find out which set of *estimators* is the most successful in estimating correctly the tasks that the corresponding non-learning agent completed. In the next step, the calculated k^θ_ω is normalised to convert it to a probability estimation:

$$\mathsf{P}(\theta)_\omega = \frac{k^\theta_\omega}{\sum_{\theta' \in \mathbf{\Theta}} k^{\theta'}_\omega} \tag{6.2}$$

After measuring the probability distribution over types for each $\omega \in \mathbf{\Omega}$, some of the aggregation rules like median, mode, or mean is used across all parameter vectors $\mathbf{p}_e$ of each set of *estimators* $\mathbf{E}^\theta_\omega$. As a result, there will be one estimated parameter vector $\mathbf{p}$ per $\theta \in \mathbf{\Theta}$ for each $\omega \in \mathbf{\Omega}$.

Update

As stated earlier, there is a possible issue that might arise in the estimation process. The problem will appear when a non-learning agent is targeting a particular task τ. However, before completing it, the agent will notice that its target task is accomplished by any of the team members (including agent ϕ). Consequently, whenever the ω agent in any state (s) notices that its task is not existing anymore, it will attempt to choose a different task at the same state. Hence, s would be a new *Choose Target State* for the agent ω. This problem would affect all *estimators* as well. Therefore, once a task τ is completed by any agent in the team, every τ_e in all sets $\mathbf{E}_\omega^\theta$ for all non-learning agents ($\omega \in \mathbf{\Omega}$) that *have not* just completed τ, will be assessed to evaluate whether there is any *estimator* e that predicts the same task as τ. If there is any e with the same task, the state s will be considered as the *Choose Target State* $\mathfrak{s}_e$ of e, and its target task τ_e will be updated accordingly based on the current parameters of the *estimator* $\mathbf{p}_e$ and the type θ of the set.

6.3 Example

For a better understanding of the method, I will explain every step with a simple example. Let us consider a foraging domain [2], [102], in which there are a set of agents in a grid-world environment as well as some items. Agents in this domain are supposed to collect items displaced in the environment.

I demonstrate a simple scenario in Figure 6.5, in which there are one learning agent ϕ, two non-learning agents ω_1, ω_2, and four items, which are in two sizes. As in all foraging problems, each task is defined as collecting a particular item, so in this scenario, there are four tasks τ^i. In addition, all non-learning agents could have two possible types θ_1 and θ_2, and two different parameters (p_1, p_2), where $p_1, p_2 \in [0, 1]$. To keep the example simple, I consider that only p_1 affects the decision-making process of ω_1 at each state, and its behaviour is as follows:

- If the type is θ_1, and $p_1 \geq 0.5$, then ω_1 goes towards the smallest and farthest item (τ^0).

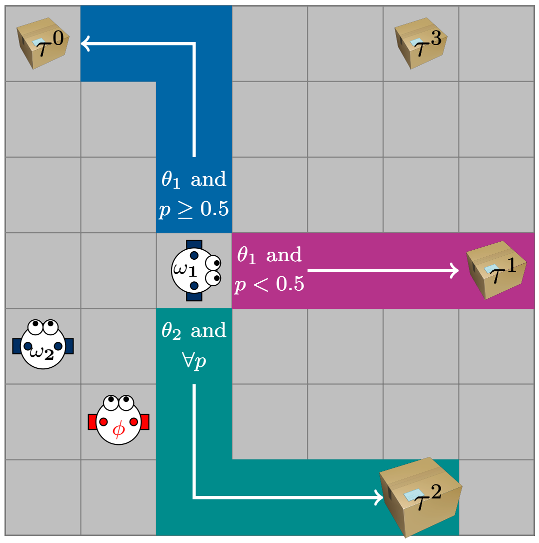


Figure 6.5: Example showing the learning agent ϕ thinking about the ω agents' behaviour, when performing foraging.

- If the type is θ_1, and $p_1 < 0.5$, then ω_1 goes towards the smallest and closest item (τ^1).
- If the type is θ_2, $\forall p_1 \in [0, 1]$, ω_1 goes towards the biggest and closest item (τ^2).

Accordingly, in the example scenario, there are four sets of *estimators*, two for each non-learning agent: $\mathbf{E}^{\theta_1}_{\omega_1}$, $\mathbf{E}^{\theta_2}_{\omega_1}$, $\mathbf{E}^{\theta_1}_{\omega_2}$, $\mathbf{E}^{\theta_2}_{\omega_2}$. I assume that the total number of *estimators* in each set is 5 ($N = 5$). Additionally, I suppose that the true type of the ω_1 agent is θ_1, and the true parameter vector is $(0.2, 0.5)$. Here, I will focus on the set of *estimators* for agent ω_1.

First step is the *Initialisation* step, where I start creating random *estimators*, as indicated in Table 6.1. To make the example simple, I define the *state* as only the position of agent ω_1. Therefore, I set each $\mathfrak{s}_e$ with the initial position of ω_1, which is $(3, 4)$. Afterwards, I create the parameter vectors $\mathbf{p}_e$ by randomly sampling from the uniform distribution, which should be done separately for both p_1 and p_2. After generating the parameter vector, the ϕ agent simulates ω_1's task decision-making process for each *estimator* in the sets $\mathbf{E}_{\omega_1}^{\theta_1}$ and $\mathbf{E}_{\omega_1}^{\theta_2}$, and obtain the corresponding target task τ_e based on the type and parameter of each *estimator*. In addition, all f_e and c_e will be initialised as zero. All initial *estimators* for both sets are shown in Table 6.1.

$\mathbf{p}_e(p_1, p_2)$	$\mathfrak{s}_e$	τ_e	c_e	f_e
$(0.4, 0.6)$	$(3, 4)$	τ^1	0	0
$(0.5, 0.3)$	$(3, 4)$	τ^0	0	0
$(0.6, 0.2)$	$(3, 4)$	τ^0	0	0
$(0.2, 0.5)$	$(3, 4)$	τ^1	0	0
$(0.9, 0.8)$	$(3, 4)$	τ^0	0	0

(a) Initial *estimators* for type θ_1

$\mathbf{p}_e(p_1, p_2)$	$\mathfrak{s}_e$	τ_e	c_e	f_e
$(0.1, 0.3)$	$(3, 4)$	τ^2	0	0
$(0.8, 0.7)$	$(3, 4)$	τ^2	0	0
$(0.3, 0.5)$	$(3, 4)$	τ^2	0	0
$(0.6, 0.9)$	$(3, 4)$	τ^2	0	0
$(0.2, 0.1)$	$(3, 4)$	τ^2	0	0

(b) Initial *estimators* for type θ_2

Table 6.1: *Estimator* sets $\mathbf{E}_{\omega_1}^{\theta_1}$, $\mathbf{E}_{\omega_1}^{\theta_2}$ after *Initialisation* step.

After some iterations, based on the true type and parameters of the agent ω_1, it gets the item that corresponds to the task τ^1. As I previously stated, whenever a task is done by an agent the process of estimation will start. The process starts with the *Evaluation* step, where all *estimators* of two sets $\mathbf{E}_{\omega_1}^{\theta_1}$, $\mathbf{E}_{\omega_1}^{\theta_2}$ will be evaluated. If the task τ of any *estimator* e equals to τ^1 then its success counter c_e increases by 1, otherwise it decreases. Moreover, in failure cases, the counter of consecutive failures f_e increases with one unit. All new values are shown in the Table 6.2.

If we suppose that the threshold for removing *estimators* is equal to *one* ($\xi = 1$), then there will be two surviving *estimators* at $\mathbf{E}_{\omega_1}^{\theta_1}$ and no one in $\mathbf{E}_{\omega_1}^{\theta_2}$. The results are displayed in Table 6.3. Hence, the bags for θ_1 are: $\mathbf{B}_{\omega_1}^{\theta_1,1} = \{0.4, 0.2\}$; $\mathbf{B}_{\omega_1}^{\theta_1,2} = \{0.6, 0.5\}$ but the ones for θ_2 are empty.

$\mathbf{p}_e(p_1,p_2)$	$\mathfrak{s}_e$	τ_e	c_e	f_e
$(0.4, 0.6)$	$(3,4)$	τ^1	1	0
$(0.5, 0.3)$	$(3,4)$	τ^0	-1	1
$(0.6, 0.2)$	$(3,4)$	τ^0	-1	1
$(0.2, 0.5)$	$(3,4)$	τ^1	1	0
$(0.9, 0.8)$	$(3,4)$	τ^0	-1	1

(a) *Estimators* for type θ_1

$\mathbf{p}_e(p_1,p_2)$	$\mathfrak{s}_e$	τ_e	c_e	f_e
$(0.1, 0.3)$	$(3,4)$	τ^2	-1	1
$(0.8, 0.7)$	$(3,4)$	τ^2	-1	1
$(0.3, 0.5)$	$(3,4)$	τ^2	-1	1
$(0.6, 0.9)$	$(3,4)$	τ^2	-1	1
$(0.2, 0.1)$	$(3,4)$	τ^2	-1	1

(b) *Estimators* for type θ_2

Table 6.2: *Estimator* sets $\mathbf{E}^{\theta_1}_{\omega_1}$, $\mathbf{E}^{\theta_2}_{\omega_1}$ after updating c_e and f_e.

$\mathbf{p}_e(p_1,p_2)$	$\mathfrak{s}_e$	τ_e	c_e	f_e
$(0.4, 0.6)$	$(3,4)$	τ^1	1	0
$(0.2, 0.5)$	$(3,4)$	τ^1	1	0

(a) *Estimators* for type θ_1

$\mathbf{p}_e(p_1,p_2)$	$\mathfrak{s}_e$	τ_e	c_e	f_e

(b) *Estimators* for type θ_2

Table 6.3: *Estimator* sets $\mathbf{E}^{\theta_1}_{\omega_1}$, $\mathbf{E}^{\theta_2}_{\omega_1}$ after *Evaluation* step.

After *Evaluation* step, it is time for the *Generation* step. By supposing $m = 3$, then $(1 - \frac{1}{3}) \times (5 - 2) = 2$ new *estimators* are generated by randomly sampling from these bags, while $\frac{1}{3} \times (5 - 2) = 1$ *estimator* is generated randomly from the uniform distribution. Hence, I may create new *estimators* with the following parameters: $(0.4, 0.5); (0.2, 0.6); (0.8, 0.7)$, where the last vector is fully random. For $\mathbf{E}^{\theta_2}_{\omega_1}$, as all *estimators* were removed then the corresponding bags are empty. Consequently, the whole set $\mathbf{E}^{\theta_2}_{\omega_1}$ will be generated using the uniform distribution as in the initialisation process.

Note that ω_1's new position will be $(5, 4)$, next to the box τ^1 it has just collected (Figure 6.6). Therefore, the current state will be the new *Choose Target State* for the agent ω_1, and all $\mathfrak{s}_e$ for all *estimators* of both sets will be updated by the current state. Now as I have *Choose Target State*, type and parameter vector of the agent ω_1, it is possible to find a new target τ_e for each *estimator* in the sets $\mathbf{E}^{\theta_1}_{\omega_1}$ and $\mathbf{E}^{\theta_2}_{\omega_1}$. All new *estimators* and updated values are shown in Table 6.4.

$\mathbf{p}_e(p_1, p_2)$	$\mathfrak{s}_e$	τ_e	c_e	f_e
$(0.4, 0.6)$	$(5, 4)$	τ^3	1	0
$(0.2, 0.5)$	$(5, 4)$	τ^3	1	0
$(0.2, 0.5)$	$(5, 4)$	τ^3	0	0
$(0.2, 0.6)$	$(5, 4)$	τ^3	0	0
$(0.6, 0.7)$	$(5, 4)$	τ^0	0	0

(a) *Estimators* for type θ_1

$\mathbf{p}_e(p_1, p_2)$	$\mathfrak{s}_e$	τ_e	c_e	f_e
$(0.1, 0.3)$	$(5, 4)$	τ^2	0	0
$(0.8, 0.7)$	$(5, 4)$	τ^2	0	0
$(0.3, 0.5)$	$(5, 4)$	τ^2	0	0
$(0.6, 0.9)$	$(5, 4)$	τ^2	0	0
$(0.2, 0.1)$	$(5, 4)$	τ^2	0	0

(b) *Estimators* for type θ_2

Table 6.4: $\mathbf{E}^{\theta_1}_{\omega_1}$, $\mathbf{E}^{\theta_2}_{\omega_1}$ sets after *Generation* step.

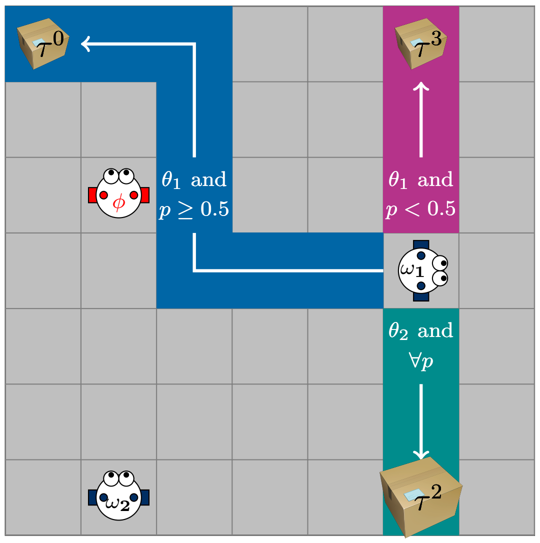


Figure 6.6: Updated scenario when agent ω_1 completes task τ^1.

After *Generation* step, it is time to update the *History of Tasks* $\mathbf{H}_{\omega_1}$ for agent ω_1. As the agent ω_1 completed the task τ_1, which was chosen in state $(3, 4)$, I add $((3, 4), \tau^1)$ to $\mathbf{H}_{\omega_1}$. Now I can do the *Estimation* step to have a probability distribution over types, and one parameter vector per type of ω_1. At this step, to find the probability of being either θ_1 or θ_2, I apply the Equation 6.1. By considering the non-negatives c_e of all *estimators*, I have:

$$k^{\theta_1} = 2, k^{\theta_2} = 0,$$

Finally, to find the probability of each type, I use the Equation 6.2. Accordingly, the probabilities are:

$$\mathsf{P}'(\theta_1) = \frac{2}{2+0} = 1, \mathsf{P}'(\theta_2) = \frac{0}{2+0} = 0,$$

which means that the probability of being θ_1 is higher. After having the probability for each type, I use $\mathbf{E}^{\theta_1}_{\omega_1}$ for estimating parameters. Assuming aggregation by averaging, the parameter p_1 will be estimated as:

$$p_1 = (0.4 + 0.2 + 0.2 + 0.2 + 0.6)/5 = 0.32,$$

and for p_2 will be:

$$p_2 = (0.6 + 0.5 + 0.5 + 0.6 + 0.7)/5 = 0.58.$$

Concerning $\mathbf{E}^{\theta_2}_{\omega_1}$, the aggregation for estimating parameters are:

$$p_1 = (0.1 + 0.8 + 0.3 + 0.6 + 0.2)/5 = 0.4$$

$$p_2 = (0.3 + 0.7 + 0.5 + 0.9 + 0.1)/5 = 0.5$$

Thus, for type θ_1, the estimated parameter vector is $(0.32, 0.58)$ and for θ_2, it is $(0.4, 0.5)$. For deciding on the next action, in the root of the MCTS tree, agent ϕ will sample the type of ω_1 from the calculated type probabilities, which are $\{1.0, 0.0\}$, and get the corresponding parameter vector.

Note that the *estimators* of agent ω_2 also need to be updated, even though it did not collect any item. Some *estimators* in $\mathbf{E}^{\theta_1}_{\omega_2}$ and $\mathbf{E}^{\theta_2}_{\omega_2}$ may have τ_1 as the estimated task τ_e, and that is not a valid task anymore since it was already completed by ω_1. Hence, for each *estimator* e where $\tau_e = \tau^1$, agent ϕ will again simulate ω_2's task decision-making process, assuming the parameters in e, and the current state. Note that for these *estimators*, both the target task τ_e and the *Choose Target State* $\mathfrak{s}_e$ need to be updated.

6.4 Analysis

I show that as the number of tasks goes to infinite, OEATA perfectly identifies the type and parameters of all agents ω, given some assumptions. First, I consider that parameters have a finite number of decimal places. This is a light assumption, as any real number x can be closely approximated by a number x' with finite precision, without much impact in a real application (e.g., any computer has a finite precision). Hence, as each element p_i in the parameter vector is in a fixed range, there is a finite number of possible values for it. To simplify the exposition, I consider ψ as possible values per element (in general they can have different sizes). Let n be the dimension of the parameter space.

I will consider three different aggregation rules: *mean*, *median*, and *mode*. For each aggregation rule, different assumptions are necessary. Let $\mathbf{p}^*$ be the correct parameter, and θ^* be the correct type of a specific ω agent. I define $\theta^- \neq \theta^*$, and $\mathbf{p}^- \neq \mathbf{p}^*$, representing wrong types and parameters, respectively. I will also use tuples $(\mathbf{p}, \theta)$ to represent a pair of parameter estimation and type. Here we have two assumptions:

Assumption 1: Aggregation by *mode* needs the lightest assumption. I just require that any $(\mathbf{p}, \theta^-)$, and any $(\mathbf{p}^-, \theta^*)$ has a lower probability of making a correct task estimation than $(\mathbf{p}^*, \theta^*)$. This assumption is very light because if a certain pair $(\mathbf{p}, \theta^-)$ or $(\mathbf{p}^-, \theta^*)$ has a higher probability of making correct task predictions, then it should indeed be the one used for planning, and could be considered at the correct parameter and type pair.

Assumption 2: For the *mean* and *median*, I further assume that any $(\mathbf{p}, \theta^-)$, and any $(\mathbf{p}^-, \theta^*)$ will not succeed infinitely often. That is, as $|\mathcal{T}| \rightarrow \infty$ there will be cases where it successfully predicts the task, but the number of cases is limited by a finite constant c.

Additionally, I will consider the case with the lack of previous knowledge, so parameters and types will be initially sampled from the uniform distribution. As before, I denote the estimated probability of a certain agent having type θ by $\mathsf{P}(\theta)$, but I drop the subscript ω for clarity.

Theorem 1. *OEATA estimates the correct parameter for all agents as* $|\mathcal{T}| \rightarrow \infty$*. Regarding type estimation, if Assumption 1 holds,* $\mathsf{P}(\theta^*) > \mathsf{P}(\theta^-)$ *(for a sufficiently large N). Furthermore, if Assumption 2 holds,* $\mathsf{P}(\theta^*) \rightarrow 1$*.*

Proof. Because of the mutation proportion m, we always have new *estimators* with random $\mathbf{p}_e$ (since wrong parameters eventually reach the failure threshold, so new ones are generated). As we sample from the uniform distribution, $\mathbf{p}^*$ will be sampled with probability $1/\psi^n > 0$. Hence, eventually it will be generated as $|\mathcal{T}| \rightarrow \infty$. As the generation defines a Bernoulli experiment, from the geometric distribution, we have that in expectation we need ψ^n trials.

Therefore, eventually, there will be an *estimator* with the correct parameter vector $\mathbf{p}^*$. Furthermore, since $(\mathbf{p}^*, \theta^*)$ has the highest probability of making correct predictions (Assumption 1), it has the lowest probability of reaching the failure threshold ξ. Hence, as $|\mathcal{T}| \rightarrow \infty$, there will be more *estimators* $(\mathbf{p}^*, \theta^*)$, than any other *estimator*. Therefore, when considering *mode* aggregation, OEATA will correctly estimate $\mathbf{p}^*$ when assuming type θ^*.

For *mean* and *median*, any $(\mathbf{p}^-, \theta^*)$ will eventually reach the failure threshold, and will be discarded, since it succeeds at most c times by Assumption 2. Hence, when $|\mathcal{T}| \rightarrow \infty$ the *mean* or *median* across $\mathbf{E}_\omega^{\theta^*}$ will be $\mathbf{p}^*$.

Concerning type estimation, in the case of *mode*, I refer the reader to Proposition 1, which shows that OEATA gives a higher probability to θ^* when we consider only the Assumption 1.

When we consider the stronger Assumption 2 (for *mean* and *median*), then the probability of the correct type $\mathsf{P}(\theta^*) \rightarrow 1$. That is, we have that $c_e \rightarrow \infty$ in the set $\mathbf{E}_\omega^{\theta^*}$. Hence, $k_\omega^{\theta^*} \rightarrow \infty$, while $c_e < c$ for θ^- (by assumption). Therefore:

$$\mathsf{P}(\theta^*) = \frac{k_\omega^{\theta^*}}{\sum_{\theta' \in \mathbf{\Theta}} k_\omega^{\theta'}} \rightarrow 1,$$

while $\mathsf{P}(\theta^-) \rightarrow 0$, as $|\mathcal{T}| \rightarrow \infty$. □

When Assumption 2 does not hold, we may have that $k_\omega^{\theta^-} \rightarrow \infty$. However, as mentioned, I can still show that the correct type will receive a higher probability:

Proposition 1. *If a parameter estimation in the wrong type θ^- succeeds infinitely often, OEATA still gives a higher probability to the correct type θ^*, for a sufficiently large N.*

Proof. As I mentioned earlier, the parameter estimation in the correct type will eventually converge to the true parameter. Hence, as the correct parameter estimation in θ^* succeeds more frequently than parameter estimations in θ^- for sufficiently large N, $k_\omega^{\theta^*}$ would be bigger than $k_\omega^{\theta^-}$. If I consider $k_\omega^{\theta}(x)$ to denote k_ω^{θ} for x tasks. Then:

$$k_\omega^{\theta^*}(x+1) - k_\omega^{\theta^*}(x) > k_\omega^{\theta^-}(x+1) - k_\omega^{\theta^-}(x)$$

By applying the Stolz–Cesàro theorem, the limit of $\mathsf{P}(\theta)$ as $|\mathcal{T}| \rightarrow \infty$ is:

$$\lim_{|\mathcal{T}| \rightarrow \infty} \frac{k_\omega^{\theta}(|\mathcal{T}|+1) - k_\omega^{\theta}(|\mathcal{T}|)}{\sum_{\theta' \in \mathbf{\Theta}} k_\omega^{\theta'}(|\mathcal{T}|+1) - \sum_{\theta' \in \mathbf{\Theta}} k_\omega^{\theta'}(|\mathcal{T}|)}$$

Therefore:

$$\lim_{|\mathcal{T}| \rightarrow \infty} \mathsf{P}(\theta^*) > \lim_{|\mathcal{T}| \rightarrow \infty} \mathsf{P}(\theta^-)$$

□

We saw in Theorem 1 that a random search from the mutation proportion takes ψ^n trials in expectation. OEATA, however, finds $\mathbf{p}^*$ much quicker than that, since a proportion of *estimators* are sampled from the corresponding bags $\mathbf{B}_\omega^{\theta,i}$. To simplify the exposition, I will denote the bags by $\mathbf{B}^i$, since I focus on a particular agent ω, and the correct type θ^*.

To show my result formally, I make the following assumptions: (i) a correct value p_i^* in any position i may still predict the task incorrectly (since other vector positions may be wrong), but it will eventually predict at least one task correctly in at most t trials, where t is a constant; (ii) a wrong value p_i^- in any position i may still predict the task correctly (since other vector positions may be correct), but that would happen at most $\mathfrak{b}$ times for each bag, across all wrong values. Furthermore, $\mathfrak{b} \ll \psi$.

That is, if one of the vector positions i is correct, $\mathbf{p}$ will not fail infinitely, even though other elements may be incorrect. That is valid in many applications, as in some cases only one element is enough to make a correct prediction. E.g., if a task were nearby, for almost any vision radius it would be predicted as the next one if the vision angle were correct. On the other hand, the wrong values will not always succeed. That is also true in many applications: although by the argument above wrong values may make correct predictions, these are a limited number of cases in the real world. E.g., eventually, all tasks nearby will be completed, and a correct vision radius estimation becomes more important to make correct predictions. As usually ψ would be large (e.g., they may approximate real numbers), we would have $\mathfrak{b} \ll \psi$.

Proposition 2. *In expectation, OEATA finds* $\mathbf{p}^*$ *in* $O(n \times \psi \times (\mathfrak{b}+1)^n)$.

Proof. Sampling the correct value for element p_i would take ψ trials in expectation. Once a correct value is sampled, it will be added to $\mathbf{B}^i$ if it makes at least one correct task prediction. It may still make incorrect predictions because of wrong values in other elements, and it would be removed if it reaches the failure threshold ξ. However, for a constant number of trials $t \times \psi$, it would be added to $\mathbf{B}^i$. Similarly, sampling at least one time the correct value for all n dimensions would take $n \times \psi$ trials in expectation, and in at most $t \times n \times \psi$ trials all $\mathbf{B}^i$ would have at least one sample of the correct value in position i. The bags store repeated values, but in the worst case, there is only one correct example at each $\mathbf{B}^i$, leading to at least $1/(\mathfrak{b}+1)$ probability to sample the correct value per bag. Hence, given the bag sampling operation, we would find $\mathbf{p}^*$ with at most $t \times n \times \psi \times (\mathfrak{b}+1)^n$ trials in expectation. □

Hence, the complexity is close to $O(\psi)$, instead of $O(\psi^n)$ as the random search (since $\mathfrak{b} \ll \psi$).

6.5 Ad-hoc Team with Partial Observability

Assuming full visibility for the learning agent is a strong presupposition, and it rarely occurs in a real application (due to data or technology limitations). Thus, to make the application more realistic, I will consider now that the agent ϕ has limited visibility of the environment. Therefore, I formalise my problem as a *Partially Observable Markov Decision Process* (POMDP). Similar to the MDP model, I define a *single agent* POMDP model, which will allow me to adapt POMCP [89] with my *Online Estimators for Ad-hoc Task Allocation.* As before, the pdfs of non-learning agents will define the transition and reward functions.

In this section, I will outline the main changes compared to my previous MDP model (Section 4) and how I designed my POMCP-based solution to the distributed task allocation scenario.

6.5.1 POMDP model

My POMDP model also considers one agent ϕ acting in the same environment as a set of non-learning agents ($\omega \in \mathbf{\Omega}$), and the agent ϕ tries to maximise the team performance without any initial knowledge about ω agents' types and parameters.

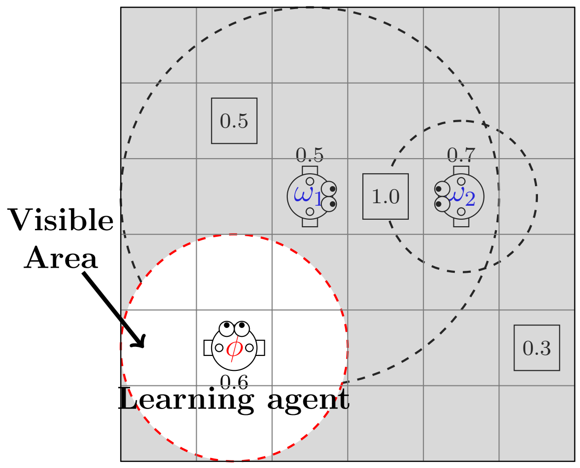


Figure 6.7: In foraging domain, I can assume an agent with a *visibility region*, like a circular sector, with a certain radius and angle, centred on the agent's position.

I consider the same set of states $\mathcal{S}$, action $\mathcal{A}$, transition probability $\mathcal{P}$ and reward function $\mathcal{R}$ defined previously. Additionally, the ϕ agent's objective is still to maximise the expected sum of discounted rewards. However, now the agent ϕ has a set of observations $\mathcal{O}$. Every action a produces an observation $o \in \mathcal{O}$, which is the visible environment in ϕ agent's point of view (all of the environment within the *visibility region*, in the state s' reached after taking action a). I assume the agent ϕ can perfectly observe the environment within the *visibility region*, but it cannot observe anything outside the *visibility region*. Hence, my POMDP model does not require an observation probability function. As before, agents' true types and parameters are not observable.

Hence, the current state cannot be observed directly by the agent ϕ, so it builds a history $\mathcal{H}$ instead. $\mathcal{H}$ consists of a set of collected information h_t from the initial timestamp $t = 0$ until the current time. Each h_t is an action and observation pair ao, representing the action a taken at time t, and the corresponding observation o that was received. The current agent history will define its *belief state*, which is a probability distribution across all possible states. Therefore, the agent ϕ must find the optimal action, for each *belief state*.

6.5.2 POMCP modification

POMCP [89], which is described in more detail in Section 2.8, is an extension of UCT for problems with partial observability. The algorithm applies an unweighted particle filter to approximate the belief state at each node in the UCT tree, and requires a *simulator*, which can sample a state s', reward r and observation o, given a state and action pair.

Each time I traverse the tree, a state is sampled from the particle filter of the root. Given an action a, the simulator samples the next state s' and the observation o. The pair ao defines the next node n in the search tree, and for the current iteration, the state of the node will be assumed to be s'. This sampled state s' is added to node n's particle filter, and the process repeats recursively down the tree. I refer the reader to Silver and Veness (2010) [89] for a detailed explanation.

However, as in the UCT case, we do not know the true transition and reward functions, since they depend on the pdfs of the non-learning agents ($\omega \in \mathbf{\Omega}$). Therefore, I employ the same strategy as previously: at each time I go through the search tree, I sample a type for each agent from the estimated type probabilities, and use the corresponding estimated parameters. These remain fixed for the whole traversal, until I re-visit the root node for the next iteration. Note that these sampled types and parameters are also going to be used in the POMCP *simulator*, when I sample a next state, a reward and an observation after choosing an action in a certain node.

As stated previously, POMCP has been modified before to sample transition functions [44]. Here, however, I am employing a technique that is commonly used in UCT (for MDPs) in ad-hoc teamwork [2], [18], but now in a partial observability scenario, which allows me to work on the type/parameter space instead of directly on the complex transition function space. I can then employ *OEATA* for the type and parameter estimation.

I employ the same OEATA algorithm described earlier in this chapter, but I must handle the cases where any agent $\omega \in \mathbf{\Omega}$ is outside the ϕ agent's visibility region. Therefore, it is not observable when the ω agent is trying to complete its tasks. To do so, I sample a particle from the POMCP root, which corresponds as sampling a state from the *belief state*. Beside the *belief state*, I assume that the agent ϕ knows when the non-learning agent has completed a task, even if it is outside our visibility region. The sampled state is then used as the *current state* in OEATA, and it can then be executed as normal. Therefore, states that are considered more likely will be sampled with a higher probability for the OEATA algorithm.

6.6 Evaluating OEATA

I will compare my novel algorithm (OEATA) against two state-of-the-art parameter estimation approaches in ad-hoc teamwork: AGA and ABU [2] (Section 4.5.1 and Section 4.5.2). As I mentioned before, in both of these approaches, for estimating parameters and types, I sample sets of parameters (for a gradient ascent step or a *Bayesian* estimation), which is similar to *set of estimators* in the OEATA. Therefore, for better comparing OEATA with these methods, I use the same set size as *estimator* sets (N). Note that Albrecht and Stone (2017) also introduced an approach called Exact Global Optimisation (EGO) [2]. I do not include it in my experiments since it is significantly slower than the ABU/AGA, without outperforming them in terms of prediction performance.

Additionally, I compare my approach against using *POMCP-based estimation* (Section 4.5.3) for type and parameter estimations. As I described earlier, in estimation with POMCP, I assume that the agent ϕ can see the whole environment. However, the team-mates' type and parameters are not observable. Hence, agent ϕ applies POMCP's particle filter for estimation. I use $N \times |\mathbf{\Omega}| \times |\mathbf{\Theta}|$ particles, matching the total number of *estimators* in my approach (since I have N per agent, for each type). I executed random scenarios in level-based foraging domain (Section 4.6) for a different number of items, agents and environment size for all estimation methods. Every run was repeated 20 times, and I plot the average results and the confidence interval ($\rho = 0.01$). When I say that a result is significant, I mean statistically significant considering $\rho \leq 0.01$, according to a *t-test*.

Configuration values for parameters of OEATA in my experiments are as follows: the number of *estimators* N is 100, the threshold for removing estimators ξ is 2 and mutation rate m is 0.2. Moreover, I will use *mean* for aggregation of *estimator* sets. For UCT-H [102], I ran 100 iterations per time step, and the maximum depth is 100.

Type and parameters of agents in $\mathbf{\Omega}$ are chosen uniformly randomly. The skill level for agent ϕ is also randomly selected. However, skill levels are randomly chosen in a way that incentivises collaborations.

That is, in the created scenarios, I manage the values for the skill level of agents and weight of items in a way that all items can be collected by them individually or with their cooperation. Additionally, I design the scenario generator to increase the likelihood of collaborations being required in order to complete the scenarios. In details, the scenario generation considers the following rules:

- For **less than four agents** in the environment ($|\mathbf{\Omega} \cup \phi| < 4$):
  - The level of each agent is given by a uniform distribution sample between 0.5 and 1.
  - The weight of each task is given by a uniform distribution sample between the *highest level* in the agent's set and the *max weight value* (equal to 1.0).
- For **four or more agents** in the environment ($|\mathbf{\Omega} \cup \phi| \geq 4$):
  - The level of each agent is given by a uniform distribution sample between 0.1 and 1.
  - The weight of each task is given by a uniform distribution sample between the *sum of 2 levels* randomly sampled from the agents' set and the *lowest value in the agents' level combination set.*

The agent's set is the collection of all agents in the environment, including ϕ. The agents' level combination set represents the set of $|\mathbf{\Omega} \cup \phi|!/(4!(|\mathbf{\Omega} \cup \phi| - 4)!)$ values between 0.4 and 1.0, consisting of the sum of all possible 4-combinations of $|\mathbf{\Omega} \cup \phi|$ levels in the agent's set (constrained to the maximum 1.0 value).

These rules make cooperation highly likely in the generated scenarios since the weight of the task considers a lower bound that, for most of the agents' skill levels, may require collaboration between two or more agents. The rule changing with the number of agents in the environment is due to the available positions to complete a task in the defined discrete domain (North, South, West and East). Thus, I aimed at creating tasks that may require four agents when four or more agents are available. My approach could increase the requirement for collaboration without generating scenarios impossible to solve when agents choose to pursue a specific task. Note, however, that the collaboration is not strictly guaranteed, and there are some cases where an agent would be able to complete a task individually.

Additionally, every task is created in random positions, but I exclude the scenario's borders. That allows agents to set up their positions to perform the load action from any direction (i.e., North, South, East, West). Therefore, it is always possible for four or fewer agents to simultaneously load an item, which guarantees that all scenarios are solvable (given the tasks weights as defined above).

First, I fix the number of possible types as two ($L1$, $L2$), and later I demonstrate the impact of increasing the number of types. For each scenario, I assume one of the four estimation methods ABU/AGA/POMCP/OEATA to be an agent ϕ's estimation method. I kept a history of estimated parameters and types for all iterations of each run and calculated the errors by having true parameters and true types in hand. Then, I evaluate the mean absolute error for the parameters, and $1 - \mathsf{P}(\theta^*)$ for type; and what I show in the plots is the average error across all parameters. Additionally, since I am aggregating several results, I calculate and plot the average error across all iterations.

However, before showing these aggregated results, I will first show examples of the parameter and type estimation error for $|\mathbf{\Omega}| = 7$ (Figure 6.8 and Figure 6.9) across all iterations. In this example, the scenario size is 20×20, and the number of items is 20. As shown in Figure 6.8, my parameter estimation error is consistently significantly lower than the other algorithms from the second iteration, and it monotonically decreases as the number of iterations increases. AGA, ABU, and POMCP, on the other hand, do not show any sign of converging to a low error as the number of iterations increases. We can also see that type estimation with OEATA becomes quickly better than the other algorithms, significantly overcoming them after a few iterations.

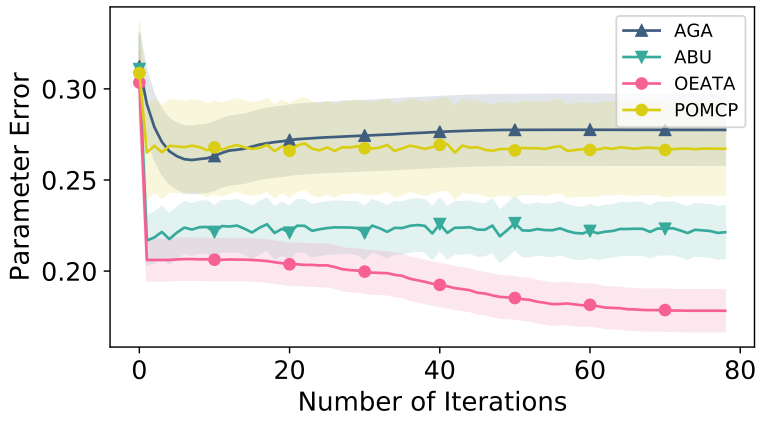


Figure 6.8: Parameter estimation errors for $|\mathbf{\Omega}| = 7$

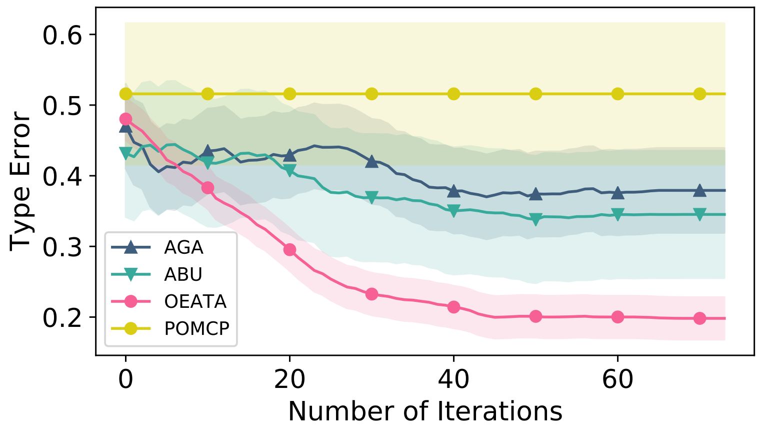


Figure 6.9: Type estimation errors for $|\mathbf{\Omega}| = 7$

6.6.1 Multiple numbers of items

Now, I display the results for different numbers of items. Therefore, I fixed the scenario size as 20×20 and the number of agents ω to 5 ($|\mathbf{\Omega}| = 5$). Then, I ran experiments for a varying number of items (20, 40, 60, 80) and the plots are shown in Figure 6.10, Figure 6.11 and Figure 6.12. As we can observe in Figures 6.10 and 6.11, OEATA has consistently lower error than the other algorithms, both in terms of parameters and type estimation. In fact, OEATA is significantly better than AGA, ABU and POMCP in terms of parameter and type estimation error for all numbers of items. The only exception is parameter estimation error for 80 items in comparison with POMCP (where significance holds with $\rho \leq 0.024$). I also figure out that OEATA can complete all tasks faster for all numbers of items, and is significantly better in almost all cases (except for AGA with 80 items, where significance holds with $\rho \leq 0.04$).

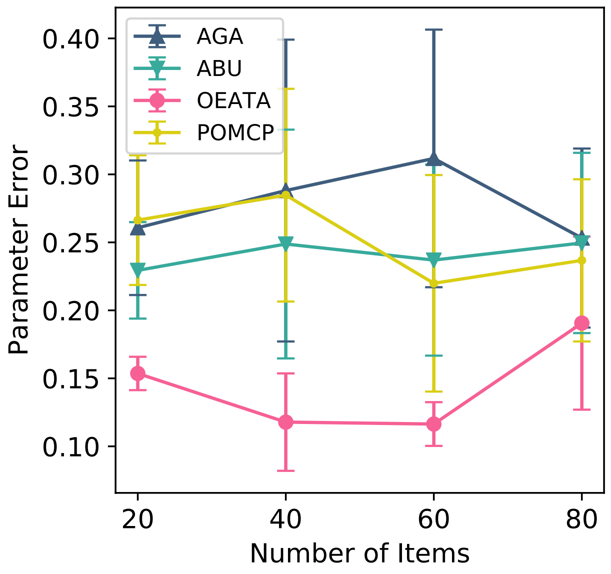


Figure 6.10: Parameter estimation errors for a varying number of items with full observability.

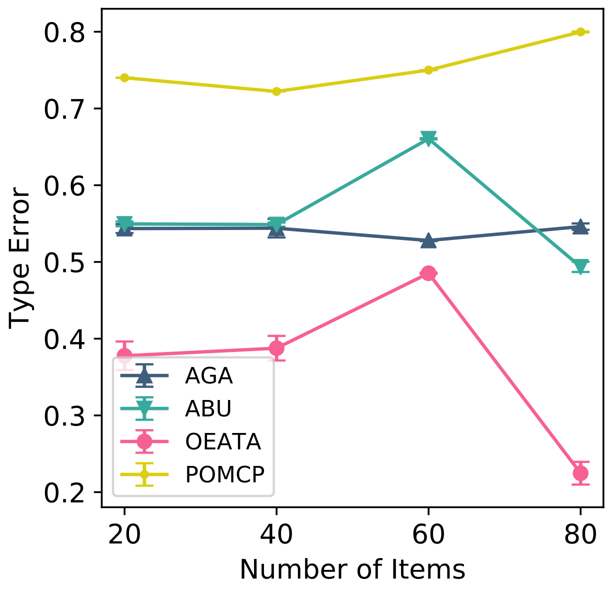


Figure 6.11: Type estimation errors for a varying number of items with full observability.

It is interesting to note that type estimation error with OEATA significantly drops for a very large number of items (80), as OEATA gets a larger number of observations. We can also note that the algorithm scales well to the number of the items and my performance (Figure 6.12) actually *significantly improves* with more than 20 items. It happens because OEATA gets observations more frequently for a larger number of items.

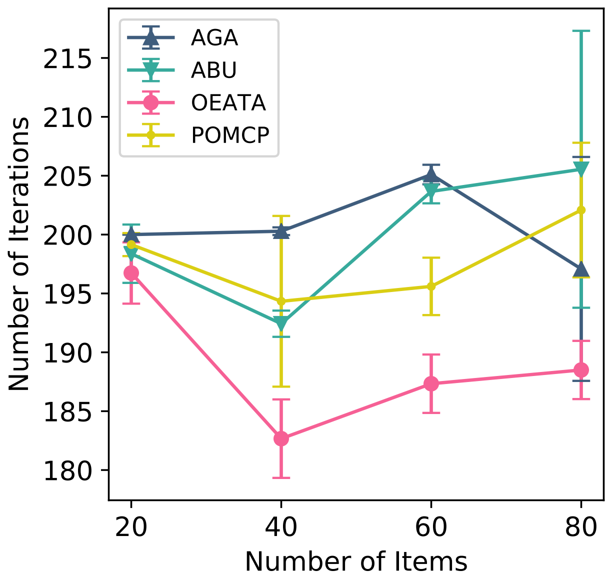


Figure 6.12: Performance for a varying number of items with full observability.

6.6.2 Multiple numbers of agents

After comparing with multiple numbers of items, I ran experiments for different numbers of agents. Here, I fixed the number of items to 40 and the scenario size to 20×20. Then, I ran experiments for a different number of agents (3, 5, 7, 10) and the plots are presented in Figures 6.13, 6.14 and 6.15. The figures tell us that again in various numbers of agents, OEATA has consistently lower error than the other algorithms, in all plots. As it is clear in Figure 6.15, the performance of the team by using OEATA is also significantly better than others. Regarding parameters and type estimation errors (Figure 6.13 and Figure 6.14), OEATA is significantly better than AGA, ABU and POMCP in almost all cases, except for parameters error with 10 agents, where $\rho \leq 0.47$, $\rho \leq 0.3$, and $\rho \leq 0.05$, against ABU, AGA, and POMCP, respectively. The reason for increasing the OEATA's parameter estimation error for a large number of agents is that, there is a lower number of observations for each agent, as the number of items is now fixed.

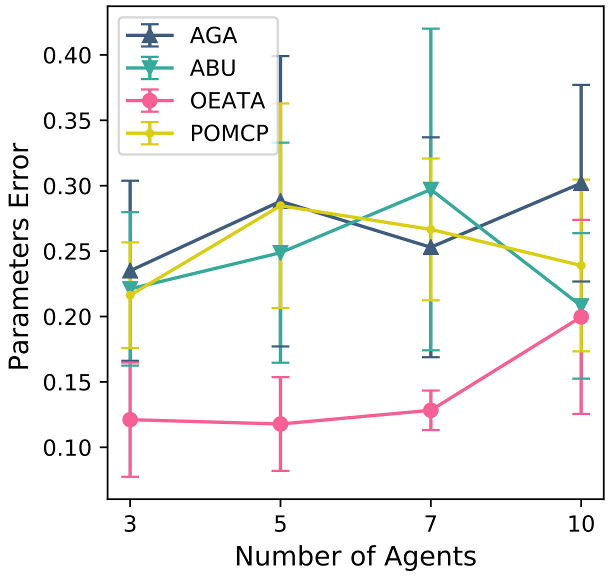


Figure 6.13: Parameter estimation errors for a varying number of agents with full observability.

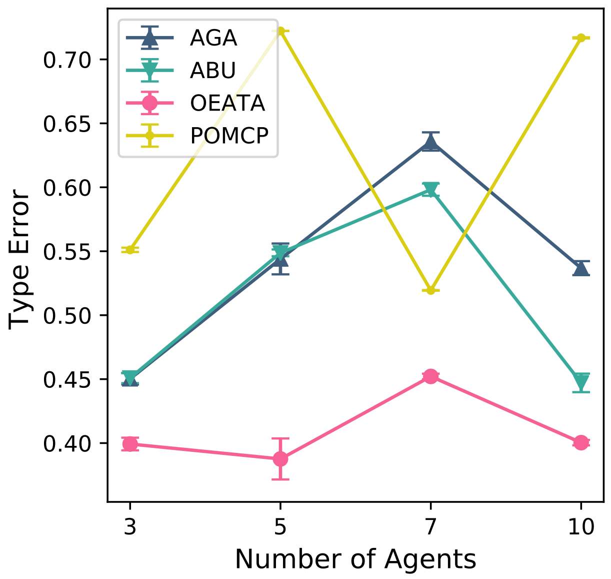


Figure 6.14: Type estimation errors for a varying number of agents with full observability.

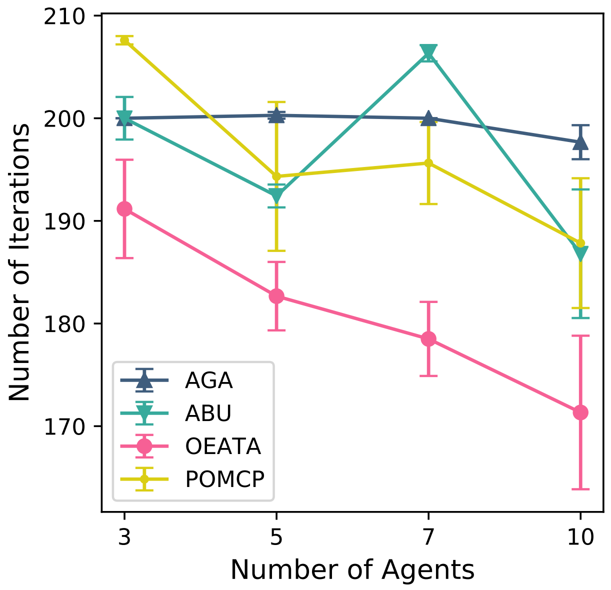


Figure 6.15: Performance for a varying number of agents with full observability.

6.6.3 Multiple scenario sizes

Following the comparison of multiple numbers of items and agents, I ran experiments for diverse scenario sizes to study the scalability of OEATA to harder problems. For that, I adjusted the number of items to 40 and the number of ω agents to 5 ($|\mathbf{\Omega}| = 5$). Afterwards, I ran experiments for a differing scenario size (30×30, 35×35, 40×40, 45×45) and the plots are displayed in Figures 6.16, 6.17 and 6.18.

As we can see, OEATA has consistently lower error than the other algorithms, both in terms of parameters and type estimation. In fact, OEATA significantly surpasses AGA, ABU and POMCP in respect of type estimation error for all scenario sizes. Regarding parameter estimation error (Figure 6.16), OEATA significantly exceeds the other algorithms, except for scenario sizes 30×30 and 40×40 against ABU, where $\rho \leq 0.24$ and $\rho \leq 0.17$, respectively. Additionally, in Figure 6.18, OEATA is able to accomplish all tasks faster for all team sizes and significantly surpasses in all cases.

It is interesting to note that the type estimation (Figure 6.17) error for OEATA significantly decreases with scenario size. It may happen because the difference between $L1$ and $L2$ decision-making might be more evident in larger scenarios. Interestingly, performance of OEATA actually significantly *improves* in larger scenarios ($\rho \leq 0.02$, when comparing 45×45 against 30×30), which may be caused by the better type estimations.

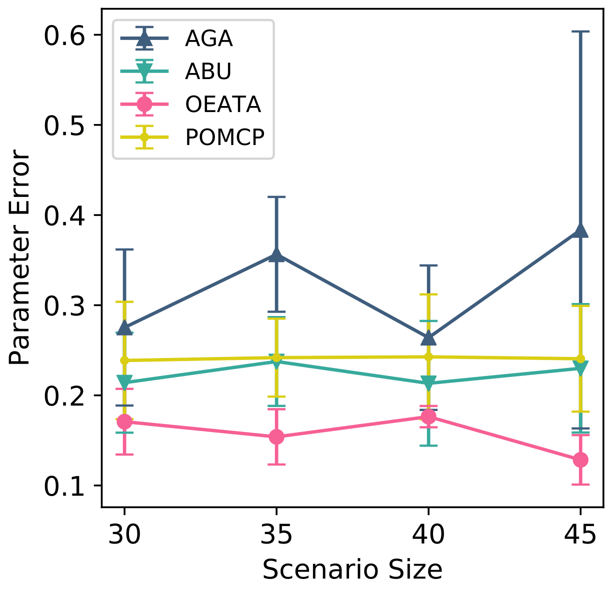


Figure 6.16: Parameter estimation errors for various environment sizes with full observability.

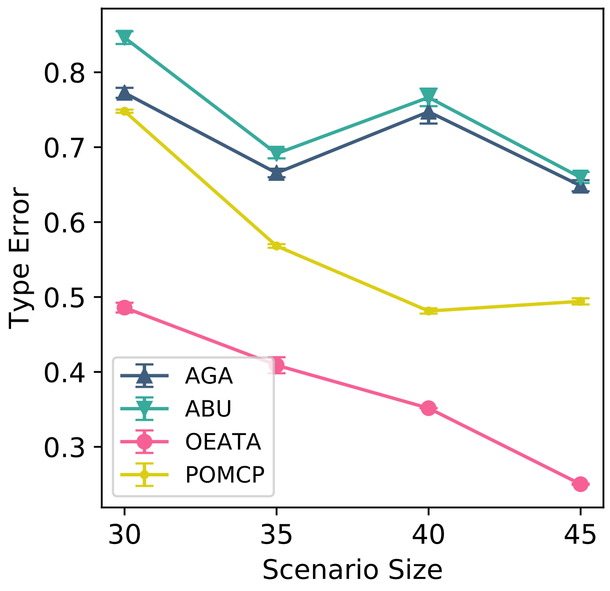


Figure 6.17: Type estimation errors for various environment sizes with full observability.

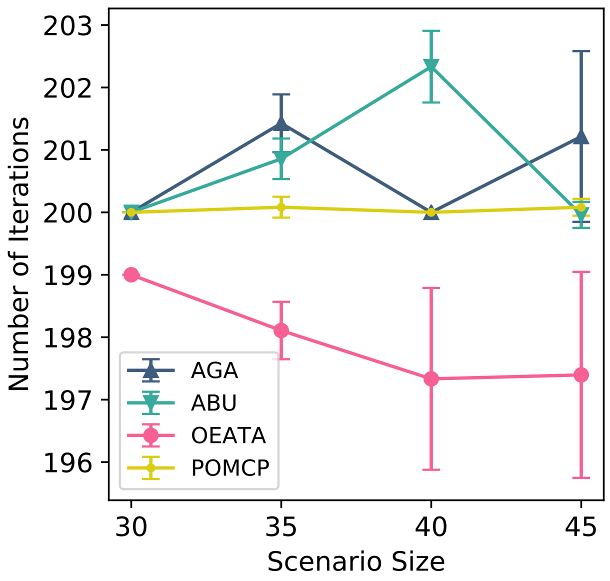


Figure 6.18: Performance for various environment sizes with full observability.

6.6.4 Multiple number of items for partial observability

I demonstrate the results for partially observable scenarios in Figures 6.19, 6.20 and 6.21. Here, the agent ϕ has partial observability of the environment and employs the POMCP modification for handling that, as described in Section 6.5.2. In these experiments, the number of ω agents is 5 and the environment size is 20×20, but the variation of items is 20, 40, 60, 80. The radius of the agent ϕ's view cone is 7 and the view angle is 360°.

Note that AGA/ABU results for partial observability are not shown in Albrecht and Stone (2017) [2], and thus are presented in my research for the first time. Hence, I applied the *modified POMCP* version, following the approach described in Section 6.5.2 to solve the POMDP model of the ϕ agent when it has a partial observation. However, by POMCP as an estimation method, I mean the *POMCP-based estimation* (Section 4.5.3), as before, which does not embed the ad-hoc teamwork algorithms for type and parameter estimation.

Again, I obtain significantly lower type estimation error than previous approaches (Figure 6.20). In the case of parameter estimation error (Figure 6.19), OEATA is significantly better, except only for 60 and 80 items. For 60 items, OEATA exceeds POMCP, but with $\rho \leq 0.38$; and the differences with other methods are significantly better with $\rho \leq 0.02$. In the case of 80 items, OEATA surpasses AGA with $\rho \leq 0.029$, and for all other cases, OEATA is significantly better.

Similarly, in Figure 6.21, as seen, OEATA obtain a significantly higher performance than previous approaches in 40 and 60 items. For 80 items, OEATA is still significantly better than the other methods, but against AGA and ABU, OEATA has $\rho \leq 0.03$. Unlike the large numbers of items, for 20 items, all methods get almost the same results.

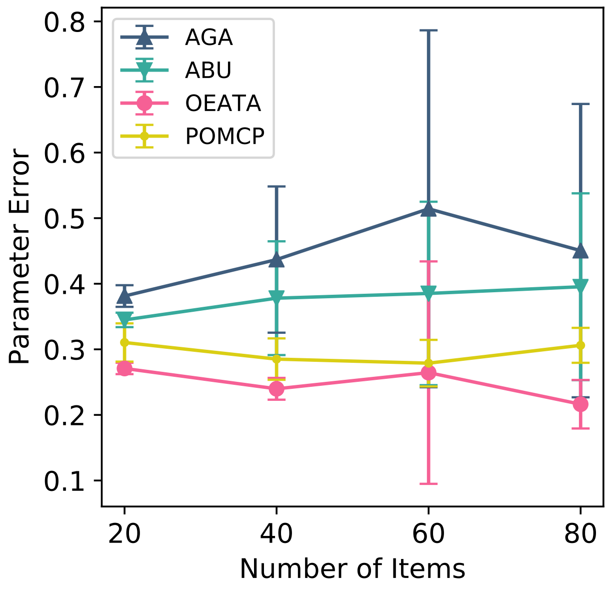


Figure 6.19: Parameter estimation errors for a varying number of items with partial observability.

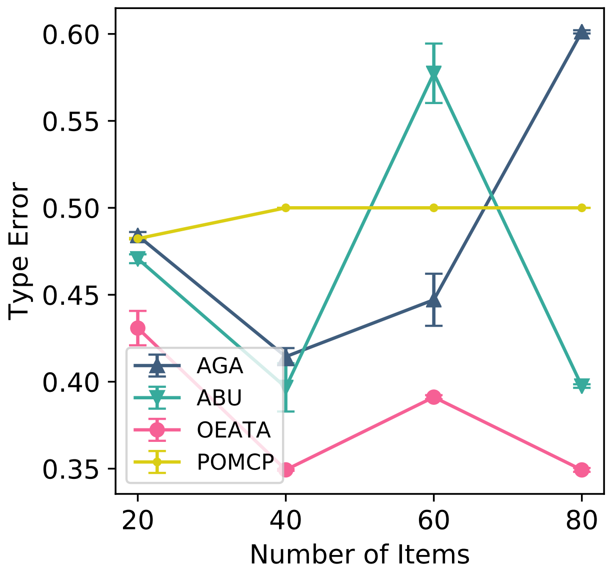


Figure 6.20: Type estimation errors for a varying number of items with partial observability.

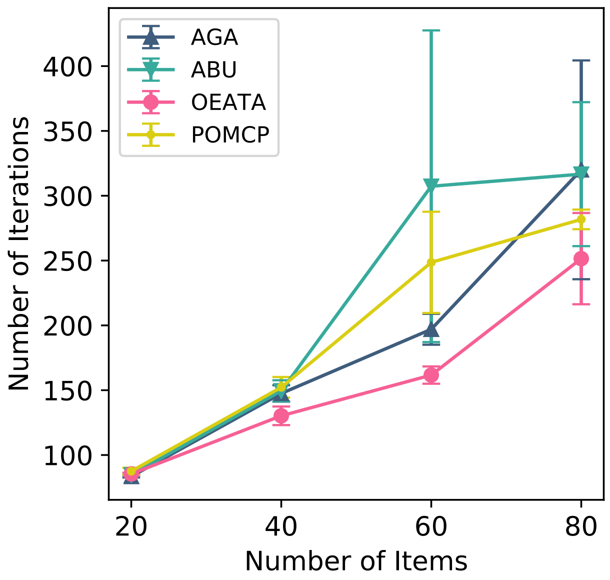


Figure 6.21: Performance for a varying number of items with partial observability.

6.6.5 Experiments with larger numbers of types

Besides trying two types ($L1$ and $L2$), I also run experiments for a larger number of potential types ($|\boldsymbol{\Theta}|$). First, I tried with four types ($L1, L2, L3, L4$). Results, displayed in Figure 6.22, demonstrates parameters error, where OEATA exceeds all other methods for all number of items with $\rho \leq 0.09$. It is clear from the results in Figure 6.23 that OEATA is always significantly better in estimating the type of the team-mates for all number of items. In the case of performance, as demonstrated in Figure 6.24, OEATA is significantly better against all algorithms for all number of items, but for 60 items, OEATA surpasses with $\rho \leq 0.06$.

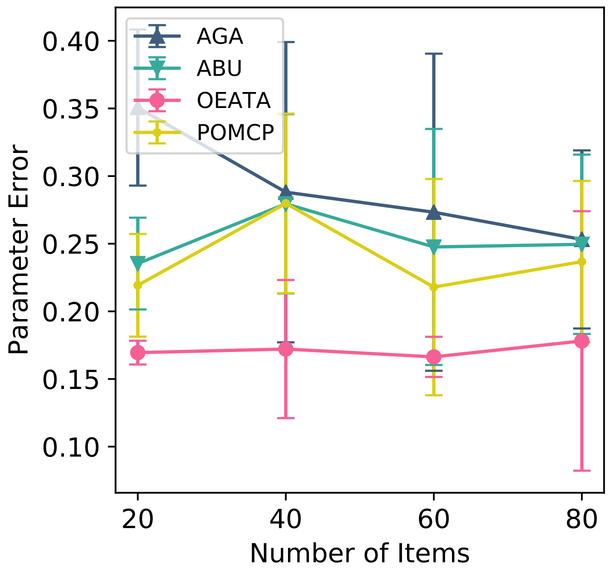


Figure 6.22: Parameter estimation errors for a varying number of items, with agents types selected randomly among 4 types.

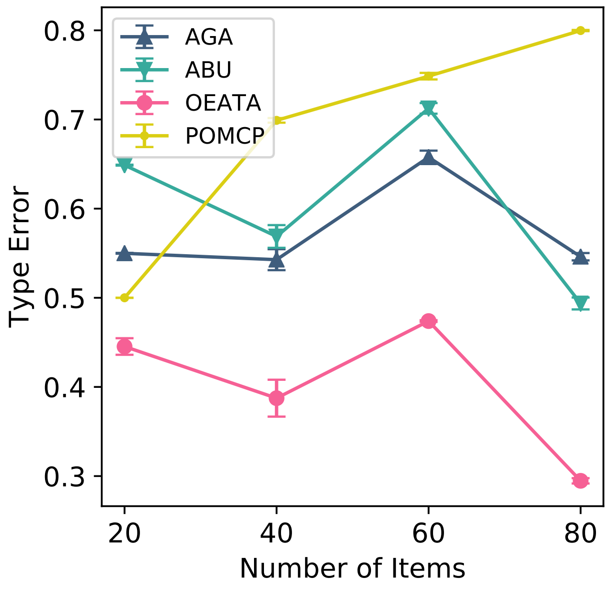


Figure 6.23: Type estimation errors for a varying number of items, with agents types selected randomly among 4 types.

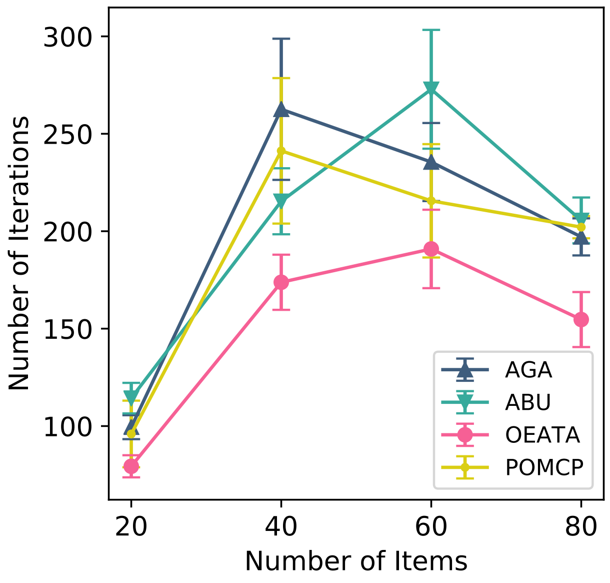


Figure 6.24: Performance for a varying number of items, with agents types selected randomly among 4 types.

After studying four different types for the ω agents, I experimented with six potential types $(L1, L2, L3, L4, L5, L6)$. The results are shown in Figures 6.25, 6.26 and 6.27. Considering parameters error, OEATA is better than the other approaches with $\rho \leq 0.07$. Taking type estimation error into account, OEATA is significantly better in all numbers of items, except for 40 items, where OEATA is significantly better than ABU and POMCP, but against AGA, OEATA exceeds with $\rho \leq 0.08$. For performance, with 20 items OEATA is better than POMCP with $\rho \leq 0.02$ and significantly outperforms other algorithms. For other item numbers, the performance of OEATA seems better than all other algorithms, but only with $\rho \leq 0.5$.

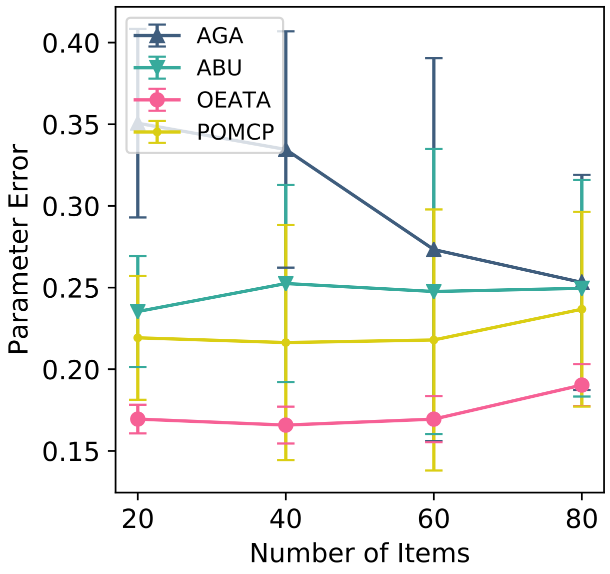


Figure 6.25: Parameter estimation errors for a varying number of items, with agents types selected randomly among 6 types.

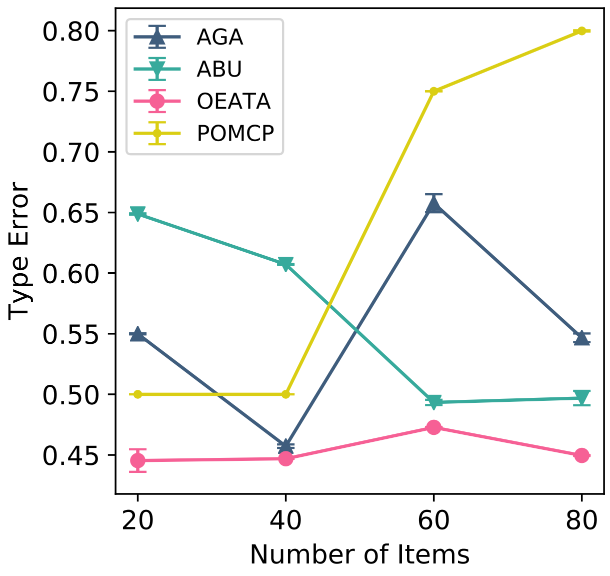


Figure 6.26: Type estimation errors for a varying number of items, with agents types selected randomly among 6 types.

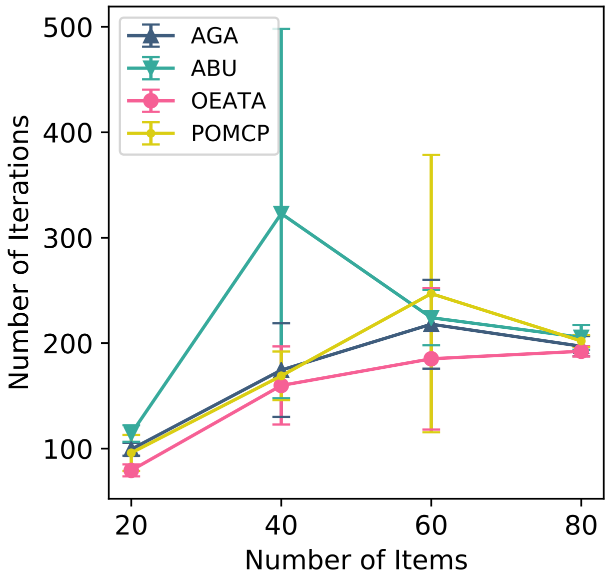


Figure 6.27: Performance for a varying number of items, with agents types selected randomly among 6 types.

Consequently, in Figures 6.28,6.29 and 6.30, I display a summary of the results for the different methods across different numbers of types θ that ω agents might have in the team. Each bar shows the average across all numbers of items for a specific method with each number of types (2, 4, 6). As demonstrated in Figure 6.30, regarding the performance of the team, as the number of types increases, it gets worse, but not significantly: 2 types are better than 4 types with $\rho \leq 0.3$, and better than 6 types with $\rho \leq 0.09$. However, for other methods, there is a significant increase in the number of iterations as the number of types grows. Likewise, in Figure 6.28, 6.29, concerning parameters and type estimation, results of the OEATA are not significantly different in all numbers of types, but other methods get significantly worse as the number of types raises (in particular from 2 to 4 types, although sometimes there is a decrease from 4 to 6). I suppose that the decrease from 4 to 6 might be caused by types $L5$ and $L6$ being easier to be learned correctly by these methods than the other types.

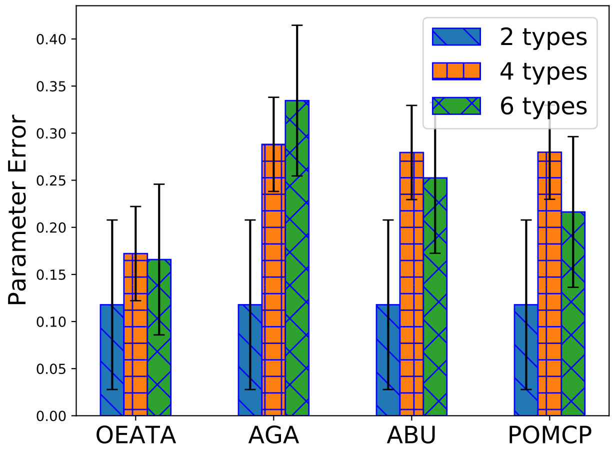


Figure 6.28: Parameter estimation errors for different number of types.

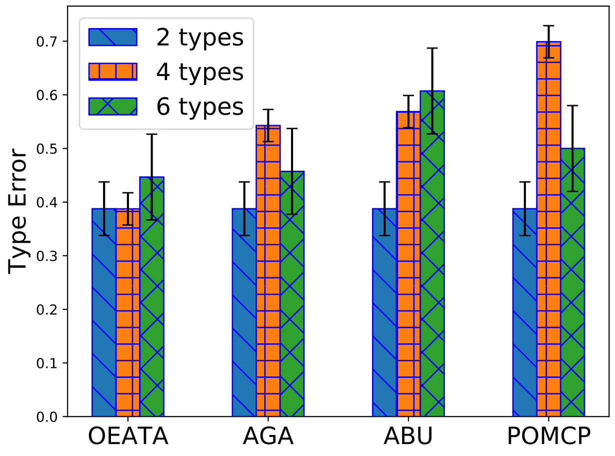


Figure 6.29: Type estimation error for different number of types.

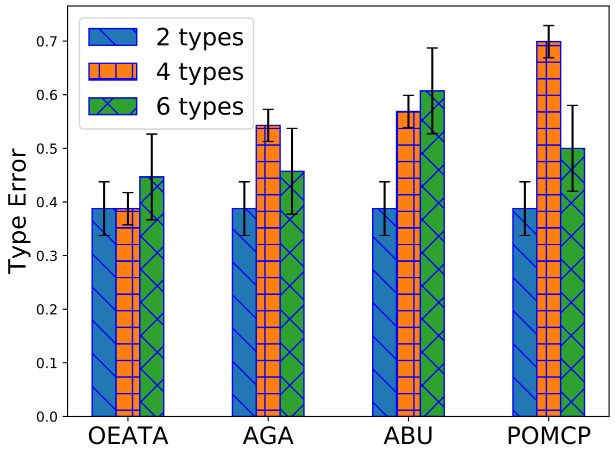


Figure 6.30: Performance for different number of types.

6.6.6 Wrong types

I also study my method's behaviour when the agent ϕ *does not* have full knowledge of the possible types of its team-mates. That is, I run experiments where all agents in $\mathbf{\Omega}$ have a type which *is not* in $\mathbf{\Theta}$. In these experiments, I assume that the agent ϕ is only aware of type $L1$ and $L2$, but I assign $L3$ and $L4$ to the ω agents as their type (sampled uniformly randomly). I ran experiments with 5 agents and fixed the size of the scenario to 20×20, with various numbers of items (20, 40, 60, 80). Figure 6.31 demonstrates the performance of the team. As the figure illustrates, even without knowing the possible types that the team-mates might have, OEATA is outperforming other methods in all numbers of items. For 80 and 60 items, OEATA is significantly better than other approaches, with the only exception for 80 items against AGA, where OEATA is better with $\rho \leq 0.1$; and for 60 items against ABU, where OEATA is better with $\rho \leq 0.08$. Additionally, for 40 items, OEATA is significantly better than the other methods, but against POMCP, OEATA has $\rho \leq 0.3$. In the case of 20 items, OEATA is only significantly better than AGA, as compared against the other methods, OEATA has $\rho \leq 0.4$.

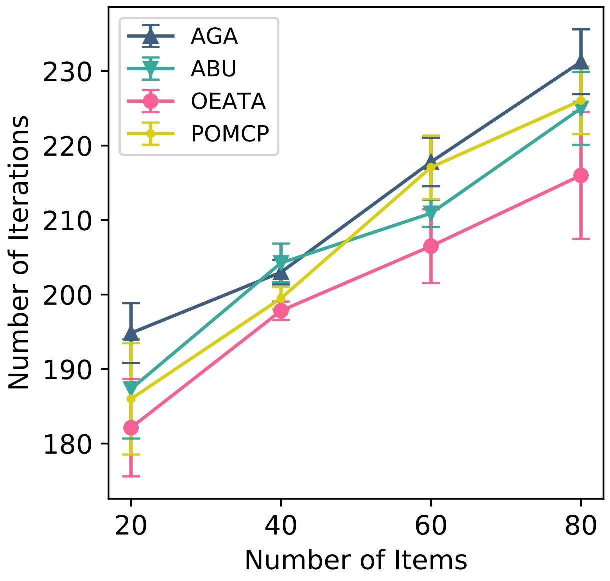


Figure 6.31: Performance of the ad-hoc team for a varying number of items without having information of correct potential team-mates types.

6.6.7 Comparing Mode, Median and Mean in OEATA

Finally, as I mentioned in Section 6.2, in *Estimation* step of OEATA, to have one estimated parameter vector $\mathbf{p}$ per $\theta \in \mathbf{\Theta}$ for each $\omega \in \mathbf{\Omega}$, I can use different aggregation rules like median, mode, or mean across all parameter vectors $\mathbf{p}_e$ of each set of *estimators* $\mathbf{E}_{\omega}^{\theta}$. As a result, I will have one estimated parameter vector $\mathbf{p}$ per $\theta \in \mathbf{\Theta}$ for each $\omega \in \mathbf{\Omega}$. To check the differences between aggregation methods, I did experiments in scenario size 20×20, applying 3, 5, and 7 agents ω to collect 20 items. As we notice in Figures 6.32, 6.33 and 6.34, *Mean* and *Median* are not significantly different, as $\rho \leq 0.65$. However, both *Mean* and *Median* are significantly better than *Mode* in almost all iterations.

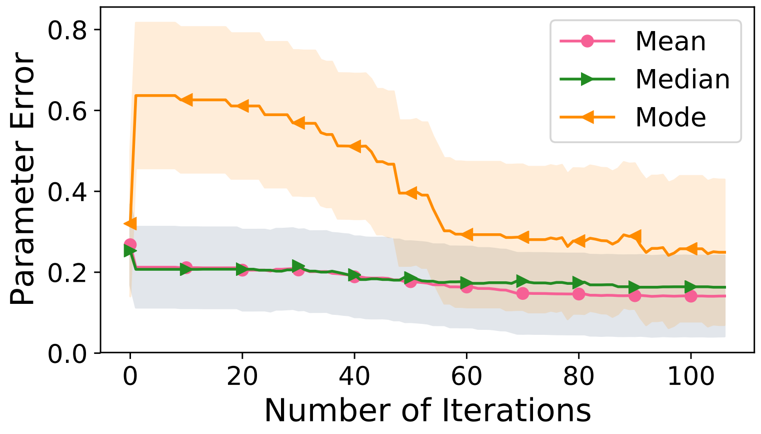


Figure 6.32: Parameter estimation errors across different aggregation rules for 3 teammates ($|\mathbf{\Omega}| = 3$).

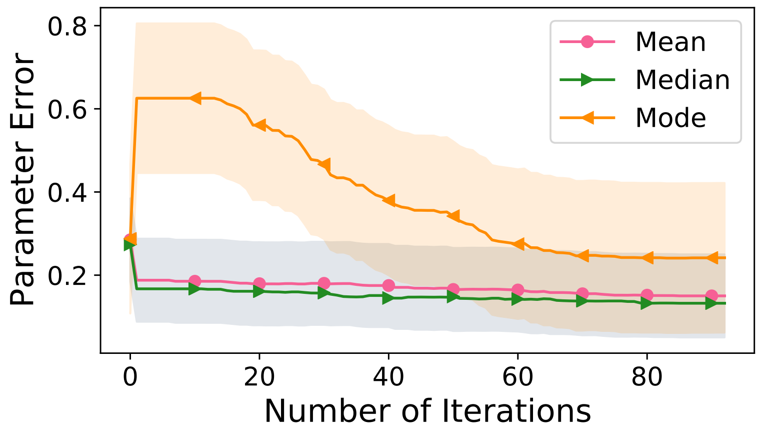


Figure 6.33: Parameter estimation errors across different aggregation rules for 5 team-mates ($|\mathbf{\Omega}| = 5$).

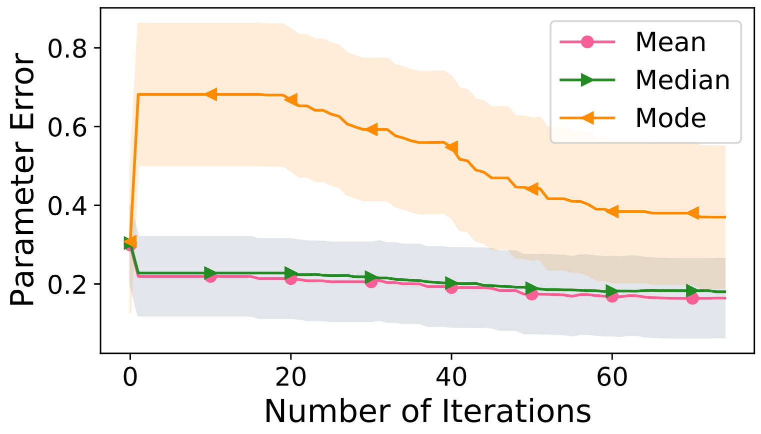


Figure 6.34: Parameter estimation errors across different aggregation rules for 7 team-mates ($|\mathbf{\Omega}| = 7$).

6.7 Conclusion

This chapter describes a novel algorithm, *On-line Estimators for Ad-hoc Task Allocation* (OEATA), which is designed to learn about the team-mate's future behaviour in a type-based ad-hoc team setting. OEATA is an algorithm for estimating types and parameters of team-mates in problems where there is a set of tasks to be completed in a scenario in a decentralised fashion. Experimental evaluation of my algorithm has been conducted in the level-based foraging domain. In my analysis, there were many different scenarios, a growing amount of items, a growing number of agents, and different-sized scenarios with various types.

Furthermore, my study evaluated how OEATA can improve the team's performance even if the correct type is not in the set of potential types of team-mates. Moreover, I investigated the impact of dealing with learning agents with partial observability. By using OEATA in the estimation of parameters and types, I demonstrate with statistical significance that it outperforms previous methods in nearly all cases.

On the other hand, having a large number of types will make it difficult since the number of estimators will increase, and in turn, evaluating them will take much more time. Furthermore, we know that the evaluation of estimators begins when a task is completed. However, there may be situations where we are unsure whether or not a task has been completed. Due to this, it becomes more difficult for the learning agent to determine when to start the process. Nevertheless, applying OEATA help us to improve the overall performance of the team comparing to the state-of-the-art.

Chapter 7

Conclusion and Discussion

7.1 Discussion

I showed in this work that by focusing on distributed task allocation problems, where agents can autonomously decide which task to perform, I can obtain better type and parameter estimations in ad-hoc teamwork than previous works in the literature, which leads to a better performance of the team. Although not all problems can be modelled as a set of tasks to be completed, it does encompass a great range of challenges. For instance, apart from the obvious warehouse management, we could think about situations such as rescuing victims after a natural disaster or even during some hazard and demining.

Although I employed both of my contributions to solve a task-based problem, UCT-H can be applied to any challenge that could be modelled as an MDP where an action can lead to a large number of potential next states.

Regarding learning other team members, note that different team-mates do not need to share the same representation of the problem, and run algorithms that explicitly "choose" tasks. That is, they could have been programmed with different paradigms, without using any explicit task representation. However, their external behaviour would still need to be understood as solving tasks distributed in an environment from the point of view of *our* ad-hoc agent. Hence, we need problems and team-mates that fit the decentralised task allocation representation for the learning agent, but the actual team-mates' internal models could be different.

Another interesting characteristic of my learning algorithm is that it allows learning from scratch at every run in an on-line manner, following the inspiration from Albrecht and Stone (2017) [2]. Therefore, I can quickly adapt to different teams and different situations, without requiring a significant pre-training. Neural network-based models, on the other hand, would require thousands (even millions) of observations, and although they may show some generalisability, eventually re-training may be required as the test situation becomes significantly different than the training cases.

On the other hand, it is true that my algorithm requires a set of potential types to be given. In the case where this set cannot be created from domain knowledge, then some training may be required to initialise this set. Afterwards, however, I would be able to learn on-line at every run, without carrying further knowledge between executions. Albrecht and Stone (2017) [2] also follow the same paradigm, and directly assumes a set of potential parametrisable types, without showing exactly how they could be learned. There are several examples of learning types in ad-hoc teamwork, but they still ignore the possibility of parametrisation. For instance, PLASTIC-Model [19] employs a supervised learning approach, and learns a probability distribution over actions given a state representation using C4.5 decision trees.

In order to better understand the impact of this assumption, I also run experiments where the set of types considered by the ad-hoc agent *does not* include the real types of team-mates. In these challenging situations, I find that my performance is either similar to the other works in the literature, or significantly better, depending on each case.

I have also shown that my algorithm scales well to a range of different variables, as I increase the number of items, number of agents, scenario sizes, and number of types. Usually, models based on neural networks (e.g., [46], [73]) are not yet able to show such scalability and present only restricted cases. A similar issue happens with I-POMDP based models (e.g., [29], [35], [42], [48]) which tend to show experiments in simplified scenarios due to the computational constraints. Therefore, by focusing on distributed task allocation scenarios, I am able to propose a light-weight algorithm, which could be more easily applied across a range of different situations.

Concerning partial observability scenarios, my algorithm still expects knowledge of which agents completed a particular task, even if outside my controlled agent visibility region. Hence, in a real application, I would still require some hardware in addition to the agent sensors, such as radio transmitters connected to the boxes that must be collected. Removing this assumption in *task-based ad-hoc teamwork* under partial observability is one of the exciting potential avenues for future work.

Considering the scenario to solve the problem of robots in an emergency situation, I could give a first step towards solving this problem assuming each robot as an agent. Different types of agents can be assumed various responsibilities and missions to save lives. I still have to apply my novel methods to real robots and real scenarios to determine how well they work. There may be some adjustments to the methods based on some issues that could arise in real-world scenarios. Additionally, we may need to have continuous action space instead of the discrete one that we applied in this research.

7.2 Conclusion

In this research, I worked on the online planning and learning of type-based ad-hoc teams. My main focus was to create novel methods to improve the performance of the ad-hoc team. Therefore, I have presented two novel techniques, *On-line Estimators for Ad-hoc Task Allocation* (OEATA) for learning the team-mates future behaviours in a type-based ad-hoc team; and *UCT-H* for better online planning in larger ad-hoc teams.

OEATA is a new algorithm for estimating types and parameters of team-mates, and it is specifically designed for problems where there is a set of tasks to be completed in a scenario. By focusing on decentralised task allocation, I can obtain a lower error in parameter and type estimation than previous works, leading to better overall performance. The alternatives for my novel method are AGA, ABU and POMCP-based estimation. These approaches do not consider their history of successes in their future calculation. However, OEATA keeps successful parameters that have better estimations during the whole scenario and rate them.

I also study my algorithm theoretically, showing that it converges to zero error as the number of tasks increases (under some assumptions), and I experimentally verify that the error does reduce with the number of iterations. My theoretical analysis also shows the importance of having parameter *bags* in my method, as it significantly decreases the computational complexity.

I experimentally evaluated my algorithm in the level-based foraging domain. I considered a range of situations, an increasing number of items, number of agents, scenario sizes, and number of types. Additionally, I evaluated the impact of having an erroneous set of potential types, and the impact of handling situations with partial observability of the scenarios. I show that in almost all cases, I outperform previous works with statistical significance. Furthermore, I find that my method scales better to an increasing number of types, and can show robustness to wrong type models, as I still overcome previous works in these challenging cases.

Concerning UCT-H, My approach introduces a more compact representation, by representing each node as a *history* instead of a *state*. As well as *OEATA*, I perform several experiments in the level-based foraging domain for evaluating *UCT-H*. I show that my approach achieves a better performance than the current state-of-the-art, using roughly the same amount of computational time, and the difference tends to increase as the number of agents grows.

Furthermore, I evaluate the memory usage of UCT-H and the state-of-the-art. I found that my approach tends to use a roughly constant amount of memory, while the memory usage of the state-of-the-art grows exponentially with the number of agents. Hence, my approach has better scalability, and better handles larger team sizes.

As an additional contribution, I provide a fully open-source version of my system to the community, making it easier for other researchers to use level-based foraging as an important benchmark problem for ad-hoc teamwork. For the interested readers that may want to explore and further extend this work, *UCT-H* and *OEATA*, my source code for both algorithms is available at `https://github.com/ElnazShy/MultiAgents/releases/tag/oeata`.

7.3 Future Works

I present a task-base ad-hoc team in this work with one learning agent collaborating with multiple non-learning agents without any prior coordination. I introduced two different contributions, *UCT-H* and *OEATA*, which assist in the improvement of the team. I did the evaluations in a level-based foraging domain. I believe these contributions can be extended to be applicable in different domains. In this section, I will discuss some possible future works to advance my current research.

Solving real-world problems I believe there are plenty of real-world problems that can be solved by applying my works on multi-agent systems. My contributions can be employed in various real problems like *Search and Rescue Teams*. We can consider that there are different types of robots, including UAVs [103], vehicles, etc without any prior knowledge about each other trying to save a victim. Robots here have various sensors to observe the environment, and there are more uncertainties which cause more complications in the problem.

Existing of an adversary agent in the team In the team of agents, most of the time, there is a possibility of existing an adversary in the team, which attempts to avoid other agents to reach their goals. Hence, the learning agent must also estimate who might be an adversary in an on-line manner, and plan its actions accordingly. Therefore, *adversary* would need to be added as one of the possible types of team-mates. As the algorithm that the adversary agent would follow is not a static one (like what I have in this work for the non-learning agents), it would be quite challenging to recognise the adversary type while it has a quite different nature than others.

Having multiple learning agents in the team Another extension would be having more than one learning agent in the team, which are all trying to reason about other team-mates to improve the team performance. The learning agents that I have encountered can either communicate or not. Whenever there is no communication, each learning agent will have its own set of estimators and independently attempt to learn the team-mate's type and parameters. Therefore, in accordance with each agent's understanding of its team-mates, each of them takes the best steps for improving team performance.

Nevertheless, having the opportunity to communicate will allow them to share their estimators. In this way, the learning agents can share their experiences and gain a better understanding of the team. Additionally, in the case of communication, we can apply Multi-agent Reinforcement Learning [70] techniques to solve decision-making the problem. The goal of MARL is to enable a team of agents to collaboratively determine the global optimal policy that maximises the sum of their local accumulated rewards. As agents communicate with one another, they obtain information about the global state and action of the team. This is because their states and rewards are generally affected by the actions of their peers.

References

[1] S. Albrecht, J. Crandall, and S. Ramamoorthy. "An empirical study on the practical impact of prior beliefs over policy types". In: *Proceedings of the 29th AAAI Conference on Artificial Intelligence*. 2015.

[2] S. Albrecht and P. Stone. "Reasoning about Hypothetical Agent Behaviours and their Parameters". In: *Proceedings of the 16th International Conference on Autonomous Agents and Multiagent Systems*. AAMAS'17. 2017.

[3] S. V. Albrecht and S. Ramamoorthy. "Exploiting causality for selective belief filtering in dynamic Bayesian networks". In: *Journal of Artificial Intelligence Research* 55 (2016).

[4] Stefano V. Albrecht, Jacob W. Crandall, and Subramanian Ramamoorthy. "Belief and truth in hypothesised behaviours". In: *Artificial Intelligence* 235 (2016), pp. 63–94.

[5] Stefano V. Albrecht, Somchaya Liemhetcharat, and Peter Stone. "Special issue on multiagent interaction without prior coordination: Guest editorial". In: *Autonomous Agents and Multi-Agent Systems* 31.4 (2017), pp. 765–766.

[6] Stefano V. Albrecht and Subramanian Ramamoorthy. *A Game-Theoretic Model and Best-Response Learning Method for Ad Hoc Coordination in Multiagent Systems*. Tech. rep. The University of Edinburgh, Feb. 2013.

[7] Stefano V. Albrecht and Subramanian Ramamoorthy. "On convergence and optimality of best-response learning with policy types in multiagent systems".

In: *Proceedings of the 30th Conference on Uncertainty in Artificial Intelligence (UAI)* (2014).

[8] Stefano V. Albrecht and Subramanian Ramamoorthy. “A game-theoretic model and best-response learning method for ad hoc coordination in multiagent systems”. In: *arXiv preprint arXiv:1506.01170* (2015).

[9] Stefano V. Albrecht and Peter Stone. “Autonomous agents modelling other agents: A comprehensive survey and open problems”. In: *Artificial Intelligence* 258 (2018), pp. 66–95.

[10] Panella Alessandro and Gmytrasiewicz Piotr. “Interactive POMDPs with finite-state models of other agents”. In: *Autonomous Agents and Multi-Agent Systems* 31.4 (July 2017), pp. 861–904.

[11] Plamen P. Angelov and Xiaowei Gu. *Empirical approach to machine learning.* Springer, 2019.

[12] Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvári. “Exploration–exploitation tradeoff using variance estimates in multi-armed bandits”. In: *Theoretical Computer Science* 410.19 (2009), pp. 1876–1902.

[13] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. “Finite-time analysis of the multiarmed bandit problem”. In: *Machine learning* 47.2-3 (2002), pp. 235–256.

[14] Tim Baarslag, Mark J. C. Hendrikx, Koen V. Hindriks, and Catholijn M. Jonker. “Learning about the opponent in automated bilateral negotiation: a comprehensive survey of opponent modeling techniques”. In: *Autonomous Agents and Multi-Agent Systems* 30.5 (2016), pp. 849–898.

[15] Sander C. J. Bakkes, Pieter H. M. Spronck, and Giel van Lankveld. “Player behavioural modelling for video games”. In: *Entertainment Computing* 3.3 (2012), pp. 71–79.

[16] Horace B. Barlow. “Unsupervised learning”. In: *Neural computation* 1.3 (1989), pp. 295–311.

[17] S. Barrett and P. Stone. "Cooperating with unknown teammates in complex domains: A robot soccer case study of ad hoc teamwork". In: *Proceedings of the 29th AAAI Conference on Artificial Intelligence*. 2015.

[18] S. Barrett, P. Stone, S. Kraus, and A. Rosenfeld. "Teamwork with limited knowledge of teammates". In: *Proceedings of the 27th AAAI Conference on Artificial Intelligence*. 2013.

[19] Samuel Barrett, Avi Rosenfeld, Sarit Kraus, and Peter Stone. "Making friends on the fly: Cooperating with new teammates". In: *Artificial Intelligence* 242 (2017), pp. 132–171.

[20] Samuel Barrett and Peter Stone. "Cooperating with Unknown Teammates in Complex Domains: A Robot Soccer Case Study of Ad Hoc Teamwork." In: *AAAI*. Vol. 15. Citeseer. 2015, pp. 2010–2016.

[21] Samuel Barrett, Peter Stone, and Sarit Kraus. "Empirical Evaluation of Ad Hoc Teamwork in the Pursuit Domain". In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*. 2011.

[22] Samuel Barrett, Peter Stone, and Sarit Kraus. "Empirical evaluation of ad hoc teamwork in the pursuit domain." In: *AAMAS*. 2011, pp. 567–574.

[23] Spring Berman, Ádám Halász, M. Ani Hsieh, and Vijay Kumar. "Optimized stochastic policies for task allocation in swarms of robots". In: *IEEE transactions on robotics* 25.4 (2009), pp. 927–937.

[24] Dimitri P. Bertsekas. "Auction algorithms for network flow problems: A tutorial introduction". In: *Computational optimization and applications* 1.1 (1992), pp. 7–66.

[25] Michael Bowling. "Convergence and no-regret in multiagent learning". In: *Advances in neural information processing systems* 17 (2004), pp. 209–216.

[26] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, et al. “A survey of monte carlo tree search methods”. In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[27] David Carmel and Shaul Markovitch. “Model-based learning of interaction strategies in multi-agent systems”. In: *Journal of Experimental & Theoretical Artificial Intelligence* 10.3 (1998), pp. 309–332.

[28] Rich Caruana and Alexandru Niculescu-Mizil. “An empirical comparison of supervised learning algorithms”. In: *Proceedings of the 23rd international conference on Machine learning.* 2006, pp. 161–168.

[29] Muthukumaran Chandrasekaran, Prashant Doshi, Yifeng Zeng, and Yingke Chen. “Team behavior in interactive dynamic influence diagrams with applications to ad hoc teams”. In: *MSDM Workshop at AAMAS 2014* (2014).

[30] Shuo Chen, Ewa Andrejczuk, Athirai A. Irissappane, and Jie Zhang. “ATSIS: Achieving the Ad hoc Teamwork by Sub-task Inference and Selection”. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19.* International Joint Conferences on Artificial Intelligence Organization, 2019, pp. 172–179.

[31] Philip R. Cohen and Hector J. Levesque. “Confirmations and Joint Action.” In: *International Joint Conference on Artificial Intelligence.* 1991, pp. 951–959.

[32] Rémi Coulom. “Efficient selectivity and backup operators in Monte-Carlo tree search”. In: *International conference on computers and games.* Springer. 2006, pp. 72–83.

[33] Steven X. Ding. *Model-based fault diagnosis techniques: design schemes, algorithms, and tools.* Springer Science & Business Media, 2008.

[34] Ali Dorri, Salil S. Kanhere, and Raja Jurdak. “Multi-agent systems: A survey”. In: *IEEE Access* 6 (2018), pp. 28573–28593.

[35] P. Doshi, Y. Zeng, and Q. Chen. “Graphical models for interactive POMDPs: Representations and solutions”. In: *Autonomous Agents and Multi-Agent Systems* 18.3 (2009), pp. 376–416.

[36] Prashant Doshi, Yifeng Zeng, and Yingke Chen. “Team Behavior in Interactive Dynamic Influence Diagrams with Applications to Ad Hoc Teams”. In: *MSDM Workshop at Autonomous Agents and Multi-Agent Systems* (2014).

[37] Adam Eck, Maulik Shah, Prashant Doshi, and Leen-Kiat Soh. “Scalable Decision-Theoretic Planning in Open and Typed Multiagent Systems”. In: *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence.* AAAI. 2019.

[38] S. Shaheen Fatima and Michael Wooldridge. “Adaptive task resources allocation in multi-agent systems”. In: *Proceedings of the fifth international conference on Autonomous agents.* 2001, pp. 537–544.

[39] Aurélien Garivier and Eric Moulines. “On upper-confidence bound policies for switching bandit problems”. In: *International Conference on Algorithmic Learning Theory.* Springer. 2011, pp. 174–188.

[40] Zulkuf Genc, Farideh Heidari, Michel A. Oey, Sander van Splunter, and Frances M.T. Brazier. “Agent-based information infrastructure for disaster management”. In: *Intelligent systems for crisis management.* Springer, 2013, pp. 349–355.

[41] John Gittins, Kevin Glazebrook, and Richard Weber. *Multi-armed bandit allocation indices.* John Wiley & Sons, 2011.

[42] P. Gmytrasiewicz and P. Doshi. “A framework for sequential planning in multiagent settings”. In: *Journal of Artificial Intelligence Research* 24 (2005), pp. 49–79.

[43] Barbara Grosz and Sarit Kraus. “Collaborative plans for complex group action”. In: *Artificial Intelligence* (1996).

[44] A. Guez, D. Silver, and P. Dayan. "Scalable and Efficient Bayes-Adaptive Reinforcement Learning Based on Monte-Carlo Tree Search". In: *Journal of Artificial Intelligence Research (JAIR)* 48 (2013).

[45] P. E. Hart, N. J. Nilsson, and B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[46] Akinobu Hayashi, Dirk Ruiken, Tadaaki Hasegawa, and Christian Goerick. "Reasoning about uncertain parameters and agent behaviors through encoded experiences and belief planning". In: *Artificial Intelligence* 280 (2020).

[47] H. Jaap van den Herik, H. H. L. M. Donkers, and Pieter H. M. Spronck. "Opponent modelling and commercial games". In: *Proceedings of the IEEE* (2005), pp. 15–25.

[48] T. N. Hoang and K. H. Low. "Interactive POMDP Lite: Towards practical planning to predict and exploit intentions for interacting with self-interested agents". In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*. IJCAI. 2013.

[49] Benjamin Hockman and Marco Pavone. "Stochastic motion planning for hopping rovers on small solar system bodies". In: *Robotics Research*. Springer, 2020, pp. 877–893.

[50] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992. ISBN: 0262082136.

[51] Junyan Hu, Parijat Bhowmick, and Alexander Lanzon. "Distributed Adaptive Time-Varying Group Formation Tracking for Multiagent Systems With Multiple Leaders on Directed Graphs". In: *IEEE Transactions on Control of Network Systems* 7.1 (2019), pp. 140–150.

[52] Constantin Hubmann, Marvin Becker, Daniel Althoff, David Lenz, and Christoph Stiller. “Decision making for autonomous driving considering interaction and uncertain prediction of surrounding vehicles”. In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2017, pp. 1671–1678.

[53] Hiroyuki Iida, Yoshiyuki Kotani, and J Uiterwijk. “Tutoring strategies in game-tree search”. In: *Games of No Chance* 29 (1996), pp. 433–435.

[54] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. “Planning and acting in partially observable stochastic domains”. In: *Artificial intelligence* 101.1-2 (1998), pp. 99–134.

[55] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W Moore. “Reinforcement learning: A survey”. In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[56] L. Kocsis and C. Szepesvári. “Bandit based Monte-Carlo Planning”. In: *Proceedings of the 17th European Conference on Machine Learning*. 2006.

[57] N. K. Krothapalli and Abhijit V. Deshmukh. “Distributed task allocation in multi-agent systems”. In: *Proceedings of the Institute of Industrial Engineers Annual Conference*. 2002.

[58] Przemyslaw A. Lasota, Terrence Fong, Julie A. Shah, et al. *A survey of methods for safe human-robot interaction*. Now Publishers, 2017.

[59] Maja J. Matarić, Gaurav S. Sukhatme, and Esben H. Østergaard. “Multi-robot task allocation in uncertain environments”. In: *Autonomous Robots* 14.2-3 (2003), pp. 255–263.

[60] Francisco S. Melo and Alberto Sardinha. “Ad hoc teamwork by learning teammates’ task”. In: *Autonomous Agents and Multi-Agent Systems* 30.2 (2016).

[61] Donald Michie, David J. Spiegelhalter, C. C. Taylor, et al. “Machine learning”. In: *Neural and Statistical Classification* 13.1994 (1994), pp. 1–298.

[62] Andreza Bastos Mourão and José Francisco Magalhães Netto. "SIMROAA Multi-Agent Recommendation System for Recommending Accessible Learning Objects". In: *2019 IEEE Frontiers in Education Conference (FIE)*. IEEE. 2019, pp. 1–9.

[63] R. Nair and M. Tambe. "Hybrid BDI-POMDP framework for multiagent teaming". In: *Journal of Artificial Intelligence Research* 23 (2005), pp. 367–413.

[64] R. Nair, P. Varakantham, M. Yokoo, and M. Tambe. "Networked distributed POMDPs: A synergy of distributed constraint optimization and POMDPs". In: *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*. IJCAI. 2005.

[65] Muaz Niazi and Amir Hussain. "Agent-based computing from multi-agent systems to agent-based models: a visual survey". In: *Scientometrics* 89.2 (2011), pp. 479–499.

[66] Olayide Olorunleke and Gordon McCalla. "A condensed roadmap of agents-modelling-agents research". In: *IJCAI'05 Workshop on Modeling Other Agents From Observation*. 2005.

[67] Lukasz Pelcner, Shaling Li, Matheus Do Carmo Alves, Leandro Soriano Marcolino, and Alex Collins. "Real-time learning and planning in environments with swarms: a hierarchical and a parameter-based simulation approach". In: *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems*. AAMAS. 2020.

[68] Isaac Pinyol and Jordi Sabater-Mir. "Computational trust and reputation models for open multi-agent systems: a review". In: *Artificial Intelligence Review* 40.1 (2013), pp. 1–25.

[69] Shokoofeh Pourmehr and Chitra Dadkhah. "An overview on opponent modeling in RoboCup soccer simulation 2D". In: *Robot Soccer World Cup*. Springer. 2011, pp. 402–414.

[70] Yaohong Qu and Youmin Zhang. “Cooperative localization against GPS signal loss in multiple UAVs flight”. In: *Journal of Systems Engineering and Electronics* 22.1 (2011), pp. 103–112.

[71] Neil Rabinowitz, Frank Perbet, Francis Song, Chiyuan Zhang, S. M. Ali Eslami, and Matthew Botvinick. “Machine Theory of Mind”. In: *Proceedings of the 35th International Conference on Machine Learning*. 2018, pp. 4218–4227.

[72] Faezeh Rahimzadeh, Leyli Mohammad Khanli, and Farnaz Mahan. “High reliable and efficient task allocation in networked multi-agent systems”. In: *Autonomous Agents and Multi-Agent Systems* 29.6 (2015), pp. 1023–1040.

[73] Arrasy Rahman, Niklas Hopner, Filippos Christianos, and Stefano V. Albrecht. “Open Ad Hoc Teamwork using Graph-based Policy Learning”. In: *arXiv preprint arXiv:2006.10412* (2020).

[74] Sarvapali D. Ramchurn, Dong Huynh, and Nicholas R. Jennings. “Trust in multi-agent systems”. In: *The Knowledge Engineering Review* 19.1 (2004), pp. 1–25.

[75] Hamed Rezaee and Farzaneh Abdollahi. “Average consensus over high-order multiagent systems”. In: *IEEE Transactions on Automatic Control* 60.11 (2015), pp. 3047–3052.

[76] Patrick Riley and Manuela Veloso. “Recognizing Probabilistic Opponent Movement Models”. In: *Robot Soccer World Cup V*. Ed. by Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro. Springer Berlin Heidelberg, 2002, pp. 453–458.

[77] Herbert Robbins. “Some aspects of the sequential design of experiments”. In: *Bulletin of the American Mathematical Society* 58.5 (1952), pp. 527–535.

[78] Alex Rogers, Esther David, Nicholas R. Jennings, and Jeremy Schiff. “The effects of proxy bidding and minimum bid increments within eBay auctions”. In: *ACM Transactions on the Web (TWEB)* 1.2 (2007), 9–es.

[79] Jonathan Rubin and Ian Watson. “Computer poker: A review”. In: *Artificial intelligence* 175.5-6 (2011), pp. 958–987.

[80] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited, 2016.

[81] P. Scerri, D. Pynadath, and M. Tambe. “Towards adjustable autonomy for the real-world”. In: *Journal of Artificial Intelligence Research* 17 (2002), pp. 171–228.

[82] Paul Scerri, David Pynadath, and Milind Tambe. “Adjustable autonomy for the real world”. In: *Agent Autonomy.* Springer, 2003, pp. 211–241.

[83] Nathan Schurr, Janusz Marecki, Milind Tambe, Paul Scerri, Nikhil Kasinadhuni, and John P. Lewis. “The Future of Disaster Response: Humans Working with Multiagent Teams using DEFACTO.” In: *AAAI spring symposium: AI technologies for homeland security.* 2005, pp. 9–16.

[84] Elnaz Shafipour Yourdshahi, Matheus Do Carmo Alves, Leandro Soriano Marcolino, and Plamen Angelov. “Decentralised Task Allocation in the Fog: Estimators for Effective Ad-hoc Teamwork”. In: *11th International Workshop on Optimization and Learning in Multiagent Systems.* 2020.

[85] Elnaz Shafipour Yourdshahi, Matheus Do Carmo Alves, Leandro Soriano Marcolino, and Plamen Angelov. “On-line Estimators for Ad-hoc Task Allocation”. In: *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems.* AAMAS. 2020.

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

[87] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, et al. “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587 (2016), pp. 484–489.

[88] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, et al. “Mastering chess and shogi by self-play with a general reinforcement learning algorithm”. In: *arXiv preprint arXiv:1712.01815* (2017).

[89] David Silver and Joel Veness. “Monte-Carlo Planning in Large POMDPs”. In: *Proceedings of the Twenty-Fourth Annual Conference on Neural Information Processing Systems*. 2010.

[90] Reid G. Smith. “The contract net protocol: High-level communication and control in a distributed problem solver”. In: *Readings in distributed artificial intelligence*. Elsevier, 1988, pp. 357–366.

[91] Finnegan Southey, Michael P. Bowling, Bryce Larson, Carmelo Piccione, Neil Burch, Darse Billings, and Chris Rayner. “Bayes’ bluff: Opponent modelling in poker”. In: *in Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence* (2005).

[92] Peter Stone, Gal A. Kaminka, Sarit Kraus, and Jeffrey S. Rosenschein. “Ad Hoc Autonomous Agent Teams: Collaboration without Pre-Coordination”. In: *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence (AAAI)*. 2010.

[93] Ron Sun and Isaac Naveh. “Simulating organizational decision-making using a cognitively realistic agent model”. In: *Journal of Artificial Societies and Social Simulation* 7.3 (2004).

[94] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[95] Milind Tambe. “Towards flexible teamwork”. In: *Journal of artificial intelligence research* 7 (1997), pp. 83–124.

[96] Su-Yan Tang, Yi-Fan Zhu, Qun Li, and Yong-Lin Lei. “Survey of task allocation in multi Agent systems”. In: *Systems Engineering and Electronics* 32.10 (2010), pp. 2155–2161.

[97] Maulesh Trivedi and Prashant Doshi. “Inverse Learning of Robot Behavior for Collaborative Planning”. In: *Proceedings of the 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IROS. 2018.

[98] Ben G. Weber and Michael Mateas. “A data mining approach to strategy prediction”. In: *2009 IEEE Symposium on Computational Intelligence and Games*. IEEE. 2009, pp. 140–147.

[99] Douglas J. White. “A survey of applications of Markov decision processes”. In: *Journal of the operational research society* 44.11 (1993), pp. 1073–1096.

[100] Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.

[101] Mohan Yogeswaran and SG Ponnambalam. “Reinforcement learning: exploration–exploitation dilemma in multi-agent foraging task”. In: *Opsearch* 49.3 (2012), pp. 223–236.

[102] E. Shafipour Yourdshahi, T. Pinder, G. Dhawan, L. Soriano Marcolino, and P. Angelov. “Towards Large Scale Ad-hoc Teamwork”. In: *2018 IEEE International Conference on Agents (ICA)*. IEEE. 2018, pp. 44–49.

[103] Elnaz Shafipour Yourdshahi, Plamen Angelov, Leandro Soriano Marcolino, and Georgios Tsianakas. “Towards evolving cooperative mapping for large-scale uav teams”. In: *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE. 2018, pp. 2262–2269.

[104] Han Yu, Z. Shen, C. Leung, C. Miao, and Victor R. Lesser. “A survey of multi-agent trust management systems”. In: *IEEE Access* 1 (2013), pp. 35–50.

[105] Yisong Yue, Josef Broder, Robert Kleinberg, and Thorsten Joachims. “The k-armed dueling bandits problem”. In: *Journal of Computer and System Sciences* 78.5 (2012), pp. 1538–1556.

[106] An-Min Zou, Krishna Dev Kumar, and Zeng-Guang Hou. “Distributed consensus control for multi-agent systems using terminal sliding mode and Chebyshev neural networks”. In: *International Journal of Robust and Nonlinear Control* 23.3 (2013), pp. 334–357.