# OBJECT DISCOVERY IN NOVEL ENVIRONMENTS FOR EFFICIENT DETERMINISTIC PLANNING

**by**

**ETHAN DANIEL FRANK**

Submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer and Data Sciences

CASE WESTERN RESERVE UNIVERSITY

May, 2023

# CASE WESTERN RESERVE UNIVERSITY

# SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis of

Ethan Daniel Frank

candidate for the degree of **Master of Science***.

Committee Chair

**Dr. Soumya Ray**

Committee Member

**Dr. M. Cenk Cavusoglu**

Committee Member

**Dr. Michael Lewicki**

Date of Defense

**April 6, 2023**

*We also certify that written approval has been obtained

for any proprietary material contained therein.

# Contents

# List of Figures

# Object Discovery in Novel Environments for Efficient Deterministic Planning

Abstract

by

ETHAN DANIEL FRANK

A central problem in applying planning to new environments is to quickly discover the characteristics of those environments. In practice, this means identifying the behaviors and properties of objects within the environment. In this thesis I investigate how to approach this problem with transfer learning for deterministic planning tasks within the Object-Oriented Markov Decision Process (OO-MDP) framework. I extend this framework with additional logical relations and describe an efficient rule learning algorithm for these domains. I then present two approaches of using the learned rules in new environments to discover the types and properties of objects under different assumptions. I show empirically that these approaches allow an agent to learn and plan in new domains with better sample complexity than if it had started from scratch.

# Chapter 1

# Introduction

If we want a robot to bring us a cookie, completing this task requires executing a precise sequence of actions in the right order. Our robot would need to go over to the pantry, grasp the handle, pull open the door, take out the cookie, close the door, release the handle, and bring the cookie back to us. But, there are many different sequences of actions that the robot could have taken. Some sequences would be faster, some slower, and some could accidentally destroy your house. Sequential decision making (SDM) is the branch of artificial intelligence that studies how agents, like our robot, choose optimal sequences of actions to accomplish a task. Ideally, our SDM agent would learn the best sequence of actions by itself, as it would be infeasible for a human "expert" to tell the agent which is the best action to take for every scenario.

Reinforcement learning (RL) overcomes this limitation by introducing *rewards*, a number which can be positive or negative. We design a system where the robot receives a reward when it picks up the cookie, another when it brings us the cookie, but receives a negative reward if it drops the cookie. An RL agent will usually start by trying many random actions, some of which will do nothing, but some by chance will lead to a situation in which the agent attains positive rewards. Eventually the agent learns from these experiences, taking actions towards positive rewards and avoiding

negative rewards. In the end it will learn to accomplish the task, which itself usually comes with a large positive reward.

The current configuration of the agent and the world around it is known as the *state*. The reward that the agent believes it will receive depends on which action it chooses to take in the current state. The state consists of everything the agent needs to keep track of, such as its location, the location of the pantry, or if the pantry door is open. If the agent leanrs how to extract a cookie from a pantry on the right side of the room, it will still be unable to do so for a pantry on the left side of the room, because the state would be different, and it would have to learn the rewards corresponding to actions in that state from scratch. Object-Oriented Markov Decision Processes (OO-MDPs) address this problem by factoring the state into objects, where all objects of the same type behave the same. Under this framework, the agent predicts the effects of its actions and the rewards it will receive based on its interactions with the objects in the environment. So, if it learns to open one pantry, it now knows how to open them all.

OO-MDPs are based on *propositional logic*, a framework of logic which contains variables that may be true or false. The agent would express its interactions with the environment as a collection of these variables, such as *holdingCookie* or *holdingDoorHandle*. However, each variable is only a true or a false, a 1 or a 0, there is no *structure*. The agent can not understand that *holdingCookie* and *holdingDoorHandle* both refer to a state in which something is grasped. *First-order logic* is a more versatile framework that extends propositional logic with objects and relations. The interactions can now be expressed as *Holding(Cookie)* and *Holding(DoorHandle)*, and the agent can understand that both *Cookie* and *DoorHandle* are objects satisfy the *Holding* relation. In this thesis, I present a method to extend OO-MDPs with first-order logic, which I call First-Order OO-MDPs (FO-OO-MDPs).

The "object-oriented" approach leads to the ability to use previous experience on

one task to get a head start on another. This is known as transfer learning, and here we apply it to object-oriented RL. In this scenario, all objects have properties, and it is reasonable to believe that objects with similar properties behave, and can be interacted with, in similar ways. If our robot encounters a new pantry door with a horizontal, instead of vertical handle, it can make a pretty good guess that pulling that handle will lead to the door opening. If tasked with getting a cookie out of this new pantry, it will not be stumped by the new handle orientation. Leveraging previous experience will allow the agent to focus on pulling the handle in order to solve this new task faster.

The real-world, full version of this algorithm is hard to implement, requiring a visual system to identify objects as well as subparts of those objects. In the 2D, simulated worlds of this thesis, there is no vision and all interactions between objects are expressed only via first-order logic. That is why I introduce a reduction of the above problem: Given previous experience interacting with objects in a 2D world, can the agent re-identify which object is which? This "object discovery" task can be seen as a step towards developing even more advanced and capable algorithms.

My specific contributions are as follows:

- An extension to the OO-MDP framework that incorporates first-order logic
- An inductive learner that rapidly solves FO-OO-MDP domains
- A demonstration of how first-order logic enables object discovery with a "logic-based" object discovery algorithm
- A "simplest-explanation" object discovery algorithm that overcomes many of the limitations of the logic-based algorithm
- An empirical evaluation and comparison of my techniques

The remainder of this work consists of five chapters. In Chapter 2, I introduce the relevant background and related work. In Chapter 3, I describe the first-order extension to OO-MDPs and introduce the inductive learner. In Chapter 4, I define the object discovery problem and introduce the logic-based object discovery agent. In Chapter 5, I present the simplest-explanation object discovery algorithm. Finally, Chapter 6 presents the empirical evaluation of these algorithms.

# Chapter 2

# Background and Related Work

This chapter will review background information necessary to understand the work in this thesis. First, I will introduce the frameworks of reinforcement learning, Markov Decision Processes, and Object-Oriented Markov Decision Processes, which help to formalize different types of decision making problems. Next, I describe the differences between propositional and first-order logic and introduce another related framework, that of Relational Markov Decision Processes. Finally, I discuss transfer learning and lifelong learning, which are frameworks for defining and evaluating agents that use previous experience to improve performance on future tasks.

## 2.1 Reinforcement Learning

Reinforcement learning [2] is a branch of machine learning that focuses on sequential decision making problems, where at each time step, the reinforcement learning agent must choose an action to take. A maze-solving agent may choose between moving up, down, left, or right, whereas a car-driving agent may choose between accelerating, breaking, or turning. Unlike in supervised machine learning, the agent is not given the "right answer" of which action to take in a given situation. Instead, they are given positive and negative rewards, and must learn to choose actions that maximize

the received reward. Because rewards can be given out sparsely (the maze-solving agent is only rewarded upon exiting the maze), the agent may need to plan many steps ahead to receive positive rewards.

### 2.1.1 Markov Decision Processes

These sequential decision making problems are often formalized as a *Markov Decision Process* (MDP) [2]. An MDP consists of a set of states $S$, representing all possible configurations of the agent and its environment. $S_0 \in S$ is the set of initial states that the agent starts in, and the agent has available a set of actions $A$ that it can choose from in each state. $P : S \times A \times S \to [0, 1]$ is the *transition function*, which for each state, action, and next state tuple $(s, a, s')$ defines the probability of the next state being $s'$ when action $a$ is taken in state $s$. $R : S \times A \to \mathbb{R}$ is the *reward function*, which returns the agent's reward for each possible $(s, a)$ pair. $\gamma \in (0, 1]$ is the *discount factor*, which acts as a tradeoff between immediate and future rewards. In *episodic* MPDs, there are a set of goal states $S_g$, which when reached indicate the end of the current "episode". In episodic MDPs the agent's goal is to achieve the highest average reward per episode [3]. In contrast, non-episodic MDPs have no end condition, and the agent's goal is to receive the largest *discounted reward* over a given time.

### 2.1.2 Illustrative Example

For an example application of an MDP, we will use the Maze domain shown in Figure 2.1. The agent starts in the bottom left corner, and needs to navigate to the top right corner to reach the exit and get a reward. Because the walls are fixed in place, the state of the environment can be fully described by the location of the agent, stored as a tuple $S = \{(x, y) \mid x \in \{1, 2, 3\}, y \in \{1, 2, 3\}\}$, and $S_0 = \{(0, 0)\}$. In any state, the agent can choose from the four available actions $A = \{\mathit{Up}, \mathit{Down}, \mathit{Left}, \mathit{Right}\}$. Each

action will move the agent in the corresponding direction, unless there is a wall in the way. This environment is *deterministic*, not *stochastic*, so there is no randomness in the outcomes of actions.

The agent receives a positive reward when it has reached the end of the maze and exits with *Up*, so the reward function can be defined like this:

$$R(s,a) = \begin{cases} +1 & \text{if } a = \mathit{Up} \wedge s = (2,2) \\ -1 & \text{otherwise} \end{cases}$$

Notice the default reward of $-1$, which encourages the agent to be efficient with its selection of actions as every extra action decreases the total reward.

The transition function has a total of $|S| * |A| * |S| = 324$ entries, so for brevity I will only show the function for the state $(1,1)$,

$$T(s=(1,1),a,s') = \begin{cases} 1 & \text{if } a = \mathit{Left} \wedge s' = (0,1) \\ 1 & \text{if } a = \mathit{Down} \wedge s' = (1,0) \\ 1 & \text{if } a = \mathit{Up} \wedge s' = (1,1) \\ 1 & \text{if } a = \mathit{Right} \wedge s' = (1,1) \\ 0 & \text{otherwise} \end{cases}$$

where each 1 indicates the 100% chance the agent has to move to the left or down

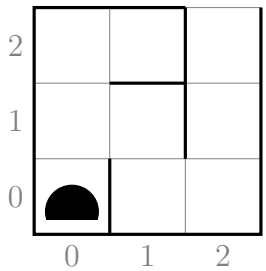


Figure 2.1: The Maze domain.

when executing *Left* or *Down*, and the 100% chance that $s' = s$ in the cases where the agent executes *Right* or *Up* and bumps into a wall. No other successor state can be reached, so all other $s'$ have a probability of 0.

### 2.1.3 Optimal Policies

To achieve positive rewards, the agent must decide upon the best action to take for any state in which it finds itself. The function that the agent uses to map each state to an action is known as the *policy* $\pi : S \to A$. The agent uses its policy to choose an action, then takes that action, transitions to the next state, and receives a reward. Now in the next state, it uses its policy to take another action, and so on. This creates a sequence of states $\{s_0, s_1, \ldots, s_n\}$ and rewards $\{r_0, r_1, \ldots, r_n\}$. The *utility* of this sequence starting at $s_0$ is defined as $U(s_0) = r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots + \gamma^n r_n = \sum_{t=0}^{\infty} \gamma^t r_t$, where $\gamma$ is the discount factor [2].

Even with an unchanging policy $\pi$, if an action can lead to many successor states (as is the case with stochastic environments), then the states reached under $\pi$ could be different each time. We modify the equation to become an expectation over the sequences of states:

$$U^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t))\right] \tag{2.1}$$

The *optimal policy* $\pi^*$ is defined as the policy which yields the highest *expected* utility,

$$\pi_s^* = \underset{\pi}{\operatorname{argmax}}\, U^{\pi}(s) \tag{2.2}$$

If the utilities of each state are known, the agent can recover the optimal policy by picking actions that are expected to lead to states with the highest utility. Because each action $a$ could have multiple successor states $s'$, we must weight the utility of each successor state by the probability of transitioning to it:

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}} \sum_{s'} P(s' \mid s, a) U(s') \tag{2.3}$$

The *Bellman equation* can be used to calculate the utility of a state from the utilities of its neighbors. The utility of a state is equal to the maximum over all actions of the reward for that action $a$, plus the utility of each next potential successor state $s'$, weighted by the probability of transitioning to that state.

$$U^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s, \pi(s), s') U^{\pi}(s') \tag{2.4}$$

In the above equations, we directly use the transition function $P(s, a, s')$ for the calculations. An agent that attempts to learn a *model* of the transition function is known as *model-based reinforcement learning. Model-free reinforcement learning* does not maintain a model of the environment, instead directly calculating the utilities of actions for given states. One such method is the well-known Q-Learning. In this thesis, I focus on model-based reinforcement learning.

## 2.2 Object-Oriented Markov Decision Processes

MDPs often fall victim to the *curse of dimensionality*: as the size of the problem grows, the required calculations increase exponentially. Imagine if the Maze domain was doubled to a $6 \times 6$ grid. Instead of 324 entries, the transition function would now consist of $|S| * |A| * |S| = 5{,}184$ entries (though most would be 0). This is a $16x$ increase, even for this two-dimensional problem. In order for agents to make decisions in a reasonable amount of time, *abstractions* that compress their knowledge of the environment are required.

One such abstraction treats state dynamics as arising from the interactions of *objects* within the environment. As there are only so many ways these objects can interact, the agent need not keep track of a full state transition table. This framework

is known as the *Object-Oriented Markov Decision Process* (OO-MDP) [4]. An OO-MDP consists of a set of object classes $C = \{C_1, \ldots, C_c\}$, where each class has a set of *attributes* $Att(C) = \{C.a_1, \ldots, C.a_a\}$ and each attribute has a *domain* $Dom(C.a)$. An environment consists of a set of objects $O = \{o_1, \ldots, o_o\}$ where each object is an instance of a class $o \in C_i$. The state of an object $o.state$ is the set of values assigned to its attributes, and the state of the OO-MDP is the union of the states of the objects $s = \bigcup_{o \in O} o.state$.

A *term* is a Boolean function over the relations between objects and/or object attributes. For example, one could define the term $touchLeft(o_1, o_2)$ to be true when one object is to the left of another.

Terms are defined on a class level, meaning all objects of the same class behave and interact with other objects in the same way. Terms are assigned values when instantiated with specific instances of objects, and which terms are and are not true in a given state determines the *effect* an action will have. For example, in the Maze domain, any time there is a *wall* object to the right of the *agent* object so $touchRight(agent, wall)$ is true, the action *Right* will have no effect. Which specific wall (or agent, though there is only one agent in this domain) is referenced does not matter. Both $touchRight(agent_1, wall_4)$ and $touchRight(agent_1, wall_7)$, being true would cause *Right* to have no effect. So, in general we can refer to the *ungrounded* term $touchRight(Agent, Wall)$.

An *effect* is an operation on a single attribute of a single object and there are two types: *arithmetic* (increment att by 1, decrement att by 2) and *assignment* (set att to 0, set att to false). However, an action may cause a change in more than one attribute of more than one object at the same time, so an *outcome* is defined as the union of all effects that occurred during a transition. This is known as the *outcome assumption* [5]. If no attributes change, that is called the *no-change* outcome. In addition, the *frame assumption* states that any state variables not described in the

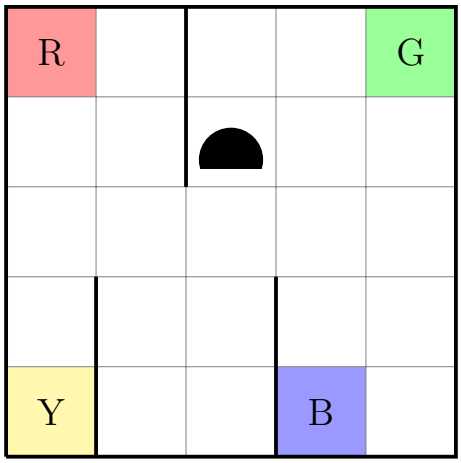


Figure 2.2: The Taxi domain

outcome of a state transition are assumed to remain the same.

In the original MDP formulation, the effect of an action like *Right* is seen as a property of specific states. Executing *Right* in the states $\{(0,0),(1,1),(1,2),(2,0),(2,1),(2,3)\}$ leads to no-change, but the agent does not know *why* this occurs and must re-learn this fact for every new state it encounters. In OO-MDPs, the agent is able to learn that when *touchRight*(*Agent*, *Wall*) is true, the outcome is *no-change*, and can extend that to *any* similar state regardless of if it has been there before.

### 2.2.1 The Taxi Domain

For an example OO-MDP, let us look at the Taxi domain [4] in Figure 6.1. The Taxi domain consists of a $5 \times 5$ grid of cells containing an agent, passenger, and four possible pick-up and drop-off locations labeled *Red*, *Green*, *Blue*, and *Yellow*. Each episode, a single pick-up and drop-off location are randomly selected from the four choices. The goal of the agent is to navigate to and pick up the passenger and then drop them off at the desired destination.

The objects in this environment are of the classes *Agent*, *Wall*, *Passenger*, and *Destination*. (The pick-up location does not need to be an object, because it is defined to be where the passenger object is waiting). Each object has an $x$ and $y$ attribute,

walls have an additional *orientation* attribute that indicates which side of the grid cell they are on, and the passenger has an additional attribute *held* to determine if it has been picked up.

The agent has the available actions *Up*, *Down*, *Left*, *Right*, *Pickup*, and *Dropoff*. The four movement actions will move the agent in the corresponding direction unless there is a wall in the way, the passenger is picked up when the agent executes the *Pickup* action while on the pick-up location, and the passenger is dropped off when the agent executes the *Dropoff* action while holding the passenger and on the destination location. Dropping off the passenger earns the agent +10 reward and ends the episode. The agent earns −1 for every other action.

In Taxi, there are six relation types: $touchUp(o_1, o_2)$, $touchDown(o_1, o_2)$, $touchLeft(o_1, o_2)$, $touchRight(o_1, o_2)$, $on(o_1, o_2)$, and $holding(o_1, o_2)$. Though it is possible to instantiate these with any objects, we only need the seven terms $touchUp(Agent, Wall)$, $touchDown(Agent, Wall)$, $touchLeft(Agent, Wall)$, $touchRight(Agent, Wall)$, $holding(Agent, Passenger)$, $on(Agent, Passenger)$, and $on(Agent, Destination)$ to fully describe transition and reward dynamics. For clarity, we will generally leave out the word *Agent* when writing relations from now on.

Next, $D_P$ is the set of *rules* that govern the transition dynamics for each action. Each rule $r$ in $D_P$ consists of an action $a$, condition $c$, and outcome $o$. A condition is some set of terms describing when an action has an effect. A condition matches, or is applicable to, another condition denoted $c_1 \models c_2$ if every positive term in $c_1$ is positive in $c_2$ and no negated terms in $c_1$ are positive in $c_2$. The function $cond(s)$ returns the subset of terms that are true in state $s$. For example, in the state shown in Figure 6.1 $cond(s)$ returns only $touchLeft(Agent, Wall)$. By the *closed world assumption*, only true terms need to be included in the state description, as any terms not listed are assumed to be false. The rules dictate that if action $a$ is executed in state $s$ while $c \models cond(s)$, outcome $o$ will occur [5]. If no rules are applicable, then the *no-change*

outcome occurs. Conditions must be mutually exclusive, otherwise more than one rule could apply to a given state, and the outcome could not be predicted.

The full set of rules in $D_P$ for Taxi are below:

$$Up : \neg touchUp(Wall) \longrightarrow \{Agent.y \leftarrow Agent.y + 1 \qquad Right : \neg touchRight(Wall) \longrightarrow \{Agent.x \leftarrow Agent.x + 1$$

$$Down : \neg touchDown(Wall) \longrightarrow \{Agent.y \leftarrow Agent.y - 1 \qquad Pickup : on(Passenger) \longrightarrow \{Passenger.held \leftarrow true$$

$$Left : \neg touchLeft(Wall) \longrightarrow \{Agent.x \leftarrow Agent.x - 1 \qquad Dropoff : on(Destination), holding(Passenger) \longrightarrow \{Passenger.held \leftarrow false$$

## 2.3 Transfer Learning

As we have seen with OO-MDPs, abstractions allow an agent to more efficiently learn complex tasks. *Transfer learning* posits the idea that abstractions are present not only within a task, but *between* tasks as well. By transferring knowledge from one or more *source task(s)*, an agent hopes to improve performance on one or more *target task(s)* [3]. Transfer learning algorithms are commonly distinguished from one another via the following dimensions:

- What are the goals of the algorithm, and how is success measured? For example, more rapid initial learning on the target task, or a higher long-term reward?
- What assumptions are made about the similarities between tasks? How do the state spaces, transitions, actions, and reward functions differ between tasks?
- Which previous tasks are relevant? Does the agent assume that all source tasks are helpful, or only some?
- What information does the agent transfer between tasks? Low level knowledge, like the transition function or higher level information like the policy?

### 2.3.1 Inter-Task Mappings

When the transfer learning methods are *not* able to assume that the state variables and actions remain the same between tasks, it is helpful to define *inter-task mappings* [3] to relate the target task to the source task. For example, if our maze-solving agent was moved to an environment where the state variables $\{x, y\}$ were the same but instead of $A_s = \{Up, Down, Left, Right\}$, the action set was $A_t = \{North, South, West, East\}$, then the action mapping $\mathcal{X}_A : A_t \rightarrow A_s$ would need to map *South* to *Down*, *West* to *Left*, and so on. These mappings can be provided by humans, or learned by the agent itself. This thesis will investigate methods for the agent to learn *object mappings* by itself, which I introduce in Chapter 4.

## 2.4 Propositional and First-Order Logic

OO-MDPs use *propositional logic* to represent information. In propositional logic, there are only *facts*, *propositional variables*, and *logical connectors*. For example, if we have the propositional variables *Tuesday* and *Raining*, the logical connector $\wedge$ (and), we can state the fact that "it is Tuesday and raining" with $Tuesday \wedge Raining$. In Taxi, each term *touchLeft*(*Wall*), *touchRight*(*Wall*), *touchUp*(*Wall*), *touchDown*(*Wall*), *holding*(*Passenger*), *on*(*Passenger*) and *on*(*Destination*) is a propositional variable. The key point is to not be fooled by the individual components (*holding*, *Passenger*, or *Wall* for example) - because our human brains are biased to treat the world in terms of objects and relations. To the agent, these are just Boolean variables it receives from $cond(s)$. If each term was named *Apple*, *Banana*, *Grape*, *Cucumber*, *Orange*, *Watermelon*, and *Cherry* respectively, then the agent would successfully learn the rule set:

$$Up : \neg Grape \longrightarrow \left\{ Agent.y \leftarrow Agent.y + 1 \right. \qquad Right : \neg Banana \longrightarrow \left\{ Agent.x \leftarrow Agent.x + 1 \right.$$

$$Down : \neg Cucumber \longrightarrow \left\{ Agent.y \leftarrow Agent.y - 1 \right. \qquad Pickup : Watermelon \longrightarrow \left\{ Passenger.held \leftarrow true \right.$$

$$Left : \neg Apple \longrightarrow \left\{ Agent.x \leftarrow Agent.x - 1 \right. \qquad Dropoff : Cherry, Orange \longrightarrow \left\{ Passenger.held \leftarrow false \right.$$

*First-order logic* extends the logical formulation to capture the intuition our human brains use. It adds *objects*, *relations*, and *functions* [2]. The statement "I am standing next to the door to the red house" contains the objects *I*, *Door*, and *House*, the unary relations (aka *properties*) *Standing* and *Red*, the binary relation *NextTo*, and the function *DoorTo*. The fact can be expressed as a *sentence* of $NextTo(I, DoorTo(House))$, $Red(House)$, and $Standing(I)$. Note the use of capital letters for relations to distinguish them from their propositional version. All relations in an OO-MDP are replaceable with first-order logic, which I discuss in detail in Chapter 3.

## 2.5 Related Work

Applying first-order logic to OO-MDPs creates a representation very similar to that of relational Markov Decision Processes [6] and relational reinforcement learning [7]. Relational MDPs also have sets of classes, objects, and attributes, but the key difference is how transition dynamics are represented. Depending on the specific implementation, the transition function either directly predicts the attributes of the objects, or the active relations, for the next state. This differs from the outcomes and effects in OO-MDPs in that effects dictate the *change* in an object's attributes, and do *not* need to predict how the relations will change.

### DOO$R_{\max}$

DOO$R_{\max}$ (Deterministic Object-Oriented $R_{\max}$) [4] is a deductive learning algorithm for deterministic object-oriented environments that follows the KWIK (Knows What it Knows) framework [8]. A KWIK algorithm is a model-based reinforcement learning algorithm that always correctly predicts the effect of an action, unless the information it has acquired so far is not sufficient to do so in which case the prediction returns $\perp$ to indicate an unknown transition. Following these constraints can guarantee convergence to an optimal solution in polynomial time. DOO$R_{\max}$ accomplishes this by going from the specific to the general: if the *Pickup* action is executed in the state *touchUp*(*Wall*), *touchLeft*(*Wall*), *on*(*Passenger*) and the agent observes the outcome $<Passenger.held \leftarrow True>$, DOO$R_{\max}$ assumes that *all* of those relations (including the negated ones that aren't listed) are the condition for the outcome. If the agent then tries *Pickup* in the state *touchUp*(*Wall*), *on*(*Passenger*) and observes the same outcome, it can rule out *touchLeft*(*Wall*) as a necessary condition. In this way, DOO$R_{\max}$ hones in on the true environment dynamics. This process only works in limited scenarios, and can be slow to converge as it must rule out all combinations. Improving on the efficiency of a DOO$R_{\max}$-like algorithm is a major motivating factor in my work.

### Schema Networks

*Schema Networks* [1] are graphical models that capture the causal relationships within environment dynamics. This allows the agent to simulate state transitions forward as well as backwards, so it can work backwards from future goal states. As an object-oriented planning approach, the algorithm is able to achieve rewards even when the environment is modified. The state of the environment is represented with a sets of entities, all of which have the same attributes. All entity attributes are Booleans, including numerical values, which are encoded as a collection of Booleans with one

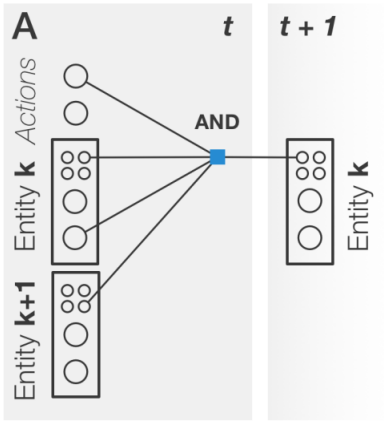


Figure 2.3: Schemas are Boolean operations on the values of entity attributes and actions at time step $t$ used to predict the value of entity attributes at time $t + 1$ [1].

variable for each possible value.

Schemas (Figure 2.3) represent case and effect with Boolean calculations over entity attributes, the result of which predicts the value of entity attributes at the next time step. Schemas are a template that can be copied across time, space, and objects, so they can capture interactions between different instances of the same object type, in different locations, at different times. To learn schemas from data, the authors use a greedy solution of linear programming relaxations, choosing schemas that have perfect precision while increasing recall on the training set. This is similar to the method I implement as discussed in Chapter 3.

Schema networks are able to predict the next state in terms of interactions between objects, and can plan actions even when the environment is changed. This is a key characteristic of my algorithms as well, however, the underlying representation is different. Each of the entity attributes in a schema network is a Boolean, which are combined with AND/ORs, whereas mine use the full representational power of first-order logic. In some sense, this could allow schema networks to discover arbitrary relations, such as $\mathit{Diagonal}(X, Y)$ using Boolean operations.

### Rule Graphs

The authors of [9] have investigated the problem of transfer learning in General Game Playing (GGP) agents. A GGP agent is given descriptions of games in the *Game Description Language* and must learn to play the game with no additional human input. Ideally, as an agent plays more and more games, it will be able to use prior experience to master newly seen games faster. The authors accomplish this by generating a *rule graph* from the game description, and searching within the graph for isomorphisms with previously seen rule graphs. This allows the agent to identify common game components, such as a turn timer. They also manually specify predetermined modifiers that can be applied to the graph to capture commonly occurring game transformations, such as a shift in game piece locations. After identifying these isomorphisms, the expected utility of the states they represent can be transferred to the target task with modifications based on which predetermined modifiers were applied to the structure of the graph. This algorithm works even when game components have been obfuscated (given different names).

A key difference is that the rule graph algorithm is given a description of the game rules ahead of time, unlike the algorithms I develop in this thesis which must explore their environment to discover environment dynamics. However, the use of isomorphisms to find similar situations could be a key factor in enabling object discovery behavior. For example, an agent could learn common configurations that objects tend to appear in in order to to make informed guesses about their identities.

# Chapter 3

# First-Order OO-MDPs

We have seen that OO-MDPs' use of propositional logic to represent relations is insufficient for reasoning over the objects in those relations, leading to an inability to transfer previous experience. In this chapter, I introduce the use of first-order logic to OO-MDPs (FO-OO-MDPs) and describe a learning algorithm that leverages this representation for efficient learning.

## 3.1 Applying First-Order Logic to OO-MDPs

To convert an OO-MDP to an FO-OO-MDP, we replace the propositional variables with first-order relations. In the Taxi domain, we have the general relations $\mathit{TouchLeft}(X, Y)$, $\mathit{TouchRight}(X, Y)$, $\mathit{TouchUp}(X, Y)$, $\mathit{TouchDown}(X, Y)$, $\mathit{On}(X, Y)$, and $\mathit{Holding}(X, Y)$. Like in the OO-MDP formulation, we could define a subset of these relations and have the agent learn only with those. Instead, to force the development of a more general learning algorithm, the agent receives *all* active relations including ones that are not explicitly required to model transitions and rewards, such as $\mathit{TouchDown}(\mathit{Passenger})$.

This representation allows the agent to distinguish specific relations and objects, which could benefit the exploration phase of learning. For example, the agent

may observe that executing *Pickup* while *On* objects often leads to the outcome $<object.held \leftarrow True>$. The agent could then prioritize this relation/action pair when interacting with new objects, to quickly learn if the object is able to be be picked up.

In the real world, the agent (such as a robot) would receive which relations are active from a visual processing system. However, for simulated environments within a computer, the designer has more control over defining relations. For example, though the relation *On* can be generally defined with object attributes as $On(A, B) \leftarrow (A.y = B.y) \wedge (A.x = B.x)$, an exception is made for walls. Although the agent may be in the same square as a wall, it makes more sense to think of walls as on the *side* of the square, so $On(Wall)$ is never true. In the computer environment, this is encoded into the $cond(s)$ function which returns the active relations for a state.

### 3.1.1 The Heist Domain

I now present the second domain of this thesis, Heist [10], as it is useful for providing examples for the upcoming algorithms. In Heist, the agent's goal is to reach and pick up the gem, which is behind three locked locks that must be opened with keys. While there are five keys, in each episode only three randomly chosen ones appear. The objects in this domain are of the classes *Agent*, *Wall*, *Key*, *Lock*, and *Gem*, and the actions available to the agent are *Up*, *Down*, *Left*, *Right*, *Pickup*, and *Unlock*. The movement actions behave as usual, with the addition of the fact that the agent can not move through a locked lock. *Pickup* will pick up a key or gem unless the agent is already holding another object, and *Unlock* will unlock a lock in a square adjacent to the agent unless blocked by a wall. The agent receives a $+5$ reward for unlocking a lock, $+10$ for picking up the gem, and $-1$ otherwise. Locks have an additional attribute *open*, and the unary relation $Open(Y)$ returns the value of this attribute.

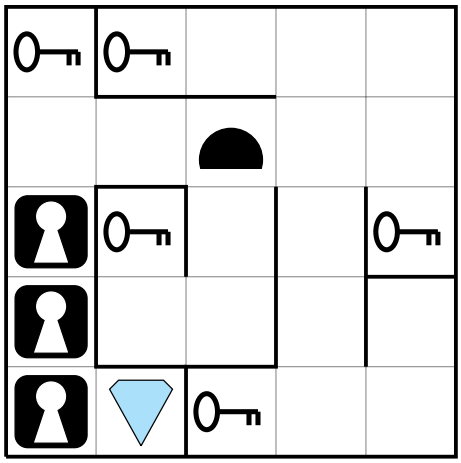

Figure 3.1: The Heist domain

## 3.2 First-Order Logic as Graphs

Environments described by first-order logic can be intuitively represented as graphs. The nodes of the graph are the objects in the environment, edges are the binary relations among objects, and unary relations are attributes attached to nodes. A full description of the state would contain every object in the environment and every established relation between them, as shown in the example in Figure 3.2. However, because the frame assumption and outcome assumption state that objects' attributes only change when the agent interacts with them, we can limit our view to a graph centered on the agent object. This is also known as the *isolated interaction* assumption [11]. As an example, a first-order graph for Heist where the relations $Holding(Key)$, $TouchDown(Wall)$ are established is shown in Figure 3.3.

The closed world assumption states that unlisted relations are assumed to be false, and so unestablished relations do not need to be included in their negated form in the state description. However, the relations used to describe the state are also used to describe a rule's context, where the use of negated relations is necessary to describe actions like *Right* with the context $\neg TouchRight(Wall)$. In graph form, these *negated edges* are drawn with dashes as illustrated in Figure 3.3.

The benefits of a graph representation include human-interpretability, the ability

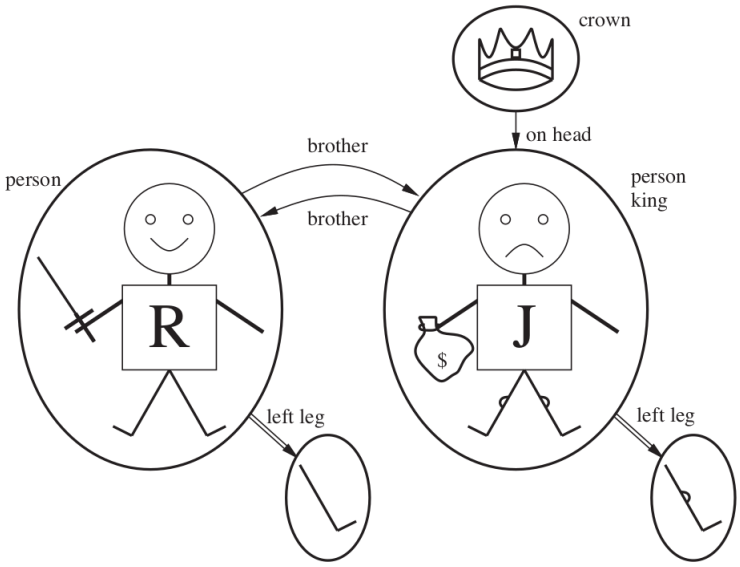


Figure 3.2: From [2], a first-order world describing King John and Richard the Lionheart, containing five objects, two binary relations, three unary relations, and a unary function.

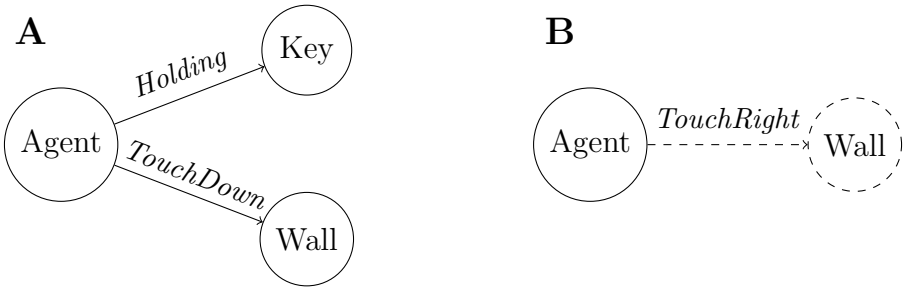


Figure 3.3: An example first-order graph (A) in Heist. The condition for the action *Right* contains a negated edge (B). The requirement that the relation not be established means that the object in question must not exist, as indicated with dashes.

to define complex algorithms with recursion, and access to the large history of graph algorithm research. Although every graph can be represented in text form, which I do in this thesis to save space, storing data as a graph allows powerful algorithms, such as the isomorphic matching from [9], to operate on and learn from patterns.

### 3.2.1 Construction

First-order graphs for a state are constructed iteratively in a breadth-first manner. Starting with the root agent node, a new object node and its relation edge are added for every object that establishes a relation with the root. All unary relations are added to the node as well. After every object node has been added, the creation process is run again with each newly-added node as a root. Limiting the depth of this recursive process creates graphs that can capture more or less complicated dynamics. As an example, consider the one-dimensional version of the classic puzzle game Sokoban, shown in Figure 3.4. In Sokoban, the agent needs to maneuver blocks onto goal squares by pushing them, but blocks can not be pushed if there is a wall or another block in the way. Therefore, describing transitions requires knowledge of more than just the objects directly interacting with the agent, and a depth limit of two is required to fully capture this structure. The iterative process of constructing the graph for the state shown in Figure 3.4 is shown in Figure 3.5, and that graph can be equivalently represented in text as:

$$TouchRight(Agent, Block), TouchRight(Agent, Goal), On(Block, Goal)$$

$$TouchRight(Block, Wall), TouchRight(Goal, Wall)$$

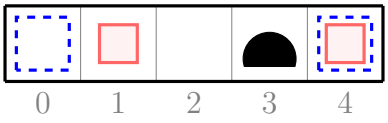


Figure 3.4: A simple Sokoban domain.

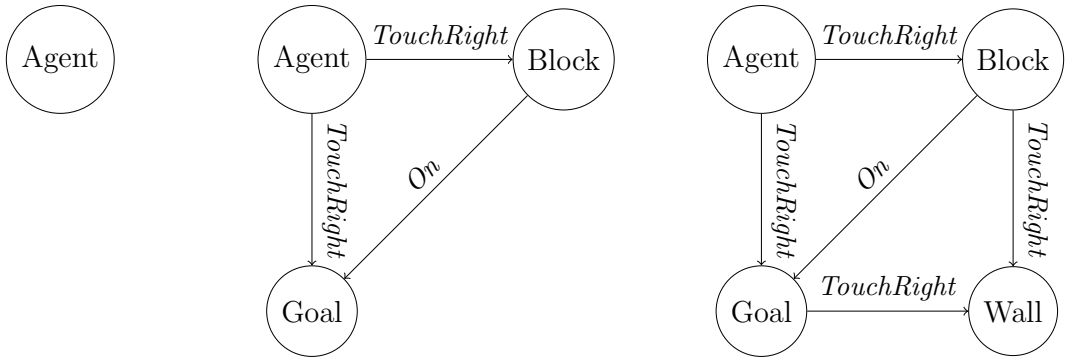


Figure 3.5: Iterative construction of first-order graphs in the Sokoban domain.

Note that there may be cycles in the graph because if $A$ is to the right of $B$, then $B$ is to the left of $A$. I do not include these cyclic edges for clarity.

The algorithms I introduce are recursive in nature, and I conjecture they will work on graphs of any size. However, a depth limit of one is sufficient to capture the transition dynamics of the domains in this thesis. I formalize this and one other assumption about first-order graphs here:

- Environment dynamics and conditions require graphs with depth limit one or less to describe.
- No two objects satisfy the same relation with the agent in a given state. For example, *TouchLeft*(*Wall*) and *TouchLeft*(*Destination*) will never be true at the same time.

Both of these assumptions are false in Sokoban. I believe that my algorithms would only need slight modifications to overcome these simplifications.

### 3.2.2 Operations on First-Order Graphs

A set of relations $c_1$ *matches* another set $c_2$, denoted $c_1 \models c_2$, if every positive relation in $c_1$ has a match in $c_2$, and no negated relation in $c_1$ has a match in $c_2$. In terms of graph operations, this matching process can be stated recursively as shown in Algorithm 1. See Figure 3.6 for examples.

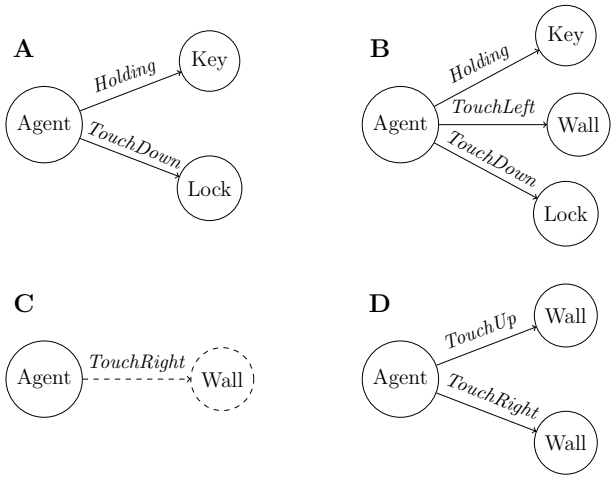


Figure 3.6: Graph A matches Graph B because every positive edge has a match. Graph C does not match Graph D, because there is a negated edge that matches.

### 3.2.3 Deictic References and Skolem Constants

Sometimes, more than one object of the same class interact with the agent. For example, if $key_5$ is to the right of $key_2$, and the agent is on $key_5$, then the state would be described as: $On(key_5)$, $TouchLeft(key_2)$. Now, if the *Pickup* action, with the following rule, is executed in this state, which key has its *held* attribute set to true?

$$Pickup : On(Key)$$
$$\longrightarrow \Big\{ Key.held \leftarrow True$$

Intuitively, the key referenced by the *On* relation will be picked up. This is captured via a *deictic reference* (pronounced die-ktik), which refers to objects through their relations. Instead of the rule using an object id to refer to a specific instance of a *Key* object, the "the-key-that-the-agent-is-on" (as it is referred to deictically) is the one whose *held* attribute is modified [5].

For real-world learning algorithms, there may be nothing that visually distinguishes objects of the same class from each other. This forces the agent to abstract transition dynamics independent of the identities of objects. To mimic this property in a simulated world, instead of referring to state objects by their unique id, they are given arbitrary identifying constants known as *Skolem constants* [5]. Instead

**Algorithm 1** Matches

**input:** graph node $n_1$, graph node $n_2$

**if** all properties of $n_1$ do not match those of $n_2$ **then** ▷ Check unary relations
  **return** False
**end if**

**for** positive edge $e_1$ in $n_1$ **do** ▷ Check positive relations
  **if** there is no edge $e_2$ with the same type and object in $n_2$ **then**
    **return** False
  **else if not** Matches($e_1.n$, $e_2.n$) **then** ▷ Recursively match on the connected nodes
    **return** False
  **end if**
**end for**

**for** negated edge $e_1$ in $n_1$ **do** ▷ Check negated relations
  **if** there is an edge $e_2$ with the same type and object in $n_2$ **then**
    **return** False
  **end if**
**end for**

**return** True

of $On(key_5)$, $TouchLeft(key_2)$, the state is represented to the agent as $On(key_{001})$, $TouchLeft(key_{002})$, where 001 and 002 are the Skolem constants. A different state like $On(key_1)$, $TouchLeft(key_4)$ would be represented the same way.

## 3.3 Learning Rule Sets with FOIL

Similar to OO-MDPs, a FO-OO-MDP rule $r$ is a tuple ($r.a$, $r.c$, $r.o$). If action $r.a$ is executed in state $s$ while the context $r.c \models cond(s)$, then outcome $r.o$ will occur. All objects in the outcome are referenced deictically, except for the agent, which is referenced in the action (For example, $Up$ implicitly refers to $Up(Agent)$).

An *example* $e$ is a tuple ($e.a$, $e.c$, $e.o$) representing a transition the agent has observed. The agent took action $e.a$, $e.c$ stores the active relations from that state

($e.c = cond(s)$), and $e.o$ was the resulting outcome. The *example set* $E$ stores all examples that the agent has experienced so far. This is similar to a memory trace, except I discarded sequential information for computational efficiency, instead only storing the count of how many times each unique example has occurred.

The goal of the *LearnRuleset* algorithm is to generate a rule set that concisely describes the examples in $E$. I take an outcome-based approach as in [5], phrasing learning as "For a given outcome, what are the situations that cause it?" instead of the usual action-oriented approach of "When an action is executed, in what situations do different outcomes occur?" This restructuring leads naturally to a learning algorithm that reasons about "causes", which is a desired property. It also allows the agent to treat different outcomes uniquely, such as skipping learning rules for *no-change* outcomes, which saves computational power. Though, an agent that did learn *no-change* rules might be able to leverage that information to avoid exploring actions that are likely to do nothing.

FOIL (First-Order Inductive Learning) [12] is a greedy algorithm for choosing literals (similar to relations) that explain examples in a training set. Consider the following example set for Heist:

$Pickup : On(Gem), TouchLeft(Wall) \longrightarrow <Gem.held \leftarrow True>$

$Pickup : TouchRight(Wall), TouchUp(Key), On(Gem) \longrightarrow <Gem.held \leftarrow True>$

$Pickup : On(Gem), TouchRight(Lock) \longrightarrow <Gem.held \leftarrow True>$

$Pickup : Holding(Key) \longrightarrow <no\text{-}change>$

A valid rule set describing these examples *could* be:

$$Pickup : On(Gem), TouchLeft(Wall)$$
$$\longrightarrow \Big\{ Gem.held \leftarrow True$$

$$Pickup : TouchRight(Wall), TouchUp(Key), On(Gem)$$
$$\longrightarrow \Big\{ Gem.held \leftarrow True$$

$$Pickup : On(Gem), TouchRight(Lock)$$
$$\longrightarrow \Big\{ Gem.held \leftarrow True$$

But we can capitalize on the fact that the difference between examples with the desired outcome (positive examples) and those without (negative examples) is the presence of the relation $On(Gem)$, to create the more concise rule set:

$$Pickup : On(Gem)$$
$$\longrightarrow \Big\{ Gem.held \leftarrow True$$

Starting at the lowest level, FOIL learns a single rule describing part of a single outcome. This is because more than one rule may be needed to describe an outcome, such as the rules for moving downwards in Heist:

$$Down : \neg TouchDown(Wall), \neg TouchDown(Lock)$$
$$\longrightarrow \Big\{ Agent.y \leftarrow Agent.y - 1$$

$$Down : TouchDown(Lock), Open(Lock)$$
$$\longrightarrow \Big\{ Agent.y \leftarrow Agent.y - 1$$

The *Learn Single Context* method (Algorithm 2) starts with a list of positive examples and a list of negative examples. Beginning with an empty context, relations are added until the context becomes specific enough to not be applicable to *any* examples in the negative list. The relation that is chosen to be added from all possible relations is the one with the highest *FOIL metric* (which I have slightly modified for our use case), a value that represents the information gained about the target outcome when the new relation is added. If $c'$ is the new context with the added relation, $p_1$ is the

**Algorithm 2** Learn Single Context

**input:** positive examples $E_+$, negative examples $E_-$, outcome $o$
Initialize new test contexts $contexts \leftarrow \emptyset$
Initialize empty best context $c^*$

**while** $E_-$ is not empty **do**
  $p_0 \leftarrow |\{e \in E_+ \mid c^* \models e.c\}|$
  $n_0 \leftarrow |\{e \in E_- \mid c^* \models e.c\}|$

  $contexts \leftarrow$ generate new contexts from $c*$
  **for** $c' \in$ contexts **do**
    $p_1 \leftarrow |\{e \in E_+ \mid c' \models e.c\}|$
    $n_1 \leftarrow |\{e \in E_- \mid c' \models e.c\}|$
    $gain \leftarrow p_1 * \log(\frac{p1}{p1+n1})$
  **end for**

  $c^* \leftarrow \text{argmax}_{c'}\, gain$
  $E_- \leftarrow \{e \in E_- \mid c^* \models e.c\}$
**end while**

**return** $c^*$

number of positive examples covered by $c'$, and $n_1$ is the number of negative examples covered by $c'$, then:

$$gain = p_1 * log(\frac{p_1}{p1 + n1}) \tag{3.1}$$

The method *Learn Rules for Outcome* (Algorithm 3) repeatedly calls *Learn Single Context*. Starting by splitting the example set into the positive examples $E_+$ and negative examples $E_-$, every time *Learn Single Context* returns a new context $c^*$ the examples that $c^*$ applied to are moved out of $E_+$ into $E_-$, and the process repeats until $E_+$ is empty. This staged construction ensures that rules are mutually exclusive. The domains in this thesis have actions that only lead to one type of outcome, so $E_-$ can additionally exclude any examples with a different action.

The full *Learn Rule Set* method simply joins the sets of rules created by *Learn Rules for Outcome* as shown in Algorithm 4. Note that FOIL requires there to be

**Algorithm 3** Learn Rules for Outcome

**input:** example set $E$, outcome $o$
Initialize empty rule set $R$

$a \leftarrow$ the action that lead to outcome $o$
$E_+ \leftarrow \{e \in E \mid e.o = o\}$
$E_- \leftarrow \{e \in E \mid e.o \neq o \wedge e.a = a\}$

**while** $E_+$ is not empty **do**
    $c \leftarrow$ Learn Single Context $(E_+, E_-)$
    $R \leftarrow R \cup \mathit{Rule}(a, c, o)$
    $C \leftarrow \{e \in E_+ \mid r \text{ is applicable to } e\}$
    $E_- \leftarrow E_- \cup C$
    $E_+ \leftarrow E_+ \setminus C$
**end while**

**return** $R$

**Algorithm 4** Learn Rule Set

Initialize $O \leftarrow$ all outcomes in $E$ except for *no-change*
Initialize empty rule set $R$
**for** $o \in O$ **do**
    $R \leftarrow R \cup$ Learn Rules for Outcome $(E, o)$
**end for**
**return** $R$

no contradictions in the example set (no two examples where $e_1.c = e_2.c$ may have different outcomes). This is not a problem in deterministic worlds, but contradictions can arise under imagined object mappings as we will discuss in Chapter 5.

This process can be improved with deictic references. In the above example, the object affected by the *Pickup* action is the gem that satisfies the relation $\mathit{On}(\mathit{Gem})$. So $\mathit{On}(\mathit{Gem})$ *must* be part of the context and is automatically added, reducing the size of the search space. This does not apply to the agent object as the agent object is already referenced from the action itself.

## 3.4 Determining State Transitions

In order to plan actions that lead to goals the agent needs to be able to predict the outcomes of its actions. The *SimulateTransition*$(R, c, a)$ method takes a rule set $R$, context $c = cond(s)$ and action $a$ and predicts the outcome $o$, from which the next state $s'$ can be calculated. First, the agent finds the rule $r \in R$ such that $r.a = a$ and $r.c \models c$. There will be at most one rule that satisfies these conditions because rules' contexts are mutually exclusive, but if no rules apply the agent predicts a *no-change* outcome and so $s' = s$. If a rule was found, the agent applies the effects in $r.o$ to the attributes of the referred-to objects, producing the next state.

## 3.5 The Main Learning Algorithm

While DOO$R_{max}$ uses deductive reasoning to find rule contexts, I propose that an inductive learning algorithm like FOIL is a more natural choice. This allows the agent to make assumptions from the data it has observed, which can lead to natural exploration and discovery. For example, a lucky agent that first executes *Right* while not next to a wall will generate a rule set with a rule that says that *Right* always moves the agent right, as it has no examples to prove otherwise. The agent now believes it can go through walls, so if there is a desired reward behind a wall, the agent will go straight for it, bumping into the wall and immediately learning the true transition dynamics!

Even so, the agent needs to be aware that it is making assumptions about environment dynamics, and in order to find new, useful, dynamics, the agent needs to track what interactions it has experienced so far. This stems from the belief that interesting things tend to happen when objects interact [13].

I define an *experience* as a tuple of an action $a$ and graph $c$. The *size* of the experience is the number of relations in the graph. The experiences generated from

an $(a, s)$ pair correspond to all possible subsets of the relations in $cond(s)$. As an example, taking the action *Dropoff* in state *TouchLeft*(*Wall*), *Holding*(*Passenger*), *TouchDown*(*Destination*) creates the following experiences:

*Dropoff* : $\emptyset$

*Dropoff* : *TouchLeft*(*Wall*)

*Dropoff* : *Holding*(*Passenger*)

*Dropoff* : *TouchDown*(*Destination*)

*Dropoff* : *TouchLeft*(*Wall*), *Holding*(*Passenger*)

*Dropoff* : *TouchLeft*(*Wall*), *TouchDown*(*Destination*)

*Dropoff* : *Holding*(*Passenger*), *TouchDown*(*Destination*)

*Dropoff* : *TouchLeft*(*Wall*), *Holding*(*Passenger*), *TouchDown*(*Destination*)

This is $O(2^N)$ in the number of relations in the state. However, in the domains in this thesis, size-two or less experiences are sufficient to capture all dynamics, so the agent only needs to keep track of this subset. This is because no transition requires interacting with more than two objects at a time. In addition, unary relations like *Open*(*Lock*) must refer to a specific lock, so are not added unless a relation that refers to a lock is also present. Like the example set, the experience set disregards sequential information and only stores how many times each experience has occurred.

With experiences defined, the agent has everything it needs for the main FO-OO-MDP learning loop (shown in Algorithm 5).

To choose actions, the agent uses *SimulateTransition* to do a breadth-first search to the nearest $(s, a)$ pair that gives a positive reward. If it is unable to find one, it searches to the nearest size-one experience that it has not yet encountered, and if it can't find that, it looks for a size-two experience. Once it encounters any of these scenarios, it executes the series of actions to reach that state. However, if at any point during the sequence the rule set changes, it restarts from the search phase. This is

**Algorithm 5** FO-OO-MDP Learner

Initialize empty rule set, example set, and experience set
**repeat**
  Observe current state $s$
  Select an action with *Choose Action*($s$)
  Execute $a$ and observe reward $r$ and outcome $o$
  Create a new example ($a$, *cond*($s$), $o$) and add it to the example set
  Add generated size-one and size-two experiences to the experience set
  Learn new rule set $R$ on the example set
**until** termination condition

**Algorithm 6** Choose Action

**input:** state $s$, rule set $R$
Use $R$ to do a breadth-first search from $s$ to the nearest state with positive reward
**if** a path is found **then**
  **return** the first action in the path
**end if**

Use $R$ to search from $s$ to the nearest unexperienced size-one experience
**if** a path is found **then**
  **return** the first action in the path
**end if**

Use $R$ to search from $s$ to the nearest unexperienced size-two experience
**if** a path is found **then**
  **return** the first action in the path
**end if**

**return** a random action

detailed in Algorithm 6. An important assumption for this method is that the agent is given the reward values of transitions. This is similar to a planning problem in that the agent knows about goal states in advance, but unlike in planning the agent has to learn for itself which actions produce which effects in order to simulate transitions to those goal states. Future work should investigate how the agent can incorporate rewards into its model of the environment.

The reason for the reward, size-one, size-two search ordering is that the agent's primary goal is to solve the environment, so it should always seek a reward if it can

find one. In addition, size-one interactions are sometimes enough to learn sufficient dynamics for this task, without spending excess time with the combinatorically larger amount of size-two experiences. However, if the agent is unlucky it will not discover the correct transitions via experimenting with size-one experiences. Consider an agent that first looks for the experience $Pickup : On(Passenger)$, then $Dropoff : On(Destination)$. Because it will have already picked up the passenger when it tries the *Dropoff* action, it will learn it can drop off the passenger. However, an agent that first attempts $Dropoff : On(Destination)$ will assume that action does nothing, and will not try again when it is holding the passenger. This is why the complete algorithm does keep track of size-two experiences while going through the environment, but only seeks them out after all size-one experiences have been exhausted.

Note that negated relations are not tracked in experiences, even though many rules use negated edges. This reduces the complexity of tracking all experiences, and works very well experimentally. This might be because size-one experiences are the equivalent of a size-two experience with any negated edge, as negated edges indicate the absence of an object. In addition, the agent does not bother with size-zero experiences, which are simply equivalent to any action with an empty graph.

# Chapter 4

# Object Discovery

The goal of an object discovery agent is to determine useful properties of unknown objects by interacting with them by using experience from objects the agent has interacted with before. In general, objects can have many unique characteristics. For example, consider a shoe. That shoe could be used to carry something, or prop open a door, or thrown as a weapon, and more. In the simple simulated environments within this thesis, objects do not have all of these different characteristics limited. So, instead of directly trying to identify unique properties of objects, I simplify the task to that of re-identifying previously seen objects. Object identities are hidden to the agent so it must identify objects through exploration, using the outcomes of the actions it takes.

To formalize this task, an object discovery agent is a transfer learning agent who needs to discover the *object mapping* $\mathcal{X}$ that relates target task objects to source task objects, similar to an action mapping or state mapping [3]. The agent trains on the source environment with object classes $C_1$, and learns a rule set $R_1$ as well as an experience and example set. The agent is then dropped into the target environment containing object classes $C_2$. Object classes could be kept, removed, added, renamed (to hide their identity from the agent), or duplicated (so that different names refer

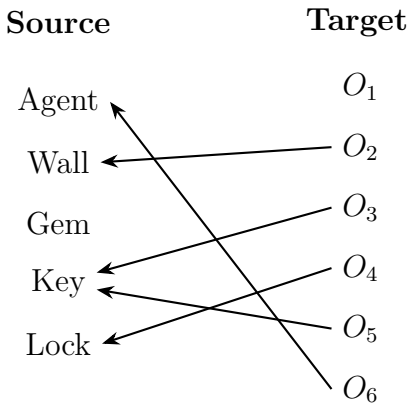


Figure 4.1: Potential object mapping showing the presence of new, removed, and duplicated objects.

to different instances of the same object type), as shown in Figure 4.1. The agent uses its observations from the source task to discover which objects are which, and learn the rules that describe the new objects (as well as update any rules from $R_1$ that describe new interactions with new objects), in order to plan actions that lead to rewards. To start, I introduce an algorithm for a simplified version of the problem, before removing many of these limitations in Chapter 5.

## 4.1 Logic-Based Object Discovery

The *logic-based* object discovery algorithm directly uses the source task rule set to identify objects and operates with the following intuition: "If I executed *Right* and didn't move, the object to my right could be a *Wall*" or "If I executed *Unlock* and nothing happened, the object I am holding is not a *Key*, and/or the object I am touching is not a *Lock*". Here, I introduce the assumptions that the logic-based object discovery algorithm will work under:

- The object map is one-to-one. Every object class in the target task has a counterpart in the source task, and vice versa.
- Trying to identify objects without fully knowing their dynamics and interactions

with other objects can lead to misidentification when they do not behave as predicted. We assume the agent has had sufficient experience in the source task to learn an accurate rule set.

- The target task has the same set of actions as the source task, and action identities are known to the agent.
- Object attributes are not obfuscated and remain identifiable by the agent.
- The agent object is not obfuscated. All actions apply to an agent, so the agent object must be known to take any action.
- As usual, all actions are deterministic.
- The behavior of objects does not change between environments. However, new objects may interact with old objects in new ways.

In the next chapter, we will relax the first three assumptions.

## 4.2 Object Maps, Assignments, and Assignment Lists

Here I describe the data structures that the logic-based object discovery agent uses to represent its knowledge of the world. The *object map* $M(C) : C_2 \rightarrow \{C_1\}$ maps each object class $c \in C_2$ to a set of object classes $\{c_1, \ldots, c_n\} \in C_1$ representing the agent's belief of the possible identities of that object. For example, in Heist, if the agent is confident that it has identified the *Wall* object but is unsure about the identities of the others, the object map may look like:

$$M = \begin{cases} O_1 : (Lock, Gem, Key) \\ O_2 : (Lock, Gem, Key) \\ O_3 : (Wall) \\ O_4 : (Lock, Gem, Key) \end{cases}$$

Because the agent object is not obfuscated, it can be excluded from the object map. Note that although the mapping operates on object classes, I often opt to describe the following concepts with the simpler word "object" and the more intuitive notation of $O_1$ instead of $c_1$.

An *object assignment* is a conjunction of statements of the form $O_i = O_j$ or $O_i \neq O_j$, representing a single possibility of object identities. Here, the notation $O_i = O_j$ represents the statement $M(O_i) = \{O_j\}$ and $O_i \neq O_j$ represents $O_j \notin M(O_i)$. A target task object can not be mapped to more than one source task object, so an object assignment can only have at most one equality assignment per object (but may have multiple inequality assignments). Here are some example object assignments:

$$(O_1 = Wall)$$

$$(O_3 \neq Key \wedge O_3 \neq Gem)$$

$$(O_1 = Lock \wedge O_2 = Key)$$

An *object assignment list* is a disjunction of object assignments, representing the multiple sets of object identities that could explain an outcome the agent observed. For example, the statement "If I executed *Unlock* and nothing happened, the object I am holding is not a *Key*, and/or the object I am touching is not a *Lock*" is expressed with two object assignments in an object assignment list:

$$(O_1 \neq Key) \vee (O_2 \neq Lock)$$

Note that at least one, but perhaps both, of the object assignments in the object assignment list must be the the true cause behind the observed outcome.

## 4.3 Calculating Object Assignment Lists

Given an example $e$ from the target task and rule set $R_1$ from the source task, the agent can generate object assignment lists that explain the observed outcome. The method $GetObjectAssignmentLists(R, e)$ returns a collection of object assignment lists, one for each $r \in R \mid r.a = e.a \wedge r.o = e.o$. However, some of these object assignment lists may be empty, because some rules are unable to give information about the identities of the objects present in $e$. This is because it may be impossible for the rule to have been applicable to $e.c$ (for example, executing *Pickup* while not *On* any object). If the rule could have applied, then the process of finding object assignments changes depending on if the outcome was or was not <*no-change*>. In total, there are three cases:

### 1) The rule is unable to apply

When object identities are not known, it is not possible to know if $r.c$ does match $e.c$, only if it *could* match. If there is a relation type in $r.c$ that does not have a corresponding relation in $e.c$, or a property of an object does not match, then it was impossible for $r$ to apply. Consider the following *Unlock* rule for Heist:

$$Unlock: TouchDown(Lock), Holding(Key), \neg Open(Lock) \longrightarrow \begin{cases} Lock.open \leftarrow True \\ Key.held \leftarrow False \end{cases}$$

Now suppose the agent observes this example:

$$Unlock: TouchDown(O_1), Open(O_1), TouchRight(O_2) \longrightarrow <no\text{-}change>$$

The *Unlock* rule could not have applied to this example because $O_1$, the object referred to deictically in *TouchDown*, has the wrong value of the property *Open*, and there is no object that satisfies the *Holding* relation. In this case we return an empty object assignment list.

Note that outcomes have to be matched deictically because object names are obfuscated. For example, a rule to pick up a key will have $r.o = <Key.held \leftarrow True>$, whereas $e.o$ might be $<O_3.held \leftarrow True>$

**2) Outcomes that are not *no-change***

If the rule could have applied and the outcome is not *no-change*, the object assignment list consists of a single object assignment where every matched positive edge turns into a positive assignment and any matched negative edge turns into a negative assignment. For instance, if the agent observes the example,

$$Unlock: TouchDown(O_1), Holding(O_2) \longrightarrow <O_1.open \leftarrow True, O_2.held \leftarrow False>$$

then this produces the object assignment list ($O_1 = Lock \wedge O_2 = Key$) when combined with the rule for *Unlock*. As another example, the rule for moving left is:

$$\begin{aligned} &Left: \neg TouchLeft(Wall) \\ &\longrightarrow \Big\{ Agent.x \leftarrow Agent.x - 1 \end{aligned}$$

and if the agent observes the example,

$$Left: TouchRight(O_1), TouchLeft(O_2) \longrightarrow <Agent.x \leftarrow Agent.x - 1>$$

then the object assignment list is ($O_2 \neq Wall$).

The assumption that there is at most one relation of each type in a state guarantees that object $O_2$ here corresponds to the *TouchLeft*(*Wall*) relation in $r$. If this assumption didn't hold, the algorithm could be modified to make one object assignment for each object referenced in each copy of the duplicated relation.

**3) *No-change* outcomes**

In this scenario, the rule could have applied, but didn't. This creates an object assignment list that is essentially the negation of the one that would have been created if the expected outcome had occurred. Every positive edge becomes an object assignment with a single negative assignment statement, and every negative edge becomes an object assignment with a single positive assignment statement. These are combined into the full object assignment list. For example, for the *Unlock* rule, if the agent observes the example:

$$Unlock: TouchDown(O_1), Holding(O_2) \longrightarrow <no\text{-}change>$$

This produces the object assignment list $(O_1 \neq Lock) \vee (O_2 \neq Key)$.

## 4.4 Reducing Object Maps with Object Assignment Lists

As the agent takes actions and observes outcomes, it grows a collection of object assignment lists which it uses to narrow down the identities of objects in its object map. The method $ReduceObjectMap(M, L) \rightarrow M'$ takes as input an object map $M$, collection of object assignment lists $L$, and returns a new object map $M'$ created by iteratively applying the following axioms with object assignment lists in $L$ until no more can be applied:

- (1) If all but one object assignment in an object assignment list are known to be false, the remaining one is true and we can apply it to the object map. An object assignment $O_1 = O_2$ can be labeled false if $O_2 \notin M(O_1)$, and an object assignment $O_1 \neq O_2$ is false if $M(O_1) = \{O_2\}$. Applying an object assignment consists of removing objects that are deemed to not be, or removing all but the

objects it is deemed to be.

- (2) The one-to-one mapping means that once an object's identity is known, no other object can be that object, so it can be removed as a possibility from the other objects.

- (3) If there is more than one non-false object assignment in an object assignment list, but each refers to the same object and contains only positive assignments, we can remove the objects that are not mentioned. For example, if $(O_1 = \mathit{Wall}) \vee (O_1 = \mathit{Lock})$, then $(O_2 \neq \mathit{Key} \wedge O_1 \neq \mathit{Gem})$.

For example, in Heist, assume the agent bumps into a wall, then picks up a key, then unlocks a lock. The real underlying object mappings are $O_1 \rightarrow \mathit{Wall}$, $O_2 \rightarrow \mathit{Key}$, $O_3 \rightarrow \mathit{Lock}$, $O_4 \rightarrow \mathit{Gem}$. Under this mapping the agent's observations produce the following set of object assignment lists,

$$(O_1 = \mathit{Wall})$$
$$(O_2 = \mathit{Key}) \vee (O_2 = \mathit{Gem})$$
$$(O_2 = \mathit{Key} \wedge O_3 = \mathit{Lock})$$

which can be used to narrow down the object map via a sequence of steps shown in Figure 4.2.

It turns out these axioms are equivalent to common operations with Boolean logic. The entire process can be reduced to a satisfiability problem with the following method: For every pair of object classes $c_i \in C_2, c_j \in C_1$, create a Boolean variable $c_{ij}$ which is true if $c_i = c_j$ and false otherwise. In Heist, with four objects excluding the agent, there are 16 variables, one for each possible entry in the object map. The constraint that every object $c_i \in C_2$ is mapped to exactly one object $c_j \in C_1$ can be written as:

$$(c_{i1} \vee c_{i2} \vee c_{i3} \vee c_{i4}) \wedge \neg(c_{i1} \wedge c_{i2}) \wedge \neg(c_{i1} \wedge c_{i3}) \wedge$$
$$\neg(c_{i1} \wedge c_{i4}) \wedge \neg(c_{i2} \wedge c_{i3}) \wedge \neg(c_{i2} \wedge c_{i4}) \wedge \neg(c_{i3} \wedge c_{i4})$$

$$M_1 = \begin{cases} O_1 : (Lock, Key, Gem, Wall) \\ O_2 : (Lock, Key, Gem, Wall) \\ O_3 : (Lock, Key, Gem, Wall) \\ O_4 : (Lock, Key, Gem, Wall) \end{cases} \quad M_4 = \begin{cases} O_1 : (Wall) \\ O_2 : (Key, Gem) \\ O_3 : (Lock, Key, Gem) \\ O_4 : (Lock, Key, Gem) \end{cases}$$

$$M_2 = \begin{cases} O_1 : (Wall) \\ O_2 : (Lock, Key, Gem, Wall) \\ O_3 : (Lock, Key, Gem, Wall) \\ O_4 : (Lock, Key, Gem, Wall) \end{cases} \quad M_5 = \begin{cases} O_1 : (Wall) \\ O_2 : (Key) \\ O_3 : (Lock) \\ O_4 : (Lock, Key, Gem) \end{cases}$$

$$M_3 = \begin{cases} O_1 : (Wall) \\ O_2 : (Lock, Key, Gem) \\ O_3 : (Lock, Key, Gem) \\ O_4 : (Lock, Key, Gem) \end{cases} \quad M_6 = \begin{cases} O_1 : (Wall) \\ O_2 : (Key) \\ O_3 : (Lock) \\ O_4 : (Gem) \end{cases}$$

Figure 4.2: $M_1$: Initial object map. $M_2$: After applying Axiom 1 with object assignment list 1. $M_3$: After applying Axiom 2. $M_4$: After applying Axiom 1 with list 2. $M_5$: After applying Axiom 2 with list 3. $M_6$: After applying Axiom 1.

A similar expression describes the constraint that every object class $c_j$ is mapped to exactly one class $c_i$. All of the object assignment lists can be directly converted to Boolean expressions by replacing each equality/inequality statement with its variable, and concatenated together with ANDs. To reduce the object map, we need which object mappings are *possible*, so we set each variable one-by-one to true, and ask the satisfiablity solver if the formula is satisfiable. In this way, we weed out object mappings which are not possible, reducing the object map. For example, Axiom 3 is equivalent to the statement $(a \vee b \vee c \vee d) \wedge (a \vee b) \rightarrow (\neg c \wedge \neg d)$.

## 4.5 Taking Actions to Reveal Object Identities

Only certain actions taken in certain scenarios allow the agent to gain information about the identities of objects. In order to plan actions that lead to gaining this information, the agent imagines what outcome it would observe under different object mappings and how that would effect the information it gains. The agent uses the

method *Remap*, which replaces all objects referred to in relations with their mapped value. *Remap* can be applied to any structure with relations, such as a rule, example, or outcome. For example, if the agent has the (partial) mapping $\mathcal{X} = (O_1 : \mathit{Wall}, O_2 = \mathit{Gem})$ and observes the relations,

$$\mathit{TouchDown}(O_1), \mathit{Holding}(O_2), \mathit{TouchLeft}(O_1)$$

then *Remap* would return

$$\mathit{TouchDown}(\mathit{Wall}), \mathit{Holding}(\mathit{Gem}), \mathit{TouchLeft}(\mathit{Wall})$$

The agent uses its current object map to generate all possible mappings for a given state. The $\mathit{GenerateMappings}(s, M)$ creates permutations of each list of possible object identities for every object referenced in the relations in the state. For example, if a set of relations contains the objects $(O_2, O_3)$ and the current object map is:

$$M = \begin{cases} O_1 : (\mathit{Lock}, \mathit{Key}, \mathit{Gem}) \\ O_2 : (\mathit{Lock}, \mathit{Wall}) \\ O_3 : (\mathit{Key}, \mathit{Gem}) \\ O_4 : (\mathit{Lock}, \mathit{Key}, \mathit{Gem}, \mathit{Wall}) \end{cases}$$

then *GenerateMappings* produces the mappings:

$$\mathcal{X}_1 = (O_2 : \mathit{Lock}, O_3 : \mathit{Key})$$

$$\mathcal{X}_2 = (O_2 : \mathit{Lock}, O_3 : \mathit{Gem})$$

$$\mathcal{X}_3 = (O_2 : \mathit{Wall}, O_3 : \mathit{Key})$$

$$\mathcal{X}_4 = (O_2 : \mathit{Wall}, O_3 : \mathit{Gem})$$

Though none were created in this example, mappings with duplicate objects can be discarded because of the one-to-one mapping assumption.

**Algorithm 7** Information Gain of State

**input:** state $s$, current object map $M$
Persistent: Set of object assignment lists $L$, agent's rule set $R$
**function** INFORMATION GAIN OF MAPPING($s$, $a$, $M$, $\mathcal{X}$)
  Remap state literals $c \leftarrow \mathit{Remap}(\mathit{cond}(s), \mathcal{X})$
  Simulate transition $o \leftarrow \mathit{SimulateTransition}(R, c, a)$
  Create new imagined example $e \leftarrow (a, c, o)$
  Get object assignment lists $L' \leftarrow \mathit{GetObjectAssignmentLists}(R, e)$
  Add to the current set $L \leftarrow L \cup L'$
  Reduce object map $M' \leftarrow \mathit{ReduceObjectMap}(M, L)$
  **return** $IG(M, M')$
**end function**

**function** INFORMATION GAIN OF ACTION($s$, $a$, $M$)
  **return** $\frac{1}{N_{mappings}} \sum_{\mathcal{X} \in GenerateMappings(s,M)} \mathit{Information\ Gain\ of\ Mapping}(s, a, M, \mathcal{X})$
**end function**

**return** $\frac{1}{|A|} \sum_{a \in A} \mathit{Information\ Gain\ of\ Action}(s, a, M)$

### 4.5.1 Information Gain

When the agent finds itself in some state, it uses *Remap* with its current object map to generate all possible variations of that state, and calls *SimulateTransition* on each to see what the outcome of all of its actions would be in that state. Using that outcome, it applies *ReduceObjectMap* to imagine how much its object map would be reduced if that outcome occurred for real. To make this quantitative, the *information gain* of reducing an object map is:

$$IG(M, M') = \sum_{c \in C_2} log_2(|M(c)|) - log_2(|M'(c)|) \tag{4.1}$$

where the log is taken to represent the number of bits needed to encode the information present in the object possibilities, and helps mathematically capture the intuition that more information is gained when reducing one object from $3 \rightarrow 1$ possibilities than two objects to $3 \rightarrow 2$ possibilities. The full algorithm is shown in Algorithm 7.

The agent plans a path to states with high information gain, and once in those

states it can take the action with the highest expected information gain.

### 4.5.2 Illustrative Example

For this example, we switch back to the Taxi domain as there are less objects to keep track of. Consider a state in Taxi where $On(Destination)$ and $TouchRight(Wall)$ are true (the agent does not know these true identities, it only sees the objects $O_3$ and $O_2$). The subset of Taxi rules that will be relevant for this example are:

$$Right : \neg TouchRight(Wall) \longrightarrow \Big\{ Agent.x \leftarrow Agent.x + 1$$

$$Pickup : On(Passenger) \longrightarrow \Big\{ Passenger.held \leftarrow true$$

To start, the agent's initial object map is:

$$M_1 = \begin{cases} O_1 : (Passenger, Destination, Wall) \\ O_2 : (Passenger, Destination, Wall) \\ O_3 : (Passenger, Destination, Wall) \end{cases}$$

The agent observes the current state and sees the relations $On(O_3)$, $TouchRight(O_2)$ and the objects in the state are $(O_3, O_2)$. $GenerateMappings(s, M_1)$, after removing the permutations with duplicate assignments, returns:

$$\mathcal{X}_1 = (O_3 : Passenger, O_2 : Destination)$$

$$\mathcal{X}_2 = (O_3 : Passenger, O_2 : Wall)$$

$$\mathcal{X}_3 = (O_3 : Destination, O_2 : Passenger)$$

$$\mathcal{X}_4 = (O_3 : Destination, O_2 : Wall)$$

$$\mathcal{X}_5 = (O_3 : Wall, O_2 : Passenger)$$

$$\mathcal{X}_6 = (O_3 : Wall, O_2 : Destination)$$

Next, *Remap* uses these mappings to convert the relations to:

$$On(Passenger),\ TouchRight(Destination)$$

$$On(Passenger),\ TouchRight(Wall)$$

$$On(Destination),\ TouchRight(Passenger)$$

$$On(Destination),\ TouchRight(Wall)$$

$$On(Wall),\ TouchRight(Passenger)$$

$$On(Wall),\ TouchRight(Destination)$$

The agent then simulates transitions for each action and permutation. Starting with the action *Right*, the only rule with the same action and outcome is:

$$\begin{aligned} &Right : \neg TouchRight(Wall) \\ &\longrightarrow \Big\{ Agent.x \leftarrow Agent.x + 1 \end{aligned}$$

With the first mapping $\mathcal{X}_1 = (O_3 : Passenger, O_2 : Destination)$, the imagined state is $On(Passenger),\ TouchRight(Destination)$. The rule is applicable in this state, so the agent predicts the outcome $<Agent.x \leftarrow Agent.x + 1>$. Combining this information into an example $e$ and calling $GetObjectAssignmentLists(R_1, e)$ returns $(O_2 \neq Wall)$. *ReduceObjectMap* then takes this object assignment list and returns an updated object map:

$$M_2 = \begin{cases} O_1 : (Passenger, Destination, Wall) \\ O_2 : (Passenger, Destination) \\ O_3 : (Passenger, Destination, Wall) \end{cases}$$

for an $IG(M_1, M_2) = (log(3) + log(3) + log(3)) - (log(3) + log(2) + log(3)) = 0.585$.

Next, the mapping $(O_3 : Passenger, O_2 : Wall)$ creates the imagined state $On(Passenger)$, $TouchRight(Wall)$, which leads to an outcome of *no-change*, object assignment list $(O_2 = Wall)$, and new object map of:

$$M_3 = \begin{cases} O_1 : (Passenger, Destination) \\ O_2 : (Wall) \\ O_3 : (Passenger, Destination) \end{cases}$$

for an $IG(M_1, M_3) = (log(3)+log(3)+log(3))-(log(2)+log(1)+log(2)) = 2.755$. In the next four mappings, the other mapping where $O_2 = Wall$ will produce the same gain of 2.755, and the other three where $O_2 \neq Wall$ produces the same 0.585. So, the expected information gain for the *Right* action is $\frac{1}{6}(2 * 2.755 + 4 * 0.585) = 1.308$.

The *Pickup* action follows much the same pattern, as just as there is only one object that stops movement of the agent to the right ($Wall$), there is only one object in Taxi that can be picked up ($Passenger$). So the expected information gain for *Pickup* is also 1.308. The other four actions give an information gain of 0, as each action will provide no information about the objects. If the agent tries a movement action, it will imagine that it will move, as there are no objects to block its way no matter the object mapping. For *Dropoff*, the agent is not holding anything, so it wouldn't expect anything to happen regardless of object mappings. In total, the expected information gain for this state is $\frac{1}{6}(1.308 + 1.308 + 0 + 0 + 0 + 0) = 0.436$.

Say the agent chooses to execute *Pickup*. Nothing happens, as the real identity of $O_3$ is *Destination*. The agent now knows that $(O_3 \neq Passenger)$, and updates its

object map:

$$M' = \begin{cases} O_1 : (Passenger, Destination, Wall) \\ O_2 : (Passenger, Destination, Wall) \\ O_3 : (Destination, Wall) \end{cases}$$

Going through the same process with the updated object map would return an information gain of 0 for *Pickup*, as it has already ruled out the possibility that ($O_3 = Passenger$), *GenerateMappings* would not create any permutations as such, and all simulated *Pickup* actions would lead to the same *no-change* outcome. The agent would instead choose the only action with positive information gain, *Right*. It would observe a *no-change* outcome, learn that $O_2 = Wall$, which allows $O_3$ to be assigned *Destination*, leaving only *Passenger* for $O_1$. In this way, the agent has solved a simple object discovery problem by taking two carefully chosen actions.

Note that the agent treats all permutations as equally likely, regardless of the fact that the agent has never been *On* a *Wall*, nor (for example) simultaneously been *On* the *Destination* while next to a *Passenger* (as the layout of the Taxi domain prevents this). Using heuristics like these could be used to skip certain permutations to save computational power, or calculate a "weighted information gain" based on the estimated likelihood of each permutation. Finding all of the permutations is $O(N^N)$ time complexity in general, however as objects are narrowed down the number of permutations dramatically decreases.

## 4.6 Simulating Transitions with Unknown Objects

When every object's identity is known, *SimulateTransition* works as usual. However, even with unknown objects it is still possible to predict the outcome of certain actions. For example, an object that is undecided between *Key* and *Gem* will still be picked

**Algorithm 8** Simulate Transition with Unknown Objects

**input:** state $s$, action $a$, current object map $M$
Persistent: agent's rule set $R$
**for** $\mathcal{X} \in \mathit{GenerateMappings}(s, M)$ **do**
    Remap state literals $c \leftarrow \mathit{Remap}(\mathit{cond}(s), \mathcal{X})$
    Simulate transition $o \leftarrow \mathit{SimulateTransition}(R, c, a)$
**end for**

**if** all $o$ are the same **then**
    **return** o
**else**
    **return** None
**end if**

up when *Pickup* is executed. Or, the object's identity may not matter, like if there is an object to the left of the agent when it executes *Up*. To simulate transitions with unknown objects, the agent imagines the outcome for every mapping. If they all agree, then the agent can predict that that will be the outcome. Otherwise, it does not know exactly what will happen. This is shown in Algorithm 8. The calculations are the same as in *Information Gain of Action*, which presents a future opportunity for optimization.

## 4.7 Putting It All Together

In the full logic-based object discovery algorithm, the agent uses its current object map to simulate transitions and search until it finds a state containing an action with a non-zero information gain. It then executes the plan to get to that state and takes the action, repeating until the identity of every object is known. Along the way, it keeps track of the object assignment lists it gains after each action, and uses them to narrow down the object map. After every object is identified, the agent can fully simulate all transitions, and it searches for states with positive reward in order to end the episode. Note that the agent can never get "stuck" and is always able to reach a state with information gain. If, at any point it is unable to simulate a transition to

the next state, this must mean it is unsure about the identities of the objects. But then, the outcome of that action must reveal information about one of the objects!

# Chapter 5

# Simplest-Explanation Object Discovery

The one-to-one mapping assumption means that logic-based object discovery is unequipped to handle new, duplicate, or missing objects. In addition, the algorithm can not update its rule set if new object behaviors are discovered. To address these limitations, I introduce *simplest-explanation object discovery*, which combines rule set learning with object identification. Instead of observing outcomes one by one and using the rule set $R_1$ to rule out or confirm certain object mappings, this method takes a more holistic approach.

## 5.1 Simplicity Scores

To determine which object mappings are more likely to be correct, the agent generates a potential mapping $\mathcal{X}$ and uses it to remap the examples in the target task's example set $E_2$ to create a new set, $E_m$. Then, the agent calls *LearnRuleset* on the union of the source task example set $E_1$ with $E_m$. The resulting rule set and mapping that produced it are given a *simplicity score*,

$$S(\mathcal{X}) = S(R) = -\sum_{r \in R} |r.context| \tag{5.1}$$

defined as the negation of the total number of relations in the rules' contexts. The guiding principle is that a mapping that produces a simpler rule set "meshes" with the agent's previous experience. For deterministic worlds, Equation 5.1 has been experimentally verified to perform well. For stochastic worlds, a rule set's simplicity could be based on the likelihood of explaining the example sets, as in [5]. More nuanced scores could also take into account the number of unique objects or literals present in the rule set.

Using simplicity scores, we create a new *ReduceObjectMap* algorithm. In general, many object mappings will produce rule sets with the same simplicity score, as many mappings may satisfy the explanations for what the agent has observed in the target task. To convert these "best" rule sets to the object map that the agent uses for reasoning, every object present in a mapping that generated a best rule set is given an additional point. Then, for each unknown object in the object map, the objects with the highest point total are kept. The full process is described in Algorithm 9, and a worked example is gone over in the next section. Because we are no longer assuming a one-to-one mapping, *GenerateMappings* is allowed to return mappings with duplicate objects. In addition, instead of creating mappings containing only the objects in the current state, the mappings contain every object encountered so far because the simplest rule set must fit everything the agent has observed.

### 5.1.1 A Worked Example

Consider a two-dimensional world consisting only of an *Agent*, *Gem*, *Key*, and *Wall*. Assume the agent has already executed *Pickup* on both *Gem* and *Key*, as well as executing *Right* both with and without a wall in the way. This produces the example set $E_1$:

**Algorithm 9** Reduce Object Map

**Inputs:** source task examples $E_1$, target task examples $E_2$, object map $M$
Initialize empty object counts $C$
**for** each object mapping $\mathcal{X}$ in *GenerateMappings*($M$) **do**
  Remap $E_2$ examples with $\mathcal{X}$, $E_m = Remap(E_2, \mathcal{X})$
  Create joint examples $E' = E_1 \cup E_m$
  Learn new rule set on joint examples $R = LearnRuleset(E')$
  Calculate score $s = S(R)$
**end for**
**for** mapping $\mathcal{X}$ in the best scored mappings **do**
  **for** object $o_2$ in $Dom(\mathcal{X})$ **do**
    Increment counts $C(o_2, \mathcal{X}(o_2)) = C(o_2, \mathcal{X}(o_2)) + 1$
  **end for**
**end for**
**for** object $o_2$ in $Dom(C)$ **do**
  Keep the best objects in $M$, $M(o_2) = \{o_1 \in M(o_2) \mid C(o_2, o_1) = \max(C(o_2))\}$
**end for**
**return** updated object map $M'$

$$Pickup : On(Gem) \longrightarrow <Gem.held \leftarrow True>$$

$$Pickup : On(Key) \longrightarrow <Key.held \leftarrow True>$$

$$Right : \emptyset \longrightarrow <Agent.x \leftarrow Agent.x + 1>$$

$$Right : TouchRight(Wall) \longrightarrow <no\text{-}change>$$

Calling *LearnRuleset* on these examples would produce a rule set with a simplicity score of $-3$:

$$Pickup : On(Gem) \longrightarrow \Big\{ Gem.held \leftarrow True$$

$$Pickup : On(Key) \longrightarrow \Big\{ Key.held \leftarrow True$$

$$Right : \neg TouchRight(Wall) \longrightarrow \Big\{ Agent.x \leftarrow Agent.x + 1$$

The agent is then transferred to the target task where the object names have been obfuscated as *Agent* → *Agent*, *Gem* → *Dpamn*, *Key* → *Idpyo*, and *Wall* → *Tyyaw* respectively. Here, I introduce object identities as random five-character strings, which are easier for the human brain to recognize and keep track of.

The agent's object map starts out as:

$$M = \begin{cases} Idpyo : (Gem, Key, Wall) \\ Dpamn : (Gem, Key, Wall) \\ Tyyaw : (Gem, Key, Wall) \end{cases}$$

If the agent is currently on a *Gem*, next to a *Wall*, and executes the action *Right*, the agent will observe *no-change*. With the obfuscation, the example is represented as:

$$Right : On(Dpamn), TouchRight(Tyyaw) \longrightarrow <no\text{-}change>$$

*GenerateMappings* will produce the following nine object mappings for this state:

$$\mathcal{X}_1 = (Dpamn : Gem, Tyyaw : Gem)$$

$$\mathcal{X}_2 = (Dpamn : Gem, Tyyaw : Key)$$

$$\mathcal{X}_3 = (Dpamn : Gem, Tyyaw : Wall)$$

$$\mathcal{X}_4 = (Dpamn : Key, Tyyaw : Gem)$$

$$\mathcal{X}_5 = (Dpamn : Key, Tyyaw : Key)$$

$$\mathcal{X}_6 = (Dpamn : Key, Tyyaw : Wall)$$

$$\mathcal{X}_7 = (Dpamn : Wall, Tyyaw : Gem)$$

$$\mathcal{X}_8 = (Dpamn : Wall, Tyyaw : Key)$$

$$\mathcal{X}_9 = (Dpamn : Wall, Tyyaw : Wall)$$

Considering $\mathcal{X}_4$ as an example mapping, $Remap(E_2, \mathcal{X}_4)$ would yield:

$$Right : On(Key), TouchRight(Gem) \longrightarrow <no\text{-}change>$$

which when added to $E_1$ would yield the joint example set $E'$ of:

$$Pickup : On(Gem) \longrightarrow <Gem.held \leftarrow True>$$

$$Pickup : On(Key) \longrightarrow <Key.held \leftarrow True>$$

$$Right : \emptyset \longrightarrow <Agent.x \leftarrow Agent.x + 1>$$

$$Right : TouchRight(Wall) \longrightarrow <no\text{-}change>$$

$$Right : On(Key), TouchRight(Gem) \longrightarrow <no\text{-}change>$$

When the agent learns a rule set on this new example set, the rule set will have to change in order to explain this additional imagined experience. Now, it has a simplicity score of $-4$:

$$Pickup : On(Gem)$$
$$\longrightarrow \Big\{ Gem.held \leftarrow True$$

$$Pickup : On(Key)$$
$$\longrightarrow \Big\{ Key.held \leftarrow True$$

$$Right : \neg TouchRight(Wall), \neg TouchRight(Gem)$$
$$\longrightarrow \Big\{ Agent.x \leftarrow Agent.x + 1$$

The only mappings where $S(R)$ remains at $-3$ are where $\mathcal{X}(Tyyaw) = Wall$. This matches the agents prior experience with walls, and the rule set remains the same size. So, the best mappings are:

$$\mathcal{X}_3 = (Dpamn : Gem, Tyyaw : Wall)$$

$$\mathcal{X}_6 = (Dpamn : Key, Tyyaw : Wall)$$

$$\mathcal{X}_9 = (Dpamn : Wall, Tyyaw : Wall)$$

Each time an object is referenced, a point is added to its total:

$$C = \begin{cases} Dpamn : (1, 1, 1) \\ Idpyo : (0, 0, 0) \\ Tyyaw : (0, 0, 3) \end{cases}$$

The objects with the most points in each row are kept. So, the object map is reduced to:

$$M' = \begin{cases} Dpamn : (Gem, Key, Wall) \\ Idpyo : (Gem, Key, Wall) \\ Tyyaw : (Wall) \end{cases}$$

As a second example, if the agent had executed *Pickup* instead of *Right*, it would have picked up the *Gem* and observed:

$$Right : On(Dpamn), TouchRight(Tyyaw) \longrightarrow <Dpamn.held \leftarrow True>$$

Here, the best mappings are where *Dpamn* is mapped to either *Gem* or *Key*, as both can be picked up. Adding either of these remapped examples to $E_1$ would lead to a rule set with no changes. So, the best mappings are:

$$\mathcal{X}_1 = (Dpamn : Gem, Tyyaw : Gem)$$

$$\mathcal{X}_2 = (Dpamn : Gem, Tyyaw : Key)$$

$$\mathcal{X}_3 = (Dpamn : Gem, Tyyaw : Wall)$$

$$\mathcal{X}_4 = (Dpamn : Key, Tyyaw : Gem)$$

$$\mathcal{X}_5 = (Dpamn : Key, Tyyaw : Key)$$

$$\mathcal{X}_6 = (Dpamn : Key, Tyyaw : Wall)$$

Which would produce point totals of,

$$C = \begin{cases} Dpamn : (3, 3, 0) \\ Idpyo : (0, 0, 0) \\ Tyyaw : (2, 2, 2) \end{cases}$$

And a new object map,

$$M' = \begin{cases} Dpamn : (Gem, Key) \\ Idpyo : (Gem, Key, Wall) \\ Tyyaw : (Gem, Key, Wall) \end{cases}$$

The agent has successfully identified *Dpamn* as being a *Gem* or a *Key*, just as logic-based object discovery would have. In fact, for scenarios with an accurate rule set and no new objects these methods produce equivalent results.

## 5.2 Handling New Objects and Inaccurate Rules

Until now, mappings consisted only of target task objects being mapped to source task objects. With the presence of new objects, the agent is now allowed to map a target task object to itself, which declares that the object is unique to the target task. However, if the agent always entertained the possibility that each target task object is a new object in addition to checking mappings where target task objects are mapped to source objects, this would be exceedingly wasteful as combinatorically more mappings would have to be tested. In addition, until a new behavior is discovered, these mappings would produce worse simplicity scores. This is because duplicate rules would be needed to describe the dynamics of each of the object's assumed identity. For example, imagine an agent trained in Heist, which is transferred to Taxi. The agent has only encountered keys and gems, never a passenger. When it first picks up the passenger, mapping *Passenger* to a unique identifier would require

an additional *Pickup* rule in addition to the already existing rules for *Gem* and *Key*. However, because of the common ability to be picked up, mapping the obfuscated *Passenger* to either a *Gem* or a *Key* would require no additional rules to describe. In some sense, the agent has extracted the *pick-up-able-ness* of the objects.

This changes when all mappings that assume that there are no new objects have worse simplicity scores than $S(R_1)$. This means the agent was not able to mesh its new observations perfectly with those from the source task, and the rule sets have had to grow more complex. Either, $R_1$ was not a complete description of the source task dynamics, or there are new objects with new dynamics in the target task. Because the agent does not know which is the case, it now has to test mappings with new objects to see if any produce a simpler rule set. However, mappings without new objects may still produce higher simplicities, even if the behavior was caused by a new object. For example, if the agent that had never seen a passenger drops of the passenger, mapping the *Passenger* to a new object would require two new rules, one for *Pickup* and one for *Dropoff*. However, the assumption that this outcome represents new, undiscovered behavior of the *Gem* or *Key* would require only one additional rule, that for *Dropoff*. It is only when the agent tries something contradictory, such as using its belief that a key can be dropped off to drop off a key and observing no effect, that mapping *Passenger* to a new object produces higher simplicity scores. In some cases, it is impossible for a new object to be mapped to a previous object, if that mapping produces contradictory examples, i.e., the same action in the same state but different outcomes occurred. This shows that simplicity doesn't always lead to correct assumptions, but does lead to finding common behavior between objects.

## 5.3 Choosing Actions

To choose actions, the agent needs to use its object map to simulate transitions, which is similar to how it is done in logic-based object discovery. To simulate a transition, the agent looks through each of its mappings that produced a best rule set. Because the mapping may be not contain every object, as the agent may not yet have interacted with every object, the mapping is used as a starting point to reduce the object map. *GenerateMappings* is called on the resulting map, and the rule set associated with the original mappings is used to simulate the transition for each mapping. If every permutation gives the same result, then the agent knows that will be the outcome. If all of the best mappings agree on the outcome, that is what the agent predicts, otherwise, it assumes *no-change*.

As in logic-based object discovery, the agent proceeds in stages. Using *SimulateTransition*, the agent starts by searching for a sequence of actions that lead to a reward. If a path can't be found, the agent then searches for information gain. Interestingly, the same information gain calculation as the logic-based object discovery agent can be used. I believe this is again due to the computational benefit of assuming that there are no new objects until it becomes simpler to do so. If a path is again unable to be found, then the agent is stuck because it does not know the full transition dynamics due to incomplete information about old objects or the presence of new objects. So, the agent begins searching for new behavior starting with size-one, then size-two experiences, just as in the first-order object-oriented learner.

# Chapter 6

# Empirical Evaluation

Here I present an empirical evaluation of the algorithms introduced in Chapters 3, 4, and 5. These evaluations will be done in Taxi (Figure 6.1), Heist (Figure 6.2), and a new environment known as Prison (Figure 6.3). In order to evaluate object discovery scenarios that contain new objects not present in the source task, I have designed a new domain combining the characteristics of Taxi and Heist. The Prison domain, so called because the agent must help the passenger escape from behind the locks, contains the objects *Agent*, *Wall*, *Key*, *Lock*, *Passenger*, and *Destination*, and the actions available to the agent are *Up*, *Down*, *Left*, *Right*, *Pickup*, *Unlock*, and *Dropoff*. All actions and objects behave as in the previous domains, except that in order to ensure that the agent unlocks the locks first, the passenger pick-up location is only ever on the red square. The episode ends when the passenger is dropped off, and the agent receives a +5 reward for unlocking a lock, +10 for dropping off the passenger, and −1 otherwise.

To summarize, here are the hypotheses I will be evaluating:

1. The inductive nature of the FO-OO-MDP learner allows it to learn much faster than DOO$R_{\max}$ and similar algorithms. In addition, removing extraneous input to the algorithm by restricting the available relations will also increase efficiency.

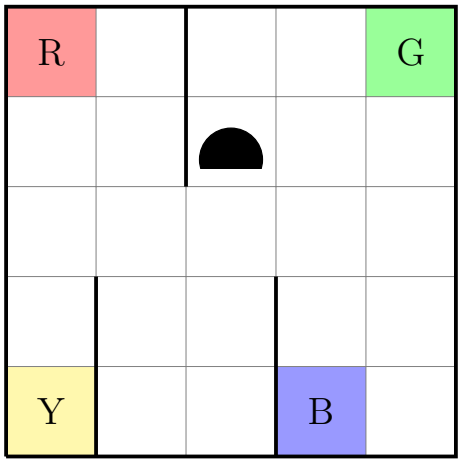


Figure 6.1: The Taxi domain

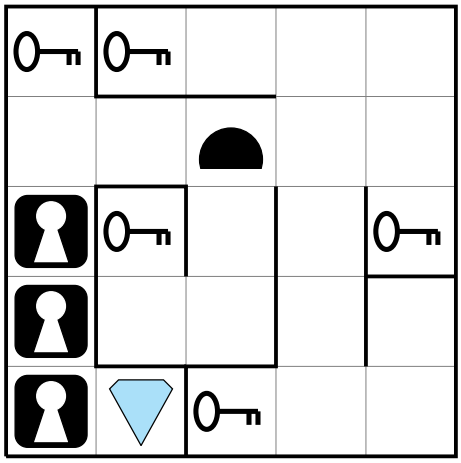

Figure 6.2: The Heist domain

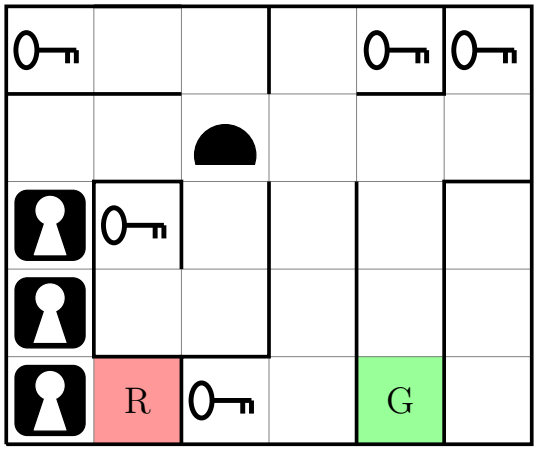


Figure 6.3: The Prison domain

2. Logic-based object discovery allows an agent to learn faster then if it was starting from scratch. In addition, if the algorithm is given prior knowledge of specific objects' identities, it can utilize this information to be even more efficient.

3. Simplest-explanation object discovery works in the presence of new objects, unlike logic-based object discovery, and also increases efficiency compared to learning from scratch.

## 6.1 Hypothesis 1: The FO-OO-MDP Learner

Because of the sparsity of the rewards in the environment and the rapidity at which the FO-OO-MDP learning algorithm is able to discover the correct transition dynamics, there is no need to report learning curves. Instead, I record how many steps it takes for the agent to complete the first episode. To recap, finishing the episode requires dropping off the passenger in Taxi and Prison, and picking up the gem in Heist. In Taxi the agent starts in a random square that is not on a pick-up or drop-off location, and pick-up and drop-off locations are randomized as described in Chapter 2. In Heist, the agent starts on a random square that is not on the key, lock, or gem, nor would cause the agent to be trapped behind a lock. Key locations are randomized as in Chapter 3. The same holds true for Prison. The max episode length for each environment is set to 200, 250, and 300 respectively, and if the agent is not able to complete the environment in time this is considered to be a failure. For each environment, I run 300 trials to get a distribution of the algorithm's performance.

In addition, to demonstrate how the availability of all possible relations impacts learning (not only those required to describe transition dynamics), I rerun the above experiments with only the necessary relations accessible to the agent. In Taxi, these are:

| | **Taxi** | **Heist** | **Prison** |
|---|---|---|---|
| Steps (All relations) | $82 \pm 26$ | $113 \pm 27$ | $191 \pm 29$ |
| Steps (Reduced relations) | $70 \pm 22$ | $116 \pm 36$ | $151 \pm 36$ |
| Failure rate (All relations) | 0% | 14% | 10% |
| Failure rate (Reduced relations) | 0% | 4% | 1% |

Table 6.1: The mean number of steps required to complete an episode, as well as the percentage of time the agent was unable to complete the episode.

$$\mathit{TouchRight}(\mathit{Wall}), \mathit{TouchLeft}(\mathit{Wall}), \mathit{TouchUp}(\mathit{Wall}), \mathit{TouchDown}(\mathit{Wall}),$$
$$\mathit{On}(\mathit{Passenger}), \mathit{On}(\mathit{Destination}), \mathit{Holding}(\mathit{Passenger})$$

In Heist,

$$\mathit{TouchRight}(\mathit{Wall}), \mathit{TouchLeft}(\mathit{Wall}), \mathit{TouchUp}(\mathit{Wall}), \mathit{TouchDown}(\mathit{Wall}),$$
$$\mathit{TouchRight}(\mathit{Lock}), \mathit{TouchLeft}(\mathit{Lock}), \mathit{TouchUp}(\mathit{Lock}), \mathit{TouchDown}(\mathit{Lock}),$$
$$\mathit{On}(\mathit{Key}), \mathit{On}(\mathit{Gem}), \mathit{Holding}(\mathit{Key}), \mathit{Open}(\mathit{Lock})$$

and in Prison:

$$\mathit{TouchRight}(\mathit{Wall}), \mathit{TouchLeft}(\mathit{Wall}), \mathit{TouchUp}(\mathit{Wall}), \mathit{TouchDown}(\mathit{Wall}),$$
$$\mathit{TouchRight}(\mathit{Lock}), \mathit{TouchLeft}(\mathit{Lock}), \mathit{TouchUp}(\mathit{Lock}), \mathit{TouchDown}(\mathit{Lock}),$$
$$\mathit{On}(\mathit{Key}), \mathit{Holding}(\mathit{Key}), \mathit{Open}(\mathit{Lock}),$$
$$\mathit{On}(\mathit{Passenger}), \mathit{Holding}(\mathit{Passenger}), \mathit{On}(\mathit{Destination})$$

Figure 6.4 shows the distribution of the single-episode duration for all three environments, for both the all-relations and reduced-relations cases. The failed runs are not included in the histograms, nor counted towards the mean and standard deviation statistics in Table 6.1 which summarizes these results.

First, we observe that all of the distributions are wide: there is a large variance in how long it takes the agent to complete an episode. Some variance is expected due to the random starting location, randomization of object locations, and the agent's random choice of action when some actions are no better than another. However,

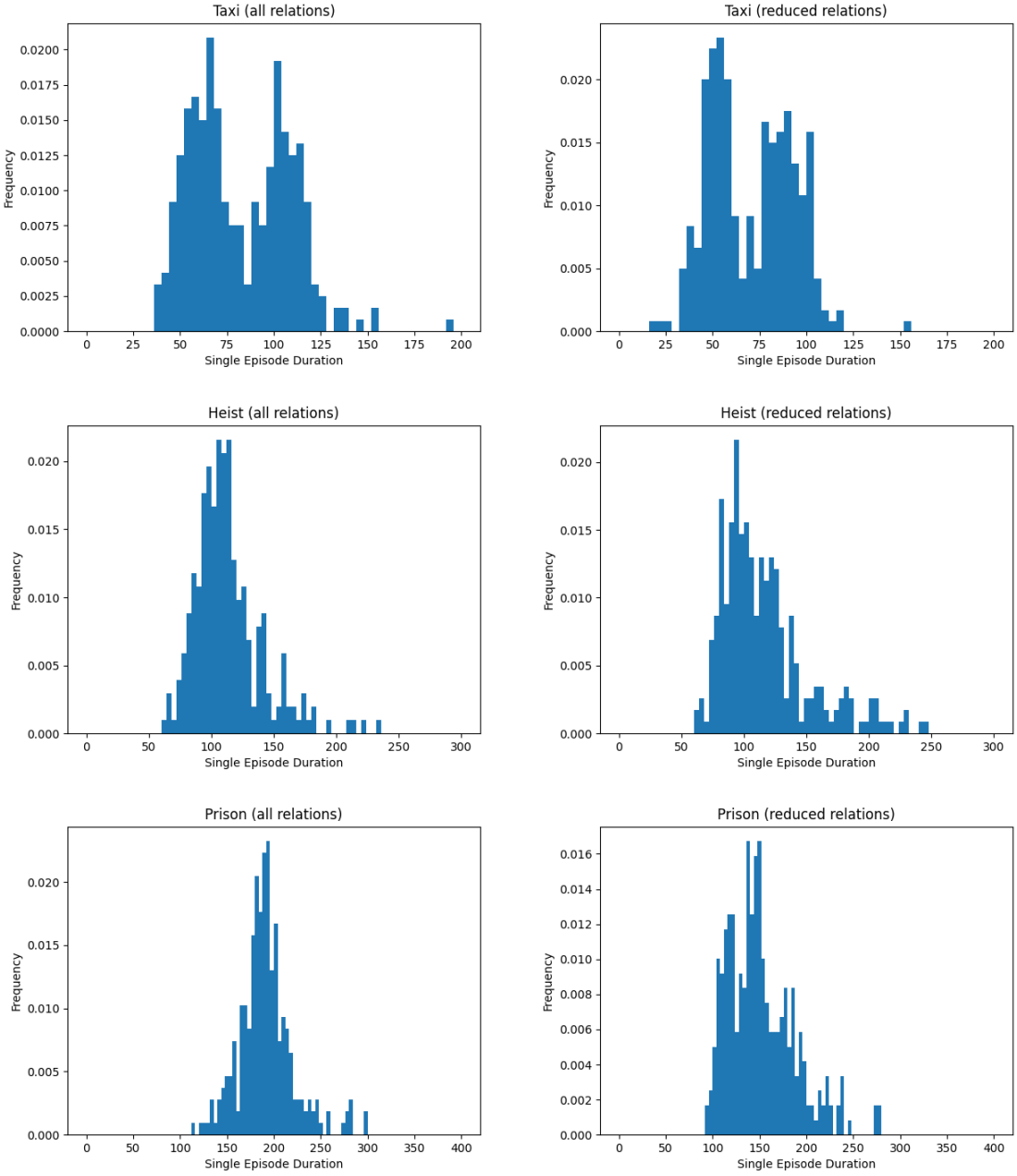


Figure 6.4: The distributions of the number of steps the FO-OO-MDP learning agent requires to complete the first episode in different domains.

another source of randomness can be seen clearly in the bimodal distribution of the Taxi distributions. The faster finishing times correspond to when the agent happens to pick up the passenger before attempting to interacting with the destination, and the slower times are when the agent interacts with the destination first. If this happens, the agent will fail to discover the correct environment dynamics by only testing size-one experiences, and will have to start testing size-two experiences. This requires backtracking and increases the number of unnecessary interactions the agent tests before it happens to pick up the passenger and drop it off at the destination.

The reason for the failures is that the FOIL-based rule set learning algorithm sometimes chooses incorrect relations to describe the dynamics of moving through an unlocked lock. This is due to the limited interactions the agent is able to have with locks, and how there may be more than one valid interpretation to start with. Looking at Figure 6.2, if the agent has unlocked the first lock and moved onto its square both *TouchDown*(*Lock*), *Open*(*Lock*) (the correct context) would be valid, but so would *TouchDown*(*Lock*), *TouchLeft*(*Wall*). An unlucky agent that has minimal experience with locks can learn an incorrect rule set, and is then unable to properly plan a path towards unlocking the other locks. Allowing the agent to plan with all rule sets of equal validity, instead of choosing just one, could be a method to resolve this.

When only necessary relations are given to the agent, the distribution for each environment is skewed to the left, decreasing the average time to completion. Without the distracting, unneeded relations the agent is more likely to test interactions that allow it to discover important dynamics. The distributions for Heist and Prison still have a long tail to the right, as even with reduced relations the larger number of object classes means there is a combinatorically larger amount of size-two experiences that the agent may have to try if it gets stuck. Interestingly, the mean number of steps for Heist went up slightly. This is because the failure rate was reduced by 10 percentage

| **Algorithm** | **# steps** |
|---|---|
| DOO$R_{\max}$ | 529 |
| Humans (non-gamers) | 101 |
| Humans (gamers) | 49 |
| FO-OO-MDP learner | 82 |

Table 6.2: Summary of results of different algorithms on Taxi world.

points, but those runs that would have failed would not have been counted towards the mean, and those runs tend to take many steps to complete. This reasoning explains the presence of the fat tail of the distribution. The median, in fact, decreased slightly from 108 to 106 steps.

The author of [4] compares DOO$R_{\max}$ against human performance in Taxi. The people in the experiment were divided into two groups: those that identified themselves as frequent players of video games ("gamers") and those who did not ("non-gamers"). I summarize these results, and compare to the FO-OO-MDP learner, in Table 6.2. Note that the statistic for DOO$R_{\max}$ is the time to learn the correct dynamics, not the first episode finish time, as DOO$R_{\max}$ requires multiple episodes to learn the dynamics. The FO-OO-MDP learner always has learned the correct dynamics by the time it finishes the first Taxi episode, so it is a fair comparison.

It can be seen that FO-OO-MDP performs between gamers and non-gamers, and is much more sample efficient than DOO$R_{\max}$. This is due to the inductive nature of FOIL, which is able to learn transition dynamics with many fewer samples. In addition, the results indicate that human gamers carry with them prior knowledge that allows them to solve the task faster. For example, after recognizing a wall, they will not need to try interacting with it as prior experience dictates that walls tend not to do anything useful. Attempting to extract similar knowledge from previous experience is an interesting area of future research for my algorithms.

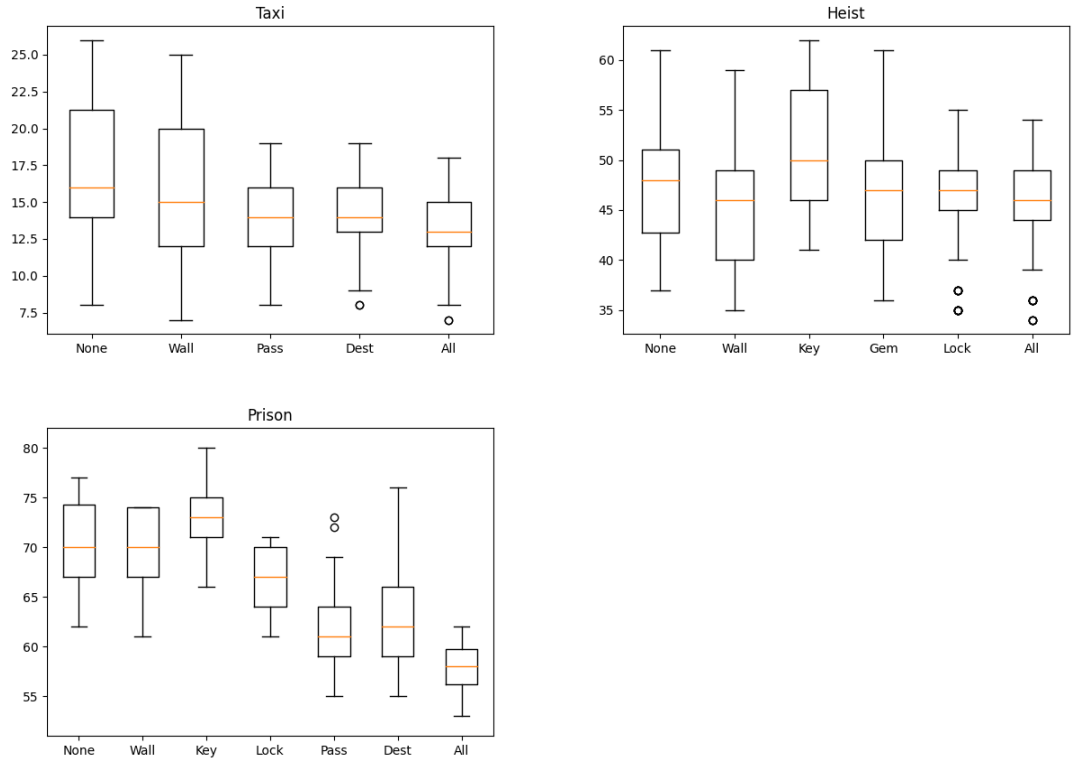


Figure 6.5: Distribution of the number of steps to finish an episode with logic-based object discovery. The labels indicate which objects' identities were given to the agent in advance.

## 6.2 Hypothesis 2: Logic-Based Object Discovery

For the logic-based object discovery evaluation, the agent is allowed to run on Taxi, Heist, or Prison until the rule set it has learned is accurate. That standard rule set is the agent's prior knowledge when it is switched to the target task, and the number of steps taken to generate this rule set are not counted in the results. The target environment is the exact same as each source environment, except that every object's identity is obfuscated. In addition, separate trials are run where the identities of one object class is *not* obfuscated, and one set of trials is run where *no* objects are obfuscated. Note that when no objects are obfuscated, it is the same as the agent using its model to plan the optimal method of finishing the episode. For each situation, 300 trials are run. The results are shown in Figure 6.5.

It can be seen that having prior knowledge of objects' behavior, and only needing to re-identify which is which, drastically reduces the number of steps taken compared to learning transitions from scratch. The median number of steps when no objects' identities are known a priori is 16, 48, and 70 in Taxi, Heist, and Prison respectively. In almost all cases, knowing the identities of one object class does allow the agent to learn more quickly, with the notable exception of keys. This is because if the agent already knows the identity of the key, it does not attempt to pick it up before interacting with the lock object. Once it discovers the identity of the lock it then needs to backtrack to get a key. This behavior is a result of the staged nature of the algorithm: first the agent discovers the identities of all objects, and once all are known it starts seeking out rewards. Combining these two phases is another area of future research.

Interestingly, having knowledge of all objects does not greatly decrease the median steps taken (except for in Prison), but does dramatically reduce the upper bound. The reason for the lack of large decrease is that the optimal sequence of actions to complete the environment and to identify objects are largely the same. For example, the agent usually starts by trying to pick up or identify the *Key* in Heist, or the *Passenger* in Taxi. However, in Prison the decrease is more noticeable because the *Destination*, the last object the agent needs to interact with to finish the episode, is far away from other objects. When its identity is not known and the agent goes to it out of order, the number of steps taken greatly increases. The lack of backtracking is also the reason that the upper bound is lower.

| | **Taxi** | **Heist** | **Prison** |
|---|---|---|---|
| Logic-based (same source) | $17 \pm 4.5$ | $47 \pm 5.9$ | $68 \pm 3.8$ |
| Simplest-explanation (same source) | $17 \pm 4.1$ | $46 \pm 6.4$ | $66 \pm 4.7$ |
| Simplest-explanation (from Heist) | $*$ | $*$ | $77 \pm 14$ |
| FO-OO-MDP learner (for reference) | $82 \pm 26$ | $113 \pm 27$ | $191 \pm 29$ |

Table 6.3: Summary and comparison of the performance of object discovery algorithms.

## 6.3 Hypothesis 3: Simplest-Explanation Object Discovery

First, I compare the performance of simplest-explanation object discovery to logic-based object discovery on the same set of source and target tasks as above, to verify that they perform equally in the scenario where no additional object are present in the target environment. Then, I evaluate simplest-explanation object discovery on the target task of Prison with a source task of Heist, to evaluate how it handles new objects. The agent is trained on the source task until its rule set is correct, then that rule set as well as the examples and experiences collected during training are used as the agent's prior knowledge when it is transferred to Prison. When trained on Heist, the agent has interacted with all of the objects in Prison except for the *Passenger* and *Destination*. For each scenario, 300 trials are run. Table 6.3 shows the results of these experiments, with the FO-OO-MDP learner for reference.

The results show that when there are no new objects, logic-based and simplest-explanation object discovery are equivalent algorithms. When the simplest-explanation agent is run with a source task of Heist, we see that the average finishing time is slightly higher then the logic-based object discovery agent that has trained on Prison, but is twice as fast as training on Prison from scratch. The extra time comes from the agent trying unneeded interactions before attempting to try dropping off the passenger at the destination. It is able to successfully discover these new interactions, whereas the logic-discovery agent would immediately break once a new object

behaved in a way not captured by the previous objects in its rule set.

The simplest-explanation agent was unable to be transferred from Taxi to Prison, and the reason is that when trained on Taxi the agent has not interacted with *Lock* or *Key*. When introduced to Prison, the first time the agent interacts with a lock, it believes it is a wall because it can not go through it. After it unlocks the lock for the first time, it still believes that it is a wall, because it is simpler to assume that walls behave in this additional manner when interacting with held keys then to invent a whole new object. The agent then starts to simulate transitions where it unlocks walls, which would provide it with a shortcut to the key. Up to this point, everything would work fine: once the agent discovers that it can't actually unlock a wall, it would discover the difference and map the lock to a new object. However: the programmers of OO-MDPs took a shortcut. Because walls never changed, the code did not have to keep track of 35 different wall objects, instead walls were implemented as static features of the environment. My work inherited this implementation, and when the agent tries to imagine unlocking a wall, the software crashes. The solution to this is to rebuild the state representation so that every wall is a unique object, where all objects have the exact same set of attributes.

# Chapter 7

# Conclusions

In this thesis, I introduced an extension of object-oriented Markov decision processes that use first-order logic, as well as an inductive rule set learning algorithm that discovers environment dynamics at an accelerated pace compared to deductive algorithms like DOO$R_{\max}$. The key insight is that inductive learning leads to an Occam's razor approach, as the algorithm tries to find the simplest set of rules to explain what it has observed. This leads to sample-efficient learning, and so does the object-oriented approach to exploration where the agent discovers unique transition dynamics by seeking out new interactions with objects. Next, I introduced the object discovery problem and the logic-based and simplest explanation object discovery algorithms, and showed that the first-order representation allows an agent to rediscover the obfuscated identities of objects, unlike previous propositional representations. With simplest explanation object discovery, the agent is able to simultaneously learn new transition dynamics while transferring knowledge about objects' identities.

## 7.1 Future work

This thesis has uncovered many areas for future work. In addition to the those mentioned in the previous chapters, I discuss four future research avenues in detail

here:

**Prioritizing Interactions:** As the experiments showed, a FO-OO-MDP learner that is given only the relations necessary to describe the environment learns faster because it does not need to explore unnecessary interactions. However, this requires human input to the algorithm, reducing its generality. Future work should investigate how the agent can use past experience to prioritize investigating relations and actions that tend to lead to useful outcomes.

**Enhanced Planning Algorithms:** Currently, the agent uses a greedy search to the nearest state in which it can achieve a positive reward or information gain, and does these searches one after another. Future work should investigate merging these two signals into one metric that the agent attempts to maximize, leading to more efficient exploration. In addition, the search stops at the first state that is found. Allowing the search to continue would allow the agent to make smarter decisions, at the cost of computational complexity. Finally, if the agent does not know the identities of the objects involved in a transition, it gives up and is unable to predict the transition. However, it could instead maintain separate predictions for each permutation of object identities, branching off the search into different imagined states to calculate the expected reward. This would further increase computational complexity, and a sampling-based planning approach like in [14] could be introduced to make the problem tractable.

**Improving Computational Efficiency:** Sampling methods could also be used to reduce the computational complexity of evaluating all object mappings in logic-based or simplest explanation object transfer. A heuristic based on previous or current observations that estimates the likelihood of each mapping could allow the agent to probabilistically skip the evaluation of mappings that are less likely to be correct.

**Stochasticity:** Finally, future work needs to investigate extending these algorithms to stochastic domains. The FO-OO-MDP learner could be adapted with

methods from [5], and I believe that simplest explanation object transfer is particularly suited to stochasticism as the simplicity score can be naturally extended to include a likelihood term.

## 7.2 Summary

In summary, first-order logic is a powerful tool for allowing agents to represent and reason about the objects and relations in their environment. By exploring unique interactions between objects, the FO-OO-MDP inductive learner is able to quickly learn the rules describing these domains. The first-order extension to OO-MDPs also enables the transfer of object-oriented knowledge in order to learn more effectively in future tasks. Both logic-based object discovery and simplest-explanation object discovery implement these ideas, with simplest-explanation object discovery being more robust and flexible.

# Bibliography


[1] K. Kansky, T. Silver, D. A. Mély, M. Eldawy, M. Lázaro-Gredilla, X. Lou, N. Dorfman, S. Sidor, S. Phoenix, and D. George, "Schema networks: Zero-shot transfer with a generative causal model of intuitive physics," in International conference on machine learning. PMLR, 2017, pp. 1809–1818.

[2] S. Russell and P. Norvig, Artificial Intelligence: a Modern Approach. Prentice hall, 2010.

[3] M. E. Taylor and P. Stone, "Transfer learning for reinforcement learning domains: A survey," Journal of Machine Learning Research, vol. 10, no. 1, pp. 1633–1685, 2009.

[4] C. Diuk, A. Cohen, and M. L. Littman, "An object-oriented representation for efficient reinforcement learning," in Proceedings of the 25th international conference on Machine learning - ICML '08. Helsinki, Finland: ACM Press, 2008, pp. 240–247.

[5] H. Pasula, L. Zettlemoyer, and L. P. Kaelbling, "Learning symbolic models of stochastic domains," in Journal of Artificial Intelligence Research, vol. 31, 2007, pp. 309–352.

[6] C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia, "Generalizing plans to new environments in relational mdps," in International Joint Conference on Artificial Intelligence, 2003.

[7] S. Džerosko, L. D. Raedt, and K. Driessnens, "Relational reinforcement learning," Machine Learning, vol. 43, pp. 7–52, 2001.

[8] L. Li, M. L. Littman, T. J. Walsh, and A. L. Strehl, "Knows what it knows: a framework for self-aware learning," Machine Learning, vol. 82, pp. 399–443, 2008.

[9] G. Kuhlmann and P. Stone, "Graph-based domain mapping for transfer learning in general games," in European Conference on Machine Learning, 2007.

[10] K. Cobbe, C. Hesse, J. Hilton, and J. Schulman, "Leveraging procedural generation to benchmark reinforcement learning," in International conference on machine learning. PMLR, 2020, pp. 2048–2056.

[11] S. Mohan and J. E. Laird, "An object-oriented approach to reinforcement learning in an action game," Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 2011.

[12] J. R. Quinlan and R. M. Cameron-Jones, "Foil: A midterm report," in European Conference on Machine Learning, 1993.

[13] W. Agnew and P. Domingos, "Unsupervised object-level deep reinforcement learning," in 32nd Conference on Neural Information Processing Systems, 2018.

[14] R. Keramati, J. Whang, P. Cho, and E. Brunskill, "Strategic object oriented reinforcement learning," ArXiv, vol. abs/1806.00175, 2018.