MOO-MDP: An Object-Oriented Representation for Cooperative Multiagent Reinforcement Learning

Felipe Leno Da Silva, Ruben Glatt, *Member, IEEE,* and Anna Helena Reali Costa, *Member, IEEE*

***Abstract*—Reinforcement Learning (RL) is a widely known technique to enable autonomous learning. Even though RL methods achieved successes in increasingly large and complex problems, scaling solutions remains a challenge. One way to simplify (and consequently accelerate) learning is to exploit regularities in a domain, which allows generalization and reduction of the learning space. While Object-Oriented Markov Decision Processes (OO-MDP) provide such generalization opportunities, we argue that the learning process may be further simplified by dividing the workload of tasks amongst multiple agents, solving problems as Multiagent Systems (MAS). In this work, we propose a novel combination of OO-MDP and MAS, called Multiagent Object-Oriented Markov Decision Process (MOO-MDP). Our proposal accrues the benefits of both OO-MDP and MAS, better addressing scalability issues. We formalize the general model MOO-MDP and present an algorithm to solve deterministic cooperative MOO-MDPs. We show that our algorithm learns optimal policies while reducing the learning space by exploiting state abstractions. We experimentally compare our results with earlier approaches in three domains and evaluate the advantages of our approach in sample efficiency and memory requirements.**

***Index Terms*—Machine Learning, Reinforcement Learning, Multiagent Systems, Cooperative Learning.**

I. Introduction

Reinforcement Learning (RL) [1] methods aim at autonomously learning how to solve tasks through interactions with the environment. While RL has been successfully applied to varied and increasingly complex applications [2], [3], [4], the classical RL approach suffers from the *curse of dimensionality*, and hence the success of RL methods is dependent on additional techniques to help with scalability.

An appropriate task description can significantly help the agent to find commonalities in the environment by generalizing knowledge, and many works depict benefits when relational techniques are used, such as *Relational MDP* (RMDP) [5], [6] and *Object-Oriented MDP* (OO-MDP) [7]. Both models describe tasks through objects and their relations. While the former relies on relational predicates, the latter defines its state-action space over objects within the environment and their attributes. OO-MDPs have been used in many works recently [8], [9], [10], [11], [12], as it provides an intuitive way to describe tasks (through observable object attributes) and enables generalization opportunities[7]. Moreover, the class descriptions required by OO-MDP usually demand less knowledge than RMDP's propositional functions.

F. L. Silva, R. Glatt, and A. R. Costa are with Intelligent Techniques Laboratory (LTI), University of São Paulo, Brazil. E-mail: {f.leno,ruben.glatt,anna.reali}@usp.br.

Another aspect to take into account when scaling RL to complex domains is the presence of multiple autonomous agents in the environment. In a world where more and more devices have computing power, many tasks can (sometimes must) be solved by Multiagent Systems (MAS) [13]. In addition to providing a decision autonomy to each agent in the environment, MAS also involve the property of interacting with other agents, which requires the ability to cooperate, coordinate, and negotiate with each other, *Multiagent Reinforcement Learning* (MARL) [14] solves the learning task in MAS; however, RL techniques are not easily-portable to MAS, because other agents actuating in parallel render the environment non-stationary. Also, the state transition becomes dependent on the joint action of all agents, instead of single actions. Some MARL approaches assume that a centralizer agent executes the entire reasoning process and determines actions to be executed for all agents [15]. However, the reasoning agent would tackle a learning task that grows exponentially according to the number of agents. Hence, such approach is infeasible for most domains, for which distributed solutions are usually more desirable [3]. Moreover, a distributed approach enables the quick deployment of new agents in the system, which is not always trivial when using a single controller.

Although the workload to solve the task can be divided among several agents, the learning task is still hard to solve. Hence, MARL can also benefit from relational techniques to generalize knowledge in the domain [16]. Although some works indicate that relational techniques can benefit learning in MAS [17], [18], OO-MDP has not been used for MAS yet. While the first effort to extend OO-MDP to MAS appeared in BURLAP [19], a library of planning and learning RL methods based on relational task descriptions, no work presented a formal OO-MDP framework for MAS nor distributed OO-MDP solutions for a generic number of agents.

We here argue that each agent in a MAS can be seen as an object, and extend OO-MDP for MAS, defining the *Multiagent Object-Oriented MDP* (MOO-MDP). We also contribute a *model-free* algorithm to solve deterministic distributed MOO-MDPs for cooperative domains, hereafter called *Distributed Object-Oriented Q-Learning* (DOO-Q). DOO-Q helps to accelerate learning by reasoning over abstract states. Moreover, under certain constraints, DOO-Q still learns optimal policies without observing the entire concrete state space. Thus, the contributions of this paper are threefold: (i) We propose the MOO-MDP model that abstracts the state space in MAS through object-oriented task descriptions; (ii) We contribute an algorithm to solve MOO-MDPs and prove that it learns optimal joint policies (under certain conditions); and (iii) We

present empirical evaluations in several domains comparing our algorithm with previous techniques.

This article extends our previous work [20] by deepening our discussion of related work and experiments, presenting additional experimental evidence, and presenting in full the theoretical proof of convergence to an optimal joint policy.

The remainder of this article is organized as follows: In Section II we define all relevant concepts for our proposal. In Section III we introduce the MOO-MDP formalism and in Section IV we present an algorithm to learn an optimal policy in deterministic cooperative MOO-MDPs. The experimental evaluation is presented in Section V and results are discussed in Section VI. Finally, we conclude our article and point toward further works in Section VII.

II. Background on Reinforcement Learning

Markov Decision Processes (MDP) are used to model sequential decision-making problems that can be solved with RL. An MDP is described by $\langle S, A, P, R, \gamma \rangle$ [21], where S is the set of environment states. A common and convenient way to describe states is through a factored description, that is, a state is composed of a Cartesian product of state variables. A is the set of actions available to an agent, $P : S \times A \times S \rightarrow [0, 1]$ is the state transition function, where $P(s, a, s')$ is the probability of observing state s' after applying action a in s (for deterministic domains $P(s, a, s') \in \{0, 1\}, \forall \langle s, a, s' \rangle$). R is the reward function, and γ, $0 \leq \gamma < 1$, is the discount factor, which represents the relative importance of future and present rewards. A decision maker (agent) takes actions at each decision step. At first, the agent observes the current state s, then it can choose an action a among the applicable ones. The chosen action causes a state transition $s' \leftarrow P(s, a)$ and the agent can observe a reward signal $r \leftarrow R(s, a, s')$. After that, this cycle is repeated until a termination condition is achieved, and the agent must infer a policy π to choose an action for each state through observing those decision-making cycles. An optimal policy π^* is the solution of an MDP, i.e., a policy that chooses the actions that maximize the discounted reward signal for every state. In this work we are interested in learning problems (i.e., P and R are unknown to the agent) that can be solved through interactions with the environment. The Q-Learning algorithm [22] is widely used to solve such learning problems. Q-Learning iteratively learns a Q-table, i.e., a function that estimates the long-term quality of each action when applied to each state: $Q : S \times A \rightarrow \mathbb{R}$. Q-Learning eventually converges to the optimal Q function[1]

$$Q^*(s, a) = E\left[\sum_{i=0}^{\infty} \gamma^i r_i\right], \tag{1}$$

where r_i is the reward received after i steps from using action a on state s and following the optimal policy on all subsequent steps. An optimal policy can be extracted from Q^* as $\pi^*(s) = \arg\max_a Q^*(s, a)$. Notice that the standard MDP only models one agent in the environment; although an MDP can be used to solve a MAS problem by ignoring all other agents, some kind of coordination is usually desired [23].

[1]The proof requires that: (i) all state-action pairs are infinitely visited; (ii) the rewards are bounded; and (iii) a proper learning rate is chosen [22].

A. Multiagent MDPs

A Stochastic Game (SG) [16], [24], [25] is an extension of MDP to MAS. As multiple agents are now present in the environment, in SGs the state and action sets are defined as the Cartesian product of local states and actions for all agents. The transition function now depends on the *joint* action, rather than one single individual action. A SG is described by the tuple $\langle S, A_{1\ldots m}, T, R_{1\ldots m}, \gamma \rangle$, where m is the number of agents in the environment. The set of states S is composed of local states from each agent: $S = S_1 \times S_2 \times \cdots \times S_m$, thus,the local states of all agents must be observed.

Several equilibrium-based MARL algorithms have been proposed to learn an equilibrium joint policy in such domains [26], [27]. These algorithms balance the reward of all agents through an equilibrium metric, instead of considering only individual rewards. However, these algorithms do not scale well as the number of agents increases, because the equilibrium computation becomes complex. On the other hand, Distributed Q-Learning [28] can be used to learn an optimal joint policy for cooperative scenarios (also called Multiagent MDPs [28], in which $R_1 = \cdots = R_m$ [29]), with only little computational complexity at each step. Distributed Q-Learning without knowledge of actions performed by other agents and stores only Q-values for the best possible *joint* action.

B. Object-Oriented MDP

The *Object-Oriented MDP* (OO-MDP) is a relational MDP extension intended to facilitate generalization in RL problems [30].We here make use of the *Goldmine* [30] domain to exemplify the object-oriented concepts applied to RL. Figure 1 illustrates the *Goldmine* domain. A team of *miners* aims at collecting as much *gold pieces* as possible. However, there are impassable *walls* in the environment to hamper miner movements. At each decision step, all miners may move or collect gold pieces that are close enough. Whenever any miner collects a gold piece, all miners receive a positive reward, hence miners always benefit from acting cooperatively.

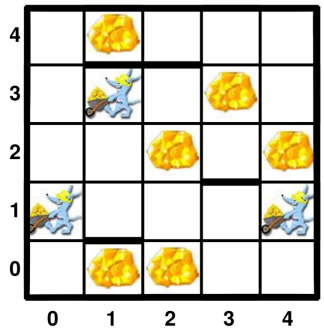
An OO-MDP is composed of $\langle C, O, A, \mathrm{T}, D, R, \gamma \rangle$. $C = \{C_1, \ldots, C_c\}$ is the set of *classes*. Each class $C_i \in C$ is composed of a set of *attributes*, $Att(C_i) = \{C_i.b_1, \ldots, C_i.b_b\}$, and each attribute b_j is restricted by a *domain* $Dom(C_i.b_j)$ specifying the set of possible values for that attribute. A possible object-oriented description of the *Goldmine* domain is defining three classes: *Miner*, *Gold*, and *Wall*, i.e., $C = \{Miner, Gold, Wall\}$. All these classes have x and y attributes, and walls also have a position attribute *pos* to indicate the position of the wall in respect to the grid cell: $Att(Miner) = Att(Gold) = \{x, y\}$, $Att(Wall) = \{x, y, pos\}$, $Dom(Miner.x) = Dom(Miner.y) = Dom(Gold.x) = Dom(Gold.y) = Dom(Wall.x) = Dom(Wall.y) = \{0, 1, 2, 3, 4\}$ (in a 5×5 grid), and $Dom(Wall.pos) = \{South, West, East, North\}$. As noticed in the example, the definition of classes and attributes must describe all types of objects in the environment and their relevant features.

$O = \{o_1, \ldots, o_o\}$ is the set of objects that exist in the task of interest. Each object is an instance of one class, so that $o_i \in C_j$ with $C(o_i) = C_j$ and, for each decision step, the object state is given by the current value of all attributes. The object uniqueness is given by an additional identification. For example, in the *Gridworld* domain, miners are distinguished by a "name", which is used to define the miner to be moved when actions are executed. Therefore, the object state is defined by the Cartesian product $o_i.state = \left(\prod_{b \in Att(C(o_i))} o_i.b\right) \times o_i.id$, where $o_i.id$ is the object identification.

Figure 1b presents an example of object-oriented state, where all objects in the environment are described by their attribute values and id. In an OO-MDP, the state of the underlying MDP is the union of all object states $s = \bigcup_{o \in O} o.state$.

The set A consists of actions that may or may not be parameterized. Parameterized actions affect any object belonging to a given set of classes in the same way, hence, parameterized actions are abstract and need to be grounded to be applied. Suppose an external agent controls all miners and can use the parameterized action *North(Miner)*. This action moves a miner towards North, however, a single miner must be specified in the action parameter in order to apply the action. For example, in the state described in Figure 1b, the action *North(miner1)* would move $miner1$ to cell (0,2).

T is a set of *terms*, which are boolean functions related to the state transition dynamics in an OO-MDP. Each term is either a *relation* between objects or an optional designer-specified function to describe domain knowledge. An example of a relation in the *Goldmine* domain is the term $on(Miner, Gold)$, which defines if (and which) miners are in the same position as gold pieces. An example of function to describe domain knowledge (not used in our experiments, though) is the function *noGold*(), which returns a true value only when all gold pieces in the environment were collected. D is a set of *rules* d, defined as tuples of $\langle condition, effect, prob \rangle$. A *condition* is a conjunction of terms of T and an *effect* f is an operation that changes with probability *prob* attribute values of an object, $f : Dom(C_i.b_j) \rightarrow Dom(C_i.b_j)$. As an example of *rule*, consider $d_N = \langle cond_N, f_N, prob_N \rangle$ related to action $North(Miner\ z)$. $cond_N(z)$ verifies if the miner z has a clear path in the north direction and will be able to move in the desired direction, thus $cond_N(z) = \neg touch_N(z, Wall)$, where the relation $touch_N$ returns a true value if the agent sees a wall in the north direction, which makes sure that f_N is only triggered when the miner's movement is not hampered by walls. $f_N(z) = z.y \leftarrow z.y + 1$ changes the position of z towards the desired direction. As *North* is an action with a single deterministic effect, $prob_N = 1$.



(a)

Object *id*	Attributes
$miner1$	$x = 0, y = 1$
$miner2$	$x = 1, y = 3$
$miner3$	$x = 4, y = 1$
$gold1$	$x = 1, y = 0$
⋮	⋮
$gold6$	$x = 4, y = 2$
$wall1$	$x = 1, y = 1,$ $pos = South$
⋮	⋮
$wall24$	$x = 4, y = 4,$ $pos = East$

(b)

Fig. 1. The *Goldmine* domain (illustration adapted from [30]). *Miners* aim to gather all *gold pieces* in the environment. Thick *walls* are impassable. (a) Graphical representation. (b) Object representation.

Finally, R and γ are, respectively, a reward function and a discount factor equivalent to the standard MDP ones. The conditions of terms, D, and R are not known by the agent in learning problems, which has to learn how to actuate through samples of interactions in the environment.

The transition dynamics in an OO-MDP is interpreted as follows. First, at each step k, the current state s_k is observed, and the agent applies one action a_k. Second, all terms are evaluated to be $true$ or $false$ at that step. And third, all rules associated to a_k are evaluated, and for all conditions that are matched, an effect is triggered. After all effects have been processed, the new object states characterize the state transition, and this cycle is repeated until a termination condition is achieved. An OO-MDP corresponds to a regular MDP, but the agent can use the extra information to generalize the learning space. Note that SGs and OO-MDP tackle specific scalability issues. In the next section, we propose to combine both methods to benefit from their advantages.

III. Multiagent Object-Oriented Representation

We now present a formal definition for an MOO-MDP, a relational MDP extension to MAS. MOO-MDP supposes that each agent can observe the other objects in the environment. We are interested in a distributed control, in which agents cannot tell other agent's actions. The main differences between OO-MDPs and MOO-MDPs are the same as the ones between MDPs and SGs, that is:

1) multiple agents are simultaneously affecting the environment. Now, the state transition depends on $joint$ actions, instead of local agent actions;
2) each agent may have a slightly different observation of the world, resulting in similar but possibly different local states; and
3) each agent has its private reward function, which means that each agent might have different goals.

Although we focus here on cooperative domains, MOO-MDP is a general model that can be used for general-sum MAS.

An MOO-MDP is described by $\langle C, O, U, \mathrm{T}, D, R^m, \gamma \rangle$. m is the number of agents and C is again the set of *classes*. We define the set of *Agent Classes* $Ag = \{Z_1, \ldots, Z_g\}, Ag \subseteq C$, meaning that each object belonging to a class $Z_i \in Ag$ represents an autonomous agent. Note that $g \leq m$, because more than one autonomous agent may belong to the same class. $\Gamma \subseteq C$ is the set of *abstracted* classes. This set is domain-specific and designer-specified. Including one class in this set means that all the objects of these classes will

be distinguishable only by their attribute values (the *ids* become invisible to the observing agent), hence each object are observed through an *abstract* state $o.\overline{state} = \prod_{b \in Att(C(o))} o.b$.

The set of objects O is divided as $O = E \cup G$, where E is the set of environment objects (not related to agents), $\forall e \in E : C(e) \notin Ag$, and G is the set of agent objects, $\forall z \in G : C(z) \in Ag$. *Concrete* states are now defined over the union of states from both environment and agent objects $s = \bigcup_{o \in O} o.state = (\bigcup_{e \in E} e.state) \cup \left(\bigcup_{z \in G} z.state\right)$. Consequently, if the state of some agents cannot be directly observed in the environment, these states must be received through communication in all decision steps. An abstract state $\tilde{s}$ is defined according to $\tilde{s} = \left(\bigcup_{o \in O, C(o) \in \Gamma} o.\overline{state}\right) \cup \left(\bigcup_{o \in O, C(o) \notin \Gamma} o.state\right)$, which means that objects belonging to abstracted classes are identified by their abstract states, while the other objects keep their concrete state in the point of view of the agent. Hence, $\tilde{s}$ is a set of concrete states ($\tilde{s} \subseteq S$). The function κ translates a concrete state s: $\tilde{s^z} = \kappa(s, z)$ to an abstract one for an agent z by suppressing the id of objects belonging to abstract classes $C_i \in \Gamma$. Note that the definition of abstract states enables knowledge generalization. Figure 2 illustrates how the abstraction works in a 2×2 *Goldmine* domain. Note that multiple concrete states are compressed into a single abstracted one. U is the set of joint actions for all agents. Joint actions are composed by a list of individual actions, which belong to an agent action set A_z, $U = A_1 \times \cdots \times A_m$. Individual actions can be parameterized or not, thus MOO-MDPs also allow action space abstraction. Moreover, individual action sets can be different.T and D have the same definition as in OO-MDPs, but they are now dependent on *joint* actions. $R^m = \{R_1, \ldots, R_m\}$ is the set of reward functions for all agents, which now is dependent on *joint* actions, instead of individual actions. While A_z is known by the agent, T, D, R^m, and U are unknown in a learning task.

The transition dynamics are illustrated in Figure 3. In each step k, each agent applies one action $a_k^z \in A_z$ in its local abstract state $\tilde{s}_k^z$. All terms are evaluated according to s_k (defined from O_k) and the joint action $\boldsymbol{u}_k$ triggers all effects related to matched conditions in the rules $d \in D$. Finally, a state transition is caused by the triggered effects, and the agent observes its reward r_k^z. Note that state transitions depend on both *term* values (defined over *concrete* states) and the *joint* action. The conditions governing transitions are unknown to the agent for learning problems. Therefore, reasoning over only abstract observations of the environment helps the agent to avoid considering all possible grounding of terms. In the next section, we present a model-free algorithm to solve deterministic cooperative MOO-MDPs with homogeneous agents.

IV. Learning in deterministic cooperative MOO-MDPs

We present here a solution for Deterministic Distributed Cooperative MOO-MDPs, a specific class of the general MOO-MDP framework presented in Section III. All agents aim at maximizing a single reward function in such domains, thus, $R_1 = \cdots = R_m$. We also assume that it is infeasible to build a central controller, and each agent takes actions without observing other agents' actions. We here propose to use a *model-free* algorithm based on Distributed Q-Learning [28] to solve such problems. In our algorithm, thereafter called *Distributed Object-Oriented Q-Learning* (DOO-Q), each agent learns a local policy in a distributed and generalized manner. Each agent z stores a local Q-table (Q^z) containing abstract states ($\tilde{s}_k^z$) and its own actions. We leave the action space abstraction for further works and assume that all actions are concrete (i.e., grounded in case of abstract actions). An agent z using DOO-Q learns a local policy that converges to an optimal joint policy[2] (under certain conditions, when all agents are using DOO-Q), even when unaware of other agent actions, through iteratively updating its Q-table using the equation proposed for Distributed Q-Learning over abstract states and concrete actions [28]:

$$Q_{k+1}^z(\tilde{s}_k^z, a_k^z) \leftarrow \max\{Q_k^z(\tilde{s}_k^z, a_k^z), r_k + \gamma \max_{a^z \in A_z} Q_k^z(\tilde{s}_{k+1}^z, a^z)\}. \tag{2}$$

Lauer and Riedmiller [28] proved that, when using only concrete states, this update-rule allows agents to learn a projection of the *joint* Q-table in a distributed manner. We apply Equation (2) with abstract states in order to learn an optimal joint policy while storing only a local Q-table, which can be done because it corresponds to the joint Q-table as

$$Q_k^z(\tilde{s}_k^z, a_k^z) \geq \max_{\boldsymbol{u} \in U, \boldsymbol{u^z} = a_k^z, s \in \tilde{s}_k^z} Q_k(s, \boldsymbol{u}), \tag{3}$$

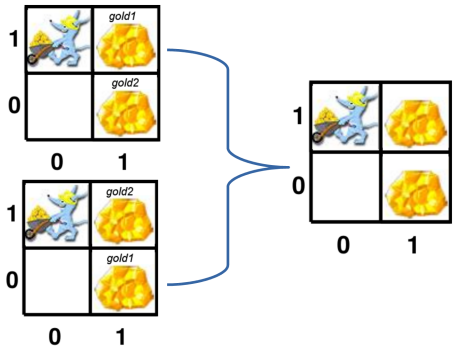
where Q_k^z is the local Q-table of agent z at step k, Q_k is the joint Q-table for all agents, where agent z chose action a_k^z ($\boldsymbol{u^z} = a^z$), and $s \in \tilde{s}_k^z$. Local Q-values are defined for a given abstract state $\tilde{s}_k^z$ and an agent action $a_k^z \in A_z$, while joint Q-values are defined for concrete states s and joint actions $\boldsymbol{u} \in U$, composed of actions of all agents. During the learning process, a single value of the local Q-table can be greater than the joint Q-table values, because another concrete state $s' \in \tilde{s}_k^z$ may have been visited before, a situation in which generalization causes a faster convergence. We prove that Equation (3) holds for MOO-MDPs under certain constraints.

Proposition 1. *Equation (3) holds for every step k, agent $z \in Ag$, state $s \in S$, and action $a^z \in A_z$, in any MOO-MDP where the following assumptions hold:*

1) *The concrete state transition and reward functions are deterministic (i.e., for a given state s_k and joint action $\boldsymbol{u_k}$ only one next state s_{k+1} and reward r_k can be achieved).*
2) *For all $s \in S, \boldsymbol{u} \in U$ and $a^z \in A_z : Q_0(s, \boldsymbol{u}) = Q_0^z(\kappa(s, z), a^z) = 0$, and $r(s, \boldsymbol{u}) \geq 0$.*
3) *The MOO-MDP is cooperative (i.e., all agents receive the same reward r_k at every step k).*
4) *For all $s \in S$, $z \in G$, and $\boldsymbol{u} \in U$, $\kappa(s, z)$ returns only one abstract state $\tilde{s}_k^z = \kappa(s, z)$. Also, the same reward r_k and next state $\tilde{s}_{k+1}^z$ are observed when applying $\boldsymbol{u}$ in any concrete state covered by $\tilde{s}_k^z$.*
5) *All state-action pairs are infinitely visited*[3].

[2]Notice that, as we are here dealing with Cooperative MOO-MDPs, the optimal policy maximizes a single reward function for all agents, rather than reaching an *equilibrium*, as when the agents have different reward functions.

[3]For this, all states must be reachable and all agents apply an exploration strategy with a non-zero probability of choosing each action in each state.



(a)

$s_1 =$	$\{miner = \{0,1\}, gold1 = \{1,1\}, gold2 = \{1,0\}\}$
$s_2 =$	$\{miner = \{0,1\}, gold1 = \{1,0\}, gold2 = \{1,1\}\}$
$\tilde{s} =$	$\{miner = \{0,1\}, gold = \{1,0\}, gold = \{1,1\}\}$

$\tilde{s} = \{s_1, s_2\}$

(b)

Fig. 2. (a) Graphical representation (illustrations adapted from [30]) and (b) textual representation of the state space abstraction. s_1 and s_2 represent two concrete states (ids on top of gold pieces) that are described by a single abstract state $\tilde{s}$ in the right side when $Gold \in \Gamma$.

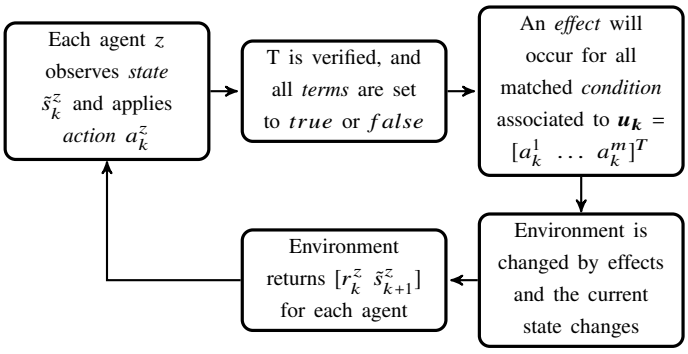


Fig. 3. Transition dynamics in an MOO-MDP.

Proof. For all agents $z \in G$:

- $k = 0$: Equation (3) is ensured by Assumption 2.
- $k \rightsquigarrow k+1$: Assumptions 1 and 3 ensure that the same experience $\langle s_k, \boldsymbol{u_k}, s_{k+1}, r_k \rangle$ is valid for all agents in k, which together with Assumption 4 guarantees that each agent always observes the same $\tilde{s}_k^z$ whenever s_k is visited. Thus, the following Q-value update is performed either in distributed or in joint control, respectively:

$$Q_{k+1}^z(\tilde{s}_k^z, a_k^z) \leftarrow \max\{Q_k^z(\tilde{s}_k^z, a_k^z), r_k + \gamma \max_{a^z \in A_z} Q_k^z(\tilde{s}_{k+1}^z, a^z)\},$$
$$Q_{k+1}(s_k, \boldsymbol{u}_k) \leftarrow \max\{Q_k(s_k, \boldsymbol{u}_k), r_k + \gamma \max_{\boldsymbol{u} \in U} Q_k(s_{k+1}, \boldsymbol{u})\}.$$

For any experience, this update can lead to two possibilities:

1) $Q_k^z(\tilde{s}_k^z, a_k^z) < r_k + \gamma \max_{a^z \in A_z} Q_k^z(\tilde{s}_{k+1}^z, a^z)$: Because Equation (3) holds for k, both $Q_k^z(\tilde{s}_k^z, a_k^z)$ and $Q_k(s, \boldsymbol{u_k})$ are updated, thus after the update: $Q_{k+1}^z(\tilde{s}_k^z, a_k^z) \geq Q_{k+1}(s_k, \boldsymbol{u_k})$.
2) *Otherwise*: No Q-value is updated on the distributed Q-table. As Equation (3) holds for k: $Q_k(s_k, \boldsymbol{u_k}) \leq Q_k^z(\tilde{s}_k^z, a_k^z)$ and $\max_{\boldsymbol{u} \in U, s \in \tilde{s}_{k+1}^z} Q_k(s, \boldsymbol{u}) \leq \max_{a^z \in A_z} Q_k^z(\tilde{s}_{k+1}^z, a^z)$. This means that, after the update in k, the following relation is valid: $Q_{k+1}(s_k, \boldsymbol{u}_k) \leq Q_{k+1}^z(\tilde{s}_k^z, a_k^z)$.

Since in both situations $Q_{k+1}(s_k, \boldsymbol{u}_k)$ and $Q_{k+1}^z(\tilde{s}_k^z, a_k^z)$ are the only Q-table entries that may be updated and the latter is always greater or equal than the former, Equation (3) also holds for $k+1$.

Assumptions 1, 3,4, and 5 also ensure that at convergence time, all trajectories starting from a given concrete state s are already known, resulting in the following equation for any $z \in G$, $s \in S$, and $a^z \in A_z$:

$$Q^{z*}(\kappa(s,z), a^z) = \max_{\boldsymbol{u} \in U, \boldsymbol{u^z} = a^z} Q^*(s, u), \tag{4}$$

where Q^{z*} and Q^* are, respectively, the distributed Q-table of agent z and the joint Q-table at convergence time.

□

However, the greedy policy applied to local Q-tables is not guaranteed to result in an optimal joint policy because some miss-coordination issues may arise depending on how each agent breaks ties in its value functions. This means that agents need an additional coordination method for optimal actuation. Thus, each agent only updates its policy when a new action results in an improvement over all other actions previously applied in the current state. This update-rule solves coordination issues since all agent policies will repeat the first joint action that received the optimal discounted reward and is described by

$$\pi_{k+1}^z(\tilde{s}_k^z) \leftarrow \begin{cases} \pi_k^z(\tilde{s}_k^z) & \text{if } \max_{a^z \in A_z} Q_k^z(\tilde{s}_k^z, a^z) = \max_{a^z \in A_z} Q_{k+1}^z(\tilde{s}_k^z, a^z) \\ a_k^z & \text{otherwise} \end{cases}. \tag{5}$$

As a greedy policy applied to a joint Q-table in cooperative scenarios leads to an optimal actuation, a distributed policy is optimal if it is greedy with respect to the joint Q-table. We can prove that Equation (5) has this property:

Proposition 2. *Let π^z be a decentralized policy learned by agent z on a cooperative MOO-MDP using Equation (5). Assume that Equation (3) holds and Equation (2) is used for Q-value updates. Let $\tilde{s}_k^z = \kappa(s_k, z)$, then for every state $s \in S$, π^z is greedy with respect to the corresponding joint Q-table at convergence time, i.e.*

$$\forall s \in S : [\pi^1(\tilde{s}^1) \ \dots \ \pi^m(\tilde{s}^m)]^T = \arg\max_{\boldsymbol{u} \in U} Q^*(s, \boldsymbol{u}). \tag{6}$$

Proof. For all agents $z \in G$, let π_0^z be arbitrarily initialized. Because Equation (2) is used for Q-value updates, Q_k^z is a monotonically increasing function; that is, $\forall s \in S, a^z \in A_z : Q_k^z(s, a^z) \leq Q_{k+1}^z(s, a^z)$. However, according to Equation (5), the policy is only updated in step k when there exists only

one action for which the Q-value related to the current state was modified:

$$\exists! a^z \in A_z : Q_k^z(\tilde{s}_k^z, a^z) < Q_{k+1}^z(\tilde{s}_k^z, a^z).$$

In this case, we know that: $\pi_{k+1}^z(\tilde{s}_k^z) \leftarrow a_k^z$. As we are dealing with cooperative MOO-MDPs, this holds for all agents in k, corresponding to a joint policy update as follows

$$\pi_{k+1}(s_k) = \begin{bmatrix} \pi_{k+1}^1(\tilde{s}_k^1) \\ \vdots \\ \pi_{k+1}^m(\tilde{s}_k^m) \end{bmatrix} = \begin{bmatrix} \underset{a^1 \in A_1}{arg\ max} Q_{k+1}^1(\tilde{s}_k^1, a^1) \\ \vdots \\ \underset{a^m \in A_m}{arg\ max} Q_{k+1}^m(\tilde{s}_k^m, a^m) \end{bmatrix} \quad (7)$$

When all state-action pairs have been explored, all Q-tables will have converged to Q^*. Hence, Equation (7) leads to:

$$\forall s \in S : \pi^*(s) = \begin{bmatrix} \underset{a^1 \in A_1}{arg\ max} Q^{1*}(\tilde{s}^1, a^1) \\ \vdots \\ \underset{a^m \in A_m}{arg\ max} Q^{m*}(\tilde{s}^m, a^m) \end{bmatrix}. \quad (8)$$

Combining Equations (4) and (8) results in

$$\forall s \in S : \pi^*(s) = \begin{bmatrix} \underset{a^1 \in A_1}{arg\ max} \underset{\boldsymbol{u} \in U, \boldsymbol{u^1} = a^1}{max} Q^*(s, u) \\ \vdots \\ \underset{a^m \in A_m}{arg\ max} \underset{\boldsymbol{u} \in U, \boldsymbol{u^m} = a^m}{max} Q^*(s, u) \end{bmatrix}. \quad (9)$$

Due to the update rule in Equation (5) and the cooperative nature of the MOO-MDP, we can say that agents coordinate by breaking ties in $\underset{a^z \in A_i}{arg\ max} Q_{k+1}^i(\tilde{s}^i, a^z)$ according to the order in which experiences occurred, which means that agents coordinate even when multiple optimal joint policies exist. Thus Equation (9) is equivalent to

$$\forall s \in S : \pi^*(s) = \underset{\boldsymbol{u} \in U}{arg\ max}\ Q^*(s, u). \quad (10)$$

Hence, a joint policy implied by decentralized policies updated as in Equation (5) eventually converges to the optimal joint policy, provided that Proposition 1 holds. □

DOO-Q solves MOO-MDPs allying the distributed Q-table update of Equation (2) with the policy update of Equation (5). DOO-Q is fully described in Algorithm 1. At first, all local Q-tables are initialized with zero values (according to Assumption 2 of Proposition 1). Then, for each decision step, each agent observes its current abstract state $\tilde{s}_k^z$ according to the state of all objects and applies an action a_k^z following an exploration strategy $ExpStr$. Any function that has a non-zero probability of executing all applicable actions can be used as $ExpStr$ (as required by Proposition 2), for example, the ϵ-greedy strategy. The $ExpStr$ arguments are $\tilde{s}_k^z$, to know which actions are applicable, and the current policy π_k^z. After all actions are applied and the state transition is processed, each agent observes the next state and reward. Finally, all local Q-tables and policies π_k^z are updated, ending the current learning step. Notice that the observation of abstract states enables state generalization, in the sense that an agent may see all objects of a class as equivalent, and only differentiate them by attribute values. Note also, that here the environment returns a single reward r_k to all agents.

Algorithm 1 Learning for a DOO-Q agent z

Require: exploration strategy $ExpStr$, discount rate γ, abstraction function κ, state space S, and action space A_z.

1: $Q_0^z(\kappa(s, z), a) \leftarrow 0, \forall s \in S, a \in A_z$.
2: Initiate π_0^z as a greedy policy.
3: Observe current abstract state $\tilde{s}_0^z$.
4: **for** Each learning step $k \geq 0$ **do**
5: Apply action $a_k^z = ExpStr(\tilde{s}_k^z, \pi_k^z)$
6: Observe reward r_k and new state $\tilde{s}_{k+1}^z$.
7: Update $Q_k^z(\tilde{s}_k^z, a_k^z)$ (Equation 2).
8: Update policy $\pi_k^z(\tilde{s}_k^z)$ (Equation 5).
9: $\tilde{s}_k^z \leftarrow \tilde{s}_{k+1}^z$.
10: **end for**

V. Experimental Evaluation

We evaluate DOO-Q in three domains. The first one is designed to represent situations where the object-oriented representation benefits from domain characteristics, while all the assumptions of our theoretical proofs hold. The second one is designed to be a simple domain with small state space, in which the task is easy to solve without abstraction. While the performance of algorithms that reason over concrete states should be maintained, our proposal has better memory requirements in such domains. The third one is a domain with partial observability and a non-deterministic environment in which some of the assumptions of our theoretical proof for learning an optimal policy are violated. This last domain provides experimental evidence of the robustness of our proposal under conditions not covered by our theoretical analysis.

For all domains, unless otherwise stated, the performance of the following algorithms was compared:

- **Single-agent Q-Learning (SAQL)**: We adapt the single-agent Q-Learning [22] to MAS. A central controller is designated to control all agents in the environment. A joint state-action space is used to build the Q-table. Here, the Q-table update is computed as: $Q_{k+1}(s_k, \boldsymbol{u_k}) \leftarrow Q_k(s_k, \boldsymbol{u_k}) + \alpha(r_k + \gamma \max_{\boldsymbol{u} \in U} Q(s_k, \boldsymbol{u}) - Q_k(s_k, \boldsymbol{u_k}))$, where α is a learning rate. The Object-Oriented representation is used to define the state space.
- **Multiagent Q-Learning (MAQL)**: Each miner is an autonomous agent in this algorithm. Agents cannot communicate, but they are able to observe each others actions in all steps. Thus, each agent stores a Q-table that has an entry for all states and *joint* actions, and every agent actuates believing that all other agents will choose the individual action which has the maximum Q-value.
- **Distributed Q-Learning (DQL)**: The standard Distributed Q-Learning [28] is similar to our proposal, but without using the Object-Oriented representation (i.e., it does not allow state abstraction).
- **DOO-Q**: In our proposal, each agent is autonomous and selects a local action based on local abstract states.

For all algorithms, we use the ϵ-greedy exploration strategy with $\epsilon = 0.1$. The experiments of Sections V-A and V-B were implemented and carried out in BURLAP [19] and graphs were printed using MATLAB [31]. The experiment described in Section V-C was implemented in Python[4]. In the following, we describe each of the evaluation domains.

A. Goldmine

A slightly modified version of the *Goldmine* is our first domain. *Goldmine* was firstly described in [30]. This domain was chosen because it has interesting multiagent qualities, since various agents must gather all gold pieces in an environment, and they benefit from cooperation (because all agents receive the same reward when any miner picks up a gold piece). This domain was originally solved through a centralized controller that moves a single miner per decision step. This controller is not directly related to objects in the environment and is rather an external agent that can control all miners, which do not perform autonomous actions. Also, there was no penalization in reward for agent collisions.

In our version of the *Goldmine* domain, at each decision step, all miners may move one position to *North*, *South*, *East* or *West* and, whenever a miner occupies the same cell as a gold piece, the action *GetGold* can be used to collect the gold piece. Episodes end when all gold pieces are collected.

As described in Section II, the *Goldmine* domain is described by three classes: *Miner*, *Gold*, and *Wall*, where *miner* objects are agents, i.e. $C = \{Miner, Gold, Wall\}$, $Ag = \{Miner\}$, and $Att(Miner) = Att(Gold) = \{x, y\}$, $Att(Wall) = \{x, y, pos\}$. In the example state illustrated in Figure 1, $E = \{gold1, \ldots, gold6, wall1, \ldots, wall24\}$, $G = \{miner1, miner2, miner3\}$ and $A_z = \{North(z), South(z), East(z), West(z), GetGold(z)\}$. The following relations are defined: $touch_N(Miner, Wall)$, $touch_S(Miner, Wall)$, $touch_W(Miner, Wall)$, $touch_E(Miner, Wall)$, and $on(Miner, Gold)$, which define whether a wall is on North, South, East or West of a miner cell, or if a miner is occupying the same cell as a gold piece. The actions, conditions and deterministic effects are defined in Table I. Note that, if a miner tries to move towards a wall, the action condition is not fulfilled and the miner does not move from the current position.

For a given triple $\langle s_k, \boldsymbol{u_k}, s_{k+1} \rangle$, we define the reward function as

$$r(s_k, \boldsymbol{u_k}) = gold \times n_{gold} \times \gamma^{(2n_{miner}+1.5n_{wall})}, \qquad (11)$$

[4]Implementations available at *https://github.com/f-leno/DOO-Q_extension*

TABLE I
Goldmine DOMAIN DYNAMICS. IF THE CONDITION FOR THE APPLIED ACTION IS NOT TRUE IN THE CURRENT STATE, NO EFFECT OCCURS.

Action	Condition	Effects
North($Miner\ m$)	$\neg touch_N(m, Wall)$	$m.y \leftarrow m.y + 1$
South($Miner\ m$)	$\neg touch_S(m, Wall)$	$m.y \leftarrow m.y - 1$
East($Miner\ m$)	$\neg touch_E(m, Wall)$	$m.x \leftarrow m.x + 1$
West($Miner\ m$)	$\neg touch_W(m, Wall)$	$m.x \leftarrow m.x - 1$
GetGold($Miner\ m$)	$on(Miner\ m, Gold\ g)$	$g.x \leftarrow \emptyset, g.y \leftarrow \emptyset$

where $gold$ is the value of each gold piece collection, γ is the discount rate, n_{gold} is the number of collected gold pieces as a result of applying the joint action $\boldsymbol{u_k}$, n_{wall} is the number of miners colliding with wall in k, and n_{miner} is the number of miner pairs occupying the same grid cell in s_{k+1}, and $gold = +100$. This reward function was designed to penalize collisions while avoiding negative rewards[5], which would invalidate Assumption 2 of Proposition 1.

In our experiments, we randomly generated 70 initial states in a 5×5 grid with 3 miners and 6 gold pieces (Figure 1 is an example of such states) and used them to compare the performance achieved by each of the algorithms. The experiment was designed in a way that every algorithm experiences the same initial states in the same order, and the next initial state is defined by swapping the position of objects of the same class after each episode. For each of the states, algorithms explore using an exploration strategy and, after every interval of 100 episodes, a single episode is assessed using the greedy policy to extract the number of steps required to reach a terminal state and the received accumulated discounted reward. The algorithms were configured as follows:

1) **SAQL**: This algorithm was used in the original *Goldmine* modeling, where an external agent sees each miner as a simple environment object (and not as an autonomous agent). A single miner can be moved at each step and all decisions are made by the external agent, which means miners do not perform actions by themselves. The Q-Learning algorithm was used to solve the task, with the parameters $\alpha = 0.2$, $\gamma = 0.9$. Here, $\Gamma = \{Gold, Wall\}$. We used the default implementation available in BURLAP.
2) **MAQL**: The following parameters were set: $\alpha = 0.2$, $\gamma = 0.9$, and $\Gamma = \{Miner, Gold, Wall\}$. The BURLAP default implementation was used.
3) **DQL**: This algorithm is implemented with a factored state description and $\gamma = 0.9$.
4) **DOO-Q**: For our proposal, $\gamma = 0.9$ and $\Gamma = \{Miner, Gold, Wall\}$.

A time limit was set for the experiment, in which an algorithm is interrupted if it takes too long to conclude a predefined number of learning episodes. In this case, the results achieved so far were still stored.

The object-oriented representation is expected to excel in *Goldmine*, as gold pieces, walls, and other agents can be abstracted, greatly reducing the state space. Hence, we also included an evaluation in a simple *Gridworld* in which the object-oriented representation is not expected to achieve better results than a factored representation, and is described in the next Section.

B. Gridworld

In our *Gridworld* domain, three agents must navigate in a shared environment aiming to reach the desired position. Figure 4 illustrates the initial state for our experiments. Each agent (circle) has a different destination (square), in which

[5]The weights for collisions were set to prioritize avoiding miner collisions, but small changes in those weights do not result in big changes when learning.

they want to be while avoiding collisions with other agents or walls. Episodes always begin in the aforementioned initial state and end when all agents reach their destination. Agents which achieved their goal cannot move anymore and must wait until all other agents arrive in their final location.

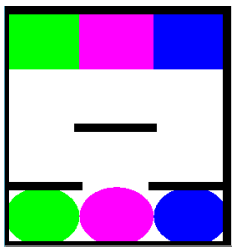


Fig. 4. Graphical representation of the Gridworld domain. Three agents (circles in the image) need to reach their destinations (squares with the same color as the agents) while avoiding collisions with other agents and walls. Thick walls are impassable.

The domain is described by the classes $C = \{Agent, Goal, Wall\}$, which have the attributes $Att(Agent) = \{x, y, agentID\}$, $Att(Goal) = \{x, y, goalID\}$, and $Att(Wall) = \{x, y, pos\}$. Agents can move in the four cardinal directions or do not move, i.e. $A_z = \{North(z), South(z), East(z), West(z), NoOp(z)\}$, where $NoOp$ means that the agent does not move and stays in the same position. An agent can execute any action until reaching its destination (i.e., a *Goal* object g and an *Agent* object z with $g.goalID = z.agentID$), after which only the $NoOp$ action is available until the episode ends.

When agents collide (more than one agent tries to enter the same cell), one random agent gets to the position and the other does not move. The reward function returns +1 to all agents when any agent arrives at its destination and 0 for all other time steps. Hence, agents benefit from coordination, since helping other agent results in positive rewards for every agent in the system. For all algorithms, $\alpha = 0.5$ and $\gamma = 0.9$. Only the Multiagent approaches were evaluated in this domain (MAQL, DQL, and DOO-Q). For both DOO-Q and MAQL, we chose $\Gamma = \{Agent, Goal, Wall\}$. The discounted cumulative reward achieved through exploiting the current learned policy was evaluated after every two episodes of learning, until 800 learning episodes were completed. The whole experiment was repeated 50 times to achieve statistical significance.

The object-oriented representation is not expected to learn faster in this domain, as the state space is very small, which renders the state abstraction unnecessary. Our last evaluation domain is described in the next Section.

C. Predator-Prey

Our last evaluation is performed in a *Predator-Prey* domain [23]. All the predators in the environment collaboratively aim at capturing randomly-moving preys in a 10×10 grid, as depicted in Figure 5a. At each step, all preys and predators can apply one of four actions $A_z = \{North(z), South(z), East(z), West(z)\}$. Predators' actions are defined by autonomous agents controlled by the aforementioned algorithms, while a random action is chosen for preys. Predators can freely move in the grid, and in case of wall collisions, the agent position is not modified. We generated 100 evaluation states where three predators and two preys were placed in random initial positions. An episode ends when all preys are captured (one predator is in the same position as the prey). When a prey is captured, a reward of +1 is given to all agents, while a reward of 0 is given otherwise. We train all algorithms for 1000 learning episodes, in which the performance is evaluated by trying to solve all evaluation episodes after every 5 learning episodes.

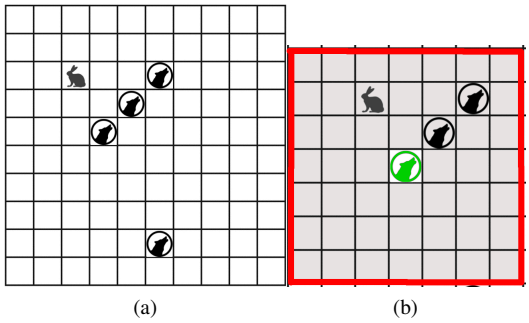


Fig. 5. An illustration of the Predator-Prey task. A group of predators aims at capturing a randomly-moving prey in a 10×10 grid. An episode ends when all preys in the environment are captured. (a) An example of a complete state. (b) The visual field of one of the predators (the reasoning predator is in green).

In this domain, the predators cannot observe the whole grid, and their visual field is limited by a parameterized visual *depth*. Figure 5b illustrates the point of view of one agent configured with $depth = 3$. The agent can observe the relative position of predators and preys inside its visual field. For example, the agent state depicted in Figure 5b is observed as: $\{Prey: (-1,2), Predator: (1,1), Predator: (2,2)\}$. The object-oriented representation is defined as: $C = \{Prey, Predator\}$ and $Att(Prey) = Att(Predator) = \{x, y\}$. Agents may occupy the same cell without penalization. Notice that, because of the random movements executed by the prey, the state transition function is non-deterministic, which means that Assumption 1 and 4 of Proposition 1 do not hold for this domain. However, although the convergence to an optimal policy is not guaranteed, we provide empirical evidence that our proposal learns how to solve the task. Here, *SAQL* can move all agents at each time step, which means that each Q-table entry contains the observations of all agents and a *joint* action. Furthermore, we chose $\Gamma = \{Prey, Predator\}$, $\gamma = 0.9$ and $\alpha = 0.1$ for all algorithms. We show the results of a task with 3 predators configured with $depth = 3$ trying to catch 2 preys.

VI. Results and Discussion

The algorithms are compared based on Q-table size and learning speed. The number of Q-table entries for an algorithm depends on the size of the state and action spaces, $|Q| = |S| \times |A|$.

However, some of the states might be very rarely observed, which means that, in practice, the agent does not need to allocate all possible Q-table entries in memory. Therefore, for all domains, we present both a theoretical definition of the maximum number of Q-table entries that may be necessary for

each algorithm and the number of actually used Q-table entries during the experiments. We compare performances through the cumulative reward achieved in the evaluation episodes for both the *Goldmine* and *Gridworld* domains and through the average number of steps to solve evaluation episodes for the *Predator-Prey* domain.

In the next sections, we present the achieved results.

A. Goldmine

We first present the number of Q-table entries for each algorithm. Let m be the number of miners, p be the number of gold pieces, q be the number of individual actions affecting a single miner state, and w be the number of possible cells inside the grid. In order to simplify calculations, we assume that Q-table entries are created even if actions are not applicable in a given state.

- **SAQL**: One agent controls all miners, but only moves one single miner per step, so we get the size of the action space $|A| = q\ m$. The state space is defined over all possible grid cells that each miner and each gold piece can occupy (gold pieces can also be in *collected* state). As *Gold* is an abstracted class, the number of ways that gold pieces can be dispersed in the grid is calculated as a permutation with repetitions, leading to $|S| = w^m \frac{(p+w)!}{p!w!}$. The memory requirement is then $O\left(w^m \frac{(p+w)!}{p!w!} q\ m\right)$.
- **MAQL**: There is one agent for each miner, which move simultaneously in every step, thus all possible combinations of joint actions determine the size of the action space $|A| = q^m$. As *Miner* and *Gold* are abstracted classes, only the agent is distinguishable by the id, resulting in $|S| = w \frac{(m+w-2)!}{(m-1)!(w-1)!} \frac{(p+w)!}{p!w!}$. The memory requirement for this algorithm is $O\left(w \frac{(m+w-2)!}{(m-1)!(w-1)!} \frac{(p+w)!}{p!w!} q^m\right)$.
- **DQL**: Here, each agent only considers its actions leading to $|A| = q$. However, no abstraction is used, leading to $|S| = w^m(w+1)^p$. Thus, the memory requirement for DQL is $O\left(w^m(w+1)^p q\right)$.
- **DOO-Q**: The state space is the same as in MAQL and the action space is the same as in DQL. Thus, the memory requirement for DOO-Q is $O\left(w \frac{(m+w-2)!}{(m-1)!(w-1)!} \frac{(p+w)!}{p!w!} q\right)$.

For example, in a 5×5 environment with three miners, six gold pieces and fixed walls (as in our experiment), the number of Q-table entries per agent for each algorithm is roughly (i) **SAQL**: 1.7×10^{11}, (ii) **MAQL**: 7.4×10^{11}, (iii) **DQL** 2.4×10^{13}, (iv) **DOO-Q** 2.9×10^{10}.

Figure 6 depicts the results of the *Goldmine* experiment described in Section V in terms of discounted cumulative reward, and Figure 7 shows the number of steps taken until a terminal state is reached. Figure 6 shows that DOO-Q learns an effective policy much faster and achieves higher rewards than all other algorithms since the beginning, maintaining better results until the end of the experiment. MAQL started with a performance comparable to SAQL, however, the high memory usage made MAQL slower to process and the time limit was exceeded after only 700 learning episodes. When compared to the object-oriented algorithms, DQL presented a very slow learning process until the end of training. As

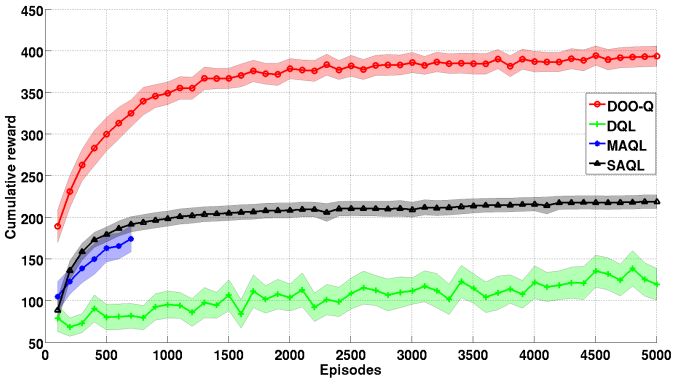


Fig. 6. Observed discounted cumulative reward in the *Goldmine* domain. The horizontal axis is the number of executed episodes using the ϵ-*greedy* policy. The vertical axis represents the metric evaluated every 100 episodes of exploration. The shaded area represents the 95% confidence interval observed in 70 repetitions.

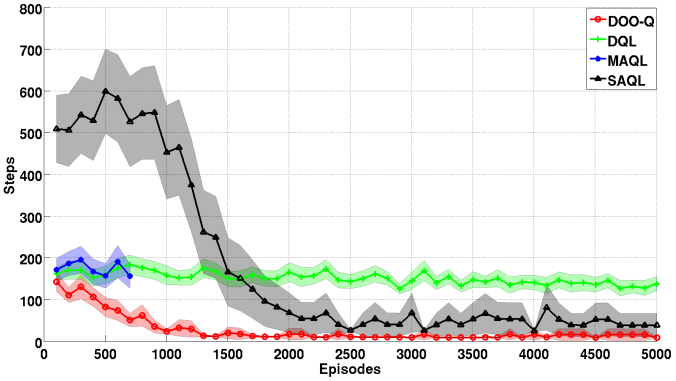


Fig. 7. Observed number of steps to complete one episode in the *Goldmine* domain. The horizontal axis is the number of executed episodes using the ϵ-*greedy* policy. The vertical axis represents the metric evaluated every 100 episodes of exploration. The shaded area represents the 95% confidence interval observed in 70 repetitions.

the only difference between DOO-Q and DQL is the object-oriented representation, the results clearly reflect the advantage of MOO-MDPs in environments similar to *Goldmine*. Figure 7 shows that the introduction of multiple agents simultaneously exploring the environment greatly improved the number of steps to complete the task. MAS approaches (DOO-Q, MAQL, and DQL) completed the task with fewer steps in the beginning of the training, and DOO-Q was never worse than SAQL during the whole training. DOO-Q learned how to complete the task with very few steps after 1300 learning episodes and SAQL surpassed DQL after around 2000 episodes because DQL presented a very slow learning. MAQL would probably present better results than SAQL in steps for task completion but it was unable to scale to this problem size because of computational limitations. Figure 8 gives better insights on why *MAQL* was unable to scale. Although the maximum number of Q-table entries is higher for *DQL*, observing new state-action pairs is much more frequent when $|A|$ is big, hence *MAQL* uses a high number of entries since the beginning. *DQL* has a less effective exploration, which makes the Q-table increase in size in a slower pace, but as a consequence,

the performance also rises very slowly. *DOO-Q* in its turn uses less memory than the other algorithms, as predicted by the theoretical analysis. After 5000 training episodes, *DOO-Q* uses roughly 5.0×10^5 entries, while *SAQL* and *DQL* use 1.8×10^6 and 3.3×10^6 respectively.

The results in this domain show that, when applicable, abstraction greatly accelerates the learning speed, as DOO-Q achieved much better results than DQL. Also, compared to SAQL, MAS algorithms were able to learn how to improve performance faster, which indicates that dividing the workload helps to solve some problems.

Thus, DOO-Q achieved the best performance of the evaluated algorithms by using the least space for the Q-table and by learning a good policy for a higher discounted cumulative reward much faster in the *Goldmine* domain.

B. Gridworld

Let m be the number of agents, w be the number of possible positions, and q be the number of actions. For a *Gridworld* with fixed goals and walls, the following memory requirements are demanded by each algorithm in the worst case:

- **MAQL**: A Q-table entry is created for all possible combinations of joint actions, hence the size of the action space is $|A| = q^m$. As *Agent* is an abstracted class, and only one agent can be at a given position at a time step, the state space size is $|S| = w\frac{(w-1)!}{(w-m)!(m-1)!}$. The memory requirement for this algorithm is $O\left(w\frac{(w-1)!}{(w-m)!(m-1)!}q^m\right)$.
- **DQL**: Each agent only considers its actions leading to $|A| = q$. However, agents are not abstracted, leading to $|S| = \frac{w!}{(w-m)!}$. Thus, the memory requirement for DQL is $O\left(\frac{w!}{(w-m)!}q\right)$.
- **DOO-Q**: The state space is the same as in MAQL, however each agent only considers its actions leading to $|A| = q$. In this case, the memory requirement for DOO-Q is $O\left(w\frac{(w-1)!}{(w-m)!(m-1)!}q\right)$.

For example, in a 4×3 grid with three agents, the number of Q-table entries per algorithm is roughly (i) **MAQL**: 8.25×10^4, (ii) **DQL** 6.6×10^3, (iii) **DOO-Q** 3.3×10^3.

Figure 9 shows the difference between the achieved discounted cumulative rewards for each algorithm. At first, all algorithms improve their policy at roughly the same speed. However, after approximately 25 episodes *MAQL* gets stuck in a suboptimal actuation, while *DOO-Q* and *DQL* reach a better performance faster. Finally, after 800 episodes, all algorithms have the same performance (the difference is not statistically significant). MAQL takes longer to improve its policy after episode 25 because of its large Q-table size, which renders the exploration less effective and slower to converge.

Figure 10 shows that the actual number of used Q-table entries is roughly the same for *DOO-Q* and *DQL*, while *MAQL* has a much higher memory requirement since the beginning of training. The average number of entries after 800 episodes was roughly 1600 for *DOO-Q* and *DQL* and 3.2×10^4 for *MAQL*. As expected, the difference between *DOO-Q* and *DQL* in terms of accumulated reward is not statistically significant. While the memory requirements for *DOO-Q* were the same as *DQL* in our experiments, in theory, *DOO-Q* may have lower memory requirements for different settings of this domain.

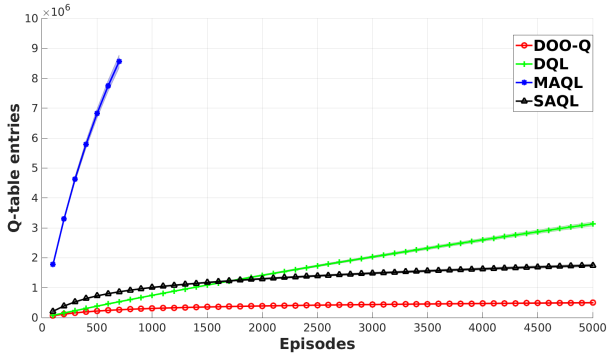


Fig. 8. The average observed number of used Q-table entries during the experiments in the *Goldmine* domain.

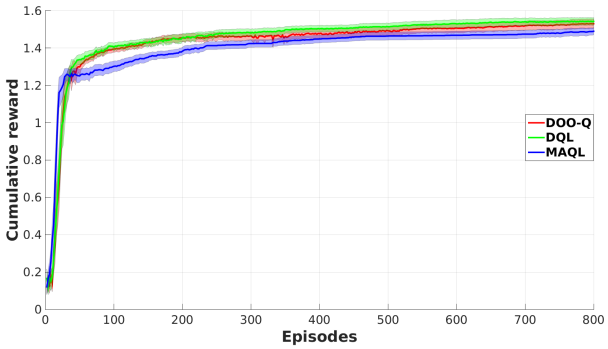


Fig. 9. Observed discounted cumulative reward in the *Gridworld* domain. The horizontal axis is the number of executed episodes using the ϵ-*greedy* policy. The vertical axis represents the distributed accumulated reward evaluated every two episodes of exploration. The shaded area represents the 95% confidence interval observed in 50 repetitions.

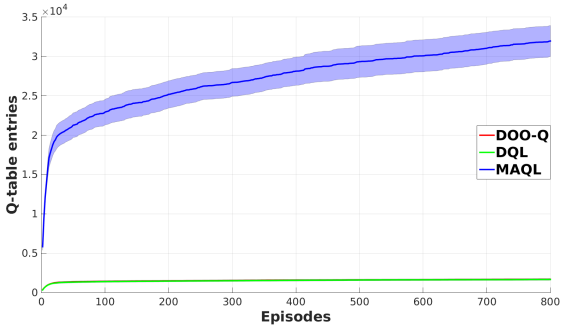


Fig. 10. The average observed number of used Q-table entries during the experiments in the *Gridworld* domain.

C. Predator-Prey

Let p be the number of preys in the environment, a be the number of predators, q be the number of possible actions, and $w = (2\ depth + 1)^2$ be the number of possible positions that the agent can observe (see Figure 5b). The maximum number of Q-table entries for each algorithm is calculated as follows:

- **SAQL**: Here the controller processes all the observation at the same time, thus, the state space is defined by all possible positions of the objects in the environment (preys and predators) that can be observed by each agent: $|S| = \frac{(a+w)!}{a!w!}\frac{(p+w)!}{p!w!}a$. As $|A| = q^a$ for this domain, the memory requirement for *SAQL* is $O\left(\frac{(a+w)!}{a!w!}\frac{(p+w)!}{p!w!}a\ q^a\right)$.
- **MAQL**: The calculation here is similar, but each agent processes its own observations, hence $|S| = \frac{(a-1+w)!}{(a-1)!w!}\frac{(p+w)!}{p!w!}$ and the memory requirement is $O\left(\frac{(a-1+w)!}{(a-1)!w!}\frac{(p+w)!}{p!w!}q^a\right)$.
- **DQL**: Here, each agent only considers its actions leading to $|A| = q$. However, without the object-oriented representation the state space is defined as $|S| = (w+1)^{p+a-1}$. Therefore, the memory requirement is $O\left((w+1)^{p+a-1}q\right)$.
- **DOO-Q**: For our proposal, the state space is the same as in MAQL and the action space is the same as in DQL. Hence, the memory requirement is $O\left(\frac{(a-1+w)!}{(a-1)!w!}\frac{(p+w)!}{p!w!}q\right)$.

In a task with $depth = 3$, $p = 2$, and $a = 3$, the number of Q-table entries per agent is roughly (i) **SAQL**: 9.3×10^7, (ii) **MAQL**: 1.1×10^8, (iii) **DQL** 2.7×10^7, (iv) **DOO-Q** 7.0×10^6.

Figure 11 shows the achieved performance for all algorithms. After 200 learning episodes *DOO-Q* solves the task with 80 steps on average, while *DQL*, *MAQL*, and *SAQL* solve the same task in 90, 92, and 85 steps, which means that *DOO-Q* learns how to solve the task more efficiently than the other algorithms. For the rest of the training process, the performance achieved by *DOO-Q* is still better than the other algorithms. While *SAQL* has a good performance at the beginning of training, after 200 episodes it improves its performance very slowly, which makes *DQL* become faster after roughly 600 learning episodes. *MAQL* is slower than all other algorithms since the start, and all algorithms are improving their policies only very slowly after 1500 learning steps. *DOO-Q* is significantly better than all other algorithms according to the Wilcoxon signed rank test with 99% of confidence since 200 learning episodes.

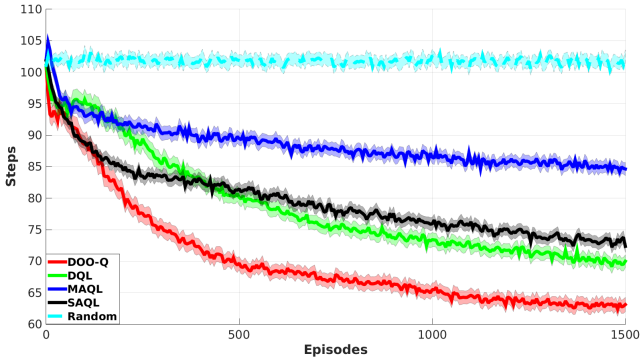


Fig. 11. Observed average number of steps to capture all preys in evaluation episodes. The horizontal axis is the number of executed episodes using the ϵ-*greedy* policy. The shaded area represents the 99% confidence interval observed in 250 repetitions. The performance achieved by a random agent is included as a baseline.

In addition to presenting a better performance, *DOO-Q* (together with *DQL*) uses much less memory than the other algorithms, as shown in Figure 12. While *DOO-Q* and *DQL*

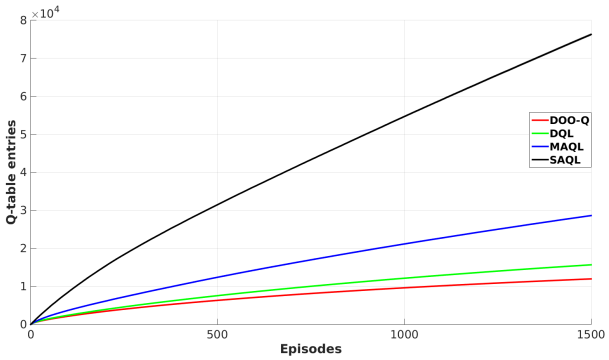


Fig. 12. Observed Q-table size in the *Predator-Prey* domain. The horizontal axis is the number of executed episodes using the ϵ-*greedy* policy. The vertical axis represents the average Q-table size at that step.

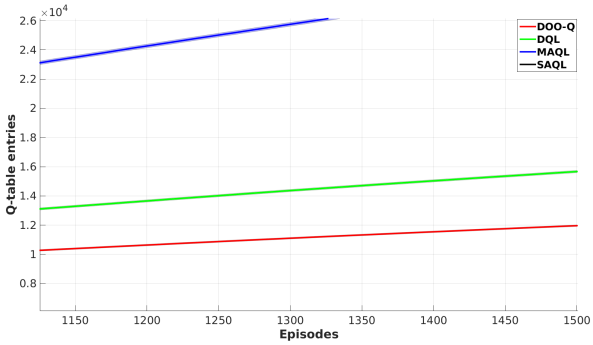


Fig. 13. Depicting differences between *DQL* and *DOO-Q* in Figure 12.

used less than 2.0×10^4 Q-table entries, *SAQL* and *MAQL* used roughly, respectively, 7.6×10^4 and 2.9×10^4 entries. The difference of memory usage between *DOO-Q* and *DQL* can be better visualized in Figure 13. While *DQL* used on average 1.6×10^4 Q-table entries, *DOO-Q* used 1.2×10^4.

In summary, our experiments show that DOO-Q achieves the best performance among the evaluated algorithms. While in the most favorable cases DOO-Q learned *faster* than the other algorithms with *fewer* memory requirements, in the most unfavorable case the performance was equivalent to the best algorithm (DQL) whereas the advantage of the reduced memory requirements remained steady.

VII. Conclusion and Further Works

We here introduced a Multiagent Object-Oriented MDP (MOO-MDP) formalism and presented a model-free algorithm to solve deterministic distributed MOO-MDPs, called Distributed Object-Oriented Q-Learning (DOO-Q). We also proved that DOO-Q learns an optimal policy while abstracting states and storing only local actions in each local Q-table (Proposition 2), and experimentally compared our proposal in three domains with other model-free algorithms. In the *Goldmine* domain, in which object-oriented approaches are expected to perform better, DOO-Q achieved a better performance both in learning speed and memory requirements. In a simple *Gridworld* domain, in which the domain allowed little state abstraction, DOO-Q achieved a learning performance equivalent to Distributed Q-Learning, while demanding less

memory. We also evaluated DOO-Q in partially observable domains with non-deterministic reward and state transition functions, in which the convergence proof does not hold.

Further works will focus on developing algorithms for MOO-MDPs for which DOO-Q is not applicable, such as general-sum games and continuous domains (such as Robot Soccer Simulations [32]). These algorithms can also explore exploration strategies that cannot be applied with DOO-Q, such as optimistic exploration [7]. MOO-MDPs could also be extended to model Partially Observable domains [33], [34], which would allow developing distributed approaches where agents reason over observations in a more robust way than taking the current observation as a state. The use of abstract policies, which achieved promising results in single-agent RMDP approaches [6] still needs to be investigated in MOO-MDPs. For complex and large problems, besides using generalization and distributed computation, state space approximation can also be explored [35]. Furthermore, the Object-Oriented representation provides generalization opportunities that could be exploited for Transfer Learning [36] proposals. Comparing the Object-Oriented description of tasks could be a promising way to compute similarity metrics for Transfer Learning frameworks as the described in [37].

ACKNOWLEDGMENT

We gratefully acknowledge financial support from São Paulo Research Foundation (FAPESP), grants 2015/16310-4 and 2016/21047-3, CAPES, and CNPq, grants 311608/2014-0 and 425860/2016-7.

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[2] G. Tesauro, "Temporal Difference Learning and TD-Gammon," *Commun. ACM*, vol. 38, no. 3, pp. 58–68, Mar. 1995.

[3] L. Busoniu, B. De Schutter, and R. Babuska, "Decentralized Reinforcement Learning Control of a Robotic Manipulator," in *Proc. 9th Int. Conf. Control, Automation, Robotics and Vision (ICARCV)*, 2006, pp. 1–6.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. Bellemare *et al.*, "Human-level Control through Deep Reinforcement Learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[5] S. Dĉeroski, L. De Raedt, and K. Driessens, "Relational Reinforcement Learning," *Machine Learning*, vol. 43, no. 1-2, pp. 7–52, 2001.

[6] M. L. Koga, V. F. Silva, and A. H. R. Costa, "Stochastic Abstract Policies: Generalizing Knowledge to Improve Reinforcement Learning," *IEEE Trans. Cybern.*, vol. 45, no. 1, pp. 77–88, 2015.

[7] C. Diuk, A. Cohen, and M. L. Littman, "An Object-oriented Representation for Efficient Reinforcement Learning," in *Proc. 26th Int. Conf. Machine Learning (ICML)*, 2008, pp. 240–247.

[8] A. Braylan and R. Miikkulainen, "Object-Model Transfer in the General Video Game Domain," in *Proc. 12th AAAI Conf. Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2016. [Online]. Available: http://nn.cs.utexas.edu/?braylan:aiide16

[9] S. Mohan and J. E. Laird, "An Object-Oriented Approach to Reinforcement Learning in an Action Game," in *Proc. 7th AAAI Conf. Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2011, pp. 164–169.

[10] N. Topin, N. Haltmeyer, S. Squire, J. Winder, M. desJardins, and J. MacGlashan, "Portable Option Discovery for Automated Learning Transfer in Object-Oriented Markov Decision Processes," in *Proc. 24th Int. Joint Conf. Artificial Intelligence (IJCAI)*, 2015, pp. 3856–3864.

[11] T. J. Walsh, I. Szita, C. Diuk, and M. L. Littman, "Exploring Compact Reinforcement-learning Representations with Linear Regression," in *Proc. 25th Conf. Uncertainty in Artificial Intelligence (UAI)*, Montreal, Canada, 2009, pp. 591–598.

[12] F. L. Silva and A. H. R. Costa, "Towards Zero-Shot Autonomous Inter-Task Mapping through Object-Oriented Task Description," in *Presented at the Transfer in Reinforcement Learning Workshop (TiRL)*, 2017.

[13] M. J. Wooldridge, *An Introduction to MultiAgent Systems (2. ed.)*, 2009.

[14] A. L. C. Bazzan, "Beyond Reinforcement Learning and Local View in Multiagent Systems," *Künstliche Intelligenz*, vol. 28, no. 3, pp. 179–189, 2014.

[15] K. Tuyls and G. Weiss, "Multiagent Learning: Basics, Challenges, and Prospects," *AI Magazine*, vol. 33, no. 3, p. 41, 2012.

[16] L. Busoniu, R. Babuska, and B. De Schutter, "A Comprehensive Survey of Multiagent Reinforcement Learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 38, no. 2, pp. 156–172, 2008.

[17] T. Croonenborghs, K. Tuyls, J. Ramon, and M. Bruynooghe, "Multi-agent Relational Reinforcement Learning," in *Learning and Adaption in Multi-Agent Systems*, 2005, pp. 192–206.

[18] S. Proper and P. Tadepalli, "Multiagent Transfer Learning via Assignment-Based Decomposition," in *Proc. 8th Int. Conf. Machine Learning and Applications (ICMLA)*, 2009, pp. 345–350.

[19] J. MacGlashan. (2015) Brown-UMBC Reinforcement Learning and Planning (BURLAP). [Online]. Available: http://burlap.cs.brown.edu/index.html

[20] F. L. Silva, R. Glatt, and A. H. R. Costa, "Object-Oriented Reinforcement Learning in Cooperative Multiagent Domains," in *Proc. 5th Brazilian Conf. Intelligent Systems (BRACIS)*, 2016, pp. 19–24.

[21] M. L. Puterman, *Markov Decision Processes : Discrete Stochastic Dynamic Programming*. Hoboken (N. J.): J. Wiley & Sons, 2005.

[22] C. J. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.

[23] M. Tan, "Multi-agent Reinforcement Learning: Independent vs. Cooperative Agents," in *Proc. 10th Int. Conf. Machine Learning (ICML)*, 1993, pp. 330–337.

[24] M. Bowling and M. Veloso, "An Analysis of Stochastic Game Theory for Multiagent Reinforcement Learning," Computer Science Department, Carnegie Mellon University, Tech. Rep., 2000.

[25] M. L. Littman, "Markov Games as a Framework for Multi-Agent Reinforcement Learning," in *Proc. 11th Int. Conf. Machine Learning (ICML)*, 1994, pp. 157–163.

[26] J. Hu and M. P. Wellman, "Nash Q-learning for General-Sum Stochastic Games," *The Journal of Machine Learning Research*, vol. 4, pp. 1039–1069, 2003.

[27] Y. Hu, Y. Gao, and B. An, "Multiagent Reinforcement Learning with Unshared Value Functions," *IEEE Trans. Cybern.*, vol. 45, no. 4, pp. 647–662, 2015.

[28] M. Lauer and M. Riedmiller, "An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems," in *Proc. 17th Int. Conf. Machine Learning (ICML)*, 2000, pp. 535–542.

[29] L. Panait and S. Luke, "Cooperative Multi-Agent Learning: The State of the Art," *Autonomous Agents and Multi-Agent Systems*, vol. 11, no. 3, pp. 387–434, 2005.

[30] C. Diuk, "An Object-Oriented Representation for Efficient Reinforcement Learning," Ph.D. dissertation, Rutgers University, 2009.

[31] MATLAB, *version 8.5.0 (R2015a)*. The MathWorks Inc., 2015.

[32] F. L. Silva, R. Glatt, and A. H. R. Costa, "Simultaneously Learning and Advising in Multiagent Reinforcement Learning," in *Proc. 16th Int. Conf. Autonomous Agents and Multiagent Systems (AAMAS)*, 2017, pp. 1100–1108.

[33] F. S. Melo and M. Veloso, "Decentralized MDPs with Sparse Interactions," *Artificial Intelligence*, vol. 175, no. 11, pp. 1757 – 1789, 2011.

[34] G. Shani, J. Pineau, and R. Kaplow, "A Survey of Point-Based POMDP Solvers," *Autonomous Agents and Multi-Agent Systems*, vol. 27, no. 1, pp. 1–51, 2012.

[35] M. Geist and O. Pietquin, "Algorithmic Survey of Parametric Value Function Approximation," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 24, no. 6, pp. 845–867, 2013.

[36] M. E. Taylor and P. Stone, "Transfer Learning for Reinforcement Learning Domains: A Survey," *Journal of Machine Learning Research*, vol. 10, pp. 1633–1685, 2009.

[37] F. L. Silva and A. H. R. Costa, "Accelerating Multiagent Reinforcement Learning through Transfer Learning," in *Proc. 31st AAAI Conf. Artificial Intelligence*, 2017, pp. 5034–5035.