Asking the Right Questions: Interpretable Action Model Learning using Query-Answering

Pulkit Verma, Shashank Rao Marpally, and **Siddharth Srivastava**
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, AZ 85281 USA
{verma.pulkit, smarpall, siddharths}@asu.edu

Abstract

This paper develops a new approach for estimating an interpretable, relational model of a black-box autonomous agent that can plan and act. Our main contributions are a new paradigm for estimating such models using a minimal query interface with the agent, and a hierarchical querying algorithm that generates an interrogation policy for estimating the agent's internal model in a vocabulary provided by the user. Empirical evaluation of our approach shows that despite the intractable search space of possible agent models, our approach allows correct and scalable estimation of interpretable agent models for a wide class of black-box autonomous agents. Our results also show that this approach can use predicate classifiers to learn interpretable models of planning agents that represent states as images.

1 Introduction

The growing deployment of AI systems leads to a pervasive problem: how would we ascertain whether an AI system will be safe, reliable, or useful in a given situation? This problem becomes particularly challenging when we consider that most autonomous systems are not designed by their users; their internal software may be unavailable or difficult to understand. Such scenarios feature *black-box* AI systems that are not aware of a user's preferred model representation.

This paper presents a new approach for estimating interpretable, relational models for such agents. Consider a situation where Hari(ette) ($\mathcal{H}$) wants their robot ($\mathcal{A}$) to clean up their lab, but s/he is unsure whether it is up to the task and wishes to estimate $\mathcal{A}$'s internal model in an interpretable representation that s/he is comfortable with (e.g., a relational STRIPS-like language (Fikes and Nilsson 1971; Fox and Long 2003)). Thus, $\mathcal{H}$ may ask $\mathcal{A}$ a series of questions, e.g., "What do you think will happen if you picked up bottle 1, bottle 2 and bottle 3 in succession?" Naïve approaches to the problem would require too many questions and place strong requirements on $\mathcal{A}$'s knowledge of $\mathcal{H}$'s preferred representations[1]. If this problem could be solved efficiently, lay persons would be able to efficiently determine the applicability of a wide class of AI systems, thus improving the overall usability as well as field-debuggability of AI systems.

We use a rudimentary class of queries to make our approach applicable to a broad class of agent implementations including simulator-based and analytical model-based agents: we use *plan outcome queries* that ask $\mathcal{A}$ what, according to it, would be the outcome of executing the longest executable prefix of a plan π on a hypothetical initial state s. These queries can also be answered by connecting the agent with a simulator and observing the result of executing π.

Our approach generates a sequence of queries ($\mathcal{Q}$) depending on the agents responses (θ) during the query process; the result of the overall interrogation process is a complete model of $\mathcal{A}$. In order to generate queries, we use a top-down process that eliminates large classes of agent-inconsistent models by computing queries that discriminate between pairs of *abstract models*. When an abstract model's answer to a query differs from the agent's answer, we effectively eliminate the the entire set of possible concrete models that are refinements of this abstract model.

In developing the first steps towards this paradigm, we assume that the user wishes to estimate $\mathcal{A}$'s internal model as a STRIPS-like relational model with conjunctive preconditions, add lists, and delete lists (and that the agent's model is expressible as such), although our framework can be extended to handle other types of formal domain representations. Further, we assume that the agent has functional definitions for the relations and actions in the user's vocabulary (these definitions can be programmed as Boolean functions over the state), and that it can be placed in a simulator to ensure that it answers truthfully. Section 2 presents our overall framework with algorithms and theoretical results about their convergence properties.

Our empirical evaluation (Section 3) shows that this method can efficiently learn correct models for IPC-model agents (International Planning Competition 2011). It also shows that our system can use image-based predicate classifiers to infer correct models for simulator based agents that respond with an image representing the result of the query plan's execution.

[1]Just 2 actions and 5 grounded propositions would yield $7^{2\times5} \sim 10^8$ possible STRIPS models – each proposition could be absent, positive or negative in the precondition and effects of each action, and cannot be positive (or negative) in both preconditions and effect simultaneously. A query strategy that inquires about each occurrence would be not only unscalable but also inapplicable on simulator-based agents that do not know their actions' preconditions and effects.

Related work A number of researchers have explored the problem of learning agent models from observations of its behavior (Yang, Wu, and Jiang 2007; Cresswell, McCluskey, and West 2009; Cresswell and Gregory 2011; Stern and Juba 2017). To the best of our knowledge, ours is the first approach to address the problem of generating query strategies for inferring relational models of black-box agents.

Camacho and McIlraith [2019] recently presented an approach for learning LTL models from an agent's observed state trajectories using an Oracle with knowledge of the target LTL representation. The oracle could also generate counterexamples when the estimated model differed from the true model. Amir and Chang [2008] use logical filtering (Amir and Russell 2003) to learn partially observable action models from action and observed traces. LOCM (Cresswell, McCluskey, and West 2009) and LOCM2 (Cresswell and Gregory 2011) present another class of algorithms that use finite-state machines to create action models from observed plan traces. LOUGA (Kučera and Barták 2018) combines a genetic algorithm with an ad-hoc method to learn planning operators from observed plan traces. FAMA (Aineto, Celorrio, and Onaindia 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. While both FAMA and LOUGA require a post-processing step to include the intersection of all states where an action is applied into its preconditions, it is not clear that such a process would necessarily converge to the correct model. Our experiments indicate that such approaches exhibit oscillating behavior in terms of model accuracy because some data traces can include spurious predicates. FAMA assumes that there are no negative literals in action preconditions. Khardon and Roth [1996] address the problem of making model-based inference faster *given a set of queries*, under the assumption that a static set of models represents the true knowledge base. Incremental Learning Model (Ng and Petrick 2019) uses reinforcement learning to learn the non-stationary model without using plan traces, and requires extensive training data to learn the full model correctly.

In contrast, our approach directly queries the agent and is guaranteed to converge to the true model while presenting a running estimate of the accuracy of the derived model; it can therefore be used in non-stationary settings where the agents model undergoes a change due to learning or a software update. In such a situation our algorithm can resume querying the system whereas approaches that derive models from observed plan traces would require arbitrarily long data-collection sessions to ensure the collection of sufficient uncorrelated data. Moreover, unlike Stern and Juba [2017] our approach does not require $\mathcal{A}$ to provide intermediate states in an execution, or to have an Oracle that could provide counterexamples as needed in Camacho and McIlraith [2019]'s approach. In contrast to approaches for white-box model-maintenance (Bryce, Benton, and Boldt 2016), our approach does not require $\mathcal{A}$ to know about $\mathcal{H}$'s preferred representation language.

The field of active learning (Settles 2009) addresses the related problem of selecting which data-labels to acquire for learning single-step decision-making models using statistical measures of information. But the effective feature set here is the set of all possible plans, which makes conventional methods for evaluating the information gain of possible feature labelings infeasible. In contrast, our approach uses hierarchical abstraction to select questions to ask, while inferring a multi-step decision-making (planning) model. Information-theoretic metrics could also be used in our approach whenever such information is available.

2 The Agent-Interrogation Task

We assume that $\mathcal{H}$ needs to estimate $\mathcal{A}$'s model as a STRIPS-like planning model (Fikes and Nilsson 1971) represented as a pair $\mathcal{M} = \langle \mathbb{P}, \mathbb{A} \rangle$, where $\mathbb{P} = \{p_1, \ldots, p_n\}$ is a finite set of predicates; $\mathbb{A} = \{a_1, \ldots, a_k\}$ is a finite set of parameterized actions (operators). Each action $a_j \in \mathbb{A}$ is represented as a tuple $\langle header(a_j), pre(a_j), \textit{eff}(a_j) \rangle$, where $header(a_j)$ is the action header representing action name and action parameters, $pre(a_j)$ represents the set of predicates that must be true in a state where a_j can be applied, $\textit{eff}(a_j)$ is the set of predicates that will change to true or false after a_j is applied.

Each predicate can be instantiated using the parameters of an action where number of parameters are bounded by the maximum arity of the action. E.g., consider the action *load_truck*$(?v1, ?v2, ?v3)$ and predicate $at(?x, ?y)$ in Logistics domain. The predicate can be instantiated using action parameters $v1$, $v2$, and $v3$ as $at(?v1, ?v1)$, $at(?v1, ?v2)$, $at(?v1, ?v3)$, $at(?v2, ?v2)$, $at(?v2, ?v1)$, $at(?v2, ?v3)$, $at(?v3, ?v3)$, $at(?v3, ?v1)$, and $at(?v3, ?v2)$. We represent the set of all such possible predicates instantiated with action parameters as $\mathbb{P}^*$.

The only information $\mathcal{H}$ has is the $header(a)$ of actions $a \in \mathbb{A}$ that $\mathcal{A}$ can perform. We denote the set of headers for all actions by $\mathbb{A}_H$. As noted in the Introduction, $\mathcal{A}$ has functional definitions of the predicates (with their parameters) in $\mathcal{H}$'s vocabulary, and therefore there is sufficient information for a dialog between $\mathcal{H}$ and $\mathcal{A}$ about the outcomes of hypothetical action sequences.

We define the overall problem of agent interrogation as follows. Given a class of queries and an agent with an unknown model who can answer these queries, determine the model of the agent. More precisely, an *agent interrogation task* is defined as a tuple $\langle \mathcal{M}^{\mathcal{A}}, \mathbb{Q}, \mathbb{P}, \mathbb{A}_H \rangle$ where $\mathcal{M}^{\mathcal{A}}$ is the true model of the agent $\mathcal{A}$ (unknown to the interrogator) being interrogated, $\mathbb{Q}$ is the class of queries that can be posed to the agent by the interrogator $\mathcal{H}$, and $\mathbb{P}$ and $\mathbb{A}_H$ are the set of predicates and action headers known to both $\mathcal{H}$ and $\mathcal{A}$. The objective of agent interrogation task is to derive the agent model $\mathcal{M}^{\mathcal{A}}$ using $\mathbb{P}$, $\mathbb{A}_H$. Let Θ be the set of possible answers to queries. Thus, strings $\theta^* \in \Theta^*$ denote the information received by $\mathcal{H}$ at any point in the query process. Query policies for agent interrogation task are functions $\theta^* \rightarrow \mathbb{Q} \cup \{\textit{Stop}\}$ that map sequences of answers to the next query that the interrogator should ask. The process stops with the *Stop* query. In other words, $\forall$ answers $\theta \in \Theta$, all valid query policies map all sequences $x\theta$ to *Stop* whenever $x \in \Theta^*$ is mapped to *Stop*. This policy is computed and executed online.

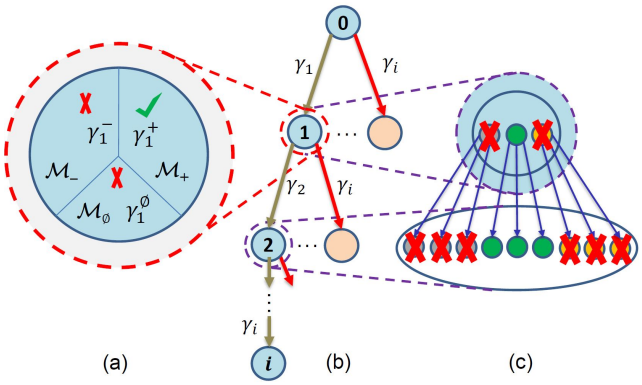

Figure 1: (b) Lattice segment explored in random order of $\gamma_i \in \Gamma$; (a) At each node, 3 abstract models are generated and 2 of them are discarded based on query responses; (c) An abstract model rejected at any level is equivalent to rejecting 3 models at the level below, 9 models two levels down, and so on.

Components of agent models In order to formulate our solution approach, we consider a model $\mathcal{M}$ to be comprised of components called *palm* tuples of the form $\lambda = \langle p, a, l, m\rangle$ where p is an instantiated predicate from the common vocabulary $\mathbb{P}^*$; a is an action from the set of parameterized actions $\mathbb{A}$, $l \in \{pre, eff\}$ and $m \in \{+, -, \emptyset\}$. For convenience, we use subscripts p, a, l or m to denote the corresponding component in a palm tuple. The presence of a palm tuple λ in a model denotes the fact that in that model, the predicate λ_p appears in an action λ_a at a location λ_l as a true (false) literal when sign λ_m is positive (negative), and is absent when $\lambda_m = \emptyset$. This allows us to define the set-minus operation $M \setminus \lambda$ on this model as removing the palm-tuple λ from the model.

We consider two palm tuples $\lambda_1 = \langle p_1, a_1, l_1, m_1\rangle$ and $\lambda_2 = \langle p_2, a_2, l_2, m_2\rangle$ to be *variants* of each other ($\lambda_1 \sim \lambda_2$) iff they differ only on 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 assignments 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 $\gamma^\emptyset = \langle p, a, l, \emptyset\rangle$.

Model abstraction We are now ready to define the notion of abstractions used in our solution approach. Several approaches have explored the use of abstraction in planning (Sacerdoti 1974; Giunchiglia and Walsh 1992; Helmert et al. 2017; Bäckström and Jonsson 2013; Srivastava, Russell, and Pinto 2016). The following definition extends the concept of predicate and propositional domain abstractions (Srivastava, Russell, and Pinto 2016) to allow for the projection of a single *palm* tuple λ.

Definition 1. Let $\mathcal{U}$ be the set of all possible models. The *abstraction of a model* $\mathcal{M}$, on the basis of a palm tuple λ, is given by $f_\lambda(\mathcal{M}) = \mathcal{M} \setminus \{\lambda\}$, where $f_\lambda : \mathcal{U} \to \mathcal{U}$. A set of models X is said to be a model abstraction of a set of models M with respect to a λ-tuple, if $X = \{f_\lambda(m) : m \in M\}$.

We also use the notation $\mathcal{M}' \sqsubset_\lambda \mathcal{M}$ to represent the situation where $f_\lambda(\mathcal{M}) = \mathcal{M}'$. We use this abstraction framework to define a subset-lattice over abstract models (Fig. 1(b)). Note that at each node we can have all possible variants of a pal tuple. E.g., in the node labeled 1 in Fig. 1, we can have models corresponding to γ_1^+, γ_1^-, and $\gamma_1^\emptyset$. Each node in the lattice represents a collection of possible abstract models at the same level of abstraction. As we move up in the lattice, we get more abstracted versions of the models and we get more concretized models as we move downward.

Definition 2. 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 $n_i \to n_j, \ell_N(n_j) = \{\Lambda \cup \{\gamma^k\} | \Lambda \in \ell_N(n_i), \gamma = \ell_E(n_i \to n_j), k \in \{+, -, \emptyset\}\}$.

The supremum $\top$ of the lattice $\mathcal{L}$ is the most abstracted node of the lattice, whereas the infimum $\bot$ is the most concretized node. Also, a node $n \in N$ in this lattice $\mathcal{L}$ can be uniquely identified as the sequence of pal tuples that label edges leading to it from the supremum. As shown in Fig. 1(a), even though theoretically $\ell : n \mapsto 2^{2^\Lambda}$, only one of the sets is stored at each node as the others are pruned out based on some query $\mathcal{Q} \in \mathbb{Q}$. Also, in these model lattices every node has an edge going out from it corresponding to each pal tuple that is not present in the paths leading to it from the most abstracted node. At any stage during the interrogation, nodes in such a lattice are used to represent the set of models that are possible given the agent's responses up to that point. At every step, our query-generation algorithm will create queries that help us determine the next descending edge to take from a lattice node.

Form of agent queries As discussed earlier, based on the responses we pose queries to the agent and infer the agent's model. We express queries as functions mapping models to answers. Recall that $\mathcal{U}$ is the set of possible models, and Θ is the set of possible responses. A *query* $\mathcal{Q}$ is a function $\mathcal{Q} : \mathcal{U} \to \Theta$.

In this paper, we utilize only one class of queries: *plan outcome queries* ($\mathcal{Q}_{PO}$), which are parameterized by a state s_I and a plan π. Let P be the set of predicates $\mathbb{P}$ instantiated with objects O in an environment. $\mathcal{Q}_{PO}$ queries ask the agent the length of the longest prefix of the plan π that can be executed successfully when starting in the state $s_I \subseteq P$ and the resulting final state $s_F \subseteq P$. E.g., "Given that the truck $t1$ and package $p1$ are at location $l1$, what would happen if you executed $load\_truck(p1, t1, l1)$, $drive(t1, l1, l2)$, $unload\_truck(p1, t1, l2)$?"

A response to these queries can be of the form "I can execute the plan till step ℓ and at the end of it $p1$ is in truck $t1$ which is at location $l1$". Formally, the response $\mathcal{R}_{PO}$ for plan outcome queries is a tuple $\langle \ell, s_\mathcal{F}\rangle$, where ℓ is the number of steps for which the plan π was successfully able to run, and $s_\mathcal{F} \subseteq P$ is the final state of the agent after executing ℓ steps of the plan. If the plan π is not executable according to the agent model $\mathcal{M}^\mathcal{A}$ then $\ell < len(\pi)$, otherwise if π is executable then $\ell = len(\pi)$. The final state $s_\mathcal{F} \subseteq P$ such that $\mathcal{M}^\mathcal{A} \models \pi[0 : \ell](s_\mathcal{I}) = s_\mathcal{F}$, i.e. starting with a state $s_\mathcal{I}$, $\mathcal{M}^\mathcal{A}$ successfully executed first ℓ steps of plan π. Thus,

$\mathcal{Q}_{PO} : \mathcal{U} \rightarrow \mathbb{W} \times 2^P$, where $\mathbb{W}$ is the set of whole numbers.

Not all queries are useful, as some of them might not increase our knowledge of the agent model at all. Hence we define some properties associated with each query to ascertain its usability. A query is useful only if it can distinguish between two models. More precisely, a query $\mathcal{Q}$ is said to *distinguish* a pair of models $\mathcal{M}_i$ and $\mathcal{M}_j$, denoted as $\mathcal{M}_i \rfloor^{\mathcal{Q}} \mathcal{M}_j$, iff $\mathcal{Q}(\mathcal{M}_i) \neq \mathcal{Q}(\mathcal{M}_j)$.

Two models $\mathcal{M}_i$ and $\mathcal{M}_j$ are said to be *distinguishable*, denoted as $\mathcal{M}_i \rfloor \mathcal{M}_j$, iff there exists a query that can distinguish between them, i.e. $\exists \mathcal{Q}\ \mathcal{M}_i \rfloor^{\mathcal{Q}} \mathcal{M}_j$.

Given a pair of abstract models, we wish to determine whether one of them can be pruned, i.e., whether there's a query for which at least one of their answer is inconsistent with the agent's answer. Since this is computationally expensive to determine and we wish to reduce the number of queries made to the agent, we first evaluate whether the two models can be distinguished by any query, independent of consistency with the agent. Pruning models under a given query class is possible only if the models are distinguishable. In determining prunability, we must consider the fact that the agent's response may be at a different level of abstraction if the given pair of models is abstract. When comparing the responses of two models at different levels of abstraction, we must also evaluate if the response of abstracted model $\mathcal{M}'$ is consistent with that of the agent.

Definition 3. Let $\mathcal{Q}$ be a distinguishing query for models $\mathcal{M}_i$ and $\mathcal{M}_j$ such that $\mathcal{M}_i \rfloor^{\mathcal{Q}} \mathcal{M}_j$, and $\mathcal{Q}(\mathcal{M}_i) = \langle \ell^i, \langle p_1^i, \ldots, p_m^i \rangle \rangle$, $\mathcal{Q}(\mathcal{M}_j) = \langle \ell^j, \langle p_1^j, \ldots, p_n^j \rangle \rangle$, and $\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) = \langle \ell^A, \langle p_1^A, \ldots, p_k^A \rangle \rangle$. When $\ell^A = len(\pi^Q)$, $\mathcal{M}_i$ is *consistent* with $\mathcal{M}^{\mathcal{A}}$, i.e. $\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}(\mathcal{M}_i)$ if $len(\pi^{\mathcal{Q}}) = \ell^i$ and $\{p_1^i, \ldots, p_m^i\} \subseteq \{p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}}\}$.

Definition 4. 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 $\exists \mathcal{Q} \in \mathbb{Q} : \mathcal{M}_i \rfloor^{\mathcal{Q}} \mathcal{M}_j \wedge (\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}(\mathcal{M}_i) \wedge \mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \not\models \mathcal{Q}(\mathcal{M}_j)) \vee (Q(\mathcal{M}^{\mathcal{A}}) \not\models \mathcal{Q}(\mathcal{M}_i) \wedge \mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}(\mathcal{M}_j))$.

2.1 Solving the Interrogation Task

Our approach iteratively generates pairs of abstract models and eliminates one of them by asking $\mathcal{A}$ queries and comparing its answer with that generated using the abstract models.

Example 1. Consider the case of a delivery robot. Assume that $\mathcal{H}$ is considering two abstract models $\mathcal{M}_1$ and $\mathcal{M}_2$ having only the predicates $at(?p, ?l)$, $at(?t, ?l)$, $in(?p, ?t)$ and the agent's model is $\mathcal{M}^{\mathcal{A}}$ (Fig. 2). $\mathcal{H}$ can ask the agent what will happen if $\mathcal{A}$ loads package $p1$ into truck $t1$ at location $l1$ twice. The agent would respond that it could execute the plan only till length 1, and the state at the time of this failure would be $at(t1, l1) \wedge in(p1, t1)$.

Algorithm 1 (AIA) shows our overall algorithm for interrogating autonomous agents. It takes the agent $\mathcal{A}$, the set of instantiated predicates $\mathbb{P}^*$, 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. We initialize $poss\_models$ as empty set (line 3) representing that we are starting at the most abstract node in model lattice.

(a) $\mathcal{M}^{\mathcal{A}}$'s `load_truck(?p,?t,?l)` action (unknown to $\mathcal{H}$)

`at(?t,?l),`	$\longrightarrow$	`in(?p,?t),`
`at(?p,?l)`		`¬(at(?p,?l))`

(b) $\mathcal{M}_1$'s `load_truck(?p,?t,?l)` action

`at(?t,?l),`	$\longrightarrow$	`in(?p,?t)`
`at(?p,?l)`		

(c) $\mathcal{M}_2$'s `load_truck(?p,?t,?l)` action

`at(?t,?l)`	$\longrightarrow$	`in(?p,?t)`

(d) $\mathcal{M}_3$'s `load_truck(?p,?t,?l)` action

`at(?t,?l)`	$\longrightarrow$	`()`

Figure 2: *load_truck* actions of the agent model $\mathcal{M}^{\mathcal{A}}$ and three abstracted models $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$. Here $X \longrightarrow Y$ means that X is the precondition of an action and Y is the effect.

In each iteration of the main loop (line 4), we keep track of the current node in the lattice. We pick a pal tuple γ corresponding to one of the descending edges in the lattice from a node given by some input ordering of Γ. The correctness of the algorithm does not depend on this ordering. We then store a temporary copy of possible models as new models (line 5) and initialize an empty set at each lattice node to store the pruned models (line 6).

The inner loop (line 7) iterates over the set of all possible models, stored as new_models. We then loop over pairs of modes (line 8), which are later used to generate queries and refine models. For the chosen pair of modes, generate_query() is called (line 9) which returns two models concretized with chosen modes and a query $\mathcal{Q}$ which can distinguish them based on their responses.

We then call *filter_models()* which poses the query $\mathcal{Q}$ to the agent and the two models. Based on their responses, we prune the models whose responses were not consistent with that of the agent (line 11). Then we update the estimated set of possible models represented by $poss\_models$ (line 18).

If we are unable to prune any models at a node (line 14), we modify the pal tuple ordering (line 15). We continue this process until we reach the most concretized node of the lattice (meaning all possible model components $\lambda \in \Lambda$ are refined). The remaining set of models represent the estimated set of models for the agent. The number of resolved palm tuples can be used as a running estimate of accuracy of the derived models. AIA would require $O(|\mathbb{A}| \times |\mathbb{P}^*|)$ queries as there are $|\mathbb{A}| \times |\mathbb{P}^*|$ pal tuples. However, our empirical studies show that we never generate so many queries. The next three sections describe the *generate_query()* (line 9) component, the *filter_models()* (line 10) component, and the *update_pal_ordering()* component (line 15) of AIA in detail.

2.2 Query Generation

The query generation process corresponds to *generate_query()* module in algorithm 1 which takes $\mathcal{M}'$,the pal tuple γ and 2 modes $i, j \in \{+, -, \emptyset\}$ as input and returns the model $\mathcal{M}'$ concretized with palm tuples γ^i and γ^j, and s_I, and π which form the query $\mathcal{Q}$ that can distinguish them,

Algorithm 1 Agent Interrogation Algorithm (AIA)

Input: $\mathcal{A}, \mathbb{A}_H, \mathbb{P}^*, \mathbb{S}$
Output: poss_models
Initialize poss_models $= \{\emptyset\}$
for γ in some input pal ordering Γ **do**
  new_models $\leftarrow$ poss_models
  pruned_models$= \{\emptyset\}$
  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
  new_models $\leftarrow$ new_models $\times \{\gamma^+, \gamma^-, \gamma^\emptyset\} \setminus$ pruned_models
end for

and if possible, satisfy prunability condition as well. As the first step of query generation, we generate two abstract models $\mathcal{M}_i$ and $\mathcal{M}_j$ from the model $\mathcal{M}'$ by adding the pal tuple γ in modes i and j respectively. We now describe how this addition is done.

Adding a palm tuple When adding a palm tuple $\langle p, a, l, m\rangle$, we also add temporary palm tuples having a predicate p_u. Predicate p_u is added at a location l in an action a when its mode is *unknown*. This avoids generating incorrect queries which assume p to be in mode $\emptyset$ instead of actually being unknown. A palm tuple $\lambda = \langle p, a, l, m\rangle$ is added to a model $\mathcal{M}'$ in the following way:

$$\mathcal{M}'' = \begin{cases} \mathcal{M}' \cup \{\langle p, a, l, m\rangle \vee \langle p_u, a, l, +\rangle\} & \text{if } l = \textit{pre}, \\ \mathcal{M}' \cup \langle p, a, l, m\rangle & \text{if } l = \textit{eff}. \end{cases}$$

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

p_u is added only temporarily and is not part of the models $\mathcal{M}_i$ and $\mathcal{M}_j$ returned by the query generation process.

Plan outcome queries have two components, an initial state s_I and a plan π. We get s_I from the input set of random states $\mathbb{S}$ (line 4). To generate the plan π, we reduce the problem of creating plan outcome queries to a planning problem (line 5). This planning problem uses conditional effects in its actions (in accordance with PDDL (Fox and Long 2003)). The idea is to maintain separate copies $\mathcal{P}^{\mathcal{M}_i}$ and $\mathcal{P}^{\mathcal{M}_j}$ of all the instantiated predicates $\mathcal{P} \in \mathbb{P}^*$, and formulate each precondition and effect of an action as a formula of predicates in both the copies of the predicates.

Let the planning problem $P_{PO} = \langle \mathcal{M}^{PO}, s_{\mathcal{I}}, G\rangle$, where $\mathcal{M}^{PO}$ is a model with predicates $\mathbb{P}^{PO} = \mathcal{P}^{\mathcal{M}_i} \cup \mathcal{P}^{\mathcal{M}_j} \cup p_\gamma$, and actions $\mathbb{A}^{PO}$ where for each action $a \in \mathbb{A}^{PO}$, $\textit{pre}(a) =$

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)
  if π **then** break **end if**
end for
return $\langle s_I, \pi\rangle, \mathcal{M}' \cup \{\gamma^i\}, \mathcal{M}' \cup \{\gamma^j\}$

$\textit{pre}(a^{\mathcal{M}_i}) \vee \textit{pre}(a^{\mathcal{M}_j})$ and $\textit{eff}(a) =$

$$\begin{aligned} &(\textit{when } (\textit{pre}(a^{\mathcal{M}_i}) \wedge \textit{pre}(a^{\mathcal{M}_j}))(\textit{eff}(a^{\mathcal{M}_i}) \wedge \textit{eff}(a^{\mathcal{M}_j}))) \\ &(\textit{when } ((\textit{pre}(a^{\mathcal{M}_i}) \wedge \neg \textit{pre}(a^{\mathcal{M}_j})) \vee \\ &\qquad (\neg \textit{pre}(a^{\mathcal{M}_i}) \wedge \textit{pre}(a^{\mathcal{M}_j}))) \, (p_\gamma)), \end{aligned}$$

$s_{\mathcal{I}} = s_{\mathcal{I}}^{\mathcal{M}_i} \wedge s_{\mathcal{I}}^{\mathcal{M}_j}$ is the initial state where $s_{\mathcal{I}}^{\mathcal{M}_i}$ and $s_{\mathcal{I}}^{\mathcal{M}_j}$ are different copies of all predicates in the initial state, and G is the goal formula expressed as $\exists p \ (p^{\mathcal{M}_i} \wedge \neg p^{\mathcal{M}_j}) \vee (\neg p^{\mathcal{M}_i} \wedge p^{\mathcal{M}_j}) \vee p_\gamma$.

With this formulation, the goal is reached when an action in the two original models differ in either precondition (making only one of them executable in a state), or effect (leading to different final states on applying the action). E.g., consider the models with differing action *load_truck*$(p1, t1, l1)$ shown in Fig. 2. From the state $at(t1, l1) \wedge \neg at(p1, l1)$, $\mathcal{M}_2$ can execute *load_truck*$(p1, t1, l1)$ but $\mathcal{M}_1$ cannot. Similarly in state $at(t1, l1) \wedge at(p1, l1)$, executing *load_truck*$(p1, t1, l1)$ will cause $\mathcal{M}^{\mathcal{A}}$ and $\mathcal{M}_1$ to end up in states differing in predicate $at(p1, l1)$. Hence, given the correct initial state, the solution to the planning problem P_{PO} will give the correct distinguishing plan.

Theorem 1. Given a pair of models $\mathcal{M}_i$ and $\mathcal{M}_j$, the planning problem P_{PO} has a solution iff $\mathcal{M}_i$ and $\mathcal{M}_j$ have a distinguishing plan outcome query $\mathcal{Q}_{PO}$.

Proof (Sketch). The input to the planning problem consists of an initial state s_I. If the planner can solve the problem P_{PO} involving s_I and give a plan π, the distinguishing plan query is simply a combination of s_I and π. Similarly, if $\mathcal{M}_i \rfloor^{\mathcal{Q}_{PO}} \mathcal{M}_j$, then giving the initial state s_I as part of planning problem P_{PO}, the solution will be plan π which is part of $\mathcal{Q}_{PO}$. □

2.3 Filtering Possible Models

This section describes the *filter_models()* module in Algorithm 1 which takes as input $\mathcal{M}^{\mathcal{A}}$, $\mathcal{M}_i$, $\mathcal{M}_j$, and the query $\mathcal{Q}$ (generated by the *generate_query()* module explained in section 2.2), and returns the subset $\mathcal{M}_{prune}$ which is not consistent with $\mathcal{M}^{\mathcal{A}}$.

First, the algorithm *asks the query* $\mathcal{Q}$ to $\mathcal{M}_i$, $\mathcal{M}_j$ and the agent $\mathcal{M}^{\mathcal{A}}$. Based on the responses of all three, it determines if the two models are prunable, i.e. $\mathcal{M}_i \langle\rangle \mathcal{M}_j$. As mentioned in Def. 4, checking for prunability involves checking if responses to the query $\mathcal{Q}$ by one of the models $\mathcal{M}_i$ or $\mathcal{M}_j$ is consistent with that of the agent or not.

Theorem 2. Let $\mathcal{M}_i, \mathcal{M}_j \in \{\mathcal{M}_+, \mathcal{M}_-, \mathcal{M}_\emptyset\}$ be the models generated by adding pal tuple γ to $\mathcal{M}'$ which is an abstraction of $\mathcal{M}^{\mathcal{A}}$. Suppose $\mathcal{Q} = \langle I^{\mathcal{Q}}, \pi^{\mathcal{Q}} \rangle$ is a distinguishing query for two distinct models $\mathcal{M}_i, \mathcal{M}_j$, i.e. $\mathcal{M}_i \rfloor^{\mathcal{Q}} \mathcal{M}_j$, 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 $\{p_1^i, \ldots, p_m^i\} \not\subseteq \{p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}}\}$.

Proof (Sketch). Proving by induction, the base case is adding a single pal tuple $\langle p, a, l \rangle$ to an empty model (which is a consistent abstraction of $\mathcal{M}^{\mathcal{A}}$) resulting in 3 models. The 2 models pruned based on Def. 3 can be shown to be inconsistent with $\mathcal{M}^{\mathcal{A}}$, leaving out the single consistent model. For inductive step, it can be shown that after adding a pal tuple to a consistent model it is not consistent with $\mathcal{M}^{\mathcal{A}}$ only if it does not execute the full plan (the precondition is not consistent), or if the end state reached by the model is not a subset of the state of the agent (the effect is not consistent). □

If the models are prunable, then the palm tuple being added in the inconsistent model cannot appear in any consistent model. As we discard such palm tuples at abstract levels (as depicted in Fig. 1 (a)), we prune out a large number of models down the lattice (as depicted in Fig. 1 (c)), hence we keep the intractability of the approach in check and end up asking less number of queries.

2.4 Updating pal ordering Γ

This section describes the *update_pal_ordering()* module in AIA (line 15). It is called when the query generated by *generate_query()* module is not executable by $\mathcal{A}$, i.e. $len(\pi^{\mathcal{Q}}) \neq \ell^{\mathcal{A}}$ in Theorem 2. E.g., consider two abstract models $\mathcal{M}_2$ and $\mathcal{M}_3$ being considered by the human interrogator $\mathcal{H}$ (Fig. 2). At this level of abstraction, $\mathcal{H}$ does not have knowledge of predicate $at(?p, ?l)$, hence it will generate a plan outcome query with initial state $at(?t, ?l)$ and plan *load_truck*$(p1, t1, l1)$ to distinguish between $\mathcal{M}_2$ and $\mathcal{M}_3$. But this cannot be executed by agent $\mathcal{A}$ as the precondition $at(?p, ?l)$ is not satisfied, and we cannot discard any of the models.

Recall that in response to the plan outcome query we get the failed action $a_F = \pi[\ell+1]$ and the final state s_F. Since the query plan π was generated using $\mathcal{M}_i$ and $\mathcal{M}_j$ (which differ only in the newly added palm tuple), they both would've reached the same state $\overline{s}_F$ after executing first ℓ steps of π. Thus, we search $\mathbb{S}$ for a state $s \supset \overline{s}_F$ where $\mathcal{A}$ can execute a_F. Then, we sequentially iterate through predicates $p \in s \setminus \overline{s}_F$ and add them to $\overline{s}_F$ to check if $\mathcal{A}$ can still execute a_F. Similar to Stern and Juba [2017], we infer that any predicate instantiation corresponding to false literals in the state will not appear in a_F's precondition in positive mode. Thus, if $\overline{s}_F$ cannot execute a_F, we add p in negative mode in a_F's precondition, otherwise in $\emptyset$ mode. All the pal tuples whose modes are correctly inferred in this way are therefore removed from pal ordering Γ.

Domain	$\lvert\mathbb{P}^*\rvert$	$\lvert\mathbb{A}\rvert$	$\lvert\hat{\mathcal{Q}}\rvert$	t_μ (ms)	t_σ (μs)
gripper	5	3	17	18.0	0.2
blocksworld	9	4	48	8.4	36
miconic	10	4	39	9.2	1.4
parking	18	4	63	16.5	806
logistics	18	6	68	24.4	1.73
satellite	17	5	41	11.6	0.87
termes	22	7	134	17.0	110.2
freecell	100	10	535	2.24†	33.4†

Table 1: The number of queries ($\lvert\hat{\mathcal{Q}}\rvert$), average time per query (t_μ), and variance of time per query (t_σ) generated by AIA with FD. Average and variance are calculated for 10 runs of AIA, each on a separate problem. †Time in sec.

2.5 Correctness of Agent Interrogation Algorithm

In this section we prove that the set of estimated models returned by AIA are correct and are functionally equivalent to the agent's model, and no correct model is discarded in the process.

Theorem 3. The Agent Interrogation Algorithm (algorithm 1) will always terminate and return a set of models, each of which are functionally equivalent to agent's model $\mathcal{M}^{\mathcal{A}}$.

Proof (Sketch). Theorem 1 and Theorem 2 prove that whenever we get a prunable query, we discard only inconsistent models, thereby ensuring that no correct model is discarded. When we don't get a prunable query, we infer the correct precondition of the failed action using *update_pal_ordering()*, hence the number of refined palm tuples always increase with the number of iterations of the algorithm, thereby ensuring the termination of the algorithm in finite time. □

3 Empirical Evaluation

We implemented AIA in Python in order to evaluate the efficacy of our approach. In this implementation, initial states ($\mathbb{S}$, line 1 in Algorithm 1) were collected by making the agent perform random walks in the simulated environment. We used a maximum of 60 such states for each domain in our experiments. The implementation assumes that domains do not have any constants and that actions and predicates do not use repeated variables (e.g. at(?v, ?v)) although these assumptions can be removed in practice without affecting the correctness of algorithms. The implementation is optimized to store the agents answers to queries; stored responses are used if a query is repeated.

We tested AIA on two types of agents: symbolic-agents that use models from the IPC (unknown to our algorithm) and simulator-agents that report states as images using PDDLGym. We wrote image classifiers for each predicate for the latter series of experiments and used them to derive state representations for use in the AIA algorithm.

All experiments were conducted on 5.0 GHz Intel i9-9900 CPUs with 64 GB RAM running Ubuntu 18.04.

The analysis presented below shows that AIA learns the correct model with a reasonable number of queries, and compares our results with the closest related work,

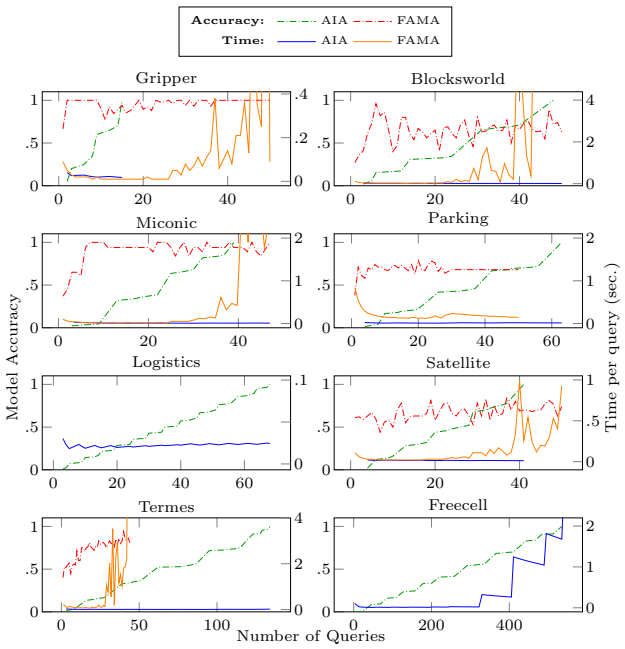

Figure 3: Graphs showing change in model accuracy and running time per query with the increase in number of queries for AIA (with FD) and FAMA (with Madagascar).

FAMA (Aineto, Celorrio, and Onaindia 2019). We use the metric of *model accuracy* in the following analysis: the number of correctly learned palm tuples normalized with the total number of palm tuples. We now explain the experiments in detail.

Experiments with symbolic-agents We initialized the agent with one of the 8 IPC domain models, and ran AIA on the resulting agent. 10 different problem instances were used to obtain average performance estimates.

Table 1 shows that the number of queries required increase with number of predicates and actions in the domain. We used Fast Downward (Helmert 2006) to solve the planning problems. Since our approach is planner-independent, we also tried using FF (Hoffmann and Nebel 2001) and the results were similar. The low variance shows that the method is stable across multiple runs.

Comparison with FAMA We compare the performance of AIA with that of FAMA in terms of stability of models learned and the time taken per query. Since the focus of our approach is on automatically generating useful traces, we provided FAMA randomly generated traces of length 3 (the length of longest plans in AIA-generated queries) of the form used throughout this paper ($\langle s_I, a_1, a_2, a_3, s_G \rangle$). Fig. 3 summarizes our findings. AIA takes lesser time per query and shows better convergence to the correct model while FAMA sometimes reaches nearly accurate models faster but its accuracy continues to oscillate, making it difficult to ascertain when the training process should be stopped (we increased the number of traces provided to FAMA until it ran out of memory). This is because solution to FAMA's internal planning problem introduces spurious palm tuples in its model if the example traces do not capture the complete dynamics of the domain. For logistics, FAMA generated an incorrect planning problem, whereas for freecell it ran out of memory owing to large complexity (AIA also took considerable time for it). Additionally, in domains with negative preconditions like termes, FAMA was unable to learn the correct model. We used Madagascar (Rintanen 2014) with FAMA as it is reported to work best with it (Aineto, Celorrio, and Onaindia 2019). We also tried FD and FF with FAMA, but as the original authors noted, it couldn't scale and ran out of memory on all but few blocksworld and gripper problems where it was much slower than with Madagascar.

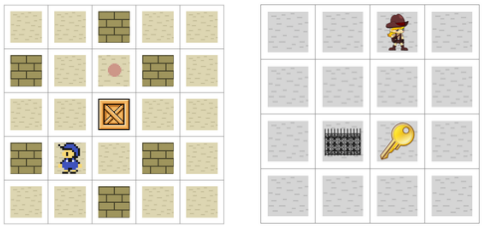

Figure 4: PDDLGym's simulated Sokoban (left) and Doors (right) environments used for the experiments.

Experiments with simulator-agents AIA can also be used with simulator-agents that dont know about predicates and report states as images. In order to test this, we wrote classifiers for detecting predicates from images of simulator-states in the PDDLGym (Silver and Chitnis 2020) framework. This framework provides ground-truth PDDL models, thereby simplifying the estimation of accuracy. We initialized the agent with one of the 2 PDDLGym environments, sokoban and doors shown in Fig. 4. AIA inferred the correct model in both cases and the number of instantiated predicates, actions, and average number of queries (over 5 runs) used to predict sokoban were 35, 3, and 217, and that for doors were 10, 2, and 188.

4 Conclusion

We presented a novel approach to efficiently learn the internal model of an autonomous agent in a STRIPS-like form using query-answering. An advantage of learning such form of models is that they permit assessments of causality (Halpern 2016). We showed that the approach works well for both symbolic and simulator agents, and is better in terms of convergence, stability, and performance than FAMA.

Extending our predicate classifiers to handle noisy state detection, similar to prevalent approaches using classifiers to detect symbolic states (Konidaris, Kaelbling, and Lozano-Perez 2014; Asai and Fukunaga 2018) is a good direction for future work. Some other promising extensions include replacing query and response communication interface between the agent and interrogator with natural language similar to Lindsay et al. [2017], or learning other representations like Zhuo, Muoz-Avila, and Yang [2014].

Acknowledgements

We thank Abhyudaya Srinet for his help with the implementation. This work was supported in part by the NSF under grants IIS 1844325 and IIS 1909370.

References

[2019] Aineto, D.; Celorrio, S. J.; and Onaindia, E. 2019. Learning action models with minimal observability. *Artificial Intelligence* 275:104–137.

[2008] Amir, E., and Chang, A. 2008. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research* 33:349–402.

[2003] Amir, E., and Russell, S. 2003. Logical filtering. In *Proc. IJCAI*.

[2018] Asai, M., and Fukunaga, A. 2018. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *Proc. AAAI*.

[2013] Bäckström, C., and Jonsson, P. 2013. Bridging the gap between refinement and heuristics in abstraction. In *Proc. IJCAI*.

[2016] Bryce, D.; Benton, J.; and Boldt, M. W. 2016. Maintaining evolving domain models. In *Proc. IJCAI*.

[2019] Camacho, A., and McIlraith, S. A. 2019. Learning interpretable models expressed in linear temporal logic. In *Proc. ICAPS*.

[2011] Cresswell, S., and Gregory, P. 2011. Generalised domain model acquisition from action traces. In *Proc. ICAPS*.

[2009] Cresswell, S.; McCluskey, T.; and West, M. 2009. Acquisition of object-centred domain models from planning examples. In *Proc. ICAPS*.

[1971] Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3-4):189–208.

[2003] Fox, M., and Long, D. 2003. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20(1):61–124.

[1992] Giunchiglia, F., and Walsh, T. 1992. A theory of abstraction. *Artificial Intelligence* 57(2-3):323–389.

[2016] Halpern, J. Y. 2016. *Actual Causality*. The MIT Press.

[2017] Helmert, M.; Haslum, P.; Hoffmann, J.; et al. 2017. Flexible abstraction heuristics for optimal sequential planning. In *Proc. ICAPS*.

[2006] Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

[2001] Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

[2011] International Planning Competition. 2011. International planning competition domains.

[1996] Khardon, R., and Roth, D. 1996. Reasoning with models. *Artificial Intelligence* 87(1-2):187–213.

[2014] Konidaris, G.; Kaelbling, L. P.; and Lozano-Perez, T. 2014. Constructing symbolic representations for high-level planning. In *Proc. AAAI*.

[2018] Kučera, J., and Barták, R. 2018. LOUGA: Learning planning operators using genetic algorithms. In *Knowledge Management and Acquisition for Intelligent Systems*.

[2017] Lindsay, A.; Read, J.; Ferreira, J.; Hayton, T.; Porteous, J.; and Gregory, P. 2017. Framer: Planning models from natural language action descriptions. In *Proc. IJCAI*.

[2019] Ng, J. H. A., and Petrick, R. P. A. 2019. Incremental learning of planning actions in model-based reinforcement learning. In *Proc. IJCAI*.

[2014] Rintanen, J. 2014. Madagascar: Scalable planning with sat. In *Proc. 8th International Planning Competition.*

[1974] Sacerdoti, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5(2):115–135.

[2009] Settles, B. 2009. Active learning literature survey. Technical Report 1648, University of Wisconsin-Madison Department of Computer Sciences.

[2020] Silver, T., and Chitnis, R. 2020. PDDLGym: Gym environments from PDDL problems.

[2016] Srivastava, S.; Russell, S.; and Pinto, A. 2016. Metaphysics of planning domain descriptions. In *Proc. AAAI*.

[2017] Stern, R., and Juba, B. 2017. Efficient, safe, and probably approximately complete learning of action models. In *Proc. IJCAI*.

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

[2014] Zhuo, H. H.; Muoz-Avila, H.; and Yang, Q. 2014. Learning hierarchical task network domains from partially observed plan traces. *Artificial Intelligence* 212:134 – 157.