Object-Oriented Reinforcement Learning in Cooperative Multiagent Domains

Felipe Leno da Silva, Ruben Glatt, and Anna Helena Reali Costa*
Escola Politécnica da Universidade de São Paulo, Brazil
{f.leno,ruben.glatt,anna.reali}@usp.br

***Abstract*—Although Reinforcement Learning methods have successfully been applied to increasingly large problems, scalability remains a central issue. While Object-Oriented Markov Decision Processes (OO-MDP) are used to exploit regularities in a domain, Multiagent System (MAS) methods are used to divide workload amongst multiple agents. In this work we propose a novel combination of OO-MDP and MAS, called Multiagent Object-Oriented Markov Decision Process (MOO-MDP), so as to accrue the benefits of both strategies and be able to better address scalability issues. We present an algorithm to solve deterministic cooperative MOO-MDPs, and prove that it learns optimal policies while reducing the learning space by exploiting state abstractions. We experimentally compare our results with earlier approaches and show advantages with regard to discounted cumulative reward, number of steps to fulfill the task, and Q-table size.**

I. Introduction

Reinforcement Learning (RL) [1] is an extensively used technique for autonomous agents with the ability to learn through experimentation. RL successes range from autonomous flight control [2] to video game playing [3]. As domains become increasingly complex, scalability is gaining more importance for RL methods.

The scalability of RL can be enhanced by relational techniques such as *Relational MDP* (RMDP) [4] and *Object-Oriented MDP* (OO-MDP) [5]. The former model relies on predicates handcrafted by the designer so as to simplify learning, while the latter defines its state space over a set of objects within the environment and its attribute values. OO-MDP is based on observable object features (like position, for instance) that can be specified naturally and intuitively, requiring less domain knowledge than RMDP's propositional functions [5]. OO-MDP has drawn significant attention recently [6]–[8].

Another approach to cope with large state-action spaces is to treat RL domains as *Multiagent Systems* (MAS), which solve tasks in a *divide and conquer* manner [9]. *Multiagent Reinforcement Learning* (MARL) solves MAS; however, RL techniques are not easily-portable to MAS, because new challenges arise from the parallel actuation of several agents: The environment becomes non-stationary as the state transition is now dependent on the joint action of all agents, rather than a single action. Many MARL algorithms assume a centralized controller that designates actions for all agents [10]. Such a controller is infeasible for most domains, and a distributed approach is usually more desirable [11]. Note that MARL also benefits from techniques that exploit regularities in the domain, as scalability is again a central issue [10].

While relational approaches have benefited multiagent domains in previous work [12], OO-MDP has not been applied in MARL so far. To the best of our knowledge, up to now the only effort toward extending OO-MDP to MAS appears in BURLAP [13], a library for the development of planning and learning algorithms based on OO-MDP. However, there is no formal OO-MDP framework for MAS nor distributed solutions based on OO-MDP for a generic number of agents.

Inspired by the insight that each agent in a MAS can be seen as an object, we extend OO-MDP for MAS, defining the *Multiagent Object-Oriented MDP* (MOO-MDP). We also present a *model-free* algorithm based on Distributed Q-Learning [14], hereafter called *Distributed Object-Oriented Q-Learning* (DOO-Q), that can solve deterministic distributed MOO-MDPs in cooperative domains in which all agents try to maximize the same reward function [15]. DOO-Q reasons over abstract states, which help to accelerate learning. We also show that, under certain constraints, DOO-Q does not need to consider all the concrete state space and still learns optimal policies. Thus, in summary, this work answers the following questions: (i) How to describe a MARL domain with multiple autonomous agents in an object-oriented manner, and (ii) How to learn, in a distributed manner, an optimal *joint* policy in deterministic cooperative MOO-MDPs.

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. MDPs and their extensions

An MDP is described by the tuple $\langle S, A, T, R\rangle$, where S is the set of environment states, A is the set of actions available to an agent, T is the transition function, and R is the reward function. At each decision step, the agent observes the state s and chooses an action a (among the applicable ones in s). Then the next state is defined by T. The agent must learn a policy π that maps the best action for each possible state. The solution of an MDP is an optimal policy π^*, a function that chooses

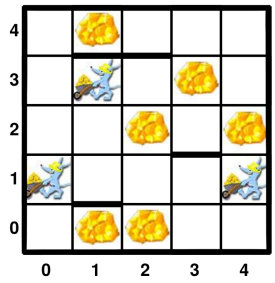
*We are grateful for the support from CAPES, CNPq (grant 311608/2014-0), and São Paulo Research Foundation (FAPESP), grant 2015/16310-4.

an action maximizing future rewards at every state. In this work we are interested in learning problems (i.e., T and R are unknown to the agent) that can be solved through interactions with the environment using the Q-Learning algorithm. Q-Learning iteratively learns a Q-table, i.e., a function that maps every combination of state and action to an estimate of the long-term reward starting from the current state-action pair: $Q : S \times A \rightarrow \mathbb{R}$. Q-Learning eventually converges to the true Q function: $Q^*(s,a) = E\left[\sum_{i=0}^{\infty} \gamma^i r_i\right]$, where γ is the discount rate and r_i is the reward received after i steps from using action a on state s. Q^* can be used to define an optimal policy as: $\pi^*(s) = \arg\max_a Q^*(s,a)$. The standard MDP takes into account only one agent; even though an MDP can be used to model a MAS problem by ignoring all other agents, some kind of coordination is needed in most domains [11].

A Multiagent MDP (MAMDP) [14], also related to Stochastic Games [10], can describe a MAS where agents are aware of each other. In an MAMDP, the state and action sets are defined as the cartesian product of local states and actions for all agents, and the transition function now depends on the joint action rather than one single individual action. Distributed Q-Learning [14] can be used to learn an optimal joint policy for deterministic cooperative scenarios with a small computing time per step. Distributed Q-Learning learns without observing actions performed by other agents and stores only Q-values for the best possible *joint* action.

OO-MDP is another MDP extension proposed with the promise to offer generalization opportunities [16]. In order to better explain the OO-MDP concepts, we firstly present the *Goldmine* domain to provide examples for theoretical definitions. Figure 1 illustrates the *Goldmine* domain. There is a certain number of *miners*, which aim to collect all *gold pieces* spread in the environment. There are also impassable *walls* that limit miner movements. At each decision step, all miners may move or collect gold pieces that are close enough. All agents must work collaboratively and the task is completed when all gold pieces in the environment are collected.

An OO-MDP consists of a tuple $\langle C, O, A, \mathrm{T}, D, R\rangle$. $C = \{C_1, \ldots, C_c\}$ is the set of *classes*, where each class C_i is composed of a set of *attributes* denoted as



(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. *Miners* aim to gather all *gold pieces* in the environment. Thick *walls* are impassable (adapted from [16]). (a) Graphical representation. (b) Object representation.

$Att(C_i) = \{C_i.b_1, \ldots, C_i.b_b\}$, and each attribute b_j has a *domain* $Dom(C_i.b_j)$, which specifies the set of values this attribute can assume. The object-oriented representation of the *Goldmine* domain has three classes: *Miner*, *Gold*, and *Wall*. They all have attributes x and y, and walls have an additional pos (position) attribute to indicate the position of the wall in respect to the compass direction. Hence, the set of classes and attributes must be enough to 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 a particular environment, where each object is an instance of one class: $o_i \in C_j, C(o_i) = C_j$ and, for each decision step, each object assumes a state given by the current value of all attributes. It may be also important to include an object identification. For example, in the *Gridworld* domain, miners may be identified by a "name", which is used to define the miner that will be moved towards a gold piece. Thus, 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. The state of the underlying MDP is the union of states of all objects: $s = \bigcup_{o \in O} o.state$. Figure 1b illustrates an object-oriented state, where all objects in the environment are described by their attribute values.

The set A consists of actions that may or may not be parameterized. Actions that are parameterized are defined at the class level (i.e., each action affects any object of that class in the same way), thus, parameterized actions are abstract and need to be grounded before applied. T is a set of *terms*, which are boolean functions representing relations between objects $t : C_i \times \cdots \times C_j \rightarrow Boolean, t \in \mathrm{T}$. 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* is an operation that changes with probability *prob* the value of attributes in an object $f : Dom(C_i.b_j) \rightarrow Dom(C_i.b_j)$. Finally, R is a reward function equivalent to the standard MDP one. As we are interested in learning problems, the agent does not know T, D, and R, and has to learn how to actuate through interactions in the environment.

The transition dynamics in an OO-MDP is interpreted as follows. First, at each step k, the agent observes the current state s_k and applies an action a_k. Second, all terms are evaluated to be $true$ or $false$ at that step. And third, all rules associated to a_k that had a matched condition trigger an effect that changes attribute values. After all effects have been processed, the change in attribute values results in a state transition, and this procedure can be repeated.

Note that MAMDP and OO-MDP use different approaches to deal with some specific scalability issues and in the following section we propose a solution which combines both methods to benefit from their advantages.

III. Multiagent Object-Oriented Representation

Here, we present a formal definition for an MOO-MDP, an extension of OO-MDP to MAS.

The main differences between OO-MDPs and MOO-MDPs are that in the latter: (i) the environment is simultaneously

affected by multiple agents, which means that the state transition now depends on $joint$ actions rather than individual agent actions; (ii) each agent may have a slightly different observation of the world, resulting in similar but possibly different local states. Also, each agent has its own reward function, which means that agents can have different goals. We focus on cooperative domains here, however MOO-MDP is a general model that can be used for both cooperative and competitive MAS (or a mix of those).

An MOO-MDP is described by the tuple: $\langle C, O, U, \mathrm{T}, D, R^M \rangle$. C is the set of *classes* and m is the number of agents. We define $Ag = \{Z_1, \ldots, Z_m\}, Ag \subseteq C$ as the set of *Agent Classes*, i.e., each object of any class $Z_i \in Ag$ is an agent able to observe the environment and perform autonomous actions. $\Gamma \subseteq C$ is the set of *abstracted* classes. Objects of these classes cannot be differentiated among themselves except by their attribute values, thus each object assumes an *abstract* state: $o.\overline{state} = \prod_{b \in Att(C(o))} o.b$.

O is the set of objects that 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, $\forall z \in G : C(z) \in Ag$. The current *concrete* state now is a composition of the state of environment and agent objects: $s = \bigcup_{o \in O} o.state = \left(\bigcup_{e \in E} e.state\right) \cup \left(\bigcup_{z \in G} z.state\right)$, i.e., we assume full observability. An abstract state $\tilde{s}$ is defined according to: $\tilde{s} = \left(\bigcup_{o \in O, C(o) \in \Gamma} o.\overline{state}\right) \cup \left(\bigcup_{o \in O, C(o) \notin \Gamma} o.state\right)$, where $\tilde{s}$ is a set of concrete states ($\tilde{s} \subseteq S$). The function κ defines the abstract state for an agent z given a concrete state, $\tilde{s^z} = \kappa(s, z)$, in which all objects of classes $C_i \in \Gamma$ have their id suppressed. Note that the definition of abstract states enables knowledge generalization. For example, in the *Goldmine* domain we set the *Gold* class as abstracted ($Gold \in \Gamma$), so that when an agent first collects a gold piece, it will learn that this action results in a positive reward and, as $Gold$ is abstracted, the agent generalizes this knowledge and reasons that any collected gold piece results in a positive reward.

U is the set of joint actions for all agents, in which each agent has its own individual action set, $U = A_1 \times \cdots \times A_m$. Individual actions can be parameterized or not, thus MOO-MDPs also allow action space abstraction. Action sets do not need to be equal, hence MOO-MDPs can be applied in both homogeneous and heterogeneous MAS. T and D have the same definition as in OO-MDPs, and $R^M = \{R_1, \ldots, R_m\}$ is the set of reward functions for all agents, which now returns reward signals taking *joint* actions into account, rather than individual actions. In learning problems, T, D, R^M, and U are unknown to the agent (it knows A_z though).

In each step k, all agents observe their abstract state $\tilde{s}_k^z$ and apply an action a_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, effects change object attributes, causing a state transition, and the agent receives a reward r_k^z.

In the next section we present an algorithm to solve deterministic cooperative MOO-MDPs with homogeneous agents.

IV. Learning in Cooperative MOO-MDPs

We are interested in a solution for Deterministic Distributed Cooperative MOO-MDPs, a specific class of the general MOO-MDP framework presented in Section III. The reward function in cooperative MAS is the same for all agents, $R_1 = \cdots = R_m$. We assume that there is no central controller, and each agent takes actions unaware of other agent actions. We here present a *model-free* algorithm based on Distributed Q-Learning [14] to solve such problems.

The proposed algorithm, hereafter called *Distributed Object-Oriented Q-Learning* (DOO-Q) learns a joint policy in a distributed and generalized manner, in which each agent z stores a local Q-table (Q^z) containing abstract states ($\tilde{s}^z$) and its own local actions (a^z). An agent z can find a suitable policy for the joint actuation, even when unaware of other agent actions, through iteratively updating its Q-table over abstract states and concrete actions:

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

However, the greedy policy applied to local Q-tables is not guaranteed to result in an optimal joint policy, because agents do not take into account each other actions and miss-coordination issues may arise. We use the following policy update in order to provide a coordination mechanism:

$$\pi_{k+1}^z(\tilde{s}^z) \leftarrow \begin{cases} \pi_k^z(\tilde{s}^z) & \text{if } \max_{a \in A_z} Q_k^z(\tilde{s}^z, a) = \max_{a \in A_z} Q_{k+1}^z(\tilde{s}^z, a) \\ a_k^z & \text{otherwise} \end{cases}. \quad (2)$$

DOO-Q solves MOO-MDPs allying distributed Q-table update of Equation (1) with policy update of Equation (2), and we can prove that DOO-Q learns an optimal joint policy under certain restrictions by:

Proposition 1. *Let π^z be a decentralized policy learned by agent z on a cooperative MOO-MDP using Equation (2), and $m = |G|$. Assume that Equation (1) 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 even when each agent's policy is defined over local Q-tables, i.e. :*

$$\forall s \in S : [\pi^1(\tilde{s}^1) \ \ldots \ \pi^m(\tilde{s}^m)]^T = \arg\max_{\boldsymbol{u} \in U} Q^*(s, \boldsymbol{u}), \quad (3)$$

where Q^* is the joint Q-table at convergence time. We assume that:

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 $\tilde{s}^z_{k+1}$ are observed when applying $\boldsymbol{u}$ in any concrete state covered by $\tilde{s}^z_k$.

Proof: For all agents $z \in G$, let π^z_0 be arbitrarily initialized. Because Equation (1) is used for Q-value updates, Q^z_k is a monotonically increasing function; that is, $\forall s \in S, a \in A_z : Q^z_k(s,a) \leq Q^z_{k+1}(s,a)$. Also, according to Equation (2), the policy is only updated in step k, when:

$$\exists! a \in A_z : Q^z_k(\tilde{s}^z_k, a) < Q^z_{k+1}(\tilde{s}^z_k, a).$$

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

$$\pi_{k+1}(s_k) = \begin{bmatrix} \pi^1_{k+1}(\tilde{s}^1_k) \\ \vdots \\ \pi^m_{k+1}(\tilde{s}^m_k) \end{bmatrix} = \begin{bmatrix} \underset{a^1 \in A_1}{arg\ max} Q^1_{k+1}(\tilde{s}^1_k, a^1) \\ \vdots \\ \underset{a^m \in A_m}{arg\ max} Q^m_{k+1}(\tilde{s}^m_k, a^m) \end{bmatrix} \tag{4}$$

which means that, at convergence time, the resulting joint policy corresponds to

$$\forall s \in S : \pi^*(s) = \begin{bmatrix} \underset{a^1 \in A_1}{arg\ max} Q^{1*}(\tilde{s}^1, a^1) \\ \vdots \\ \underset{a^m \in A_m}{arg\ max} Q^{m*}(\tilde{s}^m, a^m) \end{bmatrix}. \tag{5}$$

Assumptions 1 and 3 ensure that, for any step k, the same experience $\langle s_k, \boldsymbol{u_k}, s_{k+1}, r_k \rangle$ is valid for all agents, which together with Assumption 4 ensure that each agent always observes the same $\tilde{s}^z_k$ whenever s_k is visited. Thus, the following Q-value update is performed either in distributed or in joint control, respectively:

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

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

1) $Q^z_k(\tilde{s}^z_k, a^z_k) < r_k + \gamma \max_{a \in A_z} Q^z_k(\tilde{s}^z_{k+1}, a)$: Because of Assumption 2, Q^z and Q are initially equal, hence both $Q^z_k(\tilde{s}^z_k, a)$ and $Q_k(s, \boldsymbol{u})$ are updated, thus after the update: $Q^z_{k+1}(\tilde{s}^z_k, a^z_k) \geq Q_{k+1}(s_k, \boldsymbol{u_k})$.
2) *Otherwise*: No Q-value is updated on the distributed Q-table. As updates are done following the same procedure at any k: $Q_k(s_k, \boldsymbol{u_k}) \leq Q^z_k(\tilde{s}^z_k, a^z_k)$ and $\max_{\boldsymbol{u} \in U, s \in \tilde{s}^z_{k+1}} Q_k(s, \boldsymbol{u}) \leq \max_{a \in A_z} Q^z_k(\tilde{s}^z_{k+1}, a)$. This means that, after the update in k, the following relation is valid: $Q_{k+1}(s_k, \boldsymbol{u}_k) \leq Q^z_{k+1}(\tilde{s}^z_k, a^z_k)$.

Since for both situations $Q_{k+1}(s_k, \boldsymbol{u}_k)$ and $Q^z_{k+1}(\tilde{s}^z_k, a^z_k)$ are the only Q-table entries that may be updated and the latter is always greater or equal than the former, the following Equation holds for any k, $s \in S$, and $a^z \in A_z$:

$$Q^z_k(\tilde{s}^z_k, a^z) \geq \max_{\boldsymbol{u} \in U, \boldsymbol{u^z} = a^z, s \in \tilde{s}^z_k} Q_k(s, \boldsymbol{u}), \tag{6}$$

where Q^z_k is the local Q-table of agent z at step k, Q_k is the joint Q-table for all agents, agent z chose action a^z ($\boldsymbol{u^z} = a^z$), and $s \in \tilde{s}^z_k$. Assumptions 1, 3, and 4 also ensure that at convergence time, all trajectories starting from a given concrete state s are already known, resulting in:

$$Q^{z*}(\kappa(s,z), a^z) = \max_{\boldsymbol{u} \in U, \boldsymbol{u^z} = a^z} Q^*(s,u), \tag{7}$$

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

Combining Equations (7) and (5) results in:

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

Due to the update rule in Equation (2) and the cooperative nature of the MOO-MDP, we can say that agents coordinate by breaking ties in $\arg\max_{a \in A_i} Q^i_{k+1}(\tilde{s}^i, a)$ according to the order in which experiences occurred, which means that agents coordinate even when multiple optimal joint policies exist. Thus Equation (8) is equivalent to

$$\forall s \in S : \pi^*(s) = \underset{\boldsymbol{u} \in U}{arg\ max}\ Q^*(s,u). \tag{9}$$

Hence, a joint policy implied by decentralized policies updated as in Equation (2) eventually converges to the optimal joint policy, provided that all states are infinitely visited and all applicable actions have a *non-zero probability* of being executed by the exploration strategy. ■

DOO-Q is fully described in Algorithm 1. At first, local Q-tables are initialized with zero values according to Assumption 2 of Proposition 1. Then, each agent observes its current abstract state $\tilde{s}^z_k$ according to the state of all objects (as described in Section III) and chooses an action a^z_k according to its 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 1), e.g., the ϵ-greedy strategy. After all agents have applied their actions, each agent observes its next state and reward, and finally updates its Q-table and policy π^z_k, ending the current learning step. Note that the agent's observation of the current state over the set of objects enables state generalization; i.e., an agent may see all objects of the same class as equivalent, and only differentiate them by their attribute values.

V. Experimental Evaluation

We evaluate our proposal in a slightly modified version of the *Goldmine* [16] domain. In this 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

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 1).
8: Update policy $\pi_k^z(\tilde{s}_k^z)$ (Equation 2).
9: $\tilde{s}_k^z \leftarrow \tilde{s}_{k+1}^z$.
10: **end for**

miner objects are agents, i.e. $C = \{Miner, Gold, Wall\}$, $Ag = \{Miner\}$, $Att(Miner) = Att(Gold) = \{x, y\}$, $Att(Wall) = \{x, y, pos\}$, and $A_z = \{North(z), South(z), East(z), West(z), GetGold(z)\}$. The following terms are defined: $touch_N(m,w)$, $touch_S(m,w)$, $touch_W(m,w)$, $touch_E(m,w)$, and $on(m,g)$, 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 its current position. For a given triple $\langle s_k, \boldsymbol{u_k}, s_{k+1} \rangle$, we define the reward function as follows:

$$r(s_k, \boldsymbol{u_k}) = gold \times n_{gold} \times \gamma^{(2n_{miner}+1.5n_{wall})}, \qquad (10)$$

where $gold$ is the value for each gold collection, γ is the same discount rate as in Equation (1), n_{gold} is the number of collected gold pieces when applying the joint action $\boldsymbol{u_k}$, n_{wall} is the number of miners colliding with walls in k, 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, which would invalidate Assumption 2 of Proposition 1.

In our experiment, 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 random function 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 state, algorithms explore using an exploration strategy and, for each interval of 100 episodes, a single episode using the greedy policy is assessed to extract the number of steps required to reach a terminal state and the received accumulated discounted reward. The following algorithms were compared:

1) **Single-agent Q-Learning (SAQL)**: This algorithm follows 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.
2) **Multiagent Q-Learning (MAQL)**: Each miner is an autonomous agent for 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.
3) **DOO-Q**: In our proposal all miners are autonomous agents.
4) **Distributed Q-Learning (DQL)**: This algorithm is implemented with a factored state description [14].

For all algorithms, we used $\gamma = 0.9$ and an ϵ-greedy exploration strategy with $\epsilon = 0.1$. Also, $\alpha = 0.2$ for SAQL and MAQL. A time limit was set for the experiment, in which an algorithm is interrupted if the time limit is exceeded.The experiment was implemented in BURLAP [13]. Our implementations can be downloaded at *https://github.com/f-leno/DOO-Q_BRACIS2016*.

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($Miner\ m$)	$on(Miner\ m, Gold\ g)$	$g.x \leftarrow \emptyset, g.y \leftarrow \emptyset$

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, $|Q| = |S| \times |A|$. Table II presents $|S|$ and $|A|$ for all algorithms in the *Goldmine* domain. Hence, for 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) **DOO-Q** 2.9×10^{10}, (iv) **DQL** 2.4×10^{13}, which means that DOO-Q requires less Q-table entries than the other algorithms.

Figures 2 and 3 depict the results of our experiment, in which the shaded area represents the 95% confidence interval observed in 70 repetitions. Figure 2 shows that DOO-Q learns

TABLE II
STATE AND ACTION SPACES FOR EACH ALGORITHM IN OUR VERSION OF *Goldmine*. m IS THE NUMBER OF MINERS, p IS THE NUMBER OF GOLD PIECES, q IS THE NUMBER OF INDIVIDUAL ACTIONS, AND w IS THE NUMBER OF POSSIBLE CELLS INSIDE THE GRID.

	$\lvert S\rvert$	$\lvert A\rvert$
SAQL	$w^m \frac{(p+w)!}{p!w!}$	$q\ m$
MAQL	$w \frac{(m+w-2)!}{(m-1)!(w-1)!} \frac{(p+w)!}{p!w!}$	q^m
DOOQ	$w \frac{(m+w-2)!}{(m-1)!(w-1)!} \frac{(p+w)!}{p!w!}$	q
DQL	$w^m(w+1)^p$	q

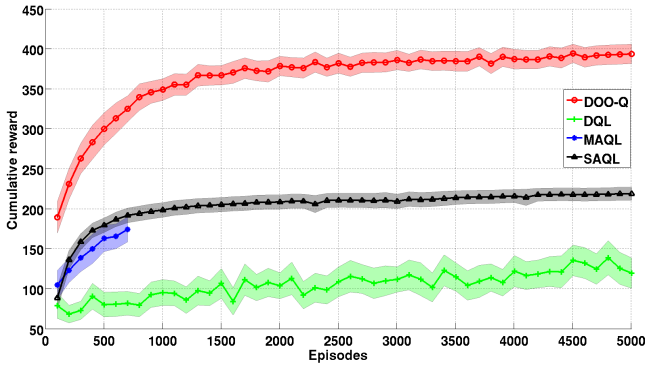


Fig. 2. Observed discounted cumulative reward in the *Goldmine* domain.

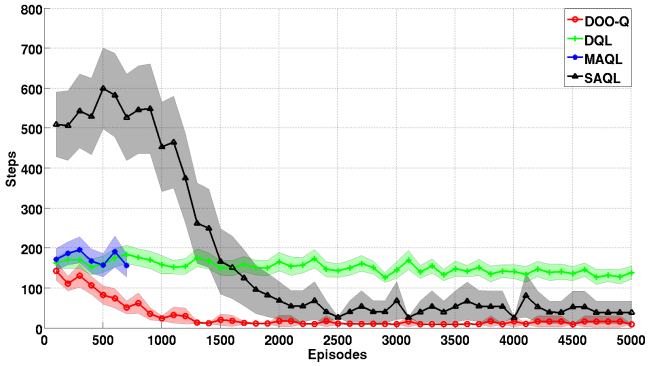


Fig. 3. Number of steps to complete one episode in the *Goldmine* domain.

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 the only difference between DOO-Q and DQL is the object-oriented representation, the results clearly reflect the advantage of MOO-MDPs.

Figure 3 shows that the MAS approaches (DOO-Q, MAQL and DQL) completed the task with less steps in the beginning of the training. Then, 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 achieve better results than SAQL in steps for task completion but it was unable to scale to this problem size. The results in this domain show that the abstraction provided by MOO-MDPs can greatly accelerate the learning speed, as DOO-Q achieved much better results than DQL. Also, compared to SAQL, MAS algorithms learned faster to reduce the number of steps needed to complete the task, which indicates that dividing the workload helps to solve some problems.

In summary, our experiment shows that DOO-Q achieves the best performance among 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.

VII. Conclusion and Further Works

In this article we 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. Our proposal was experimentally compared with other model-free algorithms in the *Goldmine* domain, and DOO-Q achieved a better performance both in learning speed and memory requirements. Further works could focus on developing algorithms for stochastic and general-sum MOO-MDPs, in which DOO-Q is not applicable. MOO-MDPs could also be extended to Partially Observable domains, which would allow to develop distributed approaches where agents do not necessarily observe the whole environment state at each step.

References

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

[2] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang, "Autonomous inverted helicopter flight via Reinforcement Learning," in *Experimental Robotics IX*. Springer, 2006, pp. 363–372.

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

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

[5] C. Diuk, A. Cohen, and M. L. Littman, "An Object-oriented Representation for Efficient Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, 2008, pp. 240–247.

[6] S. Mohan and J. E. Laird, "An Object-Oriented Approach to Reinforcement Learning in an Action Game," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2011, pp. 164–169.

[7] 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 *International Joint Conference on Artificial Intelligence (IJCAI)*, 2015, pp. 3856–3864.

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

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

[10] 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.

[11] M. Tan, "Multi-Agent Reinforcement Learning: Independent versus Cooperative Agents," in *International Conference on Machine Learning (ICML)*, 1993, pp. 330–337.

[12] 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.

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

[14] M. Lauer and M. Riedmiller, "An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems," in *International Conference on Machine Learning (ICML)*, 2000, pp. 535–542.

[15] 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.

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