Decentralised Task Allocation in the Fog: Estimators for Effective Ad-hoc Teamwork

Elnaz Shafipour¹, Matheus Aparecido do Carmo Alves², Leandro Soriano Marcolino¹, Plamen Angelov¹

¹ School of Computing and Communications, Lancaster University
² Institute of Mathematics and Computer Science (ICMC), University of São Paulo (USP)

elnaz.shafipour@lancaster.ac.uk, matheus.aparecido.alves@usp.br,
1.marcolino@lancaster.ac.uk, p.angelov@lancaster.ac.uk

ABSTRACT

In many cases (e.g., rescue) agents complete tasks in non-structured partially observable scenarios, without previous knowledge of others. Known as ad-hoc teams, they work even when agents are from different parties, without prior coordination and communication protocols. We employ parametrised type-based reasoning, where a distribution over agents types is estimated, and parameters for each are learnt. These do not need a large number of observations and can be used on-line. To increase the performance of such teams, we introduce OEATA, a novel algorithm for type and parameter estimation. We show that it converges to perfect estimations, and obtain significantly better performance than the state-of-the-art, in terms of type and parameter errors and task completion, in full and partial observability scenarios.

1 INTRODUCTION

Ad-hoc teamwork is a great model for handling challenging real-world domains, since it allows agents to dynamically collaborate to solve tasks without prior coordination rules nor communication protocols. For instance, consider rescue robots from different organisations urgently brought together to aid in a natural disaster – e.g., earthquake. Designing coordination/communication protocols would take time, and resources; avoiding such delays and funding usage could save lives.

Instead of learning on-line any policy, a common approach in the literature is to consider a set of possible agent types and parameters, reducing the problem to estimating those [4, 7]. This approach is more applicable than learning models from scratch, as it does not require such a large number of observations, and can be more easily applied in an on-line manner. Types could be built based on previous experiences [8] or may be derived from the domain [1]. The introduction of parameters for each type allowed more fine-grained models [2].

However, the previous works were not specifically designed for decentralised task allocation, missing an opportunity to obtain better performances in this relevant scenario for multi-agent collaboration. Note that individual agents do not need to share the same representation of the problem, and run algorithms that explicitly “choose” tasks. They could be developed by different parties, and could use different paradigms. All we need are problems that can be modelled as decentralised task allocation for our ad-hoc agent. Similarly, a global allocation algorithm is unfeasible in our scenario: agents developed by others would not necessarily follow commands from a central entity, and we are not assuming any communication protocol.

Hence, we present On-line Estimators for Ad-hoc Task Allocation (OEATA), a novel algorithm for estimating team-mates types and parameters in decentralised task allocation. We show that our algorithm converges to a perfect estimation when the number of tasks to be performed gets larger. Additionally, we run experiments in a collaborative foraging domain, considering (for the first time) both full and partial observability scenarios, where agents collaborate to collect “heavy” boxes together. We show that we 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.

2 RELATED WORK

Ad-hoc teamwork is an important area in multi-agent systems [5]. We consider type-based reasoning, where agents are from a known set of potential types [1, 4, 7, 8], and parameters allow fine-grained models. These parameters, however, must also be estimated in an on-line manner. [2] introduced the AGA and ABU algorithms, which sample sets of parameters for gradient ascent and Bayesian estimation. Focusing on decentralised task allocation, we surpass their parameter and type estimations, leading to better performance. We also extend for the first time the paradigm of learning types and parameters to partial observability scenarios.

Another line of work attempts to identify the task being executed by a team [17]; or an agent’s strategy for solving a repetitive task [19]. Our work is different since we focus on a set of (known) tasks which must be completed by the team. Recently, in [10] an ad-hoc agent infers which tasks its team-mates are pursuing and uses that to plan its task. However, they consider disjoint tasks and do not learn team-mates models.

I-POMDP based models [11, 12, 14] allow agents to reason over others in planning. However, they are computationally complex, assuming all agents are learning about others recursively, and they consider agents with individual rewards. We propose a lighter MDP model, with a single team reward, allowing us to tackle larger task allocation scenarios.

POMCP is usually employed in problems with partial observability [18]. However, it is originally designed for a discrete state space, making it harder to apply POMCP for parameter estimation.
We apply, however, POMCP in combination with OEATA. We also evaluate experimentally the performance of plain POMCP for our problem.

[13] proposed a Bayesian MCTS, sampling different MDP models. Our planning approach (inspired from [2, 8]) is similar, as we sample different agent models from our estimations. However, instead of directly working on the complex transition function space, we learn agents types and parameters, which would then translate to a certain transition probability.

OEATA is inspired by Genetic Algorithms [15]: we keep an estimator population, and new ones are generated either randomly or using information from previous ones. However, GAs evaluate all individuals simultaneously at each generation, and usually they are immediately eliminated according to a fitness function. Our estimators, on the other hand, are evaluated per agent at task completion, survive according to success rate, and are also used for type estimation. They are also generated in different ways than GA mutation/crossover.

3 METHODOLOGY

As usual in ad-hoc teamwork, we consider one agent φ, in the same environment as a set of agents Ω (φ ∈ Ω). φ must maximise team performance, but it does not know how agents ω ∈ Ω may behave at each state. As in previous works [2], we consider that agents in Ω can be defined by a type θ ∈ Θ, and by a vector of parameters p, each in a fixed range. Estimating θ and p allows φ to estimate ω’s behaviour, leading to better decision-making.

Note that ω agents may be using different algorithms than the ones available in our set of types Θ. Nonetheless, φ would still be able to estimate the best type θ and parameter p to approximate ω’s behaviour. For each type θ ∈ Θ, we will estimate the probability P(θ|ω) of ω having type θ.

We consider ad-hoc teamwork in the context of decentralised task allocation: there is a set of tasks to be completed, and agents are able to autonomously decide which one to perform, without being allocated a task through a centralised mechanism [9]. The decentralised allocation is quite natural in ad-hoc teamwork, as we cannot assume that other agents would follow a centralised controller.

We consider that a task may require multiple agents to be performed successfully, and may require multiple time steps to be completed. For instance, in foraging a certain heavy item could not be completed. For instance, in foraging a certain heavy item could complete a task in state s, it chooses a new task t’. Hence, we call s a choose target state. We keep track of the history of task selections for each ω: Hω = {(s0, τ0), ..., (sn, τn)}, where s’ is the choose target state and τ is the task that ω completed afterwards. We also keep a set of estimators Eθ for each type θ, and agent ω. E.g., if there are 2 potential types and 3 agents in the scenario, there would be 6 sets. Eθ have a fixed size N. An estimator e is a tuple, (pe, se, ce, fe): pe is a parameter vector; se holds the number of times that in the future, and γ ∈ (0, 1) is a discount factor. We consider that given a state s, an agent ω has a (unknown) probability distribution (pdf) across a set of actions Aω, which is given by ω’s internal algorithm. The actions of all agents define the next state and the reward obtained, but we do not directly represent that. Instead, the pdf of each ω defines the transition function. That is, in the actual problem the next state depends on the actions of all agents. However, in the point of view of single-agent MDP, we can see it as the next state being probabilistic, given ϕ’s action. The uncertainty comes from the randomness of the actions of ω, besides any stochasticity of the environment.

We assume that given a state s, a type θ, and parameters p, we can obtain the pdf of ω in s. E.g., previous works considered an algorithm “template”, which outputs the pdf given θ and p as input [2]. Hence, the problem of estimating the pdfs reduces to estimating types and parameters for each agent. We also consider that the rewards are obtained by solving tasks from a set T. That is, ϕ’s reward is ∑rτ, where rτ is the reward obtained after completing τ, and the sum goes across all tasks that were completed in a given state (not only completed by ϕ). The number of required agents for a task τ depends on each specific task, and the set of agents that are jointly trying to complete it.

We employ UCT for ϕ’s decision-making process, a Monte Carlo Tree Search based approach, which performs multiple roll-out simulations in a search tree [16]. At each node, the next child node is decided by the UCB1 equation [6], and rewards are backpropagated up the tree to update action statistics at each node.

Given ϕ’s action, the next state depends on the actions of all ω in Ω. Hence, at each node transition, we sample their actions from their estimated pdfs. Agents’ true pdfs are unknown (i.e., we do not know the transition function). As in [2], we sample a θ for each ω from our estimated type probabilities each time we re-visit the root during the tree search. We use the current estimated parameters for the corresponding sampled type. Hence, the higher the quality of our estimations, the better the result of the tree search.

3.1 On-line Estimators for Ad-hoc Task Allocation

The main idea of OEATA is to record all tasks that each agent ω accomplishes, to compare them with the predictions of a set of estimators. These contain potential parameters p, of a potential type θ, which are used to predict task selections. The ones that are not able to make good predictions are removed after a number of failures, and replaced by estimators that are created using successful ones as a basis, or purely at random.

We assume that when ω completes a task τ in state sτ, it chooses a new task t’. Hence, we call sτ a choose target state. We keep track of the history of task selections for each ω: Hω = {(s0, t0), ..., (sn, tn)}, where s’ is the choose target state and t is the task that ω completed afterwards. We also keep a set of estimators Eθ for each type θ, and agent ω. E.g., if there are 2 potential types and 3 agents in the scenario, there would be 6 sets. Eθ have a fixed size N. An estimator e is a tuple, (pe, se, ce, fe): pe is a parameter vector; se holds the number of times that
e was successful in predicting ω’s next task; fe holds the number of consecutive failures.

All e are initialised in the first step, and are updated when a task is completed. We simulate ω’s task decision process assuming pe as ω’s parameters and θ as ω’s type. The estimators are then iteratively updated depending on their success or failure in estimating the target tasks. Our algorithm has four steps: (i) Initialisation; (ii) Evaluation; (iii) Generation; (iv) Estimation. Additionally, an Update step is executed for all agents in Ω, any time a task is completed.

Initialisation: Firstly φ creates N estimators e for each type θ. If there is a lack of prior information, pe of each e can be initialised with a random value (e.g., from the uniform distribution). For all e, se is set as the initial state of the environment. Since each e has a certain type θ and a certain parameter vector pe, it allows φ to estimate ω’s task decision process in the initial state. The estimated chosen task is assigned as τe.

Evaluation: Evaluation of estimators Eθ e for a certain agent ω starts when ω completes a task τe (Algorithm 1). For every e in Eθ e, we check if τe is equal to τe. If they are equal, we save all values p_i in pe in a respective bag bj. Note that i refers to a position in the vector of parameters p, and there is a bag bj for each i. bj is not erased between iterations, and hence they may increase in size at each iteration. There are sets of bags for each Eθ e (one bj for each position i), but for a cleaner notation we refer to them as simply bj. We update ce as the number of successes across the history of ω and set fe to 0. CheckHistory counts the number of correct task predictions across the history of ω, assuming type θ and parameter Pe.

Algorithm 1 Evaluating Estimator

1: for all θ ∈ Θ do
2: for each e ∈ Eθ e do
3: if τe = τe then ▷ τe is completed task.
4: for each p_i in pe do
5: b_j ← b_j ∪ p_i ▷ Union (∪) with repetition.
6: ce ← CheckHistory(ω, θ, bj, Pe) + 1; fe ← 0
7: else
8: fe ← fe + 1; ce ← ce − 1
9: if fe > threshold ξ then
10: remove e from Eθ e
11: se ← state; τe ← new estimated target

If τe is not equal to τe, we increase fe and decrease ce. If fe is greater than a threshold ξ, we remove e. We also update the choose target state (se) and τe of all e. Line 6 of the algorithm is not ce ← ce + 1, because of the decrease when a failure happens in line 8: we decrease ce when there are consecutive failures, but we restore it to the number of successes when a correct prediction is made. This aids in our type estimation.

We also update ω’s history Hω, in order to use it for future evaluations. We add (se, τe) to the history, where τe is the task just completed and se the corresponding choose target state (i.e., when τe was chosen as next task). Note, however, that we have no access to the agent’s true choose target state. Even though when an agent completes a task, the choose target state of all e would be the same se, these can later change during the execution (as we will see later).

Therefore, we use the estimators in Eθ e to estimate sα; we set se to the one stored in the estimator e with highest ce across all sets Eθ e.

Generation: Let Eθ e be the new set with only the surviving estimators (all e that were not removed in Algorithm 1). We generate N − |Eθ e| new estimators, in order to again have sets Eθ e with size N. Let pθ be a new parameter vector associated with a new e. A proportion m of the new estimators is created by randomly sampling from the uniform distribution in the corresponding parameter’s range, for all elements p_i of pθ.

For the remaining proportion, we create e’ using the bags b. Each position i is populated by randomly sampling from b_i. Every time a new e’ is created, we check if p_e’ would have at least one success across the history. If so, e’ is added to Eθ e, otherwise it is discarded. This decreases the likelihood of wasting an estimator with a parameter that would not be able to make any correct prediction (estimated from the history). The process repeats until |Eθ e| = N (Algorithm 2).

Algorithm 2 Generate New Estimators

1: η ← 0; number of mutations ← (N − |Eθ e|) × m
2: while |Eθ e| < N do
3: if η < number of mutations then
4: pe ← random value ▷ p_i uniformly sampled.
5: else
6: pe ← random from b ▷ p_i sampled from bag b.
7: histsuccess ← CheckHistory(ω, θ, pe)
8: if histsuccess > 0 then
9: Calculate τe in CurrentState
10: ce ← histsuccess; se ← CurrentState; fe ← 0
11: Add e to Eθ e; η ← η + 1

Estimation: At each iteration, we use the average pe across all e in Eθ e as the current estimated parameter for ω, when assuming type θ. We also estimate the probabilities P(θ|ω) of each agent ω having type θ. To do so, we use the success rate of all estimators of the corresponding type. For each θ, we first calculate: kθ e = max(0, Σe∈Eθ e ce). These are then normalised as: kθ e = kθ e / Σe∈Eθ e kθ e.

Finally, we update the type estimation: P(θ|ω) ∝ kθ e × P(θ|ω), where P(θ|ω) is the previous estimation. In the first iteration, we need prior probabilities for P(θ). These would normally be initialised with the uniform distribution, in the absence of previous information.

Update: A certain r may be completed by any subset of agents (including φ), which was the target of a certain ω, who was not in the subset of agents that have just completed the task. ω would notice that r is done by other agents and would switch to a different task at that state. Hence, once τ is completed, we check every τ_i in Eθ e for all ω that have not just completed τ, to see if there is any e where τ_e = τ. If there is any, we set se as the current state, and update its target task accordingly based on the parameters pe and type θ. That is, we simulate ω choosing a new target task at state s_e.
3.2 Analysis

We show that as the number of tasks goes to infinite, we perfectly identify the type and parameters of all \( \omega \), given some assumptions: we 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 parameters have a fixed range, there is a finite number of possible values for every element \( p_i \). To simplify the exposition, we consider \( n \) possible values per element (in general they can have different sizes). Let \( d \) be the dimension of the parameter space.

We also assume that any parameter estimation in the wrong type, and any parameter \( p' \neq p^* \) (even in the correct type) will not succeed infinitely often, where \( p^* \) is the correct parameter. That is, as \( |T| \to \infty \) there will be cases where it successfully predicts the task, but the number of cases is limited by a finite constant \( c \). Let \( \theta^* \) be the correct type, and \( \theta \neq \theta^* \).

**Proposition 3.1.** OEATA estimates the correct parameter and type for all agents as \(|T| \to \infty\).

**Proof.** Because of the mutation proportion \( m \), we always have new estimators with random \( p_e \) (since wrong parameters eventually reach the failure threshold, so new ones are generated). As we sample from the uniform distribution, \( p^* \) will be sampled with probability \( 1/n^d \) > 0. Hence, eventually it will be generated as \(|T| \to \infty\). In fact, as the generation defines a Bernoulli experiment, from the geometric distribution we have that in expectation we need \( n^d \) trials. Therefore, eventually, there will be an estimator with the correct parameter vector \( p^* \), and it will eventually receive the highest \( c_e \) score across the full history since it has the highest probability of making correct predictions. Furthermore, all \( p_e \neq p^* \) will eventually reach the failure threshold \( \xi \), and will eventually be discarded. Hence, when \(|T| \to \infty\) the average across \( E_{|T|}^{p^*} \) will be \( p^* \). Concerning type estimation, we have that \( c_e \to \infty \) in the set \( E_{|T|}^{\theta} \) of the true type \( \theta^* \). Hence, \( k_{|T|}^{\theta^*} \to \infty \), while \( c_e < c \) for \( \theta \neq \theta^* \) (by assumption).

Therefore, \( k_{|T|}^{\theta*} = \frac{k_{|T|}^{\theta^*}}{\sum_{\theta \neq \theta^*} k_{|T|}^{\theta}} \to 1 \), while \( k_{|T|}^{\theta} \to 0 \), as \(|T| \to \infty\). Hence, the probability of the correct type \( P(\theta^*) \to 1 \) □

If a parameter estimation in the wrong type does succeed infinitely often, then the analysis is more intricate, since we may also have that \( k_{|T|}^{\theta} \to \infty \). However, we can still show that the correct type will receive a higher probability:

**Proposition 3.2.** If a parameter estimation in the wrong type succeeds infinitely often, OEATA still gives a higher probability to the correct type, for a sufficiently large \( N \).

**Proof.** We consider that the correct parameter estimation in the correct type succeeds more frequently than parameter estimations in the wrong type. If that is not true than the wrong type would be an even better model to be used for planning, leading to a contradiction (as \( \theta^* \) would not be the correct type).

Let \( k_{|T|}^{\theta}(x) \) denote \( k_{|T|}^{\theta} \) for \( x \) tasks. By the Stolz–Cesàro theorem, the limit of \( k_{|T|}^{\theta} \) as \(|T| \to \infty\) is: \( \lim_{|T| \to \infty} k_{|T|}^{\theta}(x) = \lim_{|T| \to \infty} \frac{k_{|T|}^{\theta}(x+1) - k_{|T|}^{\theta}(x)}{\Sigma_{\theta \neq \theta^*} k_{|T|}^{\theta}(x+1) - \Sigma_{\theta \neq \theta^*} k_{|T|}^{\theta}(x)} \).

We also have that the parameter estimation in the correct type will eventually converge to the true parameter. Hence, as the correct parameter estimation in \( \theta^* \) succeeds more frequently than parameter estimations in \( \theta^* \), there will be \( N \) such that \( k_{|T|}^{\theta^*}(x+1) - k_{|T|}^{\theta^*}(x) > k_{|T|}^{\theta^*}(x+1) - k_{|T|}^{\theta^*}(x) \). Therefore: \( \lim_{|T| \to \infty} k_{|T|}^{\theta^*} > \lim_{|T| \to \infty} k_{|T|}^{\theta} ⇒ P(\theta^*) > P(\theta^*) \) □

We saw in Proposition 3.1 that a random search from the mutation proportion takes \( n^d \) trials in expectation. OEATA, however, finds \( p^* \) much quicker than that, since a proportion of estimators are sampled from \( b \). To show that formally, we make the following assumptions: (i) a correct value \( p^*_i \) in any position \( i \) may still predict the task wrongly (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) although each bag could have at most \( n \) different values in the worst case, we assume that they have an expected fixed size \( b \) as \(|T| \to \infty\) (after removing repetitions of \( p^*_i \)).

That is, if one of the vector positions \( i \) is correct, \( p \) 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 prediction. E.g., if a task is nearby, for almost any vision radius it would be predicted as the next one if the vision angle is correct. On the other hand, wrong values do not succeed infinitely, and hence the bags have an expected size \( b < n \). 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.

**Proposition 3.3.** OEATA finds \( p^* \) in \( O(d \times n \times b^d) \).

**Proof.** By the same argument as above, sampling the correct value for element \( p_1 \) would take \( n \) trials in expectation. Once a correct value is sampled, it will be added to \( b_1 \) 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 \( \xi \). However, for a constant number of trials \( t \times n \), it would be added to \( b_1 \). Similarly, sampling at least one time the correct value for all \( d \) dimensions would take \( d \times n \) trials in expectation, and in at most \( t \times d \times n \) trials all \( b_1 \) 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 \( b_1 \), leading to a \( 1/b \) probability to sample the correct value per bag. Hence, given the bag sampling operation, we would find \( p^* \) with \( t \times d \times n \times b^d \) trials in expectation. □

Our complexity is close to \( O(n) \), instead of \( O(n^d) \) as the random search (since \( b \ll n \)). \( O(n) \) complexity could still be prohibitive in some real applications if \( n \) is large. However, \( n \) can be reduced arbitrarily: a designer can trade convergence speed with model precision, reducing the parameter space by aggregation of values (e.g., reducing the decimals).

3.3 Example

Consider Figure 1: we show one ad-hoc agent \( \phi \), two agents \( \omega_0, \omega_1 \), and three tasks \( t^1 \). They do foraging tasks are defined as collecting
We define a single agent \( \theta \) allowing us to combine the POMCP algorithm \cite{18} with OEATA:

\[ 3.4 \text{ OEATA in Partial Observability} \]

For each \( e \) that is not a valid task anymore since it was already completed. \( e \) updating them accordingly.

\[ \omega \]

ones will be generated using the uniform distribution. Therefore, we may create new estimators that made a correct prediction had the \( \omega \), hence completing that task. \( \phi \) then evaluates and updates all \( e \) if \( \tau_e \) equals \( \tau_t \), success counter \( c_e \) is increased by 1. Otherwise, counter of consecutive failures \( (f_e) \) is increased. Assuming threshold for removing estimators \( \xi = 1 \), some \( e \) will survive at \( E_{\theta_1} \), while others will be removed. Let’s assume that the estimators that made a correct prediction had the parameters \((0.4,0.6), \) and \((0.2,0.5)\). Hence, the bags for \( \theta_1 \) will be: \( b_0 = \{0.4,0.2\}; b_1 = \{0.6,0.5\} \). Assuming \( m = 1/3 \), 2 new \( e \) will be generated by sampling from these bags, while 1 will be fully random. Therefore, we may create new \( e \) with the following \( p_e \) \((0.4,0.5)\); \((0.2,0.6)\); \((0.1,0.7)\), where the last vector is fully random.

For \( E_{\theta_2} \), since no \( e \) predicted \( \tau_t \), they will all be removed, and 5 new ones will be generated using the uniform distribution.

Hence, at this iteration, \( \omega_0 \)’s \( p_0 \) parameter will be estimated as \((2 \times 0.4 + 2 \times 0.2 + 0.1)/5 = 0.26 \). \( \omega_0 \)’s probability of being \( \theta_1 \) will be updated as: \( k^{\theta_1} = 2/(2 + 0) = P(\theta_1) = 1 \). Afterwards, given \( \omega_0 \) new position \((3,6)\) (next to the box it just collected), \( \phi \) runs again a simulation of \( \omega_0 \)’s task decision process for each \( e \) in the sets \( E_{\theta_0} \), updating them accordingly.

\( \omega_1 \)’s estimators are also updated, even though it did not collect any item. Some \( e \) in \( E_{\theta_1} \) may have \( \tau_t \) as the estimated task \( \tau_e \), and that is not a valid task anymore since it was already completed. For each \( e \) where \( \tau_e = \tau_t \), \( \phi \) simulates \( \omega_1 \)’s task decision-making, assuming \( p_e \) and the current state. For these estimators, both \( \tau_e \) and the choose target state are updated.

3.4 OEATA in Partial Observability

We define a single agent POMDP (extending our previous MDP), allowing us to combine the POMCP algorithm \cite{18} with OEATA: every action \( a \) produces an observation \( o \in O \), which represents \( \phi \)’s visible environment. POMCP uses a particle filter to approximate the belief state at each node in the UCT tree. Each time we traverse the tree, a state is sampled from the particle filter of the root. Given an action \( a \), a simulator samples the next state \( s' \) and the observation \( o \). The pair \( ao \) defines the next node \( z \) in the search tree. State \( s' \) is added to \( z \)’s particle filter, and the process repeats recursively down the tree. We refer to \cite{18} for a detailed explanation. As in UCT, we do not know the true transition and reward functions (since \( \Omega \)’s pdfs are unknown). Therefore, we employ the same strategy: each time we go through the search tree, we 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 we re-visit the root node. These sampled types and parameters are also used in the POMCP simulator, when we sample a next state, a reward and an observation after choosing an action in a node. We handle partial observability by sampling a particle from the POMCP root, which corresponds as sampling a state \( s \) from the belief state. \( s \) is then used as the current state in OEATA. States that are more likely will be sampled with a higher probability.

4 EXPERIMENTS

4.1 Level-based Foraging Domain

We evaluate our approach in level-based foraging, a common problem for evaluating ad-hoc teamwork \cite{2,3,20}. In this domain, a set of agents must collect items displaced in the environment (which corresponds to the “tasks” in our model). Each item has a certain weight, and each agent has a certain (unknown) skill-level. If the sum of the skill-levels of the agents (that are trying to collect an item) surrounding a target is greater than or equal the item’s weight, the item is “loaded” by the team (Figure 2). Each agent has 5 possible actions, in a grid-world environment: North, South, East, West, and Load (which tries to load an item next to the agent, if the agent is facing that item). We consider two possible agent types, taken from previous works in this domain: We use the two “leader” types defined in Albrecht and Stone \cite{2}. Additionally, the visibility region of each \( \omega \) has an angle and a maximum radius, which are unknown. Therefore, there are 3 parameters to be learned for each \( \omega \): Skill-level, Angle and Radius. According to \( \omega \)’s type and parameters, its target item (task) will be selected. We refer the reader to Albrecht and Stone \cite{2} for a detailed description of the types and their respective parameters.
4.2 Results

We will compare our algorithm (OGE) against two state-of-the-art parameter estimation approaches in ad-hoc teamwork: AGA and ABU [2]. Both approaches also sample sets of parameters (for a gradient ascent step or a Bayesian estimation), and we use the same set size as estimator sets (N). Additionally, we compare our approach against using plain POMCP for type and parameter estimations. In this case we still consider that the agent is able to see the whole environment; however, agent type and parameters are not observable, and hence are estimated using POMCP’s particle filter. We use N × |Ω| × |Θ| particles, matching the total number of estimators in our approach (since we have N per agent, for each type). We executed 100 runs for each experiment, and plot the average results and the confidence interval (\( \rho = 0.01 \)). When we say that a result is significant, we mean statistically significant considering \( \rho \leq 0.01 \).

In plain POMCP, we calculate the type probability of a certain agent \( P(\theta) \) by counting the frequency that the type \( \theta \) is assigned to the agent in the root’s particle filter. For the parameter estimation, we use the average across the particle filter (for each type), allowing us to calculate the parameter estimation error in our evaluation.

OGE used the following parameters: \( N = 100, t = 2, m = 0.2 \). Type and parameters of agents in \( \Omega \) are chosen uniformly randomly, and the weight of each item is chosen uniformly randomly (between 0 and 1). Each scenario is also randomly generated. Agent \( \phi \)'s skill-level is fixed at 1, so every generated instance is solvable. We ran UCT for 100 iterations per time step, and maximum depth 100. We fix the scenario size as \( 20 \times 20 \), and ran experiments for a varying number of team-mates (|\( \Omega \)|). We first show results when \( \phi \) has full observability of the scenario, and later we will show with partial observability. When running experiments in partial observability, we consider a circular visibility region centered on \( \phi \), with radius 5.

We first show examples of the parameter and type error for \(|\Omega| = 5\) (Figure 3). We evaluate the mean absolute error for the parameters, and \( 1 - P(\theta^*) \) for type; and we show here the average error across all parameters. As we can see, our 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 (as may be expected from Proposition 3.1). 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 our type estimation becomes quickly better than the other algorithms, significantly overcoming them after a few iterations.

We now study the error for different \(|\Omega|\) sizes, and the execution time to complete all tasks (in terms of number of iterations), in

Figure 3: Parameter and type estimation errors for \(|\Omega| = 5\). Lower is better.

Figure 4: Parameter, type estimation errors, and performance for a varying number of agents in full observability. The graph in the bottom shows results with more than 3 agents in greater detail.
Decentralised Task Allocation in the Fog

5. CONCLUSION

We study ad-hoc teamwork for decentralised task allocation. One ad-hoc agent learns its team-mates, in order to better decisions concerning overall team performance. We propose a novel algorithm that obtains better estimations than previous works in ad-hoc teamwork, leading to better performance. OEATA converges to zero error, and in our experiments the error decreases with the number of iterations. We also show estimations with partial observability for the first time in ad-hoc teamwork, and still outperform previous works in this challenging scenario.
REFERENCES


