Artificial Intelligence 329 (2024) 104079

Contents lists available at ScienceDirect

Artificial Intelligence

journal homepage: www.elsevier.com/locate/artint

Transferable dynamics models for efficient object-oriented reinforcement learning

Ofir Marom, Benjamin Rosman [*]

University of the Witwatersrand, Johannesburg, South Africa

ARTICLE INFO

Keywords:
Reinforcement learning
Transfer learning
Object-oriented representations
Efficient learning

ABSTRACT

The Reinforcement Learning (RL) framework offers a general paradigm for constructing autonomous agents that can make effective decisions when solving tasks. An important area of study within the field of RL is transfer learning, where an agent utilizes knowledge gained from solving previous tasks to solve a new task more efficiently. While the notion of transfer learning is conceptually appealing, in practice, not all RL representations are amenable to transfer learning. Moreover, much of the research on transfer learning in RL is purely empirical. Previous research has shown that object-oriented representations are suitable for the purposes of transfer learning with theoretical efficiency guarantees. Such representations leverage the notion of object classes to learn lifted rules that apply to grounded object instantiations. In this paper, we extend previous research on object-oriented representations and introduce two formalisms: the first is based on deictic predicates, and is used to learn a transferable transition dynamics model; the second is based on propositions, and is used to learn a transferable reward dynamics model. In addition, we extend previously introduced efficient learning algorithms for object-oriented representations to our proposed formalisms. Our frameworks are then combined into a single efficient algorithm that learns transferable transition and reward dynamics models across a domain of related tasks. We illustrate our proposed algorithm empirically on an extended version of the Taxi domain, as well as the more difficult Sokoban domain, showing the benefits of our approach with regards to efficient learning and transfer.

1. Introduction

A longstanding goal in the field of Artificial Intelligence (AI) is the construction of autonomous agents that make effective decisions to complete given tasks. The Reinforcement Learning (RL) framework [1] offers a general paradigm for constructing autonomous agents with these capabilities. In the RL setting, an agent interacts with its environment with the aim of learning a policy that instructs the agent on which action to take in any given state to achieve an optimal payoff. For reasons of computational tractability, an RL task is typically modelled as a Markov Decision Process (MDP). Given an MDP description of an RL task, a set of fairly simple algorithms exist with convergence guarantees to optimal policies [1]. RL has had many successes, achieving master-level performance on a variety of complex tasks such as Backgammon, a suite of Atari 2600 games and chess [2–5].

An important area of study within RL is that of *transfer learning* [6]. In the transfer learning setting, an RL agent leverages prior knowledge learned from solving previous tasks to solve a new and related task more efficiently. Human cognition is known to transfer

[*] Corresponding author.
E-mail address: Benjamin.Rosman1@wits.ac.za (B. Rosman).

https://doi.org/10.1016/j.artint.2024.104079
Received 13 July 2021; Received in revised form 12 January 2024; Accepted 15 January 2024

Available online 26 January 2024

vast amounts of prior knowledge gained from previous experiences when solving new tasks [7]. Therefore, it is natural to seek ways to imbue RL agents with similar capabilities so that they can solve new tasks as efficiently as humans do.

While the transfer learning approach has intuitive appeal, in practice it is a challenging research area with many open questions [6]. In particular, it is not always clear what, when or how to transfer in order to achieve improved learning performance. Furthermore, the research area of transfer learning has tended to rely more on empirical methods that lack theoretical foundations [6].

One concept that has shown much promise for effective transfer is based on relational representations in RL [8,9]. These representations rely on expressing the components of an MDP in terms of logical statements over variables. Transfer is then achieved by expressing lifted rules about groups of variables, and then grounding these lifted rules for specific instantiations of variable values.

Object-oriented representations are a form of relational representation that draws inspiration from the structure of the physical world [10–15]. Object-oriented representations define a set of lifted object classes from which grounded objects are then instantiated for a particular task. Such representations aim to learn a set of lifted rules that apply to the object classes. These lifted rules then generalize to grounded instantiations of objects, and are therefore transferable between related tasks of a domain.

In particular, previous research introduced the Propositional Object-Oriented MDP (Propositional OO-MDP) framework [13,14]. Propositional OO-MDPs operate by formulating the transition dynamics of an MDP in terms of propositional preconditions over lifted object classes that map to effects over object class attributes. Since the preconditions and effects that represent the transition dynamics are fully lifted, they transfer across all tasks of a domain. Propositional OO-MDPs have limited expressive power because they require that all preconditions be propositional. However, they have the advantage that provably efficient learning algorithms exist under the formalism. In particular, the research that introduces Propositional OO-MDPs also introduces an algorithm called *DOORMAX* that operates under the Propositional OO-MDP formalism in domains with deterministic transition dynamics. The *DOORMAX* algorithm falls into the more general KWIK-R_{max} class of algorithms from the KWIK (knows what it knows) framework [16,17]. As a result, it has provably efficient learning bounds.

Unfortunately, the core restriction of Propositional OO-MDPs that the transition dynamics be described only in terms of propositional preconditions is a strong one, and precludes efficient learning for a large class of domains. Such domains include those where it is required to distinguish between different objects of the same object class for tasks of the domain. As a specific example of this, and further elaborated in subsection 3.2, consider the Sokoban domain where a person attempts to push a box but cannot do so if that box is adjacent to a wall. In this case, there is no way of tying the box that is adjacent to the person with the box that is adjacent to the wall with lifted propositions.

To accommodate this, our previous research introduced the Deictic Object-Oriented MDP (Deictic OO-MDP) framework [18]. Deictic OO-MDPs use deictic predicates for its preconditions. A deictic predicate is a predicate that is grounded only with respect to a central deictic object, therefore that object may relate itself to non-grounded object classes, but not to other grounded objects. Returning to the Sokoban domain, a deictic predicate over boxes allows a specific box to ascertain whether *any* wall is adjacent to it, but not whether a *specific* wall is adjacent to it. Deictic OO-MDPs are more general than Propositional OO-MDPs, and so are able to represent a larger class of domains. Furthermore, the research that introduces Deictic OO-MDPs extends *DOORMAX* with an algorithm called $DOORMAX_D$ so that efficient learning is possible under the Deictic OO-MDP framework as well [18].

Both previously introduced Propositional OO-MDP and Deictic OO-MDP settings assume known reward dynamics and focus on learning transition dynamics. In the most general model-based RL setting, the agent must learn both of these components. In this paper we combine Deictic OO-MDPs and Propositional OO-MDPs by leveraging the former to efficiently learn transferable transition dynamics and the latter to efficiently learn transferable reward dynamics. We then extend $DOORMAX_D$ to accommodate learning of both of these components.

We illustrate our framework empirically on the Taxi and Sokoban domains. The original Taxi task [19] takes place in a small 5×5 gridworld. In this paper we demonstrate that we can learn transferable models of both the transition and reward dynamics from such small Taxi tasks, and then zero-shot transfer them to much larger 10×10 tasks, where the taxi also has to drop of multiple passengers.[1] We further illustrate the flexibility of our framework on the Sokoban domain where we learn the dynamics from some small Sokoban tasks and then zero-shot transfer them to a larger task, that has an optimal solution depth of 209 steps, to obtain an optimal policy.

In summary, the contributions of this paper are as follows: 1) we extend the previously introduced Propositional OO-MDP formalism to a more general Deictic OO-MDP formalism for transition dynamics, providing significantly more detail compared to our previous work [18]; 2) we introduce a novel propositional object-oriented formalism for reward dynamics; 3) we extend the previously introduced KWIK-learnable *DOORMAX* algorithm for learning deterministic transition dynamics under the propositional setting to the $DOORMAX_D$ algorithm for learning deterministic transition dynamics under the deictic setting and stochastic reward dynamics under the propositional setting, and prove that our algorithm is also KWIK-learnable; 4) we demonstrate that once the dynamics are fully learned from a set of tasks of a given domain, they can be zero-shot transferred to a new task of the domain.

The rest of this paper is organized as follows: in section 2 we discuss background information for efficient learning and object-oriented representations in RL; in section 3 we introduce the Deictic OO-MDP formalism and associated efficient learning algorithms; in section 4 we extend previous work on Propositional OO-MDPs to accommodate a formalism and associated efficient learning

[1] We call this the All-Passenger Any-Destination Taxi domain. Given a gridworld of size $h \times w$ with p passengers and d destinations, and a fixed wall configuration, one can generate $\binom{hw}{p+d}$ tasks, assuming passengers and destinations cannot overlap. This makes it considerably larger than the original Taxi domain that has $p = 1$ and $d = 1$ and therefore has only $\binom{hw}{2}$ tasks.

algorithms for reward dynamics; in section 5 we introduce the complete *DOORMAX*$_D$ algorithm for learning transferable models of the transition and reward dynamics under the formalisms of sections 3 and 4 respectively; in section 6 we present a theorem for our algorithm's efficiency; in section 7 we run experiments on the Taxi and Sokoban domains illustrating the benefits of our proposed framework; in section 8 we discuss directions for future research; and in section 9 we conclude with final remarks.

2. Background

2.1. Markov decision process

For reasons of computational tractability, an RL task is typically formulated as a Markov Decision Process (MDP) [1]. An MDP is described by a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \rho)$, where:

- $\mathcal{S}$ is a finite set of states called the state-space;
- $\mathcal{A}$ is a finite set of all actions called the action-space;
- $\mathcal{P}$ is the transition dynamics for which $\mathcal{P}(s'|s,a)$ returns the transition probability for transitioning to state $s' \in \mathcal{S}$ conditional on the agent being in state s and taking action $a \in \mathcal{A}$;
- $\mathcal{R}$ is the reward dynamics for which $\mathcal{R}(s,a,s')$ returns the (stationary, finite variance) distribution that governs the reward signal that the agent receives when in state s, takes action a and transitions to state s';
- $\gamma \in [0,1)$ is a discount rate that controls the trade-off between the importance of immediate and future rewards; and
- ρ is a distribution over start states for which $\rho(s_0)$ returns the probability of a task starting in state $s_0 \in \mathcal{S}$.

In this paper, we further restrict attention to episodic tasks, so that the MDP has a set of terminal states $\mathcal{S}_{terminal}$ and a task terminates when one of these states is reached.

In RL the agent has no control over the environment dynamics, $\mathcal{P}$ and $\mathcal{R}$. What the agent does control is the policy it follows in the environment. Formally, a policy is defined by $\pi(a|s)$ and represents the probability of taking action a conditional on the agent being in state s. The goal of an RL agent is to find an optimal policy π_* that maximizes the expected future discounted rewards given any state $s \in \mathcal{S}$. Broadly speaking, algorithms that learn an optimal policy for an MDP can be categorized into model-free and model-based methods. Model-free methods go straight from experience to a policy, and so do not learn models of $\mathcal{P}$ and $\mathcal{R}$; model-based methods learn models of $\mathcal{P}$ and $\mathcal{R}$ in conjunction with learning an optimal policy. Given an MDP formulation of an RL task, a host of model-free and model-based algorithms exist with convergence guarantees to an optimal policy.

2.2. Efficient learning

Unfortunately, algorithms that only have convergence guarantees to an optimal policy are not sufficient. In practice, such algorithms may run for an infeasible amount of time before convergence. Therefore, in addition to convergence guarantees, we also care about the performance of RL algorithms. In particular, we want algorithms to be *sample efficient*. Sample efficient algorithms learn an optimal policy in as few interactions as possible.

One well-known sample efficient RL algorithm is R_{max} [20]. The R_{max} algorithm is a model-based algorithm that operates under the *optimism in the face of uncertainty* principle. Under this principle, the agent initially assumes optimistic models of the dynamics, such that taking any action in any state leads to a terminal state with optimal payoff. The agent then applies an exact planning algorithm to obtain an optimal policy under these incorrect models. This optimal policy drives exploration to the parts of the environment where the agent expects to receive these optimal payoffs. From there, the agent learns the underlying truth and updates its models with this knowledge. By applying this methodology, the agent is able to construct iteratively more accurate models of the dynamics every time it interacts with its environment. As a result of this exploration strategy, it can be shown that R_{max} has polynomial sample complexity with respect to $|\mathcal{S}|$ and $|\mathcal{A}|$ under the PAC (probably approximately correct) terms [21] – i.e. the policy produced by R_{max} is near-optimal with high probability.

The R_{max} algorithm was designed for the case of finite MDPs. Unfortunately, even with the polynomial sample complexity guarantees of R_{max}, learning a near-optimal policy may be slow. This is because R_{max} has to run an exact planning algorithm at every iteration and, when the state-space or action-space is large, such planning algorithms tend to have high computational complexity.

An alternative way to improve the sample efficiency of RL algorithms is with compact representations. Such representations do not learn the dynamics models in a tabular way, but rather use a representation that compresses the size of the state-space or action-space so that the models can be learned more efficiently.

The KWIK (knows what it knows) framework extends R_{max} with the KWIK-R_{max} algorithm [16,17]. The KWIK-R_{max} algorithm generalizes R_{max} so that it works with a broader range of representations on the condition that they respect the KWIK protocol. Under the KWIK protocol, when the agent is in some state s and takes some action a, then rather than making a next-state prediction s' the agent may instead return $\perp$, indicating that they are not yet able to make an accurate prediction with high probability. The KWIK protocol restricts the number of times that the agent may return $\perp$, called the KWIK-bound, to be a polynomial bound. Under the restriction that the KWIK-bound is a polynomial bound, KWIK-R_{max} can be shown to have polynomial sample complexity under the PAC terms as well. We provide a formal definition of the KWIK-bound below:

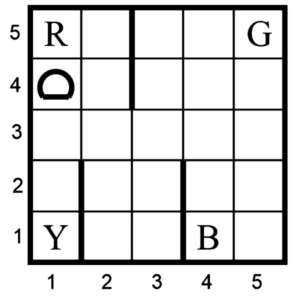


Fig. 1. The original Taxi task. Letters mark possible pickup and destination locations; marks the taxi; thicker lines mark walls.

Definition 1. Let $\mathfrak{A}$ be an algorithm that adheres to the KWIK protocol with $0 < \epsilon, \delta < 1$. Then the KWIK-bound of $\mathfrak{A}$, denoted $\mathcal{K}(\epsilon, \delta)$, is the number of times $\mathfrak{A}$ may return $\perp$ before all predictions, $\hat{y}$, made by $\mathfrak{A}$ are within ϵ of their true value, y, with probability at least $1-\delta$. That is $P(|\hat{y} - y| < \epsilon) \geq 1 - \delta$. We say that $\mathfrak{A}$ is KWIK-learnable with KWIK-bound $\mathcal{K}(\epsilon, \delta)$.

Examples of compact representations that have been used in conjunction with KWIK-R_{max} include dynamic Bayesian networks [22,23], linear models [24] and relations [13,14].

2.3. Object-oriented MDPs

Object-oriented MDPs (OO-MPDs) form part of the broader field of relational RL, and were first introduced under the Relational MDP framework for planning domains [10,11]. OO-MDPs view the state-space of an MDP in terms of objects, where each object belongs to some object class (or simply class) that has a set of attributes. The transition and reward dynamics are then expressed through the relational structure between object class attributes. The main advantage of object-oriented representations is that they allow for a compact description of an MDP that is fully lifted. Therefore, such an MDP can be used to describe an entire domain, rather than a specific task. Such a lifted MDP is called a *schema*, from which grounded MDPs can then be instantiated.

As a result, OO-MDPs are favourable for the purposes of transfer learning. In the transfer learning research problem, the agent aims to learn some knowledge from a set of source tasks that can then be used to accelerate solving a related target task [6]. Since with OO-MDPs knowledge is represented at the class-level, it is possible to learn some knowledge on source tasks of the schema and then transfer that knowledge to target tasks of the schema.

Formally, the state-space for such a schema consists of a set of object classes $\mathfrak{C} = \{C_i\}_{i=1}^{N_\mathfrak{C}}$. Each object class $C \in \mathfrak{C}$ has a set of attributes $Att(C) = \{C.\alpha_i\}_{i=1}^{N_C}$ and each attribute $C.\alpha \in Att(C)$ of an object class has a domain of possible values $Dom(C.\alpha)$. Given a schema, a grounded state-space is instantiated by first selecting a grounded object set which consists of n objects $O = \{o_i\}_{i=1}^{n}$ where each $o \in O$ is an instance of some object class C. The value of attribute $C.\alpha$ for object o is denoted by $o.\alpha$. Then the grounded state-space, denoted $\mathcal{S}_O$, is an assignment of each $o.\alpha$ for all objects in O. The schema state-space, denoted $\mathcal{S}$, is the set of all states for all possible object sets O.

To make the notion of a schema state-space concrete, consider the classical Taxi domain [19] as shown in Fig. 1. The original Taxi task takes place in a 5×5 gridworld where a taxi has to pick up a passenger that is at one of four pickup locations and drop them off at one of four destination locations. The possible pickup and destination locations as shown in Fig. 1 and are fixed upfront for the task.

The set of actions available to the agent are *North*, *East*, *South* and *West*, that control the taxi's navigation in the gridworld, as well as *Pickup* and *Dropoff*. The action *Pickup* picks up the passenger if the taxi is at the pickup location and the passenger is not already in the taxi, while the action *Dropoff* drops off the passenger if the taxi has already picked up the passenger and the taxi is on the destination location. Walls in the gridworld limit the taxi's movements. The reward dynamics for the Taxi task is -1 for each navigation step, -10 for applying the *Pickup* or *Dropoff* actions incorrectly and 20 for reaching a terminal state that drops off the passenger at the correct destination location.

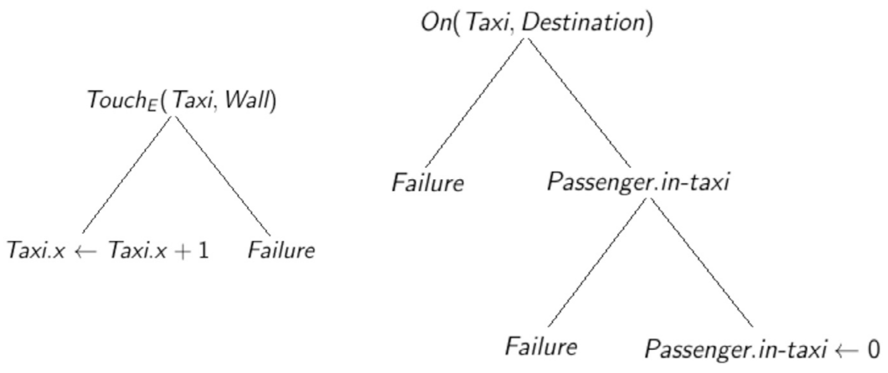
Under an object-oriented representation, we would define the schema state-space with four object classes: *Taxi*, *Wall*, *Passenger* and *Destination*. Each object class has attributes x and y for their location in the grid. Object class *Wall* has an additional attribute *pos* to mark one of four positions in a square, while *Passenger* has an additional Boolean attribute *in-taxi* to indicate if the passenger is in the taxi.

Given the schema state-space for this domain, we can then instantiate a set of grounded objects from the object classes and a resulting grounded state. For example, in Fig. 1 we can represent the taxi and its location by instantiating a grounded *taxi* object of class *Taxi* and setting $taxi.x = 1$ and $taxi.y = 4$.[2] We could similarly represent each wall, passenger and destination in Fig. 1 by instantiating an object of the required object class and setting its attributes accordingly.

[2] In this paper, we adopt the convention of using title case for object classes and lower case for grounded objects. For example, *Taxi* refers to the object class, while *taxi* refers to a grounded object of type *Taxi*.

Table 1
Transition dynamics of the Taxi domain under the Propositional OO-MDP formalism. Under the formalism, any other assignment of truth values to the propositions leads to a global *failure condition* that leaves all object class attributes unchanged.

Action	Precondition	Effect
North	$Touch_N(Taxi, Wall) = 0$	$Taxi.y \leftarrow Taxi.y + 1$
East	$Touch_E(Taxi, Wall) = 0$	$Taxi.x \leftarrow Taxi.x + 1$
South	$Touch_S(Taxi, Wall) = 0$	$Taxi.y \leftarrow Taxi.y - 1$
West	$Touch_W(Taxi, Wall) = 0$	$Taxi.x \leftarrow Taxi.x - 1$
Pickup	$On(Taxi, Passenger) = 1$	$Passenger.in\text{-}taxi \leftarrow 1$
Dropoff	$Passenger.in\text{-}taxi = 1 \wedge On(Taxi, Destination) = 1$	$Passenger.in\text{-}taxi \leftarrow 0$



(a) Full binary tree for the action *East* and attribute *Taxi.x*.

(b) Full binary tree for the action *Dropoff* and attribute *Passenger.in-taxi*.

Fig. 2. Examples of full binary tree structures under the Propositional OO-MDP formalism. The leaf node ‘Failure’ refers to a failure condition. Right branches represent a truth value of 1.

2.4. Propositional OO-MDPs

OO-MDPs were first introduced under the Relational MDP framework to learn transferable class-level value functions for planning domains [10,11]. Thereafter, they were used conjunction with KWIK-R_{max} for model-based RL under the Propositional OO-MDP framework [13,14]. The key insight that underlies Propositional OO-MDPs is that, for certain domains, it is possible to compactly represent the transition dynamics with conjunctions of propositional statements over object classes that map to effects over object class attributes as illustrated in Table 1 for the Taxi domain. Formally, an effect is defined as a change in value to a class attribute belonging to some class.

For example, the statement

$$Touch_E(Taxi, Wall) = 0 \implies Taxi.x \leftarrow Taxi.x + 1$$

for action *East* can be read as: if an object of class *Taxi* has an object of class *Wall* one square east of it is false, then all objects of class *Taxi* have their x attributes increased by 1. The same logic applies to the $Touch_N$, $Touch_S$ and $Touch_W$ expressions for the *North*, *South* and *West* actions respectively. The statement

$$On(Taxi, Passenger) = 1 \implies Passenger.in\text{-}taxi \leftarrow 1$$

for can be read as: if an object of class *Taxi* is on the same square as an object of class *Passenger* is true, then every object of class *Passenger* has its *in-taxi* attribute set to 1.

Since the preconditions in Table 1 only refer to object class attributes, the transition dynamics described in this way can be transferred to different Taxi tasks. For example, the dynamics described in this way transfer to a Taxi task with different gridworld dimensions, pickup, dropoff and wall locations.

An alternative view of transition dynamics under the Propositional OO-MDP formalism is through binary trees. For each action a and attribute $C.\alpha$, we can represent the transition dynamics as a full binary tree with propositions at the non-leaf nodes and effects at the leaf nodes. This is shown in Fig. 2a for the action *East* and attribute *Taxi.x* and in Fig. 2b for the action *Dropoff* and attribute *Passenger.in-taxi*.

The Propositional OO-MDP framework also introduces a KWIK-R_{max} based algorithm called *DOORMAX* (Deterministic Object-Oriented R_{max}) [13,14]. The *DOORMAX* algorithm operations in domains with deterministic transition dynamics that are representable under the formalism.

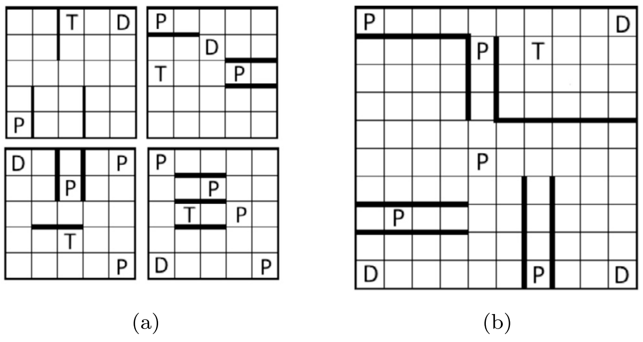


Fig. 3. 'P' marks a passenger; 'D' marks a destination; 'T' marks a taxi; thicker lines mark walls. Five possible states from the All-Passenger Any-Destination Taxi domain schema.

The algorithm learns the binary trees that represent the transition dynamics as shown in Fig. 2. Under *DOORMAX*, for each action a and attribute $C.\alpha$, a set of propositions and an effect type are hypothesized. Formally, an effect type is a defined as a function that applies a transformation to some attribute $C.\alpha$. For example, the relative effect type is defined as: $Rel_i(C.\alpha) = C.\alpha + i$ for $i = 0, 1, 2, ..., n$ and applies a shift of i to the current value of $C.\alpha$. The algorithm then learns which conjunctions over the hypothesized propositions map to which effect of the effect type. The main assumption required for *DOORMAX* to be correct is that each effect of a given effect type can occur at most at one non-leaf node in the tree. All leaf nodes that do not produce a unique effect are mapped to a global *failure condition* that leaves the state unchanged.

A further assumption made under *DOORMAX* is that effects are *invertible* so that if attribute $C.\alpha$ has some value assignment in state s and we take action a to subsequently observe a new value for $C.\alpha$ in s' then, given an effect type, there is a unique effect that can cause this transformation. This assumption is required so that, during learning, we can infer for each attribute exactly which effect occurred when transitioning from s to s'. As a practical example, consider state s where $Taxi.x = 1$ and state s' where $Taxi.x = 2$. Then under the relative effect type we have invertibility as there is only one effect, Rel_1, that can cause this transformation.

The Propositional OO-MDP framework extends *DOORMAX* to allow for the learner to hypothesize $N > 0$ effect types under the assumption that exactly one effect type is true and, furthermore, that the maximum number of effects per effect type is bounded by some $K > 0$. Under these additional assumptions one can prove that *DOORMAX* has KWIK-bound $K(D+1)+1$ to learn a single effect type for an action a and attribute $C.\alpha$, while learning given N effect types have KWIK-bound $N(K(D+1)+1)+(N-1)$ [13]. These bounds exclude the learning of failure conditions, which are learned inefficiently through memorization [16,17]. The *DOORMAX* algorithm assumes known reward dynamics and focuses on learning transition dynamics.

3. Transferable transition dynamics with deictic OO-MDPs

The Propositional OO-MDP formalism has the benefits of efficient learning, however, it lacks expressive power. In particular, Propositional OO-MDPs cannot compactly represent the transition dynamics of domains where it is required to distinguish between different objects of the same object class. To illustrate this, consider a simple extension to the Taxi domain which we call the All-Passenger Any-Destination Taxi domain. This domain is similar to the original Taxi domain, except that the taxi is tasked to pick up multiple passengers and drop each of them off at one of any destination locations. The taxi can only pick up one passenger at a time, so if a passenger is already in the taxi and the *Pickup* action is taken while the taxi is at the pickup location of another passenger, the state does not change. The state-space of the schema for this domain under an object-oriented representation is identical to that of the original Taxi domain described in section 2.3 except that we add an additional Boolean attribute to the *Passenger* object class *at-destination*, to indicate if a passenger has already been dropped off at a destination. Fig. 3 shows sample states of the schema.

Propositions over object classes are insufficient to compactly represent the transition dynamics for this version of the Taxi domain. To see why, suppose we have a task with two passenger objects and the proposition $On(Taxi, Passenger)$ with a truth value 1. This can be read as: an object of class *Taxi* is on the same square as an object of class *Passenger* is true. Clearly, this information is insufficient to determine which passenger object's *in-taxi* attribute should change given the *Pickup* action. To overcome this ambiguity under a propositional approach, we must resort to propositions over the grounded passenger objects, $passenger_1$ and $passenger_2$, of the form $On(Taxi, passenger_1)$ and $On(Taxi, passenger_2)$. Note that while this resolves the ambiguity, the number of propositions needed for the precondition now changes as we change the number of *passenger* objects in the task. This complicates both learning and transfer procedures.

As a further example, consider the Sokoban domain where a warehouse keeper is tasked to push boxes to storage locations in a warehouse, but cannot do so if a box is against a wall or in front of another box.[3] Suppose we have the propositions $Touch_W(Box, Person)$ and $Touch_E(Box, Wall)$ both with truth value 1. The first proposition can be read as: an object of class *Box* has an object of class *Person* one square west of it is true, while the second can be read as: an object of class *Box* has an object of class *Wall* one square east of it is true. Then the conjunction $Touch_W(Box, Person) = 1 \wedge Touch_E(Box, Wall) = 1$ is insufficient to determine

[3] All Sokoban images in this paper are taken from JSoko: https://www.sokoban-online.de/jsoko/credits.

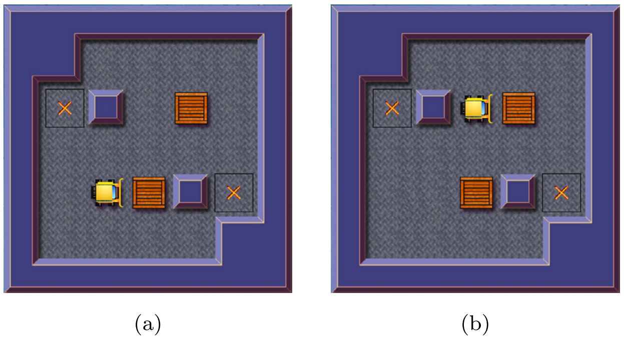


Fig. 4. For action *East* and the box adjacent to the warehouse keeper, in figure (4a) the effect is $box.x \leftarrow box.x + 0$; in figure (4b) the effect is $box.x \leftarrow box.x + 1$ while the conjunction $Touch_W(Box, Person) = 1 \wedge Touch_E(Box, Wall) = 1$ is true in both cases.

the transition dynamics of any *box* object's x attribute when taking action *East* since there is no way to know if the statements are referring to the same box. See Fig. 4 for an illustration. A similar example of ambiguity that is not resolvable under a propositional approach is also described by the original authors of the Propositional OO-MDP framework [14].

An example of a more expressive formalism that can be used to resolve this type of ambiguity is the First-Order MDP (FO MDP) [25] that can use first order predicates over grounded objects. Unfortunately, such an expressive formalism complicates efficient learning and transfer procedures.

In this section we introduce the Deictic Oject-Oriented MDP (Deictic OO-MDP) framework. Deictic OO-MDPs are more expressive than Propositional OO-MDPs, thus allowing for a broader range of domains to be represented under this formalism. While being less expressive than FO MDPs, Deictic OO-MDPs still have the provably efficient learning guarantees that underlies Propositional OO-MDPs and which are not held by FO MDPs.

The core idea behind the Deictic OO-MDP formalism is the notion of a deictic predicate. A deictic predicate is grounded only with respect to a single grounded reference object that must relate itself to non-grounded object classes. Therefore, deictic predicates are more expressive than propositions that may only depend on object classes, while being less expressive than first-order predicates that may depend on an arbitrary number of grounded objects.

3.1. Formalism

The Deictic OO-MDP framework uses the same schema state-space $\mathcal{S}$ as described in section 2.3 while deictic predicate preconditions are used to define the schema transition dynamics as described below. Let $\mathcal{A}$ be a set of actions. Then for each attribute $C.\alpha$ of a class C define a set of effects of size $K_{C.\alpha}$[4]:

$$\mathcal{E}_{C.\alpha} = \{e_i : Dom(C.\alpha) \rightarrow Dom(C.\alpha)\}_{i=1}^{K_{C.\alpha}},$$

Define a set of deictic predicates of size $D_{a,C.\alpha}$:

$$\mathcal{F}_{a,C.\alpha} = \{f_i : O[C] \times \mathcal{S} \rightarrow \mathfrak{B}\}_{i=1}^{D_{a,C.\alpha}},$$

where $O[C]$ is a set that contains objects with all possible attribute value assignments that are instances of C, and $\mathfrak{B} = \{0, 1\}$.

Then the probabilistic transition dynamics for a and $C.\alpha$ are defined by

$$\mathcal{P}_{a,C.\alpha} : \mathfrak{B}^{D_{a,C.\alpha}} \times \mathcal{E}_{C.\alpha} \rightarrow [0, 1].$$

The schema transition dynamics $\mathcal{P}$ is the set of transition dynamics for all actions and attributes,

$$\mathcal{P} = \{\mathcal{P}_{a,C.\alpha} | a \in \mathcal{A}, C \in \mathfrak{C}, C.\alpha \in Att(C)\}.$$

The schema reward dynamics are defined by

$$\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \{r \in \mathbb{R} | r \sim p(\cdot|s, a, s'), s \in \mathcal{S}, a \in \mathcal{A}, s' \in \mathcal{S}\},$$

where $p(\cdot|s, a, s')$ is a stationary reward distribution given $s \in \mathcal{S}, a \in \mathcal{A}$ and $s' \in \mathcal{S}$.

Given an object set O we can instantiate a grounded MDP $\mathcal{M}_{O,\rho} = (\mathcal{S}_O, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \rho)$. Then if the agent is currently in state $s \in \mathcal{S}_O$ and takes action a, the transition dynamics for $\mathcal{M}_{O,\rho}$ operate as follows: for each object o in s that is an instance of C and each attribute $C.\alpha$ we compute the Boolean truth values $\mathcal{B} = \{f_i(o, s)\}_{i=1}^{D_{a,C.\alpha}}$ for the deictic predicates in $\mathcal{F}_{a,C.\alpha}$. Then for an effect $e \in \mathcal{E}_{C.\alpha}$

[4] An effect set under the Deictic OO-MDP framework generalizes the notion of an effect type under the Propositional OO-MDP framework. For example, the relative effect type in section 2.4 can be written as an effect set $\mathcal{E}_{C.\alpha} = \{Rel_0, Rel_1, Rel_2, ...Rel_n\}$.

we compute $\mathcal{P}_{a,C.\alpha}(\mathcal{B}, e)$ which returns the probability of e occurring given $\mathcal{B}$. This implies a distribution over effects which in turn implies a distribution over the attribute values of o by applying the effect to $o.\alpha$ in s and obtaining $e(o.\alpha) = o.\alpha'$ in s'.

As an example, consider attribute *Taxi.x* and action *East* for the All-Passenger Any-Destination Taxi domain. We can define a set of relative effects $Rel_i(x) = x + i$ that produce a shift of i squares from the current location x, as well as a deictic predicate $Touch_E(taxi, s)$ that returns 1 if the *taxi* object has an object of class *Wall* one square east of it in s, otherwise 0. Then the transition dynamics can be described for any *taxi* object as

$$Touch_E(taxi, s) = 1 \implies taxi.x \leftarrow Rel_0(taxi.x)$$

with probability one and

$$Touch_E(taxi, s) = 0 \implies taxi.x \leftarrow Rel_1(taxi.x)$$

with probability one.

The key insight with Deictic OO-MDPs is that the parameters we pass to each deictic predicate in $\mathcal{F}_{a,C.\alpha}$ are a grounded deictic object o that must be an instance of C and s which is a state *of the schema*, not a grounded state. As a result, these deictic predicates may not refer to specific objects in s; however, they may relate o to object classes of the schema. For example, with $Touch_E(taxi, s)$ as defined above only *taxi* is grounded while we never refer to a grounded object in s.

Note that the Deictic OO-MDP formalism is more general than the Propositional OO-MDP formalism since a deictic predicate that does not make reference to its deictic object is simply a proposition. Consequently, any domain that is representable under the Propositional OO-MDP formalism is also representable under the Deictic OO-MDP formalism.

3.2. Resolving the limitations of propositional OO-MDPs

Deictic OO-MDPs are more expressive than Propositional OO-MDPs allowing us to compactly represent the transition dynamics for the All-Passenger Any-Destination Taxi and Sokoban domains. In particular, define the following effect sets:

- for attributes *Taxi.x* and *Taxi.y*: $Rel_i(x) = x + i$ for $i \in \{-1, 0, 1\}$ where x is an integer for attributes.
- for attributes *Passenger.in-taxi* and *Passenger.at-destination*: $SetBool_i(x) = x\mathbb{1}(i = 0) + (1 - x)\mathbb{1}(i = 1)$ for $i \in \{0, 1\}$ where $x \in \{0, 1\}$, where $\mathbb{1}$ is the indicator function.[5]

Define the following deictic predicates[6]:

- *AnyWallNorthOfTaxi*$(taxi, s)$: returns 1 if there is any object of class *Wall* one square north of *taxi* in s, and 0 otherwise. (f_1)
- *AnyWallEastOfTaxi*$(taxi, s)$: returns 1 if there is any object of class *Wall* one square east of $taxi$ in s, and 0 otherwise. (f_2)
- *AnyWallSouthOfTaxi*$(taxi, s)$: returns 1 if there is any object of class *Wall* one square south of $taxi$ in s, and 0 otherwise. (f_3)
- *AnyWallWestOfTaxi*$(taxi, s)$: returns 1 if there is object of class *Wall* one square west of $taxi$ in s, and 0 otherwise. (f_4)
- *AnyTaxiOnPassenger*$(passenger, s)$: returns 1 if there is any object of class *Taxi* on the same square as $passenger$ in s, and 0 otherwise. (f_5)
- *PassengerAtAnyDestination(passenger,s)*: returns 1 if *passenger.at-destination* is set to 1 in s, and 0 otherwise. (f_6)
- *AnyPassengerInAnyTaxi*$(passenger, s)$: returns 1 if any object of class *Passenger* has its *in-taxi* attribute is set to 1 in s, and 0 otherwise. (f_7)
- *PassengerInAnyTaxi*$(passenger, s)$: returns 1 if *passenger.in-taxi* is set to 1 in s, and 0 otherwise. (f_8)
- *AnyTaxiOnAnyDestination*$(destination, s)$: returns 1 if any object of class *Taxi* is on the same square as any object of class *Destination* in s, and 0 otherwise. (f_9)

Then Table 2 shows for each attribute and action the relevant preconditions and effects that describe the transition dynamics of the domain under the Deictic OO-MDP formalism.

Deictic OO-MDPs are also able to compactly represent the transition dynamics of the Sokoban domain without ambiguity. For a deictic *person* object there are four deictic predicates required when taking the action *East* that resolve the following questions: is there any object of class *Box* one square east of *person*? Is there any object of class *Box* two squares east of *person*? Is there any object of class *Wall* one square east of *person*? Is there any object of class *Wall* two squares east of *person*? Meanwhile for a deictic *box* object there are three deictic predicates required to resolve the questions: is there any object of class *Person* one square west of *box*? Is there any object of class *Wall* one square east of *box*? Is there any object of class *Box* one square east of *box*? More specifically, define the following deictic predicates:

[5] Qualitatively, *SetBool* allows us to keep or flip a Boolean value. For example, if $x = 0$ and we want to flip it to 1 we can call $SetBool_1(0) = (1 - 0)\mathbb{1}(i = 1) = 1$.

[6] The statements *AnyPassengerInAnyTaxi*$(passenger, s)$ and *AnyTaxiOnAnyDestination*$(destination, s)$ are actually propositions as they do not refer to a grounded *passenger* object. We could simplify notation by writing, for example, *AnyPassengerInAnyTaxi*(s). However, we include the *passenger* object to remain consistent with the notation described in section 3.1.

Table 2
Preconditions and effects for each action and attribute in the All-Passenger Any-Destination Taxi domain. We exclude *Wall* attributes because they never change.

Action	Attribute	Precondition	Effect
North	*Taxi.y*	$f_1 = 0$	Rel_1
North	*Taxi.y*	$f_1 = 1$	Rel_0
East	*Taxi.x*	$f_2 = 0$	Rel_1
East	*Taxi.x*	$f_2 = 1$	Rel_0
South	*Taxi.y*	$f_3 = 0$	Rel_{-1}
South	*Taxi.y*	$f_3 = 1$	Rel_0
West	*Taxi.x*	$f_4 = 0$	Rel_{-1}
West	*Taxi.x*	$f_4 = 1$	Rel_0
Pickup	*Passenger.in-taxi*	$f_5 = 1 \wedge f_6 = 0 \wedge f_7 = 0$	$SetBool_1$
Pickup	*Passenger.in-taxi*	$f_5 = 0 \vee f_6 = 1 \vee f_7 = 1$	$SetBool_0$
Dropoff	*Passenger.in-taxi*	$f_8 = 1 \wedge f_9 = 1$	$SetBool_1$
Dropoff	*Passenger.in-taxi*	$f_8 = 0 \vee f_9 = 0$	$SetBool_0$
Dropoff	*Passenger.at-destination*	$f_8 = 1 \wedge f_9 = 1$	$SetBool_1$
Dropoff	*Passenger.at-destination*	$f_8 = 0 \vee f_9 = 0$	$SetBool_0$

Table 3
Precondition and effects for action *East* and *Reset* in the Sokoban domain. We exclude *Wall* attributes since they cannot change.

Action	Attribute	Precondition	Effect
East	*Person.x*	$(\acute{f}_1 = 0 \wedge \acute{f}_3 = 0) \vee (\acute{f}_3 = 1$ $\wedge \acute{f}_2 = 0 \wedge \acute{f}_4 = 0)$	Rel_1
East	*Person.x*	$\acute{f}_1 = 1 \vee (\acute{f}_3 = 1 \wedge \acute{f}_4 = 1)$ $\vee(\acute{f}_3 = 1 \wedge \acute{f}_2 = 1)$	Rel_0
East	*Box.x*	$\acute{f}_5 = 1 \wedge \acute{f}_6 = 0 \wedge \acute{f}_7 = 0$	Rel_1
East	*Box.x*	$\acute{f}_5 = 0 \vee \acute{f}_6 = 1 \vee \acute{f}_7 = 1$	Rel_0
Reset	*Person.reset*	$\acute{f}_8 = 1$	$SetBool_1$

- *AnyWall1EastOfPerson*$(person, s)$: returns 1 if there is any object of class *Wall* one square east of *person* in state s, and 0 otherwise. $(\acute{f}_1)$
- *AnyWall2EastOfPerson*$(person, s)$: returns 1 if there is any object of class *Wall* two squares east of *person* in state s, and 0 otherwise. $(\acute{f}_2)$
- *AnyBox1EastOfPerson*$(person, s)$: returns 1 if there is any object of class *Box* one square east of *person* in state s, and 0 otherwise. $(\acute{f}_3)$
- *AnyBox2EastOfPerson*$(person, s)$: returns 1 if there is any object of class *Box* two squares east of *person* in state s, and 0 otherwise. $(\acute{f}_4)$
- *AnyPersonWestOfBox*(box, s): returns 1 if there is any object of class *Person* one square west of *box* in s, and otherwise 0. $(\acute{f}_5)$
- *AnyBoxEastOfBox*(box, s): returns 1 if there is any object of class *Box* one square east of *box* in s, and otherwise 0. $(\acute{f}_6)$
- *AnyWallEastOfBox*(box, s): returns 1 if there is any object of class *Wall* one square east of *box* in s, and otherwise 0. $(\acute{f}_7)$

Then Table 3 shows for each attribute and action the relevant preconditions and effects that describe the transition dynamics of the domain under the Deictic OO-MDP formalism for action *East*.

The actions *North*, *South* and *West* are analogous. In addition, we include a *reset* attribute for the *Person* class that activates when a *Reset* action is taken, along with the following deictic predicate.

- *ResetActivated*$(person, s)$: returns 1 if $person.reset = 1$ in s, otherwise 0. $(\acute{f}_8)$.

The *Reset* action immediately terminates the task, and its inclusion is necessary because in Sokoban it is possible to reach a deadlock state from which the task is no longer solvable.

3.3. Learning

Given a set of D deictic predicates we want to learn the transition dynamics for each action a and attribute $C.\alpha$. If the transition dynamics are deterministic this can be done using memorization with 2^D unique observations. However, this is prohibitive if D is large. As discussed in section 2.4 Propositional OO-MDPs introduce a learning algorithm called *DOORMAX* for deterministic transition dynamics that, under certain assumptions, has a KWIK-bound that is linear in D.

For *DOORMAX* to be correct, the transition dynamics for each action and attribute must be representable as a full binary tree with propositions at the non-leaf nodes and effects at the leaf nodes. Furthermore, each possible effect of an effect type can occur at most at one leaf node of the tree, except for a special effect called a *failure condition* that may occur at multiple leaf nodes. A failure

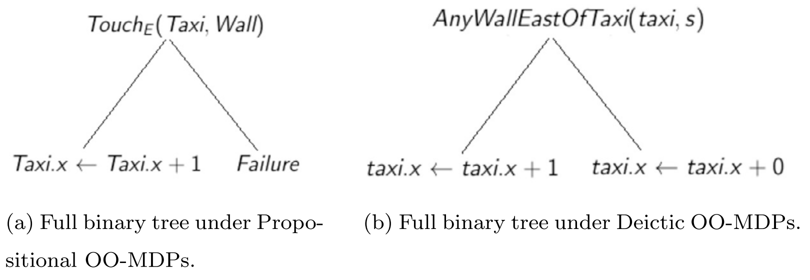

$Touch_E(Taxi, Wall)$
$Taxi.x \leftarrow Taxi.x + 1$
Failure
(a) Full binary tree under Propositional OO-MDPs.
$AnyWallEastOfTaxi(taxi, s)$
$taxi.x \leftarrow taxi.x + 1$
$taxi.x \leftarrow taxi.x + 0$
(b) Full binary tree under Deictic OO-MDPs.

Fig. 5. Full binary tree structure for the transition dynamics of $Taxi.x$ attribute and action *East*. Right branches represent a truth value of 1.

condition implies that globally no attribute changes when an action was taken i.e. $s = s'$ when a is taken. See Fig. 5a for how this represented for the *Taxi.x* attribute with action *East*.

The intuition behind *DOORMAX* is that, in many cases, the number of propositions that an effect depends on is much smaller than D. Furthermore, since an effect can occur at most once in the tree, we can invalidate multiple propositions with a single observation. To illustrate the core learning mechanism of *DOORMAX*, consider the original Taxi domain with action *East*, attribute *Taxi.x* and effect type Rel_i. We can propose four propositional statements $Touch_N(Taxi, Wall)$, $Touch_E(Taxi, Wall)$, $Touch_S(Taxi, Wall)$ and $Touch_W(Taxi, Wall)$. The aim of the KWIK-learner then is to learn that

$$Touch_E(Taxi, Wall) = 0 \implies Taxi.x \leftarrow Taxi.x + 1$$

when action *East* is taken. Suppose that agent interacts with its environment and takes action *East* to observe the effect: $Taxi.x + 1$. Prior to taking the action *East*, the agent's state determined the following assignments to the propositions:

$Touch_N(Taxi, Wall) = 0 \wedge Touch_E(Taxi, Wall) = 0 \wedge Touch_S(Taxi, Wall) = 0 \wedge Touch_W(Taxi, Wall) = 0$.

Suppose the next time the agent takes the action *East* and observes the effect $Taxi.x + 1$ and the agent's previous state determined the following assignments to the propositions:

$Touch_N(Taxi, Wall) = 1 \wedge Touch_E(Taxi, Wall) = 0 \wedge Touch_S(Taxi, Wall) = 1 \wedge Touch_W(Taxi, Wall) = 1$.

Since the *DOORMAX* algorithm assumes that each effect can occur at most once in the tree, we can infer that the propositions $Touch_N(Taxi, Wall)$, $Touch_S(Taxi, Wall)$ and $Touch_W(Taxi, Wall)$ are irrelevant to the precondition of the effect. As a result the agent has learned

$$Touch_E(Taxi, Wall) = 0 \implies Taxi.x \leftarrow Taxi.x + 1$$

with only two unique observations. Of course, this is a best-case scenario. A worst-case scenario requires 4 unique observations. In general, given $D \geq 0$ propositions, learning requires at most $D + 1$ unique observations.[7] As failure conditions can occur at multiple leaf nodes, they do not benefit from this efficient learning procedure and are learned inefficiently through memorization in *DOORMAX*.

We adapt the *DOORMAX* algorithm to Deictic OO-MDPs, which we call $DOORMAX_D$ (Algorithm 7). The $DOORMAX_D$ algorithm requires two sub-algorithms for the transition dynamics: one to learn and one to make predictions. These sub-algorithms are presented in this section while the full $DOORMAX_D$ algorithm that calls these sub-algorithms is then presented in section 5. The main difference between $DOORMAX_D$ and *DOORMAX* is that we remove the notion of a global failure condition. Instead we require that all effects apply to a single attribute. See Fig. 5b for how this is represented for the *Taxi.x* attribute with action *East*.

To achieve this, we extend the notion of an effect type under Propositional OO-MDP framework to an effect group under Deictic OO-MDP framework that includes a partition function over an effect set that groups them into those that can occur at most at one leaf node and those that can occur at multiple leaf nodes. For example, as shown in Fig. 5b, the effects $taxi.x \leftarrow taxi.x + 1$ (Rel_1) and $taxi.x \leftarrow taxi.x + 0$ (Rel_0) are both unique in the tree. Therefore, by using an appropriate partition function, we can efficiently learn both these branches. Meanwhile, the design choice used by *DOORMAX* of a global failure conditions implies that $Taxi.x \leftarrow Taxi.x + 1$ (Rel_1) is efficiently learned, while $Taxi.x \leftarrow Taxi.x + 0$ (Rel_0) is learned by memorization. A trade-off exists between these design choices. The $DOORMAX_D$ algorithm requires a partition function for each action, attribute and effect set that can improve learning efficiency. However, constructing such a partition function requires additional prior knowledge about a domain that is not required by *DOORMAX*. Similarly to *DOORMAX*, we require that all effects in an effect set are invertible under $DOORMAX_D$.

In this section we introduce two algorithms that are called by $DOORMAX_D$ to KWIK-learn the transition dynamics of an MDP under the Deictic OO-MDP formalism:

- *UpdateTree1* (Algorithm 1) updates the binary tree for action a and attribute $C.\alpha$.
- *Predict1* (Algorithm 2) makes a prediction for action a and attribute $C.\alpha$.

Before introducing the algorithms, we require some definitions. Let $\mathcal{F}$ be a set of deictic predicates and $\mathcal{E}$ be a set of effects.

[7] It is $D + 1$ and not D because an effect can depend on no propositions.

Definition 2. A term is a tuple (f, b) where $f \in \mathcal{F}$ and $b \in \mathfrak{B}$. A set of terms is denoted by $\mathcal{T}$. A set that contains sets of terms is denoted by $\mathfrak{T}$.

Definition 3. Term (f_1, b_1) mismatches term (f_2, b_2) if $f_1 = f_2$ and $b_1 \neq b_2$.

Definition 4. $\Pi : \mathcal{E} \to \mathfrak{B}$ is a binary partition function over $\mathcal{E}$ and assigns each effect in $\mathcal{E}$ to one of two partitions, 0 or 1. We call the tuple $g = (\mathcal{E}, \Pi)$ an effect group. Denote by K_0^g and K_1^g the number of effects in partition 0 and 1 respectively. We use . notation to refer to an element in a tuple g, so for example $g.\mathcal{E}$ refers to $\mathcal{E}$ in g. We denote sets of effect groups with $\mathcal{G}$.

Definition 5. Let g be an effect group. Let $M > 1$ be a constant. Then $\mathit{Tree}(g, \mathcal{F}, M)$ is the set of all full binary trees such that non-leaf nodes are elements of $\mathcal{F}$ and leaf nodes are elements of $g.\mathcal{E}$. Furthermore, if $g.\Pi$ assigns an effect in $g.E$ to partition 1 then that effect can occur at most at one leaf node and we call that effect conjunctive, otherwise that effect may occur at most at M leaf nodes and we call that effect disjunctive.

We now introduce the *UpdateTree1* (Algorithm 1) and *Predict1* (Algorithm 2) algorithms. The sets $\hat{\mathcal{F}}(a, C.\alpha)$, $\hat{\mathcal{G}}(a, C.\alpha)$, $\mathfrak{T}_e^g(a, C.\alpha)$ as well as M are initialized globally in Algorithm 5 as part of the full $\mathit{DOORMAX}_D$ algorithm that calls these sub-algorithms. The $\hat{}$ notation for $\hat{\mathcal{F}}(a, C.\alpha)$ and $\hat{\mathcal{G}}(a, C.\alpha)$ is used to emphasize that these sets are hypothesized upfront by the learner.

The set $\hat{\mathcal{F}}(a, C.\alpha)$ is initialized with hypothesized deictic predicates for action a and attribute $C.\alpha$, and does not change during learning. The set $\hat{\mathcal{G}}(a, C.\alpha)$ is initialized with hypothesized effect groups for action a and attribute $C.\alpha$, and invalid effect groups are removed from this set during learning. The set $\mathfrak{T}_e^g(a, C.\alpha)$ is initialized with the empty set, and is updated during learning with sets of terms that map to effect e of effect group g for action a and attribute $C.\alpha$.

Note that there are two equivalent views of the transition dynamics under $\mathit{DOORMAX}_D$ as the sets $\{\mathfrak{T}_e^g(a, C.\alpha)\}_{e \in g.\mathcal{E}}$, which are the data structures maintained in Algorithm 1, must at all times induce a valid binary tree in $\mathit{Tree}(g, \hat{\mathcal{F}}(a, C.\alpha), M)$. The exact conditions required for convergence of $\mathit{DOORMAX}_D$ is stated in Theorem 1.

At a high-level, the algorithm *UpdateTree1* ensures that for all $g \in \hat{\mathcal{G}}(a, C.\alpha)$ the constraint $\mathit{Tree}(g, \hat{\mathcal{F}}(a, C.\alpha), M)$ is maintained. That is, the set $\{\mathfrak{T}_e^g(a, C.\alpha)\}_{e \in g.\mathcal{E}}$ induces a binary tree subject to the constraints of the partition function $g.\Pi$. Each time we observe a set of terms $\mathcal{T}$ and an associated effect e we update $\mathfrak{T}_e^g(a, C.\alpha)$, and in doing so may discover that the resulting set $\{\mathfrak{T}_e^g(a, C.\alpha)\}_{e \in g.\mathcal{E}}$ can no longer induce an appropriate binary tree at which point we remove g from $\hat{\mathcal{G}}(a, C.\alpha)$.

Algorithm 1: *UpdateTree1*: update binary tree for action a and attribute $C.\alpha$.

```
   Input: $C.\alpha \in Att(C)$, $s \in \mathcal{S}$, $o \in O[C]$, $a \in \mathcal{A}$ , $o.\alpha' \in Dom(C.\alpha)$
1  pass $o$ and $s$ to the deictic predicates in $\hat{\mathcal{F}}(a, C.\alpha)$ to retrieve a set of terms $\mathcal{T}$
2  foreach $g \in \hat{\mathcal{G}}(a, C.\alpha)$ do
3      foreach $e \in g.\mathcal{E}$ do
4          if $e(o.\alpha) = o.\alpha'$ then
5              if $\mathfrak{T}_e^g(a, C.\alpha) = \phi$ then
6                  if $\exists e' \in g.\mathcal{E}$ with $\mathcal{T}_{e'} \in \mathfrak{T}_{e'}^g(a, C.\alpha)$ such that $\mathcal{T}_{e'} \subseteq \mathcal{T}$ then
7                      remove $g$ from $\hat{\mathcal{G}}(a, C.\alpha)$
8                  else
9                      add $\mathcal{T}$ to $\mathfrak{T}_e^g(a, C.\alpha)$
10                 end
11             else
12                 if $g.\Pi(e) = 0$ then
13                     add $\mathcal{T}$ to $\mathfrak{T}_e^g(a, C.\alpha)$
14                 else
15                     $\mathcal{T}_{temp} \leftarrow \mathcal{T}$
16                     $\mathcal{T} \leftarrow$ the only element in $\mathfrak{T}_e^g$
17                     remove from $\mathcal{T}$ any terms that mismatch terms in $\mathcal{T}_{temp}$
18                     $\mathfrak{T}_e^g(a, C.\alpha) \leftarrow \phi$
19                     add $\mathcal{T}$ to $\mathfrak{T}_e^g(a, C.\alpha)$
20                 end
21                 if $(\exists e' \in (g.\mathcal{E} - \{e\})$ with $\mathcal{T}_{e'} \in \mathfrak{T}_{e'}^g(a, C.\alpha)$ such that $(\mathcal{T} \subseteq \mathcal{T}_{e'}$ or $\mathcal{T}_{e'} \subseteq \mathcal{T}))$ or $|\mathfrak{T}_e^g(a, C.\alpha)| > M$ then
22                     remove $g$ from $\hat{\mathcal{G}}(a, C.\alpha)$
23                 end
24             end
25         end
26     end
27 end
```

At the lower-level, the algorithm *UpdateTree1* updates the binary tree for action a and attribute $C.\alpha$ given an object o from state s and the resulting attribute value $o.\alpha'$ in s'. The algorithm starts at line 1 where it computes $\mathcal{T}$, the terms for $\hat{\mathcal{F}}(a, C.\alpha)$. In lines 2-4 the algorithm iterates over each effect group $g \in \hat{\mathcal{G}}(a, C.\alpha)$ and each effect $e \in g.\mathcal{E}$ where it checks if e applied to $o.\alpha$ is equal to $o.\alpha'$. If it is, then it proceeds to update $\mathfrak{T}_e^g(a, C.\alpha)$ in lines 5-25.

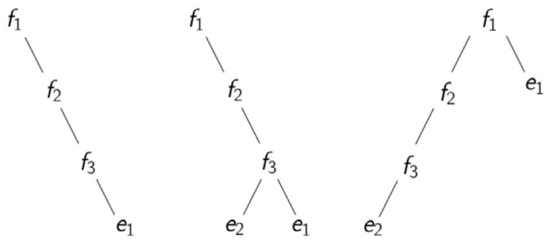


Fig. 6. Binary trees induced by $\mathfrak{T}_{e_1}$ and $\mathfrak{T}_{e_2}$. Right branches represent a truth value of 1.

If $\mathfrak{T}_e^g(a, C.\alpha)$ is empty, then as per lines 5-11, the algorithm checks if $\mathcal{T}$ already maps to different effect in g. If it does, then the effect group g is invalidated because adding $\mathcal{T}$ to $\mathfrak{T}_e^g(a, C.\alpha)$ would result in an invalid tree; otherwise, $\mathcal{T}$ is added to $\mathfrak{T}_e^g(a, C.\alpha)$.

If $\mathfrak{T}_e^g(a, C.\alpha)$ is not empty then, as per lines 12-20, it first checks if e is disjunctive or conjunctive based on the partition function $g.\Pi$. If e is disjunctive then it adds $\mathcal{T}$ to $\mathfrak{T}_e^g(a, C.\alpha)$. If e is conjunctive then it updates the existing element in $\mathfrak{T}_e^g(a, C.\alpha)$ by removing from it any terms that mismatch terms in $\mathcal{T}$ - this refines the set of terms that e depends on in the tree, and is the core mechanism for efficient learning in *DOORMAX*$_D$.

Finally, after the update to $\mathfrak{T}_e^g(a, C.\alpha)$, in lines 21-23 the algorithm checks if $\mathcal{T}$ maps to a different existing effect in g or if the number of effects in $\mathfrak{T}_e^g(a, C.\alpha)$ exceeds the limit M. In either of these cases, g is invalidated.

Note that in *DOORMAX*$_D$, only conjunctive effects are efficiently learned. Similar to failure conditions under *DOORMAX*, disjunctive effects are inefficiently learned through memorization. The parameter M restricts number of allowed occurrences for a disjunctive effect in a binary tree. However, a valid value for M in domains that require disjunctive effects under the Deictic OO-MDP formalism grows exponentially in the number of deictic predicates in order for the tree can capture all permutations of terms that cause the effect. Given no prior knowledge, a valid upper bound is $M = 2^{D_{max}}$ where

$$D_{max} = \max_{a \in \mathcal{A},\, C.\alpha \in Att(C)} D_{a,C.\alpha}.$$

To provide a better intuition on the invalidation logic of the algorithm, consider a case where $\mathcal{F} = \{f_1, f_2, f_3\}$ and there are two effects $\mathcal{E} = \{e_1, e_2\}$ where e_1 is conjunctive and e_2 is disjunctive. Consider the following examples:

- Fig. 6a: we currently have $\mathcal{T}_{e_1} = \{(f_1, 1), (f_2, 1), (f_3, 1)\}$ with $\mathfrak{T}_{e_1} = \{\mathcal{T}_{e_1}\}$ and $\mathfrak{T}_{e_2} = \phi$. Suppose we then observe e_2 with $\mathcal{T} = \{(f_1, 1), (f_2, 1), (f_3, 1)\}$. Then $\mathfrak{T}_{e_2}$ is empty and $\mathcal{T}_{e_1} \subseteq \mathcal{T}$.
- Fig. 6b: we currently have $\mathcal{T}_{e_1} = \{(f_1, 1), (f_2, 1), (f_3, 1)\}$ with $\mathfrak{T}_{e_1} = \{\mathcal{T}_{e_1}\}$ and $\mathcal{T}_{e_2} = \{(f_1, 1), (f_2, 1), (f_3, 0)\}$ with $\mathfrak{T}_{e_2} = \{\mathcal{T}_{e_2}\}$. Suppose we then observe e_1 with $\mathcal{T} = \{(f_1, 1), (f_2, 0), (f_3, 0)\}$. As e_1 is conjunctive and $\mathfrak{T}_{e_1}$ is not empty we first remove mismatching terms. Then $\mathcal{T} = \{(f_1, 1)\}$ with $\mathfrak{T}_{e_1} = \{\mathcal{T}\}$ and now $\mathcal{T} \subseteq \mathcal{T}_{e_2}$.
- Fig. 6c: we currently have $\mathcal{T}_{e_1} = \{(f_1, 1)\}$ with $\mathfrak{T}_{e_1} = \{\mathcal{T}_{e_1}\}$ and $\mathcal{T}_{e_2} = \{(f_1, 0), (f_2, 0), (f_3, 0)\}$ with $\mathfrak{T}_{e_2} = \{\mathcal{T}_{e_2}\}$. Suppose we then observe e_2 with $\mathcal{T} = \{(f_1, 1), (f_2, 0), (f_3, 1)\}$. We add $\mathcal{T}$ to $\mathfrak{T}_{e_2}$ and now $\mathcal{T}_{e_1} \subseteq \mathcal{T}$.

In all the above cases, we conclude that the effect group is invalid since the observed data can no longer induce a binary tree subject to the specified constraints. Note that we do not place any restrictions on the order in which the preconditions may appear in the tree, but there is no reordering that can recover an appropriate binary tree given the data.

Next, we discuss the *Predict1* algorithm. At a high-level the algorithm makes a prediction for attribute $C.\alpha$ of an object o of class C given a schema state s and action a. The algorithm requires that for each effect group $g \in \hat{\mathcal{G}}(a, C.\alpha)$ a prediction is made that is not $\perp$, and furthermore that all effect groups make the same prediction; otherwise the algorithm returns $\perp$. This ensures that the algorithm only makes a correct prediction if it does not return $\perp$. Once the transition dynamics are learned, then given a state s and action a the resulting state s' can be determined by calling *Predict1* for each object attribute $o.\alpha$ in s to obtain $o.\alpha'$ in s'. This algorithm is analogous to that used in *DOORMAX*.

In line 2 of the algorithm it computes $\mathcal{T}$, the terms for $\hat{\mathcal{F}}(a, C.\alpha)$. In line 3 the algorithm iterates over the effect groups $g \in \hat{\mathcal{G}}(a, C.\alpha)$. In lines 4-10 the algorithm checks whether any effect $e \in g.\mathcal{E}$ is mapped to by $\mathcal{T}$ in $\mathfrak{T}_e^g(a, C.\alpha)$. If it is, then the prediction $e(o.\alpha)$ is recorded and added to a set $\mathcal{V}$ in line 14; otherwise the algorithm returns $\perp$ in line 12. In lines 15-17 the algorithm checks if $|\mathcal{V}| > 1$. If it is, then there are two effect groups that make different predictions for $o.\alpha$ and the algorithm returns $\perp$; otherwise, all effect groups record the same prediction for $o.\alpha$ and the algorithm returns this prediction in line 20.

4. Transferable reward dynamics with propositional OO-MDPs

Propositional OO-MDPs introduce a KWIK-learning algorithm to learn the transition dynamics of a domain, while assuming known reward dynamics [13,14]. In this section we introduce an algorithm to KWIK-learn a family of reward dynamics under a propositional object-oriented approach.

Algorithm 2: *Predict1*: prediction procedure for action a and attribute $C.\alpha$.

```
Input: $C.\alpha \in Att(C)$, $s \in \mathcal{S}$, $o \in O[C]$, $a \in \mathcal{A}$
1  initialize an empty set $\mathcal{V}\langle Dom(C.\alpha)\rangle$
2  pass $o$ and $s$ to the deictic predicates in $\hat{\mathcal{F}}(a, C.\alpha)$ to retrieve a set of terms $\mathcal{T}$
3  foreach $g \in \hat{\mathcal{G}}(a, C.\alpha)$ do
4      $pred \leftarrow \perp$
5      foreach $e \in g.\mathcal{E}$ do
6          if $\exists \mathcal{T}_e \in \mathfrak{T}_e^g(a, C.\alpha)$ such that $\mathcal{T}_e \subseteq \mathcal{T}$ then
7              $pred \leftarrow e(o.\alpha)$
8              exit loop
9          end
10     end
11     if $pred = \perp$ then
12         return $\perp$
13     else
14         add $pred$ to $\mathcal{V}$
15         if $|\mathcal{V}| > 1$ then
16             return $\perp$
17         end
18     end
19 end
20 return only element in $\mathcal{V}$
```

This family consists of reward dynamics whereby for most transitions the agent receives a default reward signal, while a small number of transition groups lead to different reward signals. An example of a domain that exhibits such reward dynamics is the Taxi domain. In the Taxi domain, the agent receives a default reward of -1 for all transitions except for the groups of transitions that: apply an illegal pickup operation, which produce a reward signal of -10; apply an illegal dropoff operation, which produce a reward signal of -10; or lead to terminal state, which produces a reward signal of 20 [19]. Furthermore, it is required that these groups can be described under an object-oriented approach through propositional statements over object class attributes.

We note it is straightforward to extend the propositional formalism described in this section to a deictic formalism. However, it is not required for the domains considered in this paper.

4.1. Refactoring reward dynamics

We note that for any finite MDP, the reward dynamics can be refactored as a sum of $L+1$ terms:

$$\mathcal{R}(s,a,s') = \sum_{i=1}^{L} z_i(s,a,s')U_i + (1 - \sum_{i=1}^{L} z_i(s,a,s'))U_{L+1}, \tag{1}$$

where the $z_i(s,a,s') \in \mathfrak{B}$ are indicator variables such that $\sum_{i=1}^{L} z_i(s,a,s') \in \mathfrak{B}$ - that is, for any (s,a,s') at most one z_i has value one and all others have value zero - and U_i is a *reward token* that maps to a stationary reward distribution (or a scalar in the case of deterministic reward dynamics). We call U_{L+1} the *default reward token*.

Rewriting the reward dynamics in this way does not sacrifice generality since any arbitrary reward dynamics can be mapped to equation (1) by setting $L = |\mathcal{S}|^2|\mathcal{A}| - 1$ and mapping each $\mathcal{R}(s,a,s')$ to some U_i where $i \in [1 : L+1]$

However, for many tasks we can group transitions together and therefore define the reward dynamics with $L \ll |\mathcal{S}|^2|\mathcal{A}| - 1$. This is particularly evident in tasks with sparse rewards where the agent gets a constant default reward for almost all transitions in an MDP. For example, consider the original Taxi task. In this task the agent gets a default reward of -1 for all steps except for an illegal *Pickup* action, an illegal *Dropoff* action or for reaching a terminal state. Therefore, we can represent the reward dynamics for the entire Taxi domain with $L = 3$ given the following propositions as per Table 4:

- P_a: (s,a,s') is a transition where the *Pickup* action is applied illegally.
- P_b: (s,a,s') is a transition where the *Dropoff* action is applied illegally.
- P_c: (s,a,s') is a transition which reaches a terminal state.

Furthermore, under an object-oriented approach, each of the propositions P_a, P_b and P_c can be constructed from propositions over object class attributes as shown below:[8]

- P_a: *Pickup* action with $On(Taxi, Passenger) = 0 \vee On(Taxi, Passenger) = 1 \wedge Passenger.in\text{-}taxi = 1$.
- P_b: *Dropoff* action with $Passenger.in\text{-}taxi = 0 \vee Passenger.in\text{-}taxi = 1 \wedge On(Taxi, Destination) = 0$.

[8] These propositions are extracted from the state s of the transition tuple (s,a,s'). In general, the propositional statements for the reward dynamics can be extracted from s' as well. However, for this domain only (s,a) is needed to capture the reward dynamics.

Table 4
Expressing the reward dynamics for the Taxi domain as per equation (1) with $L = 3$.

Proposition	z_1	z_2	z_3	Reward token
$P_a = 1$	1	0	0	$U_1 = -10$
$P_b = 1$	0	1	0	$U_2 = -10$
$P_c = 1$	0	0	1	$U_3 = 20$
Default	0	0	0	$U_4 = -1$

- P_c: *Dropoff* action with $On(Taxi, Destination) = 1 \wedge Passenger.in\text{-}taxi = 1$.

4.2. Formalism

We enhance the Propositional OO-MDP formalism to include the representation of reward dynamics under equation (1). Given $L \in \mathbb{N}$, $a \in \mathcal{A}$ and $i \in [1 : L]$, define a set of propositions of size $D_{a,i}$:

$$\mathcal{F}_{a,i} = \{f_{a,i} : S \times S \to \mathfrak{B}\}_{i=1}^{D_{a,i}},$$

and a binary mapping function that is used determine whether reward token mapped to index i triggers given the truth values of the propositions in $\mathcal{F}_{a,i}$:

$$\mathcal{Z}_{a,i} : \mathfrak{B}^{D_{a,i}} \to \mathfrak{B}.$$

Define a set of $L + 1$ reward tokens:

$$\mathcal{U} = \{U_j\}_{j=1}^{L+1}.$$

Then the schema reward dynamics are defined by the set:

$$\mathcal{R} = \{\mathcal{F}_{a,i}, \mathcal{Z}_{a,i}, U_j | j \in [1 : L + 1], i \in [1 : L], a \in \mathcal{A}\}.$$

Then given a grounded MDP $\mathcal{M}_{O,\rho} = (\mathcal{S}_O, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \rho)$ the reward dynamics operate as follows: the agent is currently in state $s \in \mathcal{S}_O$, takes action a and transitions to state $s' \in \mathcal{S}_O$. For each $i \in [1 : L]$ compute the set of truth values for the propositional statements in $\mathcal{F}_{a,i}$ to get a set of binary values $\mathcal{B}_i = \{f_k(s, s')\}_{k=1}^{D_{a,i}}$. Compute $\mathcal{Z}_{a,i}(\mathcal{B}_i)$ to obtain some indicator variable z_i. The set of indicator variables is then $\mathcal{Z} = \{z_i\}_{i=1}^{L}$. The elements in the sets $\mathcal{Z}$ and $\mathcal{U}$ are then passed to equation (1) to compute a reward. Recall that equation (1) is restricted so that $\sum_{i=1}^{L} z_i \in \mathfrak{B}$. Therefore, if $\sum_{i=1}^{L} z_i = 1$ then the agent receives a reward generated from whichever reward token U_j is activated by $z_j = 1$, and if $\sum_{i=1}^{L} z_i = 0$ then the agent receives a reward generated from the default reward token U_{L+1}.

As a practical example of the formalism, consider the action *Dropoff* with $\mathcal{U} = \{U_1, U_2, U_3, U_4\}$ as described in Table 4. Suppose we have

$$\mathcal{F}_{Dropoff,2} = \mathcal{F}_{Dropoff,3} = \{AnyTaxiOnAnyDestination(s, s'), AnyPassengerInAnyTaxi(s, s')\}$$

that are defined as:[9]

- $AnyTaxiOnAnyDestination(s, s')$: returns 1 if any objects of class *Taxi* is on the same square as any object of class *Destination* in s, and otherwise 0.
- $AnyPassengerInAnyTaxi(s, s')$: returns 1 if any objects of class *Passenger* has their *in-taxi* attribute set to 1 in s, and otherwise 0.

Given assignments $\{b_1, b_2\}$ to these propositional statements the mapping functions would be defined as:

- $\mathcal{Z}_{Dropoff,2}$: return 1 if $(b_2 = 0)$ or $(b_2 = 1$ and $b_1 = 0)$, else return 0.
- $\mathcal{Z}_{Dropoff,3}$: return 1 if $(b_1 = 1$ and $b_2 = 1)$, else return 0.

Note that we would simply define $\mathcal{F}_{Dropoff,1} = \phi$ and $\mathcal{Z}_{Dropoff,1}$ would always return a value of 0. This is because an illegal pickup is not possible when the action *Dropoff* is taken.

Suppose that the agent is currently in state s, takes action *Dropoff* and the resulting state is s'. The states s and s' are passed to $AnyTaxiOnAnyDestination(s, s')$ and $AnyPassengerInAnyTaxi(s, s')$ to obtain their respective truth values. Suppose that the resulting truth values are:

[9] We can actually simplify $AnyTaxiOnAnyDestination(s, s')$ and $AnyPassengerInAnyTaxi(s, s')$ to $AnyTaxiOnAnyDestination(s)$ and $AnyPassengerInAnyTaxi(s)$ respectively because these propositions do not depend on s'. However, we include s' to remain consistent with the notation of the formalism.

Table 5
Reward tokens that activate for each action and precondition in the All-Passenger Any-Destination Taxi domain. Any other combination leads to the default reward token $\bar{U}_4 = -1$.

Action	Precondition	Reward token
Pickup	$p_1 = 0 \vee (p_1 = 1 \wedge p_2 = 1)$	$\bar{U}_1 = -10$
Dropoff	$p_2 = 0 \vee (p_2 = 1 \wedge p_3 = 0)$	$\bar{U}_2 = -10$
Dropoff	$p_4 = 1$	$\bar{U}_3 = 0$

Table 6
Reward tokens that activate for each action and precondition in the Sokoban domain. Any other combination leads to the default reward token $\acute{U}_3 = -1$.

Action	Precondition	Reward token
North	$\acute{p}_1 = 1$	$\acute{U}_1 = 300$
East	$\acute{p}_1 = 1$	$\acute{U}_1 = 300$
South	$\acute{p}_1 = 1$	$\acute{U}_1 = 300$
West	$\acute{p}_1 = 1$	$\acute{U}_1 = 300$
Reset	$\acute{p}_2 = 1$	$\acute{U}_2 - 1$

- $\{1, 0\}$: then $z_2 = 1$ and $z_3 = 0$, so the agent receives a reward from the reward token $U_2 = -10$.
- $\{1, 1\}$: then $z_2 = 0$ and $z_3 = 1$, so the agent receives a reward from the reward token $U_3 = 20$.

4.3. Examples

The reward dynamics for the All-Passenger Any-Destination Taxi domain require $L = 3$ reward tokens in addition to the default reward token. Define the following propositions:

- *AnyTaxiOnAnyPassenger*(s, s'): returns 1 if any objects of class *Taxi* is on the same square as any object of class *Passenger* in s, and otherwise 0. (p_1)
- *AnyPassengerInAnyTaxi*(s, s'): returns 1 if any objects of class *Passenger* has their *in-taxi* attribute set to 1 in s, and otherwise 0. (p_2)
- *AnyTaxiOnAnyDestination*(s, s'): returns 1 if any objects of class *Taxi* is on the same square as any object of class *Destination* in s, and otherwise 0. (p_3)
- *AllPassenagersAtAnyDestination*(s, s'): returns 1 if all objects of class *Passenger* have their *at-destination* attributes set to 1 in s', and otherwise 0. (p_4)

Then Table 5 shows the reward token that activates for each action and precondition.
For the Sokoban domain we set $L = 2$ and define the following propositions:

- *AllBoxesAtAnyStorage*(s, s'): returns 1 if all objects of class *Box* is on the same square as any object of class *Storage* in s', and otherwise 0. ($\acute{p}_1$)
- *ResetActivated*(s, s'): returns 1 if any object of class *Person* has their *reset* attribute set to 1 in s', otherwise 0. ($\acute{p}_2$)

Then Table 6 shows the reward token that activates for each action and precondition.[10]

4.4. Learning

In this section we propose an algorithm to KWIK-learn the mapping functions $\mathcal{Z}_{a,i}$ described in section 4.2 given a set of propositions $\mathcal{F}_{a,i}$ for $i \in [1 : L]$. The procedure for learning these mapping functions is analogous to learning transition dynamics as described in section 3.3.

When learning transition dynamics, we use a full binary tree structure for each action and attribute with logical statements at the leaf nodes and effects at the non-leaf nodes. For learning the mapping functions, we will also use a full binary tree structure with logical statements, in the form of propositions, at the leaf nodes; but now the non-leaf nodes are binary indicators. Each such tree is

[10] These reward dynamics encourage the agent to get all boxes to a storage location in as few steps as possible. The high reward of 300 for achieving this is necessary so that the agent does not learn an optimal policy that applies the *Reset* action prematurely.

used to indicate if a particular reward token is activated for a given transition. Therefore, we learn a full binary tree for each action $a \in \mathcal{A}$ and $i \in [1 : L]$.

Learning the mapping functions no longer requires the notion of an effect group, since the trees have binary leaf nodes. Therefore, in this setting, the partition function is $\Pi : \mathfrak{B} \to \mathfrak{B}$ that maps which of the binary indicator values is conjunctive and disjunctive in the tree. In a similar way to learning transition dynamics, the key to achieving efficiency under this framework is through the notion of conjunctive and disjunctive indicator values. That is, if an indicator value for action a and index i is known to occur at most at one non-leaf node in a tree then that branch can be KWIK-learned with at most $D_{a,i} + 1$ unique observations.

Returning to the example of the Taxi task with reward dynamics as described in Table 4, $z_3 = 1$ can be efficiently learned because the precondition that maps to the activation of U_3 is the conjunction: $\mathit{On}(\mathit{Taxi}, \mathit{Destination}) = 1 \wedge \mathit{Passenger.in\text{-}taxi} = 1$. Meanwhile, $z_1 = 1$ and $z_2 = 1$ must be learned through memorization because they occur at two leaf nodes in the tree. However, $z_1 = 0$ and $z_2 = 0$ can be efficiently learned since they occur when

$$\mathit{On}(\mathit{Taxi}, \mathit{Passenger}) = 1 \wedge \mathit{Passenger.in\text{-}taxi} = 0$$

and

$$\mathit{On}(\mathit{Taxi}, \mathit{Destination}) = 1 \wedge \mathit{Passenger.in\text{-}taxi} = 1$$

respectively.

We further assume that each U_j is KWIK-learnable with KWIK-bound B_j for $j \in [1 : L+1]$. Under this assumption we can then KWIK-learn the complete reward dynamics with the algorithms described in this section. We introduce two algorithms that are called by the $\mathit{DOORMAX}_D$ algorithm presented in section 5 to KWIK-learn the reward dynamics for an MDP under the Propositional OO-MDP formalism.

- *UpdateTree2* (Algorithm 3) updates the binary tree for action a and reward token mapped to index i.
- *Predict2* (Algorithm 4) makes a binary prediction for action a and reward token mapped to index i.

The sets $\hat{\mathcal{F}}(a,i)$, $\mathfrak{T}_b(a,i)$ as well as $\Pi(a,i)$ are initialized globally in Algorithm 5 as part of the full $\mathit{DOORMAX}_D$ algorithm. The set $\hat{\mathcal{F}}(a,i)$ is initialized with hypothesized propositions for action a and reward token mapped to index i, and does not change during learning. The set $\mathfrak{T}_b(a,i)$ is initialized with the empty set, and is updated during learning with sets of terms that map to the binary indicator value b for action a and reward token mapped to index i. Meanwhile, $\Pi(a,i)$ is initialized as a partition function for action a and reward token mapped to index i that determines if an indicator value is conjunctive or disjunctive in the tree induced by $\{\mathfrak{T}_b(a,i)\}_{b\in\mathfrak{B}}$.

The algorithm *UpdateTree2* operates in an analogous manner to the *UpdateTree1* algorithm presented in section 3.3. At a high-level, the algorithm ensures that the set $\{\mathfrak{T}^g_b(a,i)\}_{b\in\mathfrak{B}}$ at all times induces a binary tree subject to the constraints of the partition function $\Pi(a,i)$. Each time it observes a set of terms $\mathcal{T}$ and an associated indicator value z it updates $\mathfrak{T}^g_z(a,i)$ to refine tree.

At the lower-level the algorithm *UpdateTree2* updates the binary tree for action a and reward token mapped on index i given a schema state s, a resulting schema state s' and a binary indicator value z. The algorithm starts at line 1 where it computes $\mathcal{T}$, the terms for $\hat{\mathcal{F}}(a,i)$. In lines 2-3 the algorithm checks if $\mathfrak{T}_z(a,i)$ is empty, and if it is it adds $\mathcal{T}$ to $\mathfrak{T}_z(a,i)$. If $\mathfrak{T}_z$ is not empty then, as per lines 5-13, it checks if z is disjunctive or conjunctive based on the partition function $\Pi(a,i)$. If z is disjunctive then it adds $\mathcal{T}$ to $\mathfrak{T}_z(a,i)$; otherwise, if z is conjunctive then it updates the only element in $\mathfrak{T}_z(a,i)$ by removing from it any terms that mismatch terms in $\mathcal{T}$.

Algorithm 3: *UpdateTree2*: update binary tree for action a and reward token mapped to index i.

```
   Input: $i \in [1 : L]$, $s \in \mathcal{S}$, $a \in \mathcal{A}$, $s' \in \mathcal{S}$, $z \in \mathfrak{B}$
1  pass $s$ and $s'$ to the propositions in $\hat{\mathcal{F}}(a,i)$ to retrieve a set of terms $\mathcal{T}$
2  if $\mathfrak{T}_z(a,i) = \phi$ then
3      add $\mathcal{T}$ to $\mathfrak{T}_z(a,i)$
4  else
5      if $\Pi(a,i)(z) = 0$ then
6          add $\mathcal{T}$ to $\mathfrak{T}_z(a,i)$
7      else
8          $\mathcal{T}_{temp} \leftarrow \mathcal{T}$
9          $\mathcal{T} \leftarrow$ the only element in $\mathfrak{T}_z(a,i)$
10         remove from $\mathcal{T}$ any terms that mismatch terms in $\mathcal{T}_{temp}$
11         $\mathfrak{T}_z(a,i) \leftarrow \phi$
12         add $\mathcal{T}$ to $\mathfrak{T}_z(a,i)$
13     end
14 end
```

The algorithm *Predict2* takes as input a reward token index i, schema state s, action a and resulting schema state s'. The algorithm then returns a binary value indicating if the reward token mapped to index i activates, or $\perp$ if the algorithm cannot yet make an accurate prediction. Once the reward dynamics are learned, then given a state s, action a and resulting state s', the reward token that activates can be determined by calling *Predict2* for each $i \in [1 : L]$ and observing which of these returns a value of 1, or if they are all 0 then activating the default reward token. In line 1 the algorithm computes $\mathcal{T}$, the terms for $\hat{\mathcal{F}}(a,i)$. In lines 2-6 it checks whether any binary indicator value b is mapped to by $\mathcal{T}$ in $\mathfrak{T}_b(a,i)$. If it is, then b is returned; otherwise, the algorithm returns $\perp$ in line 7.

Algorithm 4: *Predict2*: prediction procedure for action a and reward token mapped to index i.

```
    Input: $i \in [1 : L]$, $s \in \mathcal{S}$, $a \in \mathcal{A}$, $s' \in \mathcal{S}$
1   pass $s$ and $s'$ to the propositions in $\hat{\mathcal{F}}(a,i)$ to retrieve a set of terms $\mathcal{T}$
2   foreach $b \in \mathfrak{B}$ do
3       if $\exists \mathcal{T}_b \in \mathfrak{T}_b(a,i)$ such that $\mathcal{T}_b \subseteq \mathcal{T}$ then
4           return $b$
5       end
6   end
7   return $\perp$
```

An important point to emphasize with regards to the *UpdateTree2* and *Predict2* algorithms is that they take the index of reward token i as one of their inputs. This information is assumed to come from the environment that the agent interacts with as per line 9 of the $DOORMAX_D$ algorithm in section 5. This additional knowledge is uncharacteristic of standard RL, where the agent only receives a scalar reward at each timestep. This assumption is necessary because the algorithm needs to know which of the trees to update with an indicator value of 1 and 0 when learning reward dynamics.

This additional knowledge inherently assumes that the agent has a built-in notion of 'reward categorization'. The idea of expanding the environment reward in RL to more than just a scalar has previously been introduced and shown to have benefits both in terms of learning performance and interpretability [26–28]. For example, by assuming that the agent receives a reward vector from its environment, where each element in the vector represents a reward type that encourages a specific behaviour (such as speed or safety), it is possible for an RL agent to learn a multi-dimensional policy that can execute any of the desired behaviours [26]. Such an approach assumes that the agent can categorize rewards, as it is able to perceive a reward vector that comprises of different reward types from its environment.

5. The $DOORMAX_D$ algorithm

In this section we combine the algorithms in sections 3 and 4 to present the full $DOORMAX_D$ algorithm, while also providing formal guarantees of $DOORMAX_D$ efficiency for the algorithm.

5.1. Algorithms

This section introduces the following algorithms / procedures:

- *InitializeGlobal* (Algorithm 5) initializes the global data structures needed by the procedures called by $DOORMAX_D$.
- *BuildFullModels* (Algorithm 6) builds the models $\mathcal{P}_{model}$ and $\mathcal{R}_{model}$.
- $DOORMAX_D$ (Algorithm 7) is the full $DOORMAX_D$ algorithm.

Note that the inputs to $DOORMAX_D$ are assumed to be globally accessible to all the procedures called by the main algorithm.

The procedure *InitializeGlobal* initializes the data structures and variables required by $DOORMAX_D$. In lines 1-8 the procedure initializes the $\hat{\mathcal{F}}(a, C.\alpha)$, $\hat{\mathcal{G}}(a, C.\alpha)$ and $\mathfrak{T}_e^g(a, C.\alpha)$ data structures as well as the variable M required for learning transition dynamics. In lines 9-14 it initializes the $\hat{\mathcal{F}}(a,i)$ and $\mathfrak{T}_b(a,i)$ data structures as well as the partition functions $\Pi(a,i)$ and reward tokens $\hat{U}_j$ for learning reward dynamics.

The algorithm *BuildFullModels* builds the full models $\mathcal{P}_{model}$ and $\mathcal{R}_{model}$. The algorithm starts at lines 1-2 where it initializes the models to the most optimistic case where every transition returns r_{max}. In line 3 the algorithm iterates over all tuples $(s,a) \in \mathcal{S}_O \times \mathcal{A}$ of the grounded MDP. In lines 4-15 the algorithms call *Predict1* for every object attribute value $o.\alpha$ in s and checks whether it can accurately predict the object attribute value $o.\alpha'$ when taking action a in order to construct the next state s'. If it can, then the algorithm proceeds to lines 16-25 where it calls *Predict2* on the tuple (s,a,s') for every reward token index $i \in [1 : L]$. The algorithm checks if it can make an accurate prediction for every $i \in [1 : L]$. If it can, the algorithm proceeds to lines 26-31 where it selects the activated reward token U_j. If U_j has been KWIK-learned, then the algorithm sets $\mathcal{P}_{model}(s,a)$ to s' and $\mathcal{R}_{model}(s,a,s')$ to $\hat{U}_j$.

The $DOORMAX_D$ algorithm is the root algorithm of this paper. In line 1 it calls *InitializeGlobal* to initialize the required data structures and variables. A start state s of the MDP is initialized in line 3 and the agent-environment interaction starts at line 4 with a while loop that ends when a terminal state is reached. In line 5 the models $\mathcal{P}_{model}$ and $\mathcal{R}_{model}$ are built by calling *BuildFullModels*. In line 6 the algorithm constructs an MDP $\hat{M}$ with transition dynamics $\mathcal{P}_{model}$ and reward dynamics $\mathcal{R}_{model}$. In line 7 the algorithm computes an optimal policy $\hat{\pi}_*$ for $\hat{M}$ by running an exact planning algorithm. In lines 8-9 the algorithm selects an action from $\hat{\pi}_*$ to observe a next state s', reward r and reward token index j from the environment. In lines 10-15 the algorithm calls *UpdateTree1* to update the appropriate binary trees for every $o.\alpha'$ in s'. In lines 16-22 the algorithm calls *UpdateTree2* to update the appropriate

Algorithm 5: *InitializeGlobal*: initialize global variables and data structures required for $DOORMAX_D$.

```
1  foreach $C \in \mathfrak{C}$ do
2      foreach $(a, C.\alpha) \in \mathcal{A} \times Att(C)$ do
3          initialize $\hat{\mathcal{F}}(a, C.\alpha) \leftarrow \{f_1, f_2, ..., f_{D_{a,C.\alpha}}\}$ globally for action $a$ and attribute $C.\alpha$ where each $f \in \hat{\mathcal{F}}(a, C.\alpha)$ is an hypothesized deictic predicates
4          initialize $\hat{\mathcal{G}}(a, C.\alpha) \leftarrow \{g_1, g_2, ..., g_{N_{a,C.\alpha}}\}$ globally for action $a$ and attribute $C.\alpha$ where each $g \in \hat{\mathcal{G}}(a, C.\alpha)$ is an hypothesized effect group; each effect group $g = (\mathcal{E}, \Pi)$ is initialized with i) an effect set $\mathcal{E} \leftarrow \{e_1, e_2, ...e_{K_{C.\alpha}}\}$ where each $e \in \mathcal{E}$ is an effect and ii) a binary partition function $\Pi$
5          foreach $g \in \hat{\mathcal{G}}(a, C.\alpha)$ do
6              foreach $e \in g.\mathcal{E}$ do
7                  initialize $\mathfrak{T}_e^g(a, C.\alpha) \leftarrow \phi$ globally as the set of sets for action $a$, attribute $C.\alpha$ and effect $e$
8  initialize $M > 1$ such that $M$ is the maximum number of elements allowed in any set $\mathfrak{T}_e^g(a, C.\alpha)$
9  foreach $(a, i) \in \mathcal{A} \times [1 : L]$ do
10     initialize $\hat{\mathcal{F}}(a, i) \leftarrow \{p_1, p_2, ...p_{D_{a,i}}\}$ globally for action $a$ and index $i$ mapped to $z_i$ where each $p \in \hat{\mathcal{F}}(a, i)$ is an hypothesized proposition
11     initialize $\mathfrak{T}_b(a, i) \leftarrow \phi$ globally as the set of sets for action $a$, index $i$ mapped to $z_i$ and Boolean $b$
12     initialize $\Pi(a, i)$ globally as a binary partition function for action $a$ and index $i$ mapped to $z_i$
13 foreach $j \in [1 : L + 1]$ do
14     initialize $\hat{U}_j$ globally as the estimated reward token mapped to index $j$
```

Algorithm 6: *BuildFullModels*: build $\mathcal{P}_{model}$ and $\mathcal{R}_{model}$ for $DOORMAX_D$.

```
1  initialize $\mathcal{P}_{model}(s, a) \leftarrow \perp$ for all $(s, a) \in \mathcal{S}_O \times \mathcal{A}$
2  initialize $\mathcal{R}_{model} \leftarrow r_{max}$ for all $(s, a, s') \in \mathcal{S}_O \times \mathcal{A} \times \mathcal{S}_O$
3  foreach $(s, a) \in \mathcal{S}_O \times \mathcal{A}$ do
4      $s' \leftarrow s$
5      $allPredKnown1 \leftarrow 1$
6      foreach object $o$ in $s'$ do
7          foreach attribute $C.\alpha \in Att(C)$ from the class $C$ of object $o$ do
8              $pred \leftarrow Predict1(C.\alpha, s, o, a)$
9              if $pred = \perp$ then
10                 $allPredKnown1 \leftarrow 0$
11             else
12                 replace attribute $C.\alpha$ of object $o$ in $s'$ with $pred$
13             end
14         end
15     end
16     if $allPredKnown1 = 1$ then
17         $allPredKnown2 \leftarrow 1$
18         $j \leftarrow L + 1$
19         foreach $i \in [1 : L]$ do
20             $pred \leftarrow Predict2(i, s, a, s')$
21             if $pred = \perp$ then
22                 $allPredKnown2 \leftarrow 0$
23             else if $pred = 1$ then
24                 $j \leftarrow i$
25         end
26         if $allPredKnown2 = 1$ then
27             if $\hat{U}_j$ with KWIK-bound $B_j$ has been KWIK-learned then
28                 $\mathcal{P}_{model}(s, a) \leftarrow s'$
29                 $\mathcal{R}_{model}(s, a, s') \leftarrow \hat{U}_j$
30             end
31         end
32     end
33 end
34 return $(\mathcal{P}_{model}, \mathcal{R}_{model})$
```

binary trees for every reward token index $i \in [1 : L]$. In lines 23-25 the algorithm updates $\hat{U}_j$ with r if $\hat{U}_j$ is not yet KWIK-learned. The loop ends after line 26 by setting the current state s to the next state s' for the start of the next iteration.

Note that while the formalisms presented in sections 3.1 and 4.2 allows for stochastic transition and reward dynamics, for $DOORMAX_D$ to be correct we require transition dynamics to be deterministic and reward dynamics that may be stochastic, so long as they are KWIK-learnable. We provide a summary of all prior knowledge and assumptions required for $DOORMAX_D$ to be correct in Appendix A.

Algorithm 7: *DOORMAX*$_D$: the *DOORMAX*$_D$ algorithm for episodic tasks.

```
Input: $\mathfrak{C}$, $L$, $\mathcal{S}_O$, $\mathcal{A}$, $\gamma$, $r_{max}$, $\rho$
InitializeGlobal()
repeat
    start episode at initial state $s \in \mathcal{S}_O$ sampled from $\rho$
    while $s \notin \mathcal{S}_{terminal}$ do
        $(\mathcal{P}_{model}, \mathcal{R}_{model}) \leftarrow$ BuildFullModels()
        initialize an MDP $\hat{M} = (\mathcal{S}_O, \mathcal{A}, \mathcal{P}_{model}, \mathcal{R}_{model}, \gamma)$
        compute an optimal policy $\hat{\pi}_*$ for $\hat{M}$ using exact planning algorithm
        choose action $a \in \mathcal{A}(s)$ from $\hat{\pi}_*$
        observe some next state $s'$ and reward $r$ generated from reward token $j$
        foreach object $o$ in $s$ do
            foreach attribute $C.\alpha \in Att(C)$ from the class $C$ of object $o$ do
                $o.\alpha' \leftarrow$ value of attribute $C.\alpha$ for object $o$ in $s'$
                UpdateTree1($C.\alpha, s, o, a, o.\alpha'$)
            end
        end
        foreach $i \in [1 : L]$ do
            $z \leftarrow 0$
            if $i = j$ then
                $z \leftarrow 1$
            end
            UpdateTree2($i, s, a, s', z$)
        end
        if $\hat{U}_j$ with KWIK-bound $B_j$ has not been KWIK-learned then
            update $\hat{U}_j$ with $r$
        end
        $s \leftarrow s'$
    end
until forever
```

6. Theory

In this section we present a theorem that the learning algorithm presented in section 3.3 KWIK-learns the transition dynamics for action a and attribute $C.\alpha$, and give the resulting KWIK-bound. The proof of the theorem is analogous to that of *DOORMAX* [13,14] and is left in Appendix B.

Theorem 1 *(KWIK-bound under DOORMAX$_D$). For action a and attribute $C.\alpha$ let $\hat{F}$ be a set of size D that contains hypothesized deictic predicate preconditions on which the transition dynamics of $C.\alpha$ when taking action a may depend. Let $\hat{\mathcal{G}} = \{g_i\}_{i=1}^{N}$ be a set of size N that contains hypothesized effect groups where each $e \in g_i.E$ has domain $Dom(C.\alpha)$. Let $K_0 = \max_{g \in \hat{\mathcal{G}}} K_0^g$ and $K_1 = \max_{g \in \hat{\mathcal{G}}} K_1^g$. Let $\mathcal{H} = \{Tree(g, \hat{F}, M) | g \in \hat{\mathcal{G}}\}$ for some constant $M > 1$. Then if some $h^* \in \mathcal{H}$ is true, the transition dynamics for a and $C.\alpha$ can be KWIK-learned under the DOORMAX$_D$ algorithm with KWIK-bound $N(K_0 M + K_1(D+1) + 1) + N - 1$.*

The algorithm learns the true transition dynamics, h^*, represented as a binary tree for action a and attribute $C.\alpha$. Since this the *schema* transition dynamics, it can be zero-shot transferred across all tasks of the schema once fully learned.

7. Experiments

In this section we conduct experiments to illustrate the benefits of the object-oriented frameworks presented in this paper. In section 7.1 we conduct experiments on the All-Passenger Any-Destination Taxi domain to efficiently learn transition dynamics, under the assumption of known reward dynamics. In section 7.2 we conduct experiments on the All-Passenger Any-Destination Taxi domain to efficiently learn reward dynamics, under the assumption of known transition dynamics. In section 7.3 we demonstrate that the reward dynamics and transition dynamics can be efficiently learned together for the Sokoban domain. In all our experiments we assume no prior knowledge of M and set $M = 2^{D_{max}}$. As our experiments are run on domains with deterministic reward dynamics, we have $B_j = 1$ for $j \in [1 : L+1]$.

7.1. All-Passenger Any-Destination Taxi domain: learning transition dynamics

We conduct two sets of experiments on the All-Passenger Any-Destination Taxi domain under the assumption that reward dynamics are known while transition dynamics need to be learned.[11] In the first set of experiments we have one destination and we fix the

[11] The source code for our implementation is available on GitHub [https://github.com/OfirMarom/DeicticOOMDPs].

Table 7
Hypothesized deictic predicates / propositions for each action and attribute as well as conjunctive effects. Any other action and attribute uses the hypothesis $\{P_1, P_2, P_3, P_4, P_5, P_6, P_7\}$ with all effects being conjunctive. Integer attributes use *Rel* effects while Boolean attributes use *SetBool* effects.

Action	Attribute	Hypothesis ($\hat{F}(a, C.\alpha)$)	Conjunctive
Any	*Taxi.x*	$\{f_1, f_2, f_3, f_4, P_5, P_6, P_7\}$	$\{-1, 0, 1\}$
Any	*Taxi.y*	$\{f_1, f_2, f_3, f_4, P_5, P_6, P_7\}$	$\{-1, 0, 1\}$
Pickup	*Passenger.in-taxi*	$\{f_5, f_6, P_1, P_2, P_3, P_4, P_7\}$	$\{1\}$
Dropoff	*Passenger.in-taxi*	$\{f_8, P_1, P_2, P_3, P_4, P_5, P_6\}$	$\{1\}$
Dropoff	*Passenger.at-destination*	$\{f_8, P_1, P_2, P_3, P_4, P_5, P_6\}$	$\{1\}$

number of passengers, n. We generate a grounded MDP with an initial state by randomly sampling n passenger locations and one destination location from one of six pre-specified locations and we also sample a random taxi start location together with one of four wall configurations as shown in Fig. 3a.

We apply 20 independent runs of the following procedure: we sample 10 test MDPs with random initial states. We then randomly sample a training MDP and run $DOORMAX_D$ on it for one episode until we reach a terminal state. Upon termination, we test performance by running $DOORMAX_D$ for one episode on each of the 10 test MDPs, stopping an episode early if we exceed 500 steps. We repeat this for 100 training MDPs. Since all the MDPs come from the same schema we can share transition dynamics between our MDPs - but we only update the transition dynamics on training MDPs.

In our experiments we start with $n = 1$ passenger and incrementally increase to $n = 4$ passengers. We run our experiments for Propositional OO-MDPs and two versions of Deictic OO-MDPs. In the first, without transfer, we relearn the transition dynamics for each n while for the second, with transfer, we transfer the previously learned transition dynamics each time we increase n. We report results in Fig. 7 that averages over the 20 independent runs the average number of steps for the 10 test MDPs with error bars included.

We use the hypothesis space as per Table 7 in our experiments, where P_1, P_2, P_3, P_4 and P_5 are propositions as defined as below. This hypothesis space is chosen so as to mimic the experimental setup of Propositional OO-MDPs on the original Taxi domain as closely as possible [13].

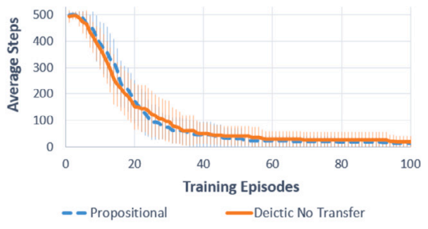
- $Touch_N(Taxi, Wall)$: returns 1 if any object of class *Taxi* has an object of class *Wall* one square north of it, and 0 otherwise. (P_1)
- $Touch_E(Taxi, Wall)$: returns 1 if any object of class *Taxi* has an object of class *Wall* one square east of i, and 0 otherwise. (P_2)
- $Touch_S(Taxi, Wall)$: returns 1 if any object of class *Taxi* has an object of class *Wall* one square south of it, and 0 otherwise. (P_3)
- $Touch_W(Taxi, Wall)$: returns 1 if any object of class *Taxi* has an object of class *Wall* one square west of it, and 0 otherwise. (P_4)
- *On*(*Taxi*, *Passenger*): returns 1 if any object of class *Taxi* is on the same square as any object of class *Passenger*, and 0 otherwise. (P_5)
- *On*(*Taxi*, *Destination*): returns 1 if any object of class *Taxi* is on the same square as any object of class *Destination*, and 0 otherwise. ($P_6 = f_9$)
- *InTaxi*(*Passenger*): returns 1 if any object of class *Passenger* has its *in-taxi* attribute set to 1, and 0 otherwise. ($P_7 = f_7$)

We see from the results that for this domain, Deictic OO-MDPs outperform Propositional OO-MDPs as we increase the number of passengers. This is because with Propositional OO-MDPs we need to add more propositions to the preconditions as we increase the number of passengers - in fact the propositional representation is unable to learn the task when $n = 4$ even after 100 training episodes.

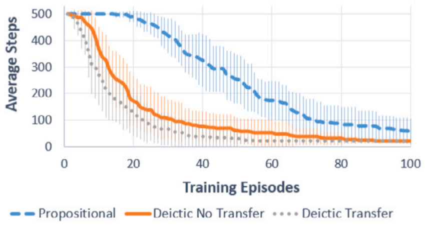
In particular, the classical Taxi domain with a single passenger and destination requires 7 propositions $\{P_1, P_2, P_3, P_4, P_5, P_6, P_7\}$ [13]. Under a propositional approach, the All-Passenger Any-Destination Taxi Domain requires an additional proposition: *Passenger-AtDestination* (P_8) that returns 1 if any object of class *Passenger* is on the same square as any object of class Destination. Furthermore, P_5, P_7 and P_8 need to be grounded for each object of class *Passenger* in order to avoid ambiguity, as further discussed in section 3.2. Therefore, in totality the All-Passenger Any-Destination Taxi Domain requires $5 + 3 \times n$ propositions for a task of n passengers.

Note that, since the number of propositions is increasing per task, it is not possible to transfer the transition dynamics between tasks under the Propositional OO-MDP framework. Meanwhile, the Deictic OO-MPD framework requires 8 deictic predicates for any task of the domain. Furthermore, as the MDPs belong to the same schema it is beneficial to transfer the previous transition dynamics under the deictic representation.

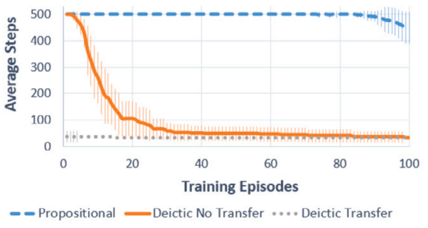
We observe from Fig. 7 that using the deictic representation without transfer is actually learning slightly faster as we add more passengers. This is somewhat misleading. What is actually happening is that as the tasks become more complex, the agent is able to learn more about the transition dynamics over a single training episode, but that episode will require many more steps to complete. To illustrate this, and also highlight the robustness of the deictic representation with transfer methodology, we conduct a second set of experiments. These experiments are similar to those conducted for the first set but we now use a larger 10×10 gridworld with five passengers and three destinations as in Fig. 3b. In these experiments we stop after 100 steps for each episode of the training MDPs. Furthermore, for the deictic representation with transfer we simply transfer the learned transition dynamics of $n = 4$ passengers and do no additional learning on the new larger gridworld.



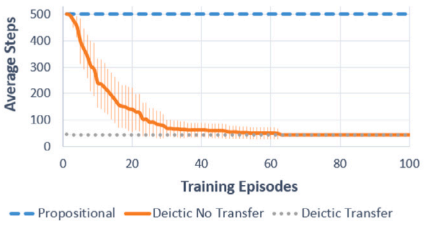
(a) One passenger experimental results.



(b) Two passenger experimental results.



(c) Three passenger experimental results.



(d) Four passenger experimental results.

Fig. 7. Experimental results for learning transitions dynamics in the All-Passenger Any-Destination Taxi domain with different number of passengers.

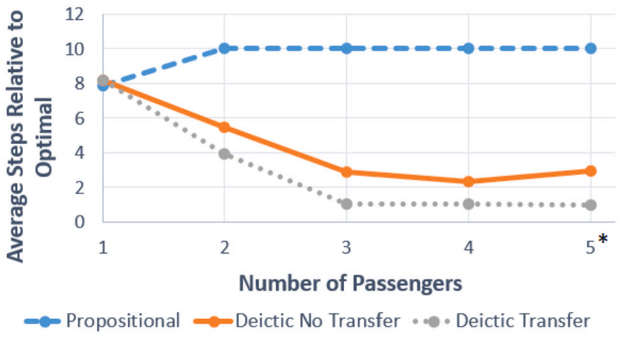


Fig. 8. Average number of steps relative to optimal number of steps as we add more passengers - for $n = 5$ we also increase the gridworld size and add more destinations hence it is marked with a $*$. The y axis is capped at 10 to make the graph more readable.

In Fig. 8 we plot for all the experiments run the average number of steps relative to optimal number of steps. We can see that once we get to $n = 4$ passengers, the transition dynamics we transfer from the $n = 3$ experiment are the fully learned schema transition dynamics and we have zero-shot transfer.[12] We also see that the deictic representation with transfer is able to solve the larger gridworld optimally with no additional learning of the transition dynamics. Meanwhile the deictic representation with no transfer, which was decreasing up to $n = 4$, now has a jump between at $n = 5$ because the agent does not have the benefit of learning for a full training episode. We cap the graph's y axis at 10 to make it more readable, but remark that Propositional OO-MDPs exhibits exponentially worse performance relative to optimal as the tasks become more complex.

To provide more intuition on how efficient learning is achieved with conjunctive effects, consider Tables 8 and 9. In these tables we show results from a simulation of running *DOORMAX*$_D$ for action *East*, attribute *Taxi.x* and effect *Rel*$_1$ as well as action *Pickup*, attribute *Passenger.in-taxi* and effect *SetBool*$_1$ respectively.

For example, in Table 8 the first time we observe the effect *Rel*$_1$ for action *East* and attribute *Taxi.x* the following assignment to the truth values are observed: $\{f_1 = 0, f_2 = 0, f_3 = 0, f_4 = 1, P_5 = 0, P_6 = 0, P_7 = 0\}$ (row 1). The next time, the following assignment to the truth values are observed: $\{f_1 = 1, f_2 = 0, f_3 = 0, f_4 = 1, P_5 = 0, P_6 = 1, P_7 = 0\}$ (row 2). Since the effect is conjunctive, f_1 and P_6 can be invalidated. As a result, the remaining possible deictic predicates that make up the precondition is $\{f_2, f_3, f_4, P_5, P_7\}$. This process continues until the true precondition is learned in line 7 i.e. $f_2 = 0$. Note that in Table 8 we are able to learn the correct precondition with only 7 observations, whereas learning by memorization would require 2^6 unique observations.

[12] It is worth pointing out that, while in this experiment we zero-shot transfer at $n = 4$, the conditions for zero-shot transfer depend on the KWIK-bound in Theorem 1, not the number of tasks the agent learns from.

Table 8
Learning the precondition for action *East*, attribute *Taxi.x* and effect Rel_1 in the All-Passenger Any-Destination Taxi domain.

Observation	f_1	f_2	f_3	f_4	P_5	P_6	P_7
1	0	0	0	1	0	0	0
2	-	0	0	1	0	-	0
3	-	0	0	-	0	-	0
4	-	0	0	-	0	-	-
5	-	0	0	-	0	-	-
6	-	0	0	-	0	-	-
7	-	0	-	-	-	-	-

Table 9
Learning the precondition for action *Pickup*, attribute *Passenger.in-taxi* and effect $SetBool_1$ in the All-Passenger Any-Destination Taxi domain.

Observation	P_1	P_2	P_3	P_4	f_5	f_6	P_7
1	0	1	1	0	1	0	0
2	-	1	-	0	1	0	0
3	-	-	-	-	1	0	0

Table 10
Hypothesized propositions for each action and reward token as well as their conjunctive indicator values. All other actions and reward token pairs use the hypothesis $\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$ with conjunctive indicators $\{0, 1\}$. The default reward token is $\bar{U}_4$.

Action	Reward token	Hypothesis ($\hat{\mathcal{F}}(a,i)$)	Conjunctive
Pickup	$\bar{U}_1$	$\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$	$\{0\}$
Dropoff	$\bar{U}_2$	$\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$	$\{0\}$
Dropoff	$\bar{U}_3$	$\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$	$\{0, 1\}$

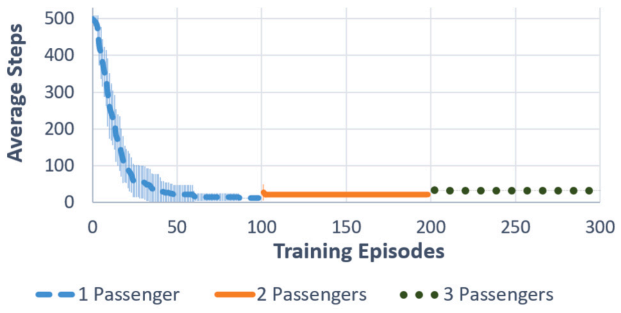


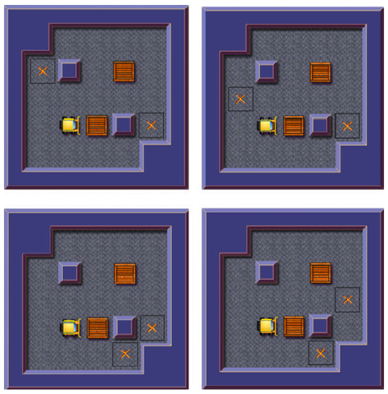
Fig. 9. Experimental results for learning reward dynamics in the All-Passenger Any-Destination Taxi domain with transfer and different number of passengers.

7.2. All-Passenger Any-Destination Taxi domain: learning reward dynamics

In this section, we run a similar experiment to that of section 7.1, but we now assume known transition dynamics and focus on learning and transfer of reward dynamics. In addition to p_1, p_2, p_3 and p_4 defined in section 4.3 we define:

- *AnyWallNorthOfAnyTaxi*(s, s'): returns 1 if any objects of class *Taxi* has an object of class *Wall* one square north of it in s', and otherwise 0. (p_5)
- *AnyWallEastOfAnyTaxi*(s, s'): returns 1 if any objects of class *Taxi* has an object of class *Wall* one square east of it in s', and otherwise 0. (p_6)
- *AnyWallSouthOfAnyTaxi*(s, s'): returns 1 if any objects of class *Taxi* has an object of class *Wall* one square south of it in s', and otherwise 0. (p_7)
- *AnyWallWestOfAnyTaxi*(s, s'): returns 1 if any objects of class *Taxi* has an object of class *Wall* one square west of it in s', and otherwise 0. (p_8)

Then in our experiments we use the hypothesis space as per Table 10.


(a) Four simple Sokoban tasks.

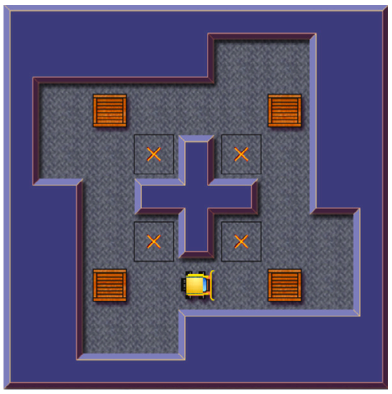

(b) A more complex Sokoban task with four boxes from the 'Micro-Cosmos' level pack.

Fig. 10. Training tasks and test task for Sokoban.

In Fig. 9 we see that, as in the case of learning transition dynamics, we can transfer the model of the reward dynamics between tasks of the domain as more passengers are added. The reward dynamics are completely learned after completing the $n = 2$ passenger MDPs.[13] From there, the reward dynamics are zero-shot transferred to the $n = 3$ passenger MDPs. In fact, the model of the reward dynamics can be zero-shot transferred to any MDP of the All-Passenger Any-Destination Taxi domain schema after learning on the $n = 2$ passenger MDPs.[14]

7.3. Sokoban domain

To demonstrate the benefits of the object-oriented frameworks presented in this paper, we conduct an experiment on a more challenging Sokoban domain. In this experiment we first learn both the transition dynamics and reward dynamics by randomly sampling MDPs with start states as shown in Fig. 10a.

As it turns out, these four simple MDPs, each with approximately 8000 states, are enough to completely learn the schema transition and reward dynamics of this domain. This is because they exhibit all possible game scenarios, such as the person adjacent to a box with that box being adjacent to another box, or the person adjacent to a box with that box being adjacent to a wall, etc.

We continue to learn on these simple MDPs until we have no $\perp$ predictions, thus having fully learned the complete schema dynamics. Once learned, we zero-shot transfer the dynamics to a more complex Sokoban task as shown in 10b. This task comes from the 'Micro-Cosmos' level pack and has approximately 10^6 states while the optimal number of steps to solve this task 209.

With no additional learning we run value iteration and solve for an optimal policy. Note that the ability to transfer here is critical. The larger MDP has approximately 125 times more states than the toy MDPs. Running R_{max} based algorithms directly on the larger MDP is very slow because at each step it is required to compute a policy with an exact planning algorithm and this has high computational complexity. By transferring the transition dynamics learned in the toy MDPs we can solve the larger MDP with only a *single run* of value iteration.

8. Future research

In this paper we have demonstrated the advantages of our proposed object-oriented frameworks. Theoretically we have shown that, under certain assumptions, our frameworks are provably efficient and guarantee zero-shot transfer. Empirically, we have demonstrated that our approach can zero-shot transfer the dynamics models learned from simple tasks to solve much more complex tasks with a single run of an exact planning algorithm, such as value iteration. However, there is much scope for future research.

As per Appendix A, our proposed method relies on certain assumptions to hold. Finding ways to relax these assumptions would make the proposed frameworks more practically applicable. For example, the most general RL setting allows for stochastic transition dynamics, while our approach is limited to deterministic transition dynamics. For the Propositional OO-MDP setting previous research has already shown that efficient learning of transition dynamics can be achieved for a family of stochastic dynamics [14]. Extending this to the Deictic OO-MDP setting and also extending the scope of permissible dynamics is an area of future research.

[13] While it appears from the figure that the reward dynamics are completely learned after $n = 1$, there is a small amount of learning required for $n = 2$ as the agent needs to learn the dynamics when picking up a passenger while another passenger is already in the taxi. One can see small error bars at the beginning of that portion of the plot indicating this.

[14] Note that as we add more passengers the optimal number of steps needed to complete a task increases, which is why you see a jump in Fig. 9 between the end of $n = 2$ and the start of $n = 3$.

Other assumptions that future work could look at relaxing, possibly by incorporating existing research, include: learning a schema's object classes and their corresponding attributes rather than having them be given [29–31]; learning the set of hypothesized propositions and deictic predicates rather than having them be given [32]; learning the dynamics models from preconditions to effects using function approximations rather than binary trees [33]; and combining object-oriented representations with value function approximations for planning in domains with large state-spaces where tabular methods used by exact planning algorithms, like value iteration, become computationally intractable [34].

The $DOORMAX_D$ algorithm works best in RL settings where effects can be represented as conjunctions over terms, as this allows the algorithm to invalidate multiple terms with few observations. However, many RL settings have effects that can only be captured with disjunctions, and $DOORMAX_D$ then requires $2^{D_{max}}$ unique observations to learn such effects without prior knowledge. This becomes prohibitive when the number of deictic predicates is large. An important area of future research is therefore to find more efficient ways to learn disjunctions.

In this paper we have combined object-oriented and deictic representations to compactly represent the dynamics models of an RL agent that transfer across related tasks of a domain. Both these representations have been leveraged in other components of RL such as policy learning [35], planning [10–12] and learning options [36,37] with the goal of improving learning efficiency and generalization. Research focused on combining multiple techniques into a unified framework could lead to enhanced RL agent capabilities in these areas.

Lastly, deep learning methods have shown much success in scaling RL to solve larger, real-world tasks [4,5,3] and recent work has investigated deep learning in the context of zero-shot transfer [38,39]. Existing research has demonstrated that deep learning can be used in conjunction with object-oriented representations to achieve improved transfer and generalization properties [40–42]. Research focusing on combining ideas from structured approaches, such as those presented in this paper, with deep learning could lead to novel methods that both generalize effectively and scale to larger, real-world tasks.

9. Conclusion

This paper has introduced and integrated two object-oriented frameworks for efficient model-based RL: Deictic OO-MDPs for transition dynamics and Propositional OO-MDPs for reward dynamics. These frameworks apply to a domain as described by a schema, and so generalize across all instantiated tasks from the schema.

The Deictic OO-MDP framework is based on deictic predicates. A deictic predicate is a predicate that is grounded with respect to a single reference object that relates itself to lifted object classes, and can be used to compactly represent the transition dynamics of domains not possible under a propositional approach. The Propositional OO-MDP framework extends previous work that focuses only on learning transition dynamics under a propositional approach to the case of learning reward dynamics.

Both of these frameworks are then combined into a KWIK-R_{max} based algorithm called $DOORMAX_D$ that efficiently learns the full dynamics models. Since the models are schema-based, they transfer across all tasks of the domains. To illustrate the benefits of our proposed frameworks we run experiments on a modified version of the Taxi domain, the All-Passenger Any-Destination Taxi domain, as well as the Sokoban domain. In both these domains we illustrate that the dynamics models can be learned on a simple set of source task from the domain and then zero-shot transferred to a more complex target task of the domain for efficient RL.

Our proposed framework requires more prior knowledge and assumptions than other approaches, such as Deep Learning. However, it does have the advantage of having provably efficient learning bounds and guarantees of zero-shot transfer. Stronger theoretical analysis has been identified as an important research direction in the area of transfer learning for Reinforcement Learning. This paper takes a step in this direction by extending previous research to produce a more general formalism that still fit into the KWIK framework.

CRediT authorship contribution statement

Ofir Marom: Conceptualization, Formal analysis, Investigation, Methodology, Validation, Writing – original draft. **Benjamin Rosman:** Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Appendix A. $DOORMAX_D$ assumptions

Assumption 1. *A schema representation of a domain is given with a fixed set of object classes, object class attributes and actions.*

Assumption 2. *Transition dynamics are deterministic.*

Assumption 3. *Prior knowledge of the deictic predicates required to represent the schema transition dynamics for each action and attribute.*

Assumption 4. *A set of effect groups is hypothesized, where each effect group consists of an effect set and a partition function that maps each effect in the effect set as either disjunctive or conjunctive. One of the hypothesized effect groups must be correct.*

Assumption 5. *All effects in an effect set are invertible.*

Assumption 6. *A constant M, which represents the maximum number of times that a disjunctive effect may occur in any binary tree, is known. However, we do provide a no-prior value $M = 2^{D_{max}}$ as well.*

Assumption 7. *Reward dynamics may be stochastic, as long as they are KWIK-learnable.*

Assumption 8. *A known decomposition of the reward dynamics into L reward tokens, as well as a default reward token.*

Assumption 9. *A partition function for each action and non-default reward token index that maps the binary indicator values associated to the activation of that reward token as either disjunctive or conjunctive.*

Assumption 10. *An exact planning algorithm is used to compute an optimal policy under the*

Appendix B. Proof of Theorem 1

Proof. Consider an effect group $g \in \hat{\mathcal{G}}$. If this is the correct effect group then it can be learned with KWIK bounds $K_0^g M + K_1^g(D+1)$. This is because the K_1^g conjunctive effects require at most $D+1$ observations each to learn the terms they depend on while the terms for the K_0^g disjunctive effects can be memorized M times each. If we then consider all N effect groups in $\hat{\mathcal{G}}$, an upper bound on the number of observations required so that all effect groups are either removed or return some prediction is $N(K_0 M + K_1(D+1)+1)$.

Now when we call *Predict1* if two effect groups provide a prediction that is not the same we remove one of them on the subsequent run of the *UpdateTree1* procedure and this can occur at most $N-1$ times. This gives a total KWIK-bound of $N(K_0 M + K_1(D+1) + 1) + N - 1$. □

An analogous proof can be constructed for the learning algorithm of reward dynamics presented in section 4.4 showing that if a binary value occurs at most once in the tree for action a and index i mapped to z_i, then that branch can be learned with KWIK-bound $D_{a,i}+1$.

References

[1] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, 2018.
[2] G.J. Tesauro, TD-Gammon, a self-teaching backgammon program, achieves master-level play, Neural Comput. 6 (2) (2004) 215–219.
[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing Atari with deep reinforcement learning, CoRR, arXiv:1312.5602 [abs].
[4] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis, Mastering the game of go with deep neural networks and tree search, Nature 529 (2016) 484–489.
[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T.P. Lillicrap, K. Simonyan, D. Hassabis, Mastering chess and shogi by self-play with a general reinforcement learning algorithm, CoRR, arXiv:1712.01815.
[6] M.E. Taylor, P. Stone, Transfer learning for reinforcement learning domains: a survey, J. Mach. Learn. Res. 10 (1) (2009) 1633–1685.
[7] D.N. Perkins, G. Salomon, Transfer of learning, in: International Encyclopedia of Education, 1992.
[8] S. Džeroski, L. DeRaedt, K. Driessens, Relational reinforcement learning, Mach. Learn. J. 43 (2001) 7–52.
[9] M. van Otterlo, A survey of reinforcement learning in relational domains, CTIT Tech. Rep. Ser. (2005) 1381–3625.
[10] C. Guestrin, D. Koller, C. Gearhart, N. Kanodia, Generalizing plans to new environments in relational MDPs, in: Proceedings of the 18th Joint Conference on Artificial Intelligence, 2003, pp. 1003–1010.
[11] C. Guestrin, Planning under uncertainty in complex structured environments, Ph.D. thesis, Stanford University, 2003.
[12] K. Kansky, T. Silver, D.A. Mèly, M. Eldawy, M. Làzaro-Gredilla, X. Lou, N. Dorfman, S. Sidor, S. Phoenix, D. George, Schema networks: zero-shot transfer with a generative causal model of intuitive physics, in: Proceedings of the 34th International Conference on Machine Learning, 2017, pp. 1809–1818.
[13] C. Diuk, A. Cohen, M.L. Littman, An object-oriented representation for efficient reinforcement learning, in: Proceedings of the 25th International Conference on Machine Learning, 2008, pp. 240–247.
[14] C. Diuk, An object-oriented representation for efficient reinforcement learning, Ph.D. thesis, Rutgers the State University of New Jersey-New Brunswick, 2010.
[15] J. Scholz, M. Levihn, C.L. Isbell, D. Wingate, A physics-based model prior for object-oriented MDPs, in: Proceedings of the 31st International Conference on Machine Learning, 2014, pp. 1089–1097.
[16] L. Li, M.L. Littman, T.J. Walsh, Knows what it knows: a framework for self-aware learning, in: Proceedings of the 25th International Conference on Machine Learning, 2008, pp. 568–575.
[17] L. Li, A unifying framework for computational reinforcement learning theory, Ph.D. thesis, Rutgers the State University of New Jersey-New Brunswick, 2009.
[18] O. Marom, B.S. Rosman, Zero-shot transfer with deictic object-oriented representation in reinforcement learning, in: Proceedings of the 32nd Conference on Neural Information Processing Systems, 2018.
[19] T. Dietterich, The MAXQ method for hierarchical reinforcement learning, in: Proceedings of the 15th International Conference on Machine Learning, 1998.
[20] R.I. Brafman, M. Tennenholtz, R-MAX - a general polynomial time algorithm for near-optimal reinforcement learning, J. Mach. Learn. Res. 3 (2002) 213–231.

[21] M.S. Kakade, On the sample complexity of reinforcement learning, Ph.D. thesis, Gatsby Computational Neuroscience Unit, University College London, 2003.
[22] A.L. Strehl, Model-based reinforcement learning in factored-state MDPs, in: Proceedings of the 2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning, 2007.
[23] A.L. Strehl, C. Diuk, M.L. Littman, Efficient structure learning in factored state MDPs, in: Proceedings of the 22nd AAAI Conference on Artificial Intelligence, 2007, pp. 645–650.
[24] A.L. Strehl, M.L. Littman, Online linear regression and its application to model-based reinforcement learning, in: Proceedings of the 21st Annual Conference on Neural Information Processing Systems, 2008.
[25] C. Boutilier, R. Reiter, B. Price, Symbolic dynamic programming for first-order MDPs, in: Proceedings of the 17th International Joint Conference on Artificial Intelligence, 2001, pp. 690–697.
[26] K. Kiguchi, T. Nanayakkara, K. Watanabe, T. Fukuda, Multi-dimensional reinforcement learning using a vector Q-net - application to mobile robots, Int. J. Control. Autom. Syst. 1 (2003) 142–148.
[27] Z. Juozapaitis, A. Koul, A. Fern, M. Erwig, F. Doshi-Velez, Explainable reinforcement learning via reward decomposition, in: Workshop on Explainable Artificial Intelligence at International Joint Conference on Artificial Intelligence, pp. 47–53, 2019.
[28] H. van Seijen, M. Fatemi, J. Romoff, R. Laroche, T. Barnes, J. Tsang, Hybrid reward architecture for reinforcement learning, CoRR, arXiv:1706.04208 [abs].
[29] C.P. Burgess, L. Matthey, N. Watters, R. Kabra, I. Higgins, M. Botvinick, A. Lerchner, Monet: unsupervised scene decomposition and representation, CoRR, arXiv:1901.11390 [abs].
[30] T. Kipf, E. van der Pol, M. Welling, Contrastive learning of structured world models, in: Proceedings of the 8th International Conference on Learning Representations, 2020.
[31] S. James, B. Rosman, G. Konidaris, Learning of object-centric abstractions for high-level planning, in: Proceedings of the 10th International Conference on Learning Representations, 2022.
[32] D.E. Hershkowitz, J. MacGlashan, S. Tellex, Learning propositional functions for planning and reinforcement learning, AAAI Fall Symp. Ser. (2015).
[33] P. Battaglia, R. Pascanu, M. Lai, D. Rezende, K. Kavukcuoglu, Interaction networks for learning about objects, relations and physics, in: Proceedings of the 30th International Conference on Neural Information Processing Systems, 2016, pp. 4502–4510.
[34] W. Woof, K. Chen, Learning to play general video-games via an object embedding network, in: IEEE Symposium on Computational Intelligence and Games (CIG), 2018.
[35] S. Finney, N.H. Gardiol, L.P. Kaelbling, T. Oates, The thing that we tried didn't work very well: deictic representation in reinforcement learning, in: Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence, 2002, pp. 154–161.
[36] B. Ravindran, A.G. Barto, Relativized options: choosing the right transformation, in: Proceedings of the 20th International Conference on Machine Learning, 2003.
[37] N. Topin, N. Haltmeyer, S. Squire, J. Winder, M. desJardins, Portable option discovery for automated learning transfer in object-oriented Markov decision processes, in: Proceedings of the 24th International Joint Conference on Artificial Intelligence, 2015.
[38] O. Junhyuk, S. Satinder, L. Honglak, K. Pushmeet, Zero-shot task generalization with multi-task deep reinforcement learning, in: Proceedings of the 34th International Conference on Machine Learning, vol. 70, 2017, pp. 2661–2670.
[39] I. Higgins, A. Pal, A.A. Rusu, L. Matthey, C. Burgess, A. Pritzel, M. Botvinick, C. Blundell, A. Lerchner, Darla: improving zero-shot transfer in reinforcement learning, in: Proceedings of the 34th InternationalConference on Machine Learning, vol. 70, 2017, pp. 1480–1490.
[40] L. Zhao, L. Kong, R. Walters, L.L.S. Wong, Toward compositional generalization in object-oriented world modeling, in: Proceedings of the 39th International Conference on Machine Learning, 2022, pp. 26841–26864.
[41] A. Creswell, R. Kabra, C. Burgess, M. Shanahan, Unsupervised object-based transition models for 3d partially observable environments, in: Proceedings of the 35th Conference on Neural Information Processing Systems, 2021.
[42] R. Veerapaneni, J.D. Co-Reyes, M. Chang, M. Janner, C. Finn, J. Wu, J. Tenenbaum, S. Levine, Entity abstraction in visual model-based reinforcement learning, in: Proceedings of the 4th Conference on Robot Learning, 2020, pp. 1439–1456.