Learning Interpretable Action Models of Simulated Agents Through Agent Interrogation

by

Shashank Rao Marpally

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2021 by the
Graduate Supervisory Committee:

Siddharth Srivastava, Chair
Yu (Tony) Zhang
Georgios E. Fainekos

ARIZONA STATE UNIVERSITY

May 2021

ABSTRACT

Understanding the limits and capabilities of an AI system is essential for safe and effective usability of modern AI systems. In the query-based AI assessment paradigm, a personalized assessment module queries a black-box AI system on behalf of a user and returns a user-interpretable model of the AI system's capabilities. This thesis develops this paradigm to learn interpretable action models of simulator-based agents. Two types of agents are considered: the first uses high-level actions where the user's vocabulary captures the simulator state perfectly, and the second operates on low-level actions where the user's vocabulary captures only an abstraction of the simulator state. Methods are developed to interface the assessment module with these agents. Empirical results show that this method is capable of learning interpretable models of agents operating in a range of domains.

DEDICATION

To my mom, dad, uncle, grandpa, and grandma for supporting me always.

ACKNOWLEDGMENTS

I would like to sincerely thank Dr. Siddharth Srivastava for guiding and supporting me throughout my research. I would like to thank Dr. Yu (Tony) Zhang for giving me the chance to participate in research in his lab and for his constant encouragement. I would also like to thank Pulkit Verma for constantly advising me through this thesis and being a valuable research collaborator and the members of the Autonomous Agents and Intelligent Robots Lab for providing a great research environment to foster ideas. Lastly, I would like to thank my mother, father, uncle, and grandparents, for their unwavering support and faith in me.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

The development of AI systems has reached an interesting milestone. It is now, more than ever, much easier to access AI systems, whether it is a digital assistant in smartphones, assistive devices, or autonomous cars. Although some of them are far more affordable than others, this does mean that overall, a broader section of society can now obtain some form of an AI system. This also poses a parallel problem: How is one to ascertain that the AI system that he/she uses is safe, reliable, and/or can perform the required task? More often than not, these AI systems are "black-box" to the end-user, meaning the end-user cannot readily ascertain the reasoning and decision-making process underlying the AI system. Thus, to judge the suitability of an AI system for a task, it is important to develop methods to approximate its internal model and gauge its capabilities, limitations, and strengths. The widespread usage of AI systems and easy accessibility make the problem of ensuring safety of AI systems an important next step before the advent of large-scale generalizable AI that humans and corporations may depend upon to take important decisions. Similarly, interpretability and explainability of AI systems are also the problems that must be addressed to hold the system accountable for its decisions. Currently, commercial robotic systems are limited in functionality to perform basic tasks. As robots become more and more autonomous and capable of performing complex tasks, it becomes even more important to ensure they operate safely and follow the expectations of a designer. For example, a robotics researcher may want to acquire a robot for performing certain tasks and thus would need to assess the capabilities and limitations of the robot, or, a robot technician may want to verify if a robot's internal model

follows his/her expectations after a firmware update. Learning the internal action model of a system is one way to learn the capabilities and limitations of the system. For agents based on symbolic-planning models, this means learning the preconditions and effects of the actions that make the internal model of the agent. Quite often, such systems may operate on low-level commands and provide interfaces to end-users to operate the system using high-level commands. For example, internally, a robot operates using motor controls but they can be controlled using frameworks that provide more user-friendly high-level control. Learning the action model of such agents provides a challenge and an opportunity at the same time since the same low-level actions in different states can result in different results, learning the model in terms of low-level actions can result in a model that is difficult to interpret and learning interpretable models can also help in interacting with such systems using user-specific interpretable high-level vocabulary. To this end, this thesis develops a framework to learn interpretable abstract action models of simulator-based agents using the agent assessment module developed in Verma and Srivastava (2020) and Verma *et al.* (2021). Two types of simulated agents are considered, namely:

- Type-1: Simulator-based agents where the user's vocabulary captures the simulator state perfectly. The agents provide access to a set of symbolic action headers that the agent assessment module can use, and a reference ground-truth model that can be used to check the accuracy of the learnt model.

- Type-2: Simulator-based agents where the user's vocabulary captures only an abstraction of the simulators state. These agents do not have a ground-truth model for reference and do not provide a set of action headers which is required as an input to the agent assessment module.

Chapter 2

RELATED WORK

The concept of action model learning has been explored greatly in the past. In contrast to many related works that explore learning action models from observations of behavior (Gil (1994); Yang *et al.* (2007); Cresswell *et al.* (2009) and Zhuo and Kambhampati (2013), The Agent Assessment Module (AAM) developed in Verma and Srivastava (2020) and Verma *et al.* (2021) is one of the first approaches to address inferring relational models of black-box agents using an agent-interrogation strategy (query-and-answer approach). Cresswell *et al.* (2009) proposes LOCM, which uses finite-state machines to create action models from a collection of plans. Each state machine represents the precondition and effect of the actions in the domain. LOCM uses one finite state machine for each action and this limitation is overcome by LOCM2 (Cresswell and Gregory (2011)), where a single object can be represented by multiple state machines. Learning static relations in a domain is a non-trivial task. Static relations refer to literals that never change over the course of a plan trace. The problem here is that since these do not appear in the add or delete lists of an action's effect, they could be overlooked. But this is a problem since static relations can appear in the preconditions of certain operators. Since LOCM2 uses dynamic properties of the domain, it is unable to learn the static relations that may appear in certain actions. This is addressed by LOP (Gregory and Cresswell (2015)) where static relations are discovered by finding the minimal set of static predicates for each action that preserves the length of the optimal plan. Stern and Juba (2017) introduce the safe model-free planning problem, learn a conservative model for safe planning, and provide only soundness guarantees. In contrast, AAM is theoretically guaran-

teed to learn the complete and correct model, which can be directly used for safe planning. Additionally, AAM does not require the intermediate states in execution traces. Konidaris *et al.* (2018) develop methods to autonomously learn the abstract representation required for a computer game domain that internally operates on low-level actions. This work assumes that a set of high-level options are available to the agent which can be used to explore the environment and learn skills. In contrast, the problem addressed in this thesis is that of learning abstract actions, given user-defined vocabulary or abstract state representations through agent interrogation. FAMA (Aineto *et al.* (2019)) reduces model recognition to a planning problem and can work with partial action sequences and/or state traces as long as correct initial and goal states are provided. It can end up learning spurious preconditions since preconditions are learned as a post-processing step. This also results in oscillating model accuracy. Bonet and Geffner (2020) is one of the few methods for learning relational models when the action schema, predicates, etc. are not available. But, this approach is viable for small state spaces only. Simulator domains often have very large state spaces. Suárez-Hernández *et al.* (2020) propose two algorithms (OARU, AU) to solve the problem of recognizing actions given a state transition and a knowledge library of relational actions. These algorithms are restricted to symbolic inputs and deterministic action effects but support partially observable transitions. AU unifies a trivial ground action with an existing action using weighted partial Max-SAT. The precision and recall of the model, however, are calculated with respect to the actual library of actions available. Čertickỳ (2014) proposes the 3SG algorithm that is shown to learn the action models of games with incomplete information. This model is learned directly from observation with known action signatures of the high-level actions. 3SG is shown to be fast, but not guaranteed to be precise.

Chapter 3

BACKGROUND

3.1 Classical Planning Task

A classical planning task is a 5 tuple $\Pi = (\mathbb{P}, \mathbb{A}, \mathcal{O}, s_0, \gamma)$ consisting of a set of predicate symbols $\mathbb{P}$, a set of Actions $\mathbb{A}$, set of Objects $\mathcal{O}$, an initial state s_0 and goal condition (or set of goal states) γ. Each predicate $p \in \mathcal{P}$ with an arity n can be instantiated with n objects from $\mathcal{O}$. A predicate p is said to be "grounded" when the parameters in p are substituted with objects $o \in \mathcal{O}$. A state consists of a collection of ground predicates. A state transitions into another state by the application of an action $a \in \mathbb{A}$ which consist of an action signature (name and the parameters), a set of preconditions and a set of add and delete effects. An action can be applied in a state if it follows the preconditions of the action. Upon application of an action in a state, the atoms in the add effects of the action are added to the state and the atoms in the delete effects of the action are removed from the state. the solution to a planning task is a plan π which is a sequence of actions such that applying π to s_0 results in a state s_F which satisfies the goal condition γ. Classical planning tasks can be encoded in a STRIPS-Like language like PDDL into a domain.pddl file which defines $\mathbb{P}$, and $\mathbb{A}$ and a problem file which defines a particular $\mathcal{O}, s_0, \gamma$ and many off-the-shelf planners can solve the task given these files.

3.2 Agent Interrogation Task

Let the agent $\mathcal{A}$'s planning model is represented as a pair $\mathcal{M} = \langle \mathbb{P}, \mathbb{A} \rangle$, where $\mathbb{P} = \{p_1^{k_1}, \ldots, p_n^{k_n}\}$ is a finite set of predicates with arities k_i; $\mathbb{A} = \{a_1, \ldots, a_k\}$ is a

finite set of parameterized actions (operators). Each action $a_j \in \mathbb{A}$ is represented as a tuple $\langle header(a_j), pre(a_j), \mathit{eff}(a_j)\rangle$, where $header(a_j)$ is the action header consisting of action name and action parameters, $pre(a_j)$ represents the set of predicate atoms that must be true in a state where a_j can be applied, $\mathit{eff}(a_j)$ is the set of positive or negative predicate atoms that will change to true or false respectively as a result of execution of the action a_j. Each predicate can be instantiated using the parameters of an action, where the number of parameters are bounded by the maximum arity of the action. Now, the Agent Interrogation task can be defined as:

Definition 1. An agent interrogation task is defined as 4 tuple: $\langle \mathcal{M}^{\mathcal{A}}, \mathbb{Q}, \mathbb{P}, \mathbb{A}_H\rangle$, where $\mathcal{M}^{\mathcal{A}}$ is the true model (unknown to AAM) of the agent $\mathcal{A}$ being interrogated, $\mathbb{Q}$ is the class of queries that can be posed to the agent by AAM, and $\mathbb{P}$ and $\mathbb{A}_H$ are the sets of predicates and action headers that AAM uses based on inputs from $\mathcal{H}$ and $\mathcal{A}$.

The objective of the agent interrogation task is to derive the agent model $\mathcal{M}^{\mathcal{A}}$ using $\mathbb{P}$ and $\mathbb{A}_H$.

3.3 Agent Assessment Module

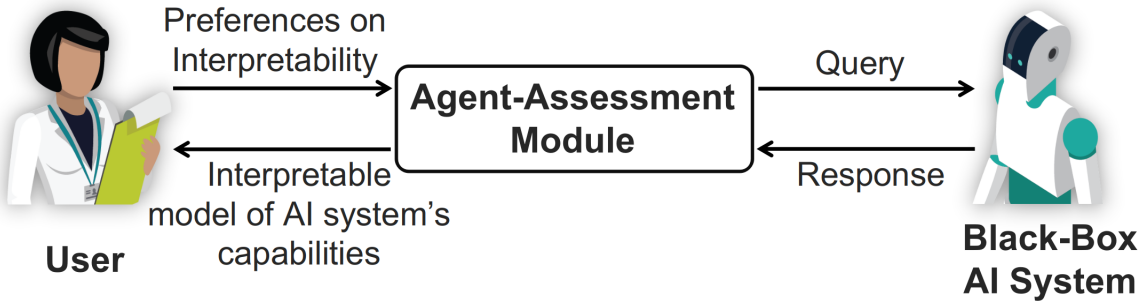


Figure 3.1: Agent Assessment Module (Verma *et al.* (2021))

As shown in Fig 3.1, the Agent Assessment Module takes as input the user's interpretable vocabulary, and uses the Agent Interrogation Algorithm to query the

agent and learn its model in this vocabulary. As in Verma and Srivastava (2020), for simplicity, it is assumed that the user understands models in a STRIPS-Like language (PDDL).

3.3.1 Agent Models

Agent models are represented as a collection of predicate(p)-action(a)-location(l)-mode(m) (*palm*) tuples. If a model $\mathcal{M}$ contains the *palm* tuple $\lambda = \langle p, a, l, m \rangle$, it represents the fact that the action $a \in \mathbb{A}$ in model $\mathcal{M}$ has the predicate $p \in \mathbb{P}$ in location l in the mode m, where $l \in \{pre, eff\}$, and $m \in \{+, -, \emptyset\}$. For example, if the model $\mathcal{M}$ has the *palm* tuple <holding,pick-up,pre,->, it means that the action 'pick-up' in model $\mathcal{M}$ has the predicate 'holding' in negative form in the precondition. Two palm tuples $\lambda_1 = \langle p_1, a_1, l_1, m_1 \rangle$ and $\lambda_2 = \langle p_2, a_2, l_2, m_2 \rangle$ are considered to be *variants* of each other ($\lambda_1 \sim \lambda_2$) iff they differ only on mode m, i.e., $\lambda_1 \sim \lambda_2 \Leftrightarrow (\lambda_{1_p} = \lambda_{2_p}) \wedge (\lambda_{1_a} = \lambda_{2_a}) \wedge (\lambda_{1_l} = \lambda_{2_l}) \wedge (\lambda_{1_m} \neq \lambda_{2_m})$. Hence, mode assignment to a *pal* tuple $\gamma = \langle p, a, l \rangle$ can result in 3 palm tuple variants $\gamma^+ = \langle p, a, l, + \rangle$, $\gamma^- = \langle p, a, l, - \rangle$, and $\gamma^\emptyset = \langle p, a, l, \emptyset \rangle$. Let $\mathbb{P}^*$ represent the set of all possible predicates instantiated with action parameters and Λ be the set of all possible palm tuples which can be generated using the predicate vocabulary $\mathbb{P}^*$ and an action header set $\mathbb{A}_H$. Let $\mathcal{U}$ be the set of all consistent (abstract and concrete) models that can be expressed as subsets of Λ, such that no model has multiple variants of the same palm tuple.

3.3.2 Model Abstraction

Given the above form of representation of models in AAM as a collection of *palm* tuples, an abstraction of a model $\mathcal{M}$ with respect to a *palm* tuple λ is another model $\mathcal{M}'$ where $\mathcal{M}'$ is obtained by removing the *palm* tuple λ from $\mathcal{M}$ i.e., $\mathcal{M}' = \mathcal{M} \setminus \lambda$. Thus, as in Verma and Srivastava (2020):

Definition 2. The *abstraction of a model* $\mathcal{M}$ with respect to a palm tuple $\lambda \in \Lambda$, is defined by $f_\lambda : \mathcal{U} \to \mathcal{U}$ as $f_\lambda(\mathcal{M}) = \mathcal{M} \setminus \lambda$.

This notion of abstraction and concretization of model now allows defining the model lattice $\mathcal{L}$ as:

Definition 3. A ***model lattice*** $\mathcal{L}$ is a 5-tuple $\mathcal{L} = \langle N, E, \Gamma, \ell_N, \ell_E \rangle$, where N is a set of lattice nodes, Γ is the set of all pal tuples $\langle p, a, l \rangle$, $\ell_N : N \to 2^{2^\Lambda}$ is a node label function where $\Lambda = \Gamma \times \{+, -, \emptyset\}$ is the set of all palm tuples, E is the set of lattice edges, and $\ell_E : E \to \Gamma$ is a function mapping edges to edge labels such that for each edge $n_i \to n_j$, $\ell_N(n_j) = \{\xi \cup \{\gamma^k\} | \, \xi \in \ell_N(n_i), \gamma = \ell_E(n_i \to n_j), k \in \{+, -, \emptyset\}\}$, and $\ell_N(\top) = \{\phi\}$ where $\top$ is the supremum containing the empty model ϕ.

Based on the agent's response to a query, an edge is selected that can be used to concretize the parent node with the selected *pal* tuple.

3.3.3 Agent Queries

In simplified terms, suppose P is the set of all predicates $\mathbb{P}^*$ instantiated by objects $\mathcal{O}$ of the domain, and $s_I \subseteq P$, the queries (called **plan-outcome queries** Q_{PO}) posed to the agent are of the form: "If you were in state s_I and were to perform an action-sequence/plan π, what would be the resulting state?". The agent's response to this query, Q_{PO} comprises of l, the longest prefix of the plan π that it is successfully able to execute, and the resulting state $s_F \subseteq \mathcal{P}$ after executing the l steps of the plan. Thus, $Q_{PO} : \mathcal{U} \to \mathbb{N} \times 2^P$ where $\mathbb{N}$ is the set of all natural numbers. In AAM, the agent's response to queries is used to ascertain which model between two possible models is inconsistent with that of the agent. This allows pruning of models and subsequently, sub-lattices originating from that model.

Definition 4. Two models $\mathcal{M}_i$ and $\mathcal{M}_j$ are said to be *distinguishable*, denoted as

$\mathcal{M}_i \wr \mathcal{M}_j$, iff there exists a query that can distinguish between them, i.e., $\exists \mathcal{Q}\ \mathcal{M}_i \wr^{\mathcal{Q}} \mathcal{M}_j$ where $\mathcal{M}_i \wr^{\mathcal{Q}} \mathcal{M}_j \implies Q(\mathcal{M}_i) \neq Q(\mathcal{M}_j)$.

Thus, queries must be able to distinguish between models to be useful; i.e., $\exists \mathcal{Q}\ \mathcal{M}_i \wr^{\mathcal{Q}} \mathcal{M}_j$. Finding the query that can distinguishes between models is framed as a planning problem. To avoid the unnecessary computation of generating queries for models that cannot be distinguished, it is important to first determine if two models can be distinguished:

Definition 5. Let $\mathcal{Q}$ be a query such that $\mathcal{M}_i \wr^{\mathcal{Q}} \mathcal{M}_j$; $\mathcal{Q}(\mathcal{M}_i) = \langle \ell^i, \langle p_1^i, \ldots, p_m^i \rangle \rangle$, $\mathcal{Q}(\mathcal{M}_j) = \langle \ell^j, \langle p_1^j, \ldots, p_n^j \rangle \rangle$, and $\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) = \langle \ell^{\mathcal{A}}, \langle p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}} \rangle \rangle$. $\mathcal{M}_i$'s response to $\mathcal{Q}$ is *consistent* with that of $\mathcal{M}^{\mathcal{A}}$, i.e. $\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}(\mathcal{M}_i)$ if $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}})$, $len(\pi^{\mathcal{Q}}) = \ell^i$ and $\{p_1^i, \ldots, p_m^i\} \subseteq \{p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}}\}$.

Similarly, given two distinguishable models, one of the models can be pruned depending on the response from the agent:

Definition 6. Given an agent-interrogation task $\langle \mathcal{M}^{\mathcal{A}}, \mathbb{Q}, \mathbb{P}, \mathbb{A}_H \rangle$, two models $\mathcal{M}_i$ and $\mathcal{M}_j$ are *prunable*, denoted as $\mathcal{M}_i \langle\rangle \mathcal{M}_j$, iff $\exists \mathcal{Q} \in \mathbb{Q} : \mathcal{M}_i \wr^{\mathcal{Q}} \mathcal{M}_j \wedge (\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}(\mathcal{M}_i) \wedge \mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \not\models \mathcal{Q}(\mathcal{M}_j)) \vee (Q(\mathcal{M}^{\mathcal{A}}) \not\models \mathcal{Q}(\mathcal{M}_i) \wedge \mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}(\mathcal{M}_j))$.

3.3.4 Agent Interrogation Algorithm

Algorithm 1 summarizes the Agent Interrogation Algorithm (AIA), that is used to solve the Agent Interrogation task. This section elaborates on the various components of the algorithm. AIA takes as input, the agent $\mathcal{A}$, the set of instantiated predicates $\mathbb{P}^*$, the set of all action headers $\mathbb{A}_H$, and a set of random states $\mathbb{S}$ as input, and gives the set of functionally equivalent estimated models represented by *poss_models* as output. $\mathbb{S}$ can be generated in a pre-processing step given $\mathbb{P}^*$. AIA initializes *poss_models* as a set consisting of the empty model ϕ (line 3) representing that AAM

Algorithm 1 Agent Interrogation Algorithm (AIA)

Input: $\mathcal{A}, \mathbb{A}_H, \mathbb{P}^*, \mathbb{S}$
Output: poss_models
Initialize poss_models = $\{\phi\}$
for γ in some input pal ordering Γ **do**
  new_models $\leftarrow$ poss_models
  pruned_models= {}
  for each $\mathcal{M}'$ in new_models **do**
    for each pair $\{i, j\}$ in $\{+, -, \emptyset\}$ **do**
      $\mathcal{Q}$, $\mathcal{M}_i$, $\mathcal{M}_j \leftarrow$ generate_query$(\mathcal{M}', i, j, \gamma, \mathbb{S})$
      $\mathcal{M}_{prune} \leftarrow$filter_models$(\mathcal{Q}, \mathcal{M}^{\mathcal{A}}, \mathcal{M}_i, \mathcal{M}_j)$
      pruned_models$\leftarrow$ pruned_models $\cup\, \mathcal{M}_{prune}$
    end for
  end for
  if pruned_models is $\emptyset$ **then**
    update_pal_ ordering$(\Gamma, \mathbb{S})$
    continue
  end if
  poss_models $\leftarrow$ new_models $\times \{\gamma^+, \gamma^-, \gamma^{\emptyset}\} \setminus$ pruned_models
end for

is starting at the supremum $\top$ of the model lattice. In each iteration of the main loop, a *pal* tuple is selected. The order in which the *pal* tuples are not important, apart from the preconditions being selected first. The reason for this will be elaborated upon in later sections. After a *pal* tuple is selected, the models in *poss_models* are refined with each of three possible modes for a *pal* tuple to obtain the the list *new_models*. Next, for every pair of models $\mathcal{M}^i, \mathcal{M}^j$ in *new_models*, a distinguishing query $\mathcal{Q}$ is generated. The agent $\mathcal{A}$, and the two models $\mathcal{M}^i, \mathcal{M}^j$ as well as the query $\mathcal{Q}$ are

then passed to *filter_models* which returns the model(s) which is inconsistent with the agent and can be pruned. In case no models can be pruned, which happens when the agent $\mathcal{A}$ is unable to run the query, the list of *pal* tuples is updated. A sample

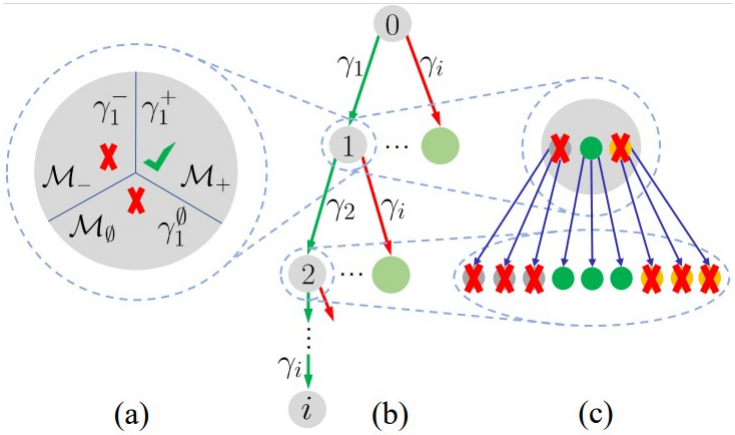


Figure 3.2: Hierarchical Abstractions Used in AIA (Verma *et al.* (2021))

run of this is show in Fig 3.2. AIA begins with the most abstract model ϕ and is concretized by pal tuples γ_1 and γ_2 sequentially. Each concretization generates 3 models, 2 of which are discarded based on the agent's responses to queries(a). The remaining possible models are propagated further(b). As shown in (c), eliminating one model in the abstract level discards 3^n models n levels down.

Query Generation

Algorithm 2 describes the query generation process. It takes as input, a model $\mathcal{M}'$, the indices i, j corresponding to the modes i.e., $i, j \in \{+, -, \emptyset\}$, a *pal* tuple γ and the set of random states for the domain $\mathbb{S}$, and outputs a query $\mathcal{Q}$ that can distinguish between the models $\mathcal{M}_i = \mathcal{M} \bigcup \gamma^i$ and $\mathcal{M}_j = \mathcal{M} \bigcup \gamma^j$ using any of the random states in $\mathbb{S}$. Generating a distinguishing query $\mathcal{Q}$ is framed as a planning problem P_{PO}. In order to distinguish between $\mathcal{M}_j$ and $\mathcal{M}_i$, $\mathcal{Q}$ must be such that $\mathcal{Q}(\mathcal{M}_i) \neq \mathcal{Q}(\mathcal{M}_j)$. Let $\gamma = \langle p, a, l \rangle$ and the two refined models are $\mathcal{M}'_m = \mathcal{M}' \cup \langle p, a, l, m \rangle$, where $m \in \{i, j\}$.

Algorithm 2 Query Generation Algorithm

Input: $\mathcal{M}', i, j, \gamma, \mathbb{S}$

Output: $\mathcal{Q}, \mathcal{M}_i, \mathcal{M}_j$

$\mathcal{M}_i, \mathcal{M}_j \leftarrow$ add_palm$(\mathcal{M}', i, j, \gamma)$

for s_I in $\mathbb{S}$ **do**

    dom, prob $\leftarrow$ get_planning_prob $(s_I, \mathcal{M}_i, \mathcal{M}_j)$

    $\pi \leftarrow$ planner(dom, prob)

    $\mathcal{Q} \leftarrow \langle s_{\mathcal{I}}, \pi \rangle$

    if π **then** break **end if**

end for

return $\mathcal{Q}$, $\mathcal{M}' \cup \{\gamma^i\}$, $\mathcal{M}' \cup \{\gamma^j\}$

Temporary models $\mathcal{M}_i''$ and $\mathcal{M}_j''$ are then created as:

$$\mathcal{M}_m'' = \mathcal{M}_m' \cup \{\langle p_u, a', l', + \rangle : \forall a', l' \ \langle a', l' \rangle \notin \{\langle a^*, l^* \rangle : \exists m^* \ \langle p, a^*, l^*, m^* \rangle \in \mathcal{M}'\}\}$$
$$\cup \{\langle p_u, a', l', - \rangle : \forall a', l' \ \langle a', l' \rangle \in \{\langle a^*, l^* \rangle : l^* = \mathit{eff} \wedge \exists m^* \langle p, a^*, l^*, m^* \rangle \in \mathcal{M}'\}\}$$

Here, p_u is a dummy predicate representing "unknown_predicate" which is added when $l = pre$ in disjunction with the predicate p to the preconditions of both models.

	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_A$
putdown			
precondition	$\langle empty \rangle$	$\langle empty \rangle$	(not(ontable ?x))
effect	$\langle empty \rangle$	$\langle empty \rangle$	(ontable ?x)
pickup			
precondition	(ontable ?x)	(not (ontable ?x))	(ontable ?x)
effect	$\langle empty \rangle$	$\langle empty \rangle$	$(not(ontable ?x))$

Table 3.1: Sample Query in AIA Without p_u

	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_A$
putdown			
precondition	p_u	p_u	(not(ontable ?x))
effect	p_u	p_u	(ontable ?x)
pickup			
precondition	(ontable ?x) $\vee p_u$	(not (ontable ?x)) $\vee p_u$	(ontable ?x)
effect	p_u	p_u	(not (ontable ?x))

Table 3.2: Sample Query in AIA With p_u

The dummy predicate p_u is added to avoid incorrect pruning of possible domain models. Suppose p_u is not used in query generation. Consider the example shown in table 3.1 for the Blocksworld domain. Here, the models $\mathcal{M}_1$ and $\mathcal{M}_2$ are the two models being considered, concretized from the abstract model using the pal tuple $\lambda = \langle \text{pickup}, \text{ontable}, pre \rangle$. The agent's true model is shown in the last column. Now, consider the query $\mathcal{Q} = \langle s_I = (\text{not } (\text{ontable}(B)), \pi = [\text{putdown ,pickup}] \rangle$. Now, $\mathcal{Q}(\mathcal{M}_2) = \langle 2, (\text{not } (\text{ontable}(B)) \rangle$ and $\mathcal{Q}(\mathcal{M}_1) = \langle 1, (not(\text{ontable}(B)) \rangle$, thus, $\mathcal{Q}(\mathcal{M}_1) \neq \mathcal{Q}(\mathcal{M}_2)$ and so $\mathcal{M}_1 \wr^{\mathcal{Q}} \mathcal{M}_2$. Now, $\mathcal{Q}(\mathcal{M}_{\mathcal{A}}) = \langle 2, (\text{not } (\text{ontable}(B)) \rangle = \mathcal{Q}(\mathcal{M}_2)$. Thus, $\mathcal{M}_1 \wr^{\mathcal{Q}} \mathcal{M}_2 \wedge (\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}(\mathcal{M}_2) \wedge \mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \not\models \mathcal{Q}(\mathcal{M}_1))$ Therefore, using definition 6, $\mathcal{M}_1$ can be pruned. But this is incorrect since the pal tuple λ's mode in $\mathcal{M}_1$ matches with that of $\mathcal{M}_a$. This discrepancy occurs since the effects of the action *putdown* can change the state but this is not modelled in the domain since the pal tuple $\lambda' = \langle$ putdown,ontable,$eff \rangle$ has not been refined yet. To overcome this, the dummy predicate p_u is added temporarily to the two models under consideration as shown in table 3.2. Now, for the same query $\mathcal{Q} = \langle s_I = (\text{not } (\text{ontable}(B)), \pi = [\text{putdown ,pickup}] \rangle$, $\mathcal{Q}(\mathcal{M}_1) = \mathcal{Q}(\mathcal{M}_2) = \langle 0, (\text{not } (\text{ontable}(B)) \rangle$ and thus the models are not prunable. The planning problem for query generation is expressed as $P_{PO} =$

$\langle \mathcal{M}^{PO}, s_{\mathcal{I}}$. The domain for P_{PO} is $M^{PO} = \langle \mathbb{P}^{PO}, \mathbb{A}^{PO} \rangle$ where $\mathbb{P}^{PO} = \mathcal{P}^{\mathcal{M}''_i} \cup \mathcal{P}^{\mathcal{M}''_j} \cup p_\psi$ and each action $a \in \mathbb{A}^{PO}$, has the same header as $\mathbb{A}_H$, $pre(a) = pre(a^{\mathcal{M}''_i}) \vee pre(a^{\mathcal{M}''_j})$ and

$$\begin{aligned} \mathit{eff}(a) = (&when\,(pre(a^{\mathcal{M}''_i}) \wedge pre(a^{\mathcal{M}''_j}))(\mathit{eff}(a^{\mathcal{M}''_i}) \wedge \mathit{eff}(a^{\mathcal{M}''_j}))) \\ &(when\,((pre(a^{\mathcal{M}''_i}) \wedge \neg pre(a^{\mathcal{M}''_j})) \vee \\ &(\neg pre(a^{\mathcal{M}''_i}) \wedge pre(a^{\mathcal{M}''_j})))\,(p_\psi)), \end{aligned}$$

Under this formulation, each action has a precondition which is a disjunction of the preconditions of the same action in both the models $\mathcal{M}''_i$ and $\mathcal{M}''_j$. The effect of each action is conditional upon whether the precondition for just one model was satisfied or that of both the models were satisfied. The former case would result in the dummy predicate p_ψ to be set to True and the latter case results in a effect which is the disjunction of the effects of the action from both the models. The initial state $s_I = s_I^{\mathcal{M}''_i} \wedge s_I^{\mathcal{M}''_j}$, where $s_I^{\mathcal{M}''_i}$ and $s_I^{\mathcal{M}''_j}$ are copies of all predicates in s_I, and G is the goal formula expressed as $\exists p\ (p^{\mathcal{M}''_i} \wedge \neg p^{\mathcal{M}''_j}) \vee (\neg p^{\mathcal{M}''_i} \wedge p^{\mathcal{M}''_j}) \vee p_\psi$. With this of the planning problem, the goal would be reached when either an action is applicable in only one of the models and not the other, or an execution of an action yields different predicates to be set to True.

Filtering Possible Models

This function takes as input a distinguishing Query $\mathcal{Q}$, the agent $\mathcal{A}$ and the two partially refined models $\mathcal{M}_i$ and $\mathcal{M}_j$ and gives the set of pruned models $\mathcal{M}_{prune}$ that are inconsistent with the agent $\mathcal{A}$. If $\mathcal{M}_i \wr^{\mathcal{Q}} \mathcal{M}_j$, then $\mathcal{Q}(\mathcal{M}_i) \neq \mathcal{Q}(\mathcal{M}_j)$ and based on $\mathcal{Q}(\mathcal{A})$, it determines if the two models are prunable i.e., $\mathcal{M}_i \langle\rangle \mathcal{M}_j$. The policy from pruning out a model $\mathcal{M}_i$ is based on the rule described in Theorem 1

Theorem 1. Let $\mathcal{M}_i, \mathcal{M}_j \in \{\mathcal{M}_+, \mathcal{M}_-, \mathcal{M}_\emptyset\}$ be the models generated by adding the pal tuple γ to $\mathcal{M}'$ which is an abstraction of the true agent model $\mathcal{M}^{\mathcal{A}}$. Suppose $\mathcal{Q} = \langle s_{\mathcal{I}}^{\mathcal{Q}}, \pi^{\mathcal{Q}} \rangle$ is a distinguishing query for two distinct models $\mathcal{M}_i, \mathcal{M}_j$, i.e. $\mathcal{M}_i \wr^{\mathcal{Q}} \mathcal{M}_j$, and the response of models $\mathcal{M}_i, \mathcal{M}_j$, and $\mathcal{M}^{\mathcal{A}}$ to the query $\mathcal{Q}$ are $\mathcal{Q}(\mathcal{M}_i) = \langle \ell^i, \langle p_1^i, \ldots, p_m^i \rangle \rangle$, $\mathcal{Q}(\mathcal{M}_j) = \langle \ell^j, \langle p_1^j, \ldots, p_n^j \rangle \rangle$, and $\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) = \langle \ell^{\mathcal{A}}, \langle p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}} \rangle \rangle$. When $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}})$, $\mathcal{M}_i$ is not an abstraction of $\mathcal{M}^{\mathcal{A}}$ if $len(\pi^{\mathcal{Q}}) \neq \ell^i$ or $\{p_1^i, \ldots, p_m^i\} \not\subseteq \{p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}}\}$.

Update PAL ordering

In case the agent $\mathcal{A}$ is unable to execute the query $\mathcal{Q}$, i.e., $len(\pi^{\mathcal{Q}}) \neq \ell^{\mathcal{A}}$, the agent replies with the failed action $a_F = \pi[\ell{+}1]$ and the final state s_F. The models $\mathcal{M}_i$ and $\mathcal{M}_j$ are generated by refining the model $\mathcal{M}$ using the *pal* tuple γ. Thus $\mathcal{M}_i$ and $\mathcal{M}_j$ only differ in terms of the mode of the *pal* tuple γ. Thus both the models would reach the same state $\overline{s}_F$ after executing the first l steps of the plan. To find the predicates that are necessary for the execution of the action a_f, a state is needed in which it is executable. This state is obtained by searching in $\mathbb{S}$ such that $s \supset \overline{s}_F$ and $\mathcal{A}$ can execute a_F. Now, the predicates $p' \subseteq s \setminus \overline{s}_F$ are sequentially iterated over and added to $\overline{s}_F$ to check if $\mathcal{A}$ can still execute a_f. Similar to Stern and Juba (2017), it is assumed that any predicate instantiation corresponding to false literals in a state will not appear in $a_{\mathcal{F}}$'s precondition in the positive mode. Thus, if $\mathcal{A}$ cannot execute a_f in state $\overline{s}_f \cup p'$, the predicates in p' which are in negative mode, are added to in a_f's precondition, and if $\mathcal{A}$ can execute a_f, the predicates in p' are added in $\emptyset$ mode. All pal tuples whose modes are correctly inferred in this way are therefore removed from the pal ordering.

Chapter 4

LEARNING ACTION MODELS OF SIMULATED AGENTS

Simulator environments are widely used to test both planning and learning algorithms. They can operate on actions of variable granularity, from high-level parameterized actions like in PDDLGym (Silver and Chitnis (2020)) to simple low-level keyboard-based commands.

Since AAM requires rudimentary query answering capabilities only (when action signatures are available), it can be connected to simulators that answer queries in any format provided suitable translators to convert the output states into symbolic state representation are available. Thus, AAM is connected to simulators in the PDDLGym suite. PDDLGym provides OpenAI gym-like simulators for standard PDDL domains like BlocksWorld, Sokoban etc. Furthermore, since PDDLGym environments are based on symbolic planning domains, the action signatures are readily available. These simulators provide the capability to set an input initial state and execute a sequence of actions and provide the end result in image format. To translate the image into predicates that AAM understands, predicate classifiers are developed to extract state information in the user-defined vocabulary. AIA learns action models directly based on the agent's replies and thus it is assumed that the predicate classifiers are noise-free and provide an accurate state representation of the output image. Although simulators provide a good approximation of realistic agents that operate in sequential decision-making domains, those that are not based on symbolic domains, operate using rudimentary actions like "Up", "Down", "Left", "Right" and "Use Item". This presents an opportunity to learn action models of agents in terms of abstracted actions, that correspond to a sequence of low-level actions within the

simulator. For example, consider the simulated environment shown below: This sim-

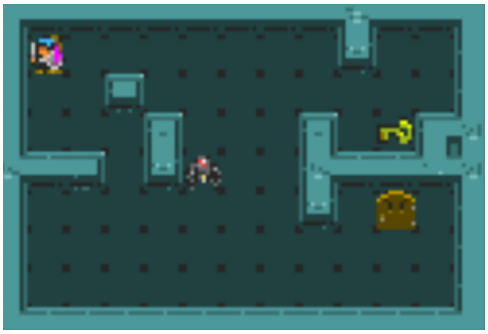


Figure 4.1: Sample State of GVGAI's Zelda Domain

ulator only operates using low-level actions input using the keyboard (Arrow Keys, Space bar). It is possible to learn an action that makes the player obtain the key using a series of low-level actions (keyboard inputs). An abstracted version of this sequence of actions is a single action that can achieve the same result. To comply with the assumptions of AIA, the following conditions are ensured for the environments:

- Simulator is "stationary": the world state does not change until the player executes an action
- Simulator dynamics are deterministic

To work directly with the Agent Assessment Module, the simulator must also possess two rudimentary functionalities:

- Random State Generator: In order to generate distinguishing queries Q_{PO}, a set of random states is to be passed as input to the query generation function. Thus the simulator must be able to provide a set of random states that are "valid" based on its internal model.
- Query Answering: The simulator must be able to run plan outcome queries $Q_{PO} = \langle s_I, \pi \rangle$. This means the simulator must allow setting a random initial

state directly and run a sequence of grounded actions which, are based directly on the actions available in $\mathbb{A}$.

- State validator: Setting the simulator to a certain initial state requires that the state be *valid* in terms of the symbolic vocabulary used to define it.

4.1 Learning action models of Type-1 agents

These simulated agents operate on high-level actions and the user's vocabulary perfectly captures the simulator state. Since AIA requires the action headers $\mathbb{A}_H$ as input, one option for learning action models of simulators would be to use those which are based directly on symbolic planning domains. PDDLGym is well suited for these requirements since it provides a simulator-like OpenAI Gym interface for some planning domains. The actions are directly derived from "domain.pddl" files and the states are rendered by using a mapping from the grounded objects in the domain to images. This image renders corresponding to objects in the domain are then arranged in a way that conveys the visual understanding of the state.

Random State Generation

Both the domains considered have a grid structure. Thus to generate a random state, each cell of a grid was randomly assigned to an object. The resulting state was then converted to a symbolic state using a simple mapping. To make sure these states are valid, these were also checked with the state validation function.

Query Answering Policy

Algorithm 3 summarizes the query answering policy for PDDLGym simulators. The function takes as input the query $\mathcal{Q}$ which consists of the initial state s_I and the plan π and outputs the length of longest executable prefix of the plan π, l and the final

state after executing l steps of the plan from s_I, s_F. First the simulator is queried to set the initial state s_I. This can fail if the state being set is "invalid". This could occur due to various reasons, some of which have been enumerated below taking the sokoban domain as an example:

- Mutually exclusive atoms being set to True in conjunction: For example, a grid cell cannot be both "clear" and contain an object or a "wall".
- Dependent predicates not being set correctly: For example, the 'at-goal(?ob)' predicate is set to true when a block is moved into the goal. Thus, if at-goal(block) is True, at(block,goal) must be True as well.
- Atoms that are necessary for defining a "meaningful" simulator state being absent: For example, For a simulator state to be meaningful/valid, the "player"'s position must be encoded in the state. Thus, if there does not exist a literal l such that at(player,l) $\in s_I$ for some $l \in O_{location}$ where $O_{location}$ is the set of all grounded objects of type "location" in the domain, then the state is invalid.

These state validation checks are added to make sure the states being set by the simulator are meaningful and semantically coherent with respect to the states encountered in a typical simulator run. If the simulator can successfully set the initial state, then each step of the plan π is sequentially run on the simulator. Since the simulator returns the result of an action execution as an image, these images are returned to AAM as query response and thus have to be converted to symbolic states.

Extracting State from Images

For the Sokoban and Doors domains, the grid-like structure of the state images was exploited to extract the state information. The grid contours were detected by thresholding the image followed by using OpenCV's *get_contour*() function. The contours

Algorithm 3 Query Answering Algorithm (PDDLGym)

Input: s_I, π
Output: l, s_F
success = simulator.set_state(s_I)
if success == True **then**
    state = s_I
    for i, a in enumerate(π) **do**
        next_state = simulator.step(a)
        if next_state == state **then**
            return i, state
        end if
        state = next_state
    end for
else
    return $0, s_I$
end if
return length(π), state

were filtered to keep the grid cells only. For example, in the Sokoban domain, state renders typically consist of the player, blocks, clear cells, walls and goal locations. Since each of these objects are unique, either template matching or simple color matching can be used to match the contents of each of the filtered contours with that of the closest object's image render. Thus, each cell of the grid is classified into whether it is clear, or has any of the aforementioned objects. After this, the symbolic state is generated by a simple process of matching the grid cells to the corresponding predicates. For example, if the grid cell "cell_1_0" is classified as a clear cell, the state would have the literal "clear(cell_1_0)" set to True and if the grid cell "cell_1_0" is

classified as a cell containing the "player", then the state would contain the literal "at-player(cell_1_0)".

4.1.1 Domains

Sokoban

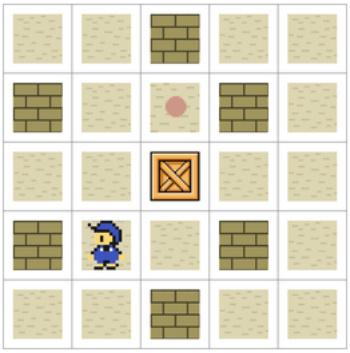


Figure 4.2: PDDLGym's Sokoban Domain

Fig 4.2. Shows a sample simulated state for the "Sokoban" domain. The world consists of blocks that can be pushed onto any of the goal cells in the grid and the episode ends when all blocks are pushed into goal cells.

Doors

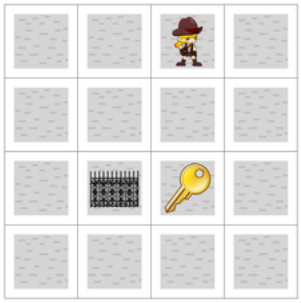


Figure 4.3: PDDLGym's Doors Domain

Fig 4.3. Shows a sample simulated state for the "doors" domain. The world consists of distinct rooms, each assigned with a key randomly spawned in the grid.

The player is to obtain the key to the final door (which may be locked within other rooms) and escape to end an episode.

4.2 Learning action models of Type-2 agents

For simulators of this type, the user's vocabulary captures only an abstraction of the state. In the context of simulator agents, "low-level actions" refer to low-level controllers that are used to directly interact with the simulator. For example, simple games such as those from the General Video Game AI competition or Atari Games suite, use keyboard arrow keys as action input to transition the state of the game. Learning abstract actions for these games presents a unique challenge since the same low-level action can operate differently depending upon the state of the simulator. For example, executing the low-level move-right action (right arrow key) in a state where the cell next to that of the player is not blocked by a wall or some other sprite results in the player's current cell becoming clear and a change in the player's location; while executing the same action in a state where the cell next to the player is blocked, results in no effect. More importantly, Low-level actions do not have an associated action signature that is required as an input to the Agent Interrogation Algorithm for learning the internal action model of the simulator. This means that actions have to be discovered before being learned. The next sections elaborate on the functionalities and methodologies for learning abstract action models of such agents. This thesis develops a grounded version of the models. Future work can explore extracting relational models from the learned grounded models.

4.2.1 Action discovery

Since the set of abstract action headers $\mathbb{A}_H$ is no longer available, it must be generated. One way to achieve this is to collect random execution traces from the

Algorithm 4 Action Discovery Algorithm(ADA)

Input: $\mathcal{A}, n$
Output: action_objects
action_objects = $\emptyset$
for $i = 1 : n$ **do**
  tr = generate_random_trace($\mathcal{A}$)
  for each state-action-next_state triple, $\{s_i^l, a_i, s_{i+1}^l\}$ in tr **do**
    $s_1^H \leftarrow$ abstract_state(s_i^l)
    $s_2^H \leftarrow$ abstract_state(s_{i+1}^l)
    if $s_1^H \neq s_2^H$ **then**
      a^H = new_action()
      a^H.state_before= s_1^H
      a^H.state_after= s_2^H
      if a^H = not in action_objects **then**
        a^H.assign_predicate_types()
        action_objects.append(a^H)
      end if
    end if
  end for
end for
return action_objects

agent and discover unique actions from the traces. An action here is uniquely defined as a transition from an abstract state to another. Note that defining an action in this way makes it difficult to obtain traces with action execution failures i.e. since an action a discovered from traces is uniquely defined as a transition from $s_1 \rightarrow s_2$, it is not possible to observe an action failing to apply in a state in a trace since it wouldn't be discovered without the $s_1 \rightarrow s_2$ transition. Algorithm 4 describes the

Action discovery algorithm to generate a list of Action objects containing arbitrarily named actions. First, a random trace is generated using the simulator (line 3-5). Note that since these state-action sequences are directly generated from the simulator, they would consist of low-level states and actions. For each of the "low-level traces", the states before and after the action execution are abstracted using the state abstraction function *abstract_state*. This function maps low-level states obtained from the simulator to high-level states in terms of the user's vocabulary (line 7,8). **Note:** For this work, the abstractions are assumed to be "pseudo-lossy" i.e., although information is lost in the abstraction of a state from the low-level simulator state to the user's interpretable vocabulary, the states generated by refining an abstract state are all connected. Future work can address true "lossy" abstractions. When the abstracted states s_1^H and s_2^H correspond to different states,an arbitrary name for the action object corresponding to the transition $s_1^H \rightarrow s_2^H$ is generated and stored in the *abstract_objects* list. Additionally, the member function *assign_predicate_type()* is run for the object a_H. Note that the states here consist of predicates grounded by objects in the domain. This means all state transitions and consequently, actions are described by literals. Inferring a relational operator from grounded actions is a non-trivial problem. This is because, in order to lift the operator to first order, its parameter list must be set. This can be arbitrarily long due to the presence of static predicates. Thus, a grounded action model is learned ie; the preconditions and effects of the actions consist of grounded literals and the parameter lists are empty. The *assign_predicate_type()* function classifies the grounded literals present in the pre-state (Stern and Juba (2017)) of an action. Let $L = s_1^H \cup s_2^H$; ie; L is the set of all literals present in either s_1^H or s_2^H. The assign_predicate_types() function classifies the literals $l \in L$ as:

- **Added literals**: $l \in s_2^H$ and $l \notin s_1^H$

- **Deleted literals**: $l \in s_1^H$ and $l \notin s_2^H$

Query Answering Policy

Algorithm 5 and 6 summarize the query answering function for simulators based on low-level actions. The function takes as input the Query Q consisting of an initial state s_I, plan π and the *action_objects* list generated from the action discovery algorithm (Algorithm 4) (which is run when the agent $\mathcal{A}$ is initialized) and outputs the length of the longest executable prefix of the plan π, l and the final state after executing l steps of the plan from s_I, s_F. Similar to the previous case, first, the simulator state is set to the initial state s_I which can fail due to the same reasons described in section 4.1. Then, for each named action in the plan, π, the action object corresponding to this name is obtained from action_objects using the get_action_object function. Recall that each action stored in action_objects had literals corresponding to added_literals and deleted_literals. Next, the literals corresponding to the added_literals are added and deleted_literals are deleted from the state. It is assumed that literals being added to the state are not already present and literals being deleted from the state are already present. If either of these is not true, the action is deemed to have failed to execute and the length of execution l and the state so far is returned. The new state obtained is then checked for validity by setting the simulator state to this state. Finally, the previous state and the updated new state are refined to the simulator's vocabulary and the agent is asked to plan from the previous state to the new state using the simulator. This is necessary because the simulator has no knowledge of the action a and can only plan using low-level actions. Thus, to check if the transition from $state \rightarrow next\_state$ is possible, the agent is asked to plan between these states. Since this functionality is not typically provided with simulators, a simple A* search algorithm is used to plan internally using a heuristic that calculates the distance between

two states as: $h(s_1, s_2) = \text{player\_manhattan\_distance}(s_1, s_2) + \text{editing\_distance}(s_1, s_2)$ where player_manhattan_distance(s_1, s_2) returns the manhattan distance between the players location in the two input states and editing_distance(s_1, s_2) returns the number of literals that are different between s_1 and s_2. If the agent can successfully plan

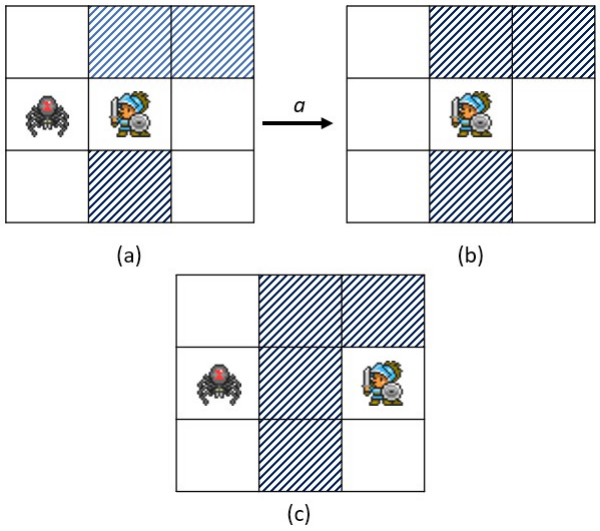


Figure 4.4: Sample Discovered Action That Can Fail to Execute in Simulator

between these states, then the execution of the action a is considered to be successful. This step is required since the simulator cannot perform the high-level actions directly and must be operated using actions from its own action space. Further, it is possible that the application of add and delete effects of a discovered action on a state s may result in a valid state but this transition may not be possible in the low-level simulator. For example, consider the action a shown in figure Fig. 4.4. The action a adds the literals *clear-cell_0_2* and deletes the literals *at-monster0-cell_0_2*, *next_to_monster*, and *monster_alive-monster0* to the state shown in (a) to yield the state (b). These effects can be applied to the state shown in (c), but there is no sequence of low-level actions that can bring about this change in the simulator's state. Thus the action a cannot be applied in the state shown in (c). Such cases mandate the additional step of querying the agent to plan between the pre and post states of

a discovered action.

AIA and Dependent Predicates

As mentioned in section 3.3.4, The update_pal_ordering function in AIA is called when the agent is unable to answer a query $Q = \langle s_I, \pi \rangle$. For non-simulator domains, this function creates a state with all predicates set to True and finds the minimal state required to run the failing action a by eliminating those literals, whose removal does not change the answer from the agent to the query Q. For simulator-based domains, this cannot be done since the state validator checks the validity of a state based on hard domain-specific rules. This also means that sequentially removing predicates and testing the agent's response to Q to infer the mode of a pal tuple can prove to be erroneous since the fact that a state is invalid does not necessarily imply that a literal is required as a precondition for the execution of an action. In order to correctly learn the mode of precondition of an action (a) in domains with derived predicates (p) and hard state validation rules, it must be checked if $\exists s \in \mathbb{S}$ such that the agent can execute the action a from state s where $l \notin s$ where l is the predicate p grounded with a set of compatible objects from the domain. If such state s does not exist, then it can be inferred that the pal tuple $\langle p, a, pre \rangle$ must be in the $+$ mode otherwise, in $\emptyset$ mode. Searching through the entire set of simulator state $\mathbb{S}$ is a computationally expensive task and is a functionality that is required in the simulator itself. Without this, some spurious preconditions may be learned in the final learned model.

4.2.2 Domains

Zelda

Fig 4.5 shows a sample simulator state for the "Zelda" domain. The world consists of a key that can be used to open a door to escape, the player, and monsters that must be eliminated before escaping through the door. An episode/trace of the game ends when the player has killed all the monsters and used the key to escape through the door. The player can move one cell at a time in the direction it is facing. Only the orientation information is lost in abstracting the simulator state to the user-interpretable vocabulary. If the player moves into the cell containing the key, the player picks up the key and if the player executes the low-level action "ACTION_USE" or the corresponding keyboard command when facing a monster in a cell adjacent to the monster, the monster is slain. Similarly, using the key when next to the door and facing the door opens it. The abstract predicates used to describe the state in this domain consist of:

- *at*(?ob-sprite, ?loc-location): True when ob is at location loc
- *monster_alive*(?ob-sprite): True when ob is alive
- *next_to_monster*(): True when the player is next to a monster
- *has_key*(): True when the player has obtained the key
- *escaped*(): True when the player has slain the monster, obtained the key, opened the door and escaped
- *wall*(?loc - loc): True when the location loc has a wall
- *clear*(?loc - loc): True when the location loc is clear (no objects)

where “sprite” refers to any of “player”,“key” and “monster” and “location” refers to any of the cells in the grid.

Door Key Monster Player

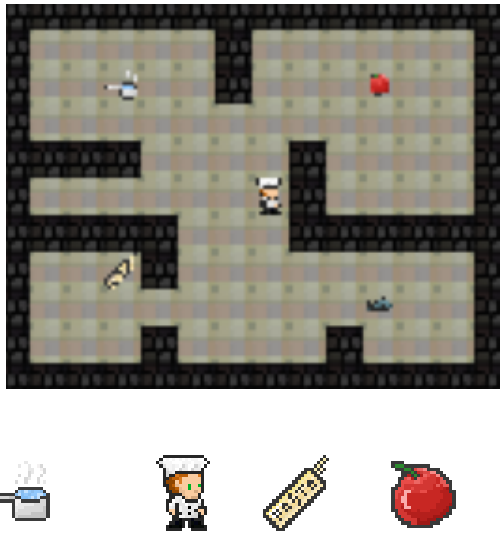
Figure 4.5: Sample Zelda State and Sprite Set

CookMePasta

Fig 4.6 shows a sample simulator state for the ‘Cook-Me-Pasta’ domain. The world consists of ‘raw_pasta’, ‘sauce’, ‘boiling_water’ and ‘tuna’. The objective of the game is to make the final product ‘pasta’, which is obtained by combining (pushing into) ‘cooked pasta’ (‘boiling_water + ‘raw_pasta’) with ‘cooked sauce’ (‘sauce’ + ‘tuna’). The abstract predicates used to describe the state in this domain consists of:

- *at*(?ob-sprite,?location): True when the sprite ob is at the grid location location. Here, a sprite refers to any objects of type *player, boiling_water, pasta, tomato, tuna, cooked_sauce* and, *cooked_pasta*.
- *wall*(?loc - location): True when a wall is at location loc
- *clear*(?loc - location): True when the cell loc is clear

- *pasta_cooked*(): True when the pasta has been created



Boiling Water Player Pasta Tomato Tuna

Figure 4.6: Sample CookmePasta State and Sprite Set

Escape

Fig 4.7 shows a sample simulated state for the "escape" domain. The world consists of movable blocks and "holes". The blocks can be pushed into the "holes" to clear out a path from the player's location to the goal to complete the game. The abstract predicates used to describe a state in this game are:

- *at*(?ob - sprite,?loc - location): True when a sprite (player/block) is at a location loc
- *wall*(?loc - location): True when a wall is at location loc
- *is_door*(?loc - location): True when a door(goal) is at the cell location loc
- *is_hole*(?loc - location): True when a hole is at the cell location loc
- *clear*(?loc - location): True when the cell location loc is clear

Block Goal Hole Player

Figure 4.7: Sample Escape State and Sprite Set

Snowman

Fig 4.8 shows a sample simulated state for the "snowman" domain. The world consists of 3 pieces of a snowman: the top, middle and bottom piece, A key which can be used to unlock a door and the goal cell. The objective of the game is to assemble the snowman in the goal location in order, constrained by the player being able to hold only 1 piece at any given time. The abstract predicates used to define a state in the snowman domain consist of:

- *at*(?ob - sprite, ?loc - location): True when the sprite (player/key/lock/any of the snowman pieces) ob is at the location loc.
- *is_goal*(?loc - location): True when the cell location loc is the goal cell
- *player_has*(?ob - sprite): True when the player is holding the any of the snowman pieces (ob)
- *has_key*(): True when the player has picked up the key

- *top_placed*(): True when the top piece has been placed in the goal location
- *middle_placed*(): True when the middle piece has been placed in the goal location
- *bottom_placed*(): True when the bottom piece has been placed in the goal location
- *wall*(?loc - location): True when the cell location loc has a wall
- *clear*(?loc - location): True when the cell location loc is clear

Figure 4.8: Sample Snowman State and Sprite Set

Note: In all the domains, the grid structure is maintained by using the predicates "leftOf(?ob1-location,?ob2-location)", "rightOf(?ob1-location,?ob2-location)", "above(?ob1-location,?ob2-location)" and "below(?ob1-location,?ob2-location)" but these predicates are added to the abstract model being used in AIA as preconditions for all discovered actions for the sake of simplicity.

Algorithm 5 Query Answering Policy (low-level simulators) Part: 1

Input: s_I, π, action_objects
Output: l, s_F
state = s_I
next_state = state
success = simulator.set_state(s_I)
if success $\neq$ True **then**
  return $0, s_I$
end if
for i, a_l in enumerate(π) **do**
  a = get_action_object(action_objects,a_l)
  for l in a.added_atoms() **do**
    if $l \in$ next_state **then**
      return i,next_state
    else
      next_state.add(l)
    end if
  end for
  for l in a.deleted_atoms() **do**
    if $l \notin$ next_state **then**
      return i,next_state
    else
      next_state.delete(l)
    end if
  end for
  success = simulator.set_state(next_state)

Algorithm 6 Query Answering Policy (low-level simulators) Part: 2

```
    if success == True then
        sim_initial_state = get_sim_state(state)
        sim_final_state = get_sim_state(next_state)
        plan_success, plan = agent.plan(sim_initial_state,sim_final_state)
        if plan_success == True then
            state = next_state
        else
            return i,state
        end if
    end if
end for
return length(π),state
```

Chapter 5

EXPERIMENTS AND RESULTS

The simulator agents were implemented in Python and all the experiments were run on a system with i9-9900 processor, 64gb RAM and an RTX 2080. To optimize the implementation, the queries and the simulator planning requests (initial and goal state) were stored in a hashmap and retrieved instead of repeatedly generating the same query or planning between the same two states.

5.1 Type-1 Agents

The agent was initialized with an instance of either the "doors" or the "Sokoban" PDDLGym simulator. Some predicates in the ground-truth domain models for PDDLGym domains are only present to aid the processing and rendering of the state image and these are added to the abstract model that AIA starts with for the sake of simplicity. Additionally, since PDDLGym domains have a ground-truth PDDL file, the accuracy of the learned models can be checked against the ground-truth domains.

5.1.1 Sokoban

The agent was initialized with 20 random valid states and AIA learned the correct model with 201 queries (over 5 runs), for 3 actions and 35 instantiated predicates.

5.1.2 Doors

The agent was initialized with 20 random valid states and AIA learned the correct model with 252 queries (over 5 runs), for 2 actions and 10 instantiated predicates.

5.2 Type-2 Agents

Since the simulators developed based on the games from the GVGAI suite are not directly based on planning domains, there is no ground-truth model to check the accuracy of the learned model. Thus, to check the correctness of the learned model, the states in the traces collected during action discovery were abstracted and the learned model was used to plan from the initial state of the trace to each of the intermediate states.In all of the domains described below, the learned model was able to run the complete traces used to discover actions. ie; given the initial state s_0 and the intermediate states s_k for $k \in len(trace)$, the learnt model M was able to plan between $s_0 \rightarrow s_k \forall k \in [0, len(trace)]$. Thus, since the learned model was found to be consistent with the generated traces and all the actions were learned (AIA is guaranteed to converge), it can be said that from empirical evidence, this method to learn action models of Type-2 agents is sound and complete. The next few sections discuss the results for the chosen domains. In all of the domains, a general trend of increase in the number of queries with increase in the number of *pal* tuples was observed. Additionally, to detect a majority of the possible actions in a domain, a random initial state was solved and used as a trace. Thus, since every initial state in a trace is random, there is no correlation between the number of actions detected and the grid-size. The tables in each section also compare the number of queries required by AIA to The number of queries required for a Naïve/brute-force approach. The latter is obtained by using the expression: number_of_*pal*_tuples $*2^{|\mathbb{P}|}$. This is because, for each predicate in each action in both precondition and location, a query would be required for each state. In all of the domains, AIA learned the model with significantly smaller number of queries as compared to the Naïve approach.

5.2.1 Zelda

Table 5.1 shows the results of the number of queries required to learn the model, the number of actions, and the number of *pal* tuples for different grid sizes. Fig. 5.1 show some of the notable actions learned by AIA (The actions are named for the sake of interpretability since discovered actions are named arbitrarily):

- (a) follows the domain dynamics of slaying a monster which is only possible when the player is next to the monster.
- (b) follows the domain dynamics of getting the key when the player moves into the cell containing the key
- (c) follows the domain dynamics of moving into a cell when the cell is clear
- (d) follows the domain dynamics of escaping through the door when the monster has been killed and the player has the key.

Number of grid cells	$\lvert\mathbb{P}\rvert$	$\lvert\mathbb{A}\rvert$	Number of *pal* tuples	Number of Queries (AIA)	Number of Queries (Naïve)
12	58	6	288	235	8.3E+19
20	96	9	612	512	4.85E+31
30	142	9	792	696	4.42E+45
42	202	18	2160	1891	1.39E+64
64	308	16	2688	2423	1.4E+96

Table 5.1: Results for Zelda Domain

```
(:action slay_monster_1_1
  :parameters ()
  :precondition (and (at-player0-cell_2_1)
        (at-monster_1_1-cell_1_1)
        (monster_alive-monster_1_1)
        (next_to_monster))
  :effect (and (not (at-monster_1_1-cell_1_1))
        (not (monster_alive-monster_1_1))
        (not (next_to_monster))
        (clear-cell_1_1)))
```

(a)

```
(:action get_key0
  :parameters ()
  :precondition (and (at-player0-cell_1_2)
        (at-key0-cell_0_2))
  :effect (and (not (at-player0-cell_1_2))
        (not (at-key0-cell_0_2))
        (at-player0-cell_0_2)
        (has_key)
        (clear-cell_1_2)))
```

(b)

```
(:action move_to_cell_1_1
 :parameters ()
 :precondition (and (at-player0-cell_2_1)
       (clear-cell_1_1))
 :effect (and (not (at-player0-cell_2_1))
       (not (clear-cell_1_1))
       (at-player0-cell_1_1)
       (clear-cell_2_1)))
```

(c)

```
(:action escape_through_door
  :parameters ()
  :precondition (and (at_0-player0-cell_1_1)
        (at_3-door0-cell_2_1)
        (clear-cell_2_1)
        (has_key-)
        (not (monster_alive-monster_0_0))
        (not (at_2-monster_0_0-cell_0_0)))
  :effect (and (not (at_0-player0-cell_1_1))
        (not (clear-cell_2_1))
        (clear-cell_1_1)
        (escaped-)
        (at_0-player0-cell_2_1)))
```

(d)

Figure 5.1: Sample Actions from Zelda Domain

5.2.2 CookMePasta

Table 5.2 shows the results for the number of queries required to learn the model, the number of actions and the number of *pal* tuples for different grid sizes. Fig. 5.2 show some of the notable actions learned by AIA:

- (a) follows the domain dynamics of combining the raw-pasta and water when

the raw-pasta is pushed into the water by the player

- (b) follows the domain dynamics of moving into a cell when the cell is clear
- (c) follows the domain dynamics of combining the cooked-pasta and cooked-sauce to obtain the finished pasta when one of them is pushed into the other by the player.
- (d) follows the domain dynamics of pushing the pot of boiling water onto a clear cell by the player when the player and boiling water pot are next to each other.

Number of grid cells	$\lvert\mathbb{P}\rvert$	$\lvert\mathbb{A}\rvert$	Number of *pal* tuples	Number of Queries (AIA)	Number of Queries (Naïve)
20	112	30	3000	2136	1.56E+37
30	160	38	4712	3523	6.89E+51
42	226	40	6720	5075	7.25E+71
64	351	58	14732	11115	6.8E+109

Table 5.2: Results for CookMePasta Domain

```
(:action combine_rawpasta0_with_water0
  :parameters ()
  :precondition (and (at_0-player0-cell_4_2)
         (at_3-raw_pasta0-cell_2_2)
         (at_4-boiling_water0-cell_3_2))
  :effect (and (not (at_0-player0-cell_4_2))
                 (not (at_3-raw_pasta0-cell_2_2))
                 (not (at_4-boiling_water0-cell_3_2))
           (at_0-player0-cell_3_2)
           (clear-cell_4_2)
           (at_5-pasta_in_place0-cell_2_2)))
```

(a)

```
(:action move_to_cell_4_1
  :parameters ()
  :precondition (and (at_0-player0-cell_4_1)
                     (clear-cell_4_2))
  :effect (and (not (at_0-player0-cell_4_1))
               (not (clear-cell_4_2))
               (clear-cell_4_1)
               (at_0-player0-cell_4_2)))
```

(b)

```
(:action finish_pasta
  :parameters ()
  :precondition (and (at_0-player0-cell_1_0)
        (at_5-pasta_in_place0-cell_1_2)
        (at_6-sauce_in_place0-cell_1_1))
  :effect (and (not (at_0-player0-cell_1_0))
        (not (at_5-pasta_in_place0-cell_1_2))
        (not (at_6-sauce_in_place0-cell_1_1))
        (pasta_cooked-)
        (clear-cell_1_0)
        (at_7-pasta_done0-cell_1_2)
        (at_0-player0-cell_1_1)))
```

(c)

```
(:action push-boiling_water0
  :parameters ()
  :precondition (and (at_0-player0-cell_0_1)
        (at_4-boiling_water0-cell_1_1)
        (clear-cell_2_1))
  :effect (and (not (at_0-player0-cell_0_1))
        (not (at_4-boiling_water0-cell_1_1))
        (not (clear-cell_2_1))
        (clear-cell_0_1)
        (at_4-boiling_water0-cell_2_1)
        (at_0-player0-cell_1_1)))
```

(d)

Figure 5.2: Sample Actions from CookMePasta Domain

5.2.3 Escape

Table 5.3 shows the results for the number of queries required to learn the model, the number of actions and the number of *pal* tuples for different grid sizes. Fig. 5.3 show some of the notable actions learned by AIA:

- (a) follows the domain dynamics of moving into a cell when the cell is clear
- (b) follows the domain dynamics of the player being able to push a block when the cell adjacent to the block is clear and the player is adjacent to the block
- (c) follows the domain dynamics of escaping (trace terminating action) when the player moves into the goal cell.
- (d) follows the domain dynamics of pushing a block into a hole when the block is adjacent to a hole and the player is adjacent to the block

Number of grid cells	$\|\mathbb{P}\|$	$\|\mathbb{A}\|$	Number of *pal* tuples	Number of Queries (AIA)	Number of Queries (Naïve)
9	41	4	136	114	2.99E+14
16	79	9	558	445	3.37E+26
20	101	12	936	740	2.37E+33
42	214	31	4464	3602	1.18E+68

Table 5.3: Results for Escape Domain

```
(:action moveto_cell_3_2
  :parameters ()
  :precondition (and (at_0-player0-cell_3_1)
                     (clear-cell_3_2))
  :effect (and (not (at_0-player0-cell_3_1))
               (not (clear-cell_3_2))
               (at_0-player0-cell_3_2)
               (clear-cell_3_1)))
```

(a)

```
(:action push_block_6
  :parameters ()
  :precondition (and (at_0-player0-cell_2_2)
                     (at_1-block6-cell_2_1)
                     (clear-cell_2_0))
  :effect (and (not (at_0-player0-cell_2_2))
               (not (at_1-block6-cell_2_1))
               (not (clear-cell_2_0))
               (at_1-block6-cell_2_0)
               (at_0-player0-cell_2_1)
               (clear-cell_2_2)))
```

(b)

```
(:action escape
  :parameters ()
  :precondition (and (at_0-player0-cell_2_3))
  :effect (and (not (at_0-player0-cell_2_3))
          (at_0-player0-cell_2_4)
          (clear-cell_2_3)
          (escaped-))))
```

(c)

```
(:action push_block2_into_hole
 :parameters ()
 :precondition (and (at_0-player0-cell_3_4)
       (at_1-block2-cell_2_4)
       (is_hole-cell_1_4))
        :effect (and (not (at_0-player0-cell_3_4))
       (not (at_1-block2-cell_2_4))
       (clear-cell_3_4)
       (at_0-player0-cell_2_4)))
```

(d)

Figure 5.3: Sample Actions from Escape Domain

5.2.4 Snowman

Table 5.4 shows the results for the number of queries required to learn the model, the number of actions and the number of *pal* tuples for different grid sizes. Fig. 5.4 show some of the notable actions learned by AIA:

- (a) follows the domain dynamics of picking up the bottom-piece when the player is adjacent to the cell containing the piece.
- (b) follows the domain dynamics of the player dropping the top piece in a clear cell when it is next to the cell.

- (c) follows the domain dynamics of stacking the top piece when the player is next to the cell, is holding the top piece and the bottom and middle pieces are already stacked.

Number of grid cells	$\|\mathbb{P}\|$	$\|\mathbb{A}\|$	Number of *pal* tuples	Number of Queries (AIA)	Number of Queries (Naïve)
20	109	57	5358	3953	3.47754E+36
25	135	45	4950	3653	2.15603E+44
30	162	64	8192	6189	4.78905E+52
42	222	74	11840	9228	7.98014E+70
64	322	51	9996	8401	8.5405E+100

Table 5.4: Results for Snowman Domain

```
(:action pickup-bottompiece
  :parameters ()
  :precondition (and (at_0-player0-cell_2_0)
        (at_5-bottom_piece0-cell_2_1)
        (not (player_has_1))
        (not (player_has_0)))
  :effect (and
        (not (at_5-bottom_piece0-cell_2_1))
        (clear-cell_2_1)
        (player_has_2)))
                    (a)
```

```
(:action drop_toppiece
 :parameters ()
 :precondition (and (at_0-player0-cell_0_2)
       (player_has_0)
       (clear-cell_0_1)
       (not (at_3-top_piece0-cell_0_2)))
  :effect (and (not (player_has_0))
       (not (clear-cell_0_1))
       (at_3-top_piece0-cell_0_1)))
                    (b)
```

```
(:action stack_toppiece
  :parameters ()
  :precondition (and (at_0-player0-cell_3_4)
        (at_4-middle_piece0-cell_2_4)
        (at_5-bottom_piece0-cell_2_4)
        (is_goal-cell_2_4)
        (player_has_0)
        (clear-cell_6_5)
        (bottom_placed-)
        (middle_placed-))
  :effect (and (not (player_has_0))
        (at_3-top_piece0-cell_2_4)
        (top_placed)))
                    (c)
```

Figure 5.4: Sample Actions from Snowman Domain

Chapter 6

CONCLUSIONS AND FUTURE WORK

This thesis developed methods to learn action models of two types of simulated agents using the Agent Assessment Module:

- Simulators operating on high-level actions, where the user's vocabulary captures the simulator state perfectly.
- Simulators operating on low-level actions, where the user's vocabulary captures an abstraction of the simulator state.

Sokoban and Doors domains from the PDDLGym suite of environments were used to showcase AAM's compatibility with typical black-box simulators commonly used for symbolic reinforcement learning tasks. This involved adding functionalities like state validation checks and converting the simulator's image output to symbolic states. Correct models for both domains were learned by AIA provided the image-to-state converter is assumed to be noiseless. For both domains, AIA learned the correct model. Simplified versions of games from the GVGAI suite are used to assess the functionalities required to learn interpretable action models of simulators based on low-level actions. It was concluded that the agent additionally required planning capabilities to answer queries and the interfacing framework must discover actions from low-level traces that can be directly obtained from the simulator. Four domains were adapted from the GVGAI collection, namely, Zelda, Cookmepasta, Escape, and Snowman, and interpretable abstract action models of these domains were learned using AAM. Since there is no ground truth model to compare the learned model

against, it was instead used to plan end-to-end as well as between the intermediate states of the traces collected during action discovery. The learned model was able to plan successfully for all state transitions within the collected traces. Thus, from empirical results, this method of learning action models of simulated agents appears to be sound and complete. There are many possible future work directions. The abstractions can be made truly lossy which would possibly result in an approximation of the true model rather than an accurate model since the application of the same action on different groundings of an abstract state can result in different states which may not be connected. Simulators with stochastic actions and a dynamic environment can be investigated to learn the functionalities and modifications required for AAM to learn the model of a realistic simulator domain. Additionally, the modifications required within AAM to accommodate noisy image-classifiers when learning models of Type-1 agents could be investigated.

BIBLIOGRAPHY

Aineto, D., S. J. Celorrio and E. Onaindia, "Learning action models with minimal observability", Artificial Intelligence **275**, 104–137 (2019).

Bonet, B. and H. Geffner, "Learning first-order symbolic representations for planning from the structure of the state space", in "Proc. ECAI", (2020).

Čertickỳ, M., "Real-time action model learning with online algorithm 3 sg", Applied Artificial Intelligence **28**, 7, 690–711 (2014).

Cresswell, S. and P. Gregory, "Generalised domain model acquisition from action traces", in "Proc. ICAPS", (2011).

Cresswell, S., T. McCluskey and M. West, "Acquisition of object-centred domain models from planning examples", in "Proc. ICAPS", (2009).

Gil, Y., "Learning by experimentation: Incremental refinement of incomplete planning domains", in "Proc. ICML", (1994).

Gregory, P. and S. Cresswell, "Domain model acquisition in the presence of static relations in the lop system", in "Proceedings of the International Conference on Automated Planning and Scheduling", vol. 25 (2015).

Konidaris, G., L. P. Kaelbling and T. Lozano-Perez, "From skills to symbols: Learning symbolic representations for abstract high-level planning", Journal of Artificial Intelligence Research **61**, 215–289 (2018).

Silver, T. and R. Chitnis, "PDDLGym: Gym environments from PDDL problems", in "ICAPS Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL)", (2020).

Stern, R. and B. Juba, "Efficient, safe, and probably approximately complete learning of action models", in "Proc. IJCAI", (2017).

Suárez-Hernández, A., J. Segovia-Aguas, C. Torras and G. Alenyà, "Online action recognition", (2020).

Verma, P., S. R. Marpally and S. Srivastava, "Asking the right questions: Learning interpretable action models through query answering", Proceedings of the AAAI Conference on Artificial Intelligence (2021).

Verma, P. and S. Srivastava, "Learning generalized models by interrogating black-box autonomous agents", in "AAAI 2020 Workshop on Generalization in Planning", (2020).

Yang, Q., K. Wu and Y. Jiang, "Learning action models from plan examples using weighted max-sat", Artificial Intelligence **171**, 2-3, 107–143 (2007).

Zhuo, H. H. and S. Kambhampati, "Action-model acquisition from noisy plan traces", in "Proc. IJCAI", (2013).